휴대장치를 위한 응용프로그램 특성에 따른 적응형 전력관리 기법

(An Application-Specific and Adaptive Power Management Technique for Portable Systems)

이 강 웅 † 이 재 진 †† 신 현 식 ††
(Bernhard Egger) (Jaejin Lee) (Heonshik Shin)

요 약 본 논문은 dynamic voltage scaling (DVS)를 지원하는 휴대장치를 대상으로 하여 응용프로그램 특성에 따라 실행 중에 전력관리 기법이 다르게 적용되는 적응형 전력관리 기법에 대하여 소개한다. 본 논문의 전력관리 기법은 멀티태스킹 시스템에서 실행되는 soft real-time 프로그램의 memory subsystem 과 프로세서의 실행 시간(run time) 및 유휴 시간(idle time)을 고려하여 프로그램 실행 중에 최적의 DVS가 적용될 수 있도록 하여 전력을 관리한다. 세부적인 전력 및 실행시간 프로파일 정보를 이용할 수 있도록 adaptive power manager(APM)를 개발하여 운영체제에 연동시켰고, Post-pass 최적화기는 APM을 위한 적응형 API를 프로그램의 실행이미지에 삽입하여 실행 중 DVS가 적용되는 코드영역을 표시한다. APM은 프로그램 실행 중에 cache miss 수 등을 측정하는 CPU의 performance counter들을 관찰한다. Performance counter들의 값을 바탕으로 CPU와 memory 중심의 코드 영역을 구분하여 프로세서의 유휴 시간에 대한 분석을 수행하고, 표시된 코드영역들에 대한 최적정 전압과 동작 클락을 결정하여 시스템에 반영한다. 제안하는 기법의 효과를 보이기 위하여 Intel의 XScale 프로세서 상에서 동작하는 Windows CE에 본 기법을 구현하였고, 실험을 통하여 본 논문에서 제시하는 기법이 영상이나 음성 데이타를 해독하는 프로그램과 같이 정기적으로 비슷한 일을 수행하는 프로그램에서 효과적임을 알 수 있었다. 실험 결과 본 기법으로 유휴시간에 프로세서를 저전력모드로 바꾸는 기존의 고전적인 전력 관리 기법보다 전체 시스템 전력 소모를 9% 더 절약할 수 있었다.

키워드 : dynamic voltage scaling, 휴대장치

Abstract In this paper, we introduce an application-specific and adaptive power management technique for portable systems that support dynamic voltage scaling (DVS). We exploit both the idle time of multitasking systems running soft real-time tasks as well as memory- or CPU-bound code regions. Detailed power and execution time profiles guide an adaptive power manager (APM) that is linked to the operating system. A post-pass optimizer marks candidate regions for DVS by inserting calls to the APM. At runtime, the APM monitors the CPU's performance counters to dynamically determine the affinity of the each marked region. For each region, the APM computes the optimal voltage and frequency setting in terms of energy consumption and switches the CPU to that setting during the execution of the region. Idle time is exploited by monitoring system idle time and switching to the energy-wise most economical setting without prolonging execution. We show that our method is most effective for periodic workloads such as video or audio decoding. We have implemented our method in a multitasking operating system (Microsoft Windows CE) running on an Intel XScale-processor. We achieved up to 9% of total system power savings over the standard power management policy that puts the CPU in a low power mode during idle periods.

Key words : dynamic voltage scaling, portable systems

· This work was supported in part by the Ministry of Education under the Brain Korea 21 Project, by the MIC/IITA/ETRI SoC Industry Promotion Center, Human Resource Development Project for IT SoC Architect, by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software], and by the Microsoft Research Innovation Excellence Award for Embedded Systems. ITC at Seoul National University provided research facilities for this study.

† 학생회원 : 서울대학교 컴퓨터공학부
bernhard@aces.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
jlee@aces.snu.ac.kr
shinhs@snu.ac.kr

논문접수 : 2007년 3월 19일
심사완료 : 2007년 5월 21일

1. Introduction

As battery-operated portable devices become ubiquitous, satisfying demands for effective energy reduction techniques is ever more important. The device should use as little energy as possible while still providing satisfactory performance; in other words, the quality of service (QoS) requirement must be met.

Dynamic voltage (and frequency) scaling (DVS) is an effective method to reduce the power consumption of the CPU [4]. DVS is supported by a number of processors on the market such as AMD's Mobile Athlon [2], Intel's XScale [15], and Transmeta's Crusoe [11]. Power dissipation of CMOS circuits is composed of static and dynamic power. Static power is caused by leakage in transistors. In this work, leakage is assumed to be constant. Dynamic power is directly proportional to the square of the input voltage

$$P \propto C \cdot V_{dd}^{2} \cdot f$$

where C is the switched capacitance, V_{dd} is the supply voltage, and f is the clock frequency. DVS enables one to reduce the supply voltage at runtime to reduce power/energy consumption. However, lowering the supply voltage increases the device delay and must therefore be accompanied by a reduction in clock frequency, i.e. the voltage and the frequency must be lowered together [16].

The goal of the vast majority of research on DVS is to conserve energy without the user noticing a performance degradation. There are basically two ways to achieve that: the first one is detecting and exploiting slack and idle time, the other one is applying DVS in memory-bound and CPU-bound code regions.

In real-time operating systems, slack time is defined as the time a periodic task finishes before its periodic deadline. Idle time, on the other hand, is the time that is left in a certain period of time after all threads have been scheduled. Idle time can occur both in real-time and non real-time systems. Exploiting slack and/or idle time is common for real-time systems where task deadlines are known in advance. By modifying the task scheduler and analyzing the (remaining) worst case execution time (WCET) of periodic real-time tasks, the CPU is switched to a lower frequency to make use of slack and/or idle time, which saves a considerable amount of power [1,3,19,23,24]. In non real-time multitasking operating systems, there is no slack time. The idle time varies with the system load and is not known in advance. There are several approaches to estimate the future idle time. Weiser's original algorithm, called PAST [26], predicts idle time based on the idle time observed in the past. More sophisticated and accurate prediction algorithms have been developed in [21,25].

The second way to save energy without severely degrading performance is to apply DVS in memory-bound and CPU-bound code regions. If the target architecture supports asynchronous memory accesses, the CPU often stalls during the execution of memory-bound code regions due to frequent cache misses. For such regions, reducing the CPU frequency does not affect performance significantly because the execution time is dominated by the memory access latency. This approach can be found in [12,22,27].

This paper introduces an application-specific and adaptive power management technique for portable systems. Our goal is to design an adaptive execution algorithm that reduces the total system energy using DVS without modifying the scheduler. An adaptive power manager (APM) is linked to the operating system, and user programs call the power manager through its API. Our work integrates both of the two methods mentioned above. It detects memory-bound code regions at run time and slows down the CPU in these regions whenever these regions are entered again. Moreover, it monitors the idle time to detect periodic behavior and then prolongs the execution time of these periodic regions by reducing the CPU clock frequency. Some previous work [6,13] specifically targets MPEG decoding and evaluates the system energy state at the frame level (i.e., the power APIs are called whenever decoding of a new video frame begins). However, these program points are difficult to detect at compile time without deep knowledge of the application. Our technique, in contrast, uses a post-pass optimizer to find likely candidates for energy sav-

ings and can detect periodic behavior even if the APM API calls do not occur at specific program points.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 explains our adaptive power management scheme in detail. Section 4 presents the energy models. Section 5 describes the evaluation environment and the benchmarks used, and Section 6 presents the results. The paper concludes with Section 7.

2. Related Work

DVS scheduling policies have been exhaustively studied at the hardware, operating system and compiler level. Weiser *et al.* [26] proposed an interval-based algorithm called PAST that predicts the workload based on the last couple of epochs. Ishihara and Yasuura [16] give an ILP formulation for task-based DVS, but do not take into account the transition costs. Swaminathan and Chakrabarty [24] make some approximations to have their ILP model incorporate the transition costs. Aydin *et al.* [3] reclaim slack time of hard real-time tasks and redistribute it to the scheduled task. In [25], Varma *et al.* present an interesting approach to set the voltage of the CPU in the scheduler based on a feedback-loop controller as commonly found in control systems. Lorch and Smith [19] introduced PACE, a framework that works in conjunction with any existing OS-level DVS scheduling policy. They show that optimal energy savings are achieved by incrementally increasing the voltage/frequency as a task approaches its deadline. The work of Xu *et al.* [28] proposes an improved version of PACE for systems with discrete voltage/frequency settings.

At the compiler level, research has targeted the use of mode-set instructions. Shin *et al.* [23] insert mode-set instructions on the edges in the control flow between basic blocks. A not taken IF branch, for example, shortens the total WCET and so presents an opportunity to lower the voltage/speed. Another possibility is presented by loops that are not repeated the maximum number of iterations. [23] considers both cases. Hsu and Kremer [12] suggest lowering the voltage/frequency in memory-bound regions that are detected by analyzing profiling information. Saputra *et al.* [22] present an ILP formulation for computing the optimal voltage/frequency mode for each loop nest. Xie *et al.* [27] extend the ILP to include practical transition costs.

In [1], Aboughazaleh *et al.* present a DVS scheduling policy that involves both the compiler and the operating system. At compile-time, the compiler inserts mode-set instructions or mode-set hints. The OS is invoked by the latter one and sets the mode based on actual slack available.

Some research explicitly targets video codecs (MPEG). Hughes *et al.* [13] measure the IPC and the power consumption for each of three MPEG frame types and set the optimal mode at the beginning of each frame type. Choi *et al.* [6] use the information provided by the performance counters of the XScale processor to determine the best setting for each frame type. Both require careful analysis of the source code to insert the instrumentation code at the correct locations (i.e., at the beginning/end of the code handling the various frame types). In [7], the same authors propose a similar approach to our previous work [8]. By monitoring the performance counters, a modified scheduler switches to the energy-optimal frequency for each application based on an energy model obtained beforehand. However, their approach does not consider idle time and requires modifications to the operating system.

This paper extends the DVS scheduling policy introduced in [8]. It exploits both idle/slack time and can also switch the mode during memory-bound or CPU-bound code regions. Unlike [6,8,13], no source code analysis is required. Instead, a post-pass optimizer inserts the instrumentation code automatically. At run-time, an APM monitors the performance counters and the idle/slack time to heuristically compute and set the optimal mode. The APM is independent of the operating system and can thus also be attached to systems where the source code is unavailable.

3. Adaptive Power Management

Our adaptive power management scheme is compromised of three distinct phases: characterization of the system, inserting APM API calls, and adaptive

execution. First, a detailed power consumption and execution time profile is obtained by running various synthetic benchmarks on the target system. The data obtained is approximated by linear models which are later used by the APM at runtime to compute the energy-optimal voltage/frequency setting. Next, a post-pass optimizer analyzes the application binary and inserts code around potential candidate regions to invoke the APM. Finally, the instrumented binary is run on a system that is linked with the APM. In a first phase, the APM gathers information about the code regions identified by the post-pass optimizer by monitoring the hardware performance counters. Once the APM has collected enough information, it computes and switches to the energy-optimal setting at the beginning of each region. The APM also monitors the system idle time. The following three sections describe these three phases in more detail.

3.1 Characterization of the System

The first phase of our adaptive power management scheme is to obtain a detailed power and execution time profile of the system.

Even for a fixed frequency/voltage setting, the CPU power consumption varies slightly depending on the cache miss ratio and the memory-to-ALU instruction ratio. In CPU-bound code regions with few memory accesses and a low cache miss ratio, the CPU consumes more power because almost no CPU stalls occur, whereas in memory-bound code regions with many memory accesses and a high cache miss ratio, the CPU consumes less power due to memory stalls. On the other hand, the memory system consumes more power in memory-bound code regions and less in CPU-bound code regions.

The execution time of CPU-bound code regions is dominated by the number of instructions, while that of memory-bound code regions is dominated by the number of memory operations and the cache miss ratio.

To obtain a detailed power and execution time profile, we run various synthetic benchmarks on the target system while we measure the execution time and the overall system power consumption. For the synthetic benchmarks, we use a simple loop that is executed a few million times. We vary both the ALU-to-memory instruction ratio as well as the (data) cache miss ratio. The extent ranges from (almost) only ALU instructions to (almost) only memory instructions, and the cache miss ratio is varied between 0% and 100%. For each voltage/frequency setting of the CPU, we measure the overall system power consumption and the execution time for each benchmark (Figure 4).

The measured power and execution time are then approximated by several linear functions. Using these functions, we construct power and execution-time estimation models that will be used by the APM at run-time to determine the energy-optimal mode. We explain our estimation models in detail later in Section 4.

3.2 Application Binary Instrumentation

The purpose of the instrumentation phase is to insert calls to the APM at suitable locations. Candidates regions for applying DVS are frequently executed code regions such as loops and loop nests. These candidate regions can be hand-picked by the programmer or selected by a compiler/post-pass optimizer based on static and/or dynamic call graph information. Each candidate region is enclosed by a pair of APM API calls as shown in Figure 1. Each region is assigned a unique identifier.

In this work, we use a post-pass optimizer do automatically detect and instrument candidate regions. Using a post-pass optimizer instead of selecting the regions by hand or at compile-time has several advantages: manually inserting candidate regions into the source code requires deep knowledge of the code to be optimized. This is especially difficult if the code has been written some time ago and the programmer inserting the API calls is not the original author of the code. These problems can be avoided by having the compiler enclose the candidate regions by API calls, which requires that the source code for the application is readily available. A post-pass optimizer, on the other hand, operates fully automatically, does not require access to the source code, and can even instrument libraries.

Our post-pass optimizer is based on the one presented in [9,10]. The inputs to the post-pass optimizer are application binaries and libraries in the ARM ELF file format. The object files are dis-

```
x = ...;
apmRegionsStart(id);
for (i=1; i<N; i++) {
  a = i * ...;
  do {
    x += a + ...;
  } while (a);
  if (x > MAX_X) x = ...;
}
apmRegionStop(id);
y = c(x);
```

Figure 1 A loop nest marked with APM calls

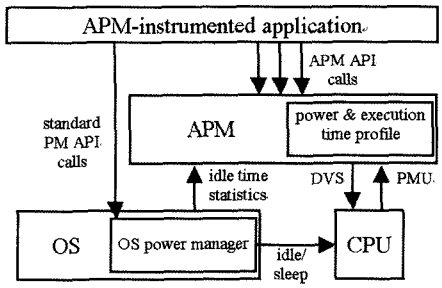


Figure 2 Interaction of OS, APM and APM-aware applications

assembled into code and data segments. Code blocks are further divided into functions composed of basic blocks. The post-pass optimizer then detects natural loops in the static call graph and inserts two call statements to the APM, one in front of the loop head and after the loop body. If a loop contains inner loops, the APM calls are only inserted before and after the outermost loop. After all loops and loop nests are instrumented with APM API calls, the post-pass optimizer re-assembles the code into an optimized executable binary.

3.3 Adaptive Execution

The adaptive power manager (APM) is part of the operating system. It enhances the standard power-down-on-idle power manager, but does not require any modifications to the scheduler or the existing power manager; it is merely linked to the operating system. This is useful in situations where not the full source code of the OS is available.

The APM is invoked by applications that have been instrumented with APM calls. For each region, the APM measures the CPU performance counters (PMU) and determines their characteristics. Based on detailed energy models of the system, the APM then selects the energy-efficient DVS setting (Figure 2). Depending on the data obtained through the CPU's performance counters and the idle time reported by the operation system, the APM operates in two energy optimization modes.

3.3.1 CPU-/Memory-bound Code Regions

The APM constantly monitors the characteristics (instruction count, cache accesses, and cache misses) of each region. Based on the CPU's performance counters and the models obtained through the system characterization (Section 3.1), it computes the energy-optimal CPU frequency by estimating the execution time and the energy consumption of the region. Whenever a region is entered, the APM switches the CPU to the energy-optimal frequency and back to the original frequency at the end of the region. The APM maintains a correction factor for each region by comparing the estimated execution time the measured time. The correction factor is used in successive computations to improve accuracy.

Whenever the cache miss ratio or the execution time of a region changes significantly, the APM recalculates the time and energy requirements of the different frequencies and selects the one that minimizes energy consumption. A maximum relative performance penalty limit can be used to ensure QoS.

3.3.2 Exploiting Idle Time

Usually, idle time is detected and exploited in the OS scheduler. Since the APM is not an integral part of the operating system, but rather just linked to it, exploiting idle time is not straightforward.

To detect periodic behavior, such as the decoding of a video frame once every $1/24^{th}$ of a second, the APM not only monitors the performance counters, but also the system idle time. The system has been idle if the idle time between two invocations of the APM increases. In this case, the APM marks the current region r_i as the head of a periodic p, r_p. The period ends as soon as the idle time increases again. Figure 3 illustrates the idea.

Once a period p has been detected, the APM measures the characteristics of the code over one full period. Then, based on an algorithm similar to PAST [26], it estimates the CPU frequency that

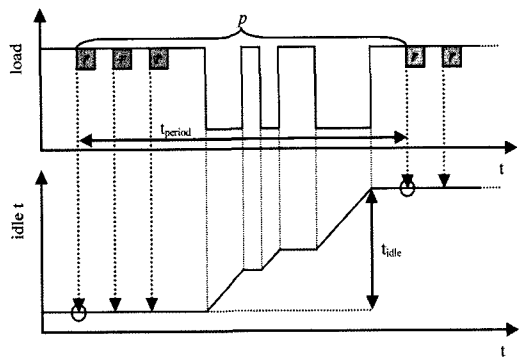


Figure 3 Detecting periodic behavior by looking at the idle time. The APM API calls are placed around the region ***r*** (grey boxes). The APM detects the periodic region ***p*** by looking at the idle time, here concluding that a new period starts every three invocations of region ***r***.

minimizes the overall system energy consumption, i.e. the frequency at which the CPU is slowed down just as much such that there is no idle time left.

The APM constantly monitors and compares the measured execution time to the estimated execution time. For each frequency, it computes a correction factor. The next estimate is multiplied by this factor to improve the accuracy of the execution time prediction.

To avoid oscillation between two frequency settings, the APM only switches to a lower or higher frequency if the idle time exceeds a threshold value. The threshold values when to lower and when to increase the CPU speed can be chosen independently.

4. Estimating Energy Consumption and Execution Time

Accurate estimation of execution time and power consumption is one of the most crucial parts of our method. We execute synthesized workloads and measure the overall system energy consumption as well as the execution time for all of these workloads. For each frequency setting, we vary the cache miss ratio and the memory-to-ALU ratio. Figure 4 shows an example of an energy profile. The measured data is approximated by several

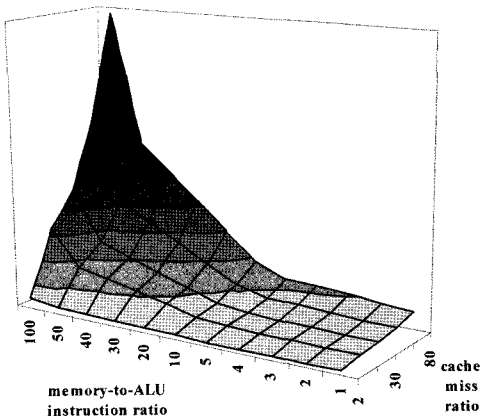


Figure 4 Energy profile

linear functions and stored in a table inside the APM.

To estimate the power consumption and execution time of a given region r, we use the following formulas:

$$P_r = P(f, instr, acc, miss)$$
$$T_r = T(f, instr, acc, miss)$$

where f, $instr$, acc, and $miss$ are the frequency, the number of instructions, the number of cache hits, and the number of cache misses respectively. By linearly interpolating the data in the tables, we can estimate the power and time consumption for any given load rate ($acc/instr$) and cache miss rate ($miss/acc$).

For a region r, the energy consumption is estimated by the following formula:

$$E_r = P(f_{new}, instr, acc, miss) \cdot T(f_{new}, instr, acc, miss) + E_{switch}(f_{old}, f_{new})$$

where $E_{switch}(f_{old}, f_{new})$ is the overhead to switch from f_{old} to f_{new} and back.

The energy consumption for a periodic region p can be computed by

$$E_p = n \cdot E_r(f_r, instr_r, acc_r, miss_r) + P(f_p, instr_p, acc_p, miss_p) \cdot T(f_p, instr_p, acc_p, miss_p) + P_{idle} \cdot (t_p - n \cdot T_r - T(f_p, instr_p, acc_p, miss_p))$$

under the constraint

$$t_p \geq n \cdot T_r + T(f_p, instr_p, acc_p, miss_p)$$

where t_p is the measured period of p; n is the number of regions per period; $instr_r$, acc_r and $miss_r$ are the performance counters for the region r; and

$instr_p$, acc_p and $miss_p$ are those for the entire period p without the counters for the regions.

5. Evaluation Environment

We have implemented our APM for Microsoft Windows CE.NET 4.2 [20] running on a development board with an Intel XScale PXA255 processor [14]. The board is equipped with a PXA255 processor, 64MB of SDRAM, 64MB of Flash memory, Ethernet, USB, Infrared, and a 320×160 pixel-wide LCD screen. The board is powered by two voltages; 3.3V and 5.0V. The power measurements were performed using a high-performance data acquisition device (DAC) that can sample several inputs at 100 KHz simultaneously (Figure 5). The synchronization of the power measurement with the start/finish of the experiment is signaled by means of the general purpose I/O (GPIO) pins: at the beginning of a measurement, a specific GPIO pin is set *high*, and the end of the measurement is signaled by setting the GPIO pin *low*.

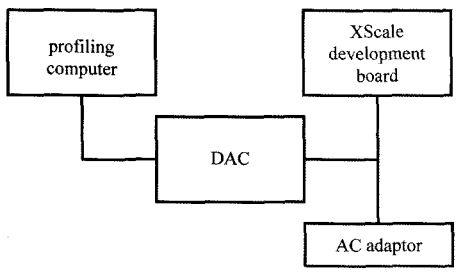


Figure 5 Experimental Setup

We use mpeg2dec and 3dview to evaluate our technique. Mpeg2dec is an MPEG2 video decoder from MediaBench [17], and 3dview renders a three-imensional view of a virtual world using only fixed-point operations.

6. Experimental Results

6.1 CPU-/Memory-bound Code Regions

The Intel XScale PXA255 processor supports a total of 10 different frequency settings. For each frequency setting, not only the CPU core clock varies, but also the CPU-to-memory bus, the memory-to-LCD, and also the SDRAM frequencies differ (Table 1). The data obtained from the system characterization reveals a surprising result: the fastest frequency setting with a CPU clock of 398MHz is the optimal setting in terms of energy consumption for any given code region independent of the instruction mix or the cache miss ratio (Figure 6). This is because not only the CPU clock, but also the CPU-to-memory bus speed is reduced. In other words, even though the CPU stalls less for high cache miss ratios at lower speeds, the slower speed of the memory bus increases the execution time so much that the additional energy consumed by the longer execution time cancels out the energy savings obtained from running at a lower speed. Note that this does not contradict the results in [12] because in their experimental setup the memory bus speed does not change with the CPU speed.

Table 1 DVS Modes of the Intel XScale PXA255 CPU

setting	f_{CPU} [MHz]	$f_{CPU-MEM}$ [MHz]	$f_{MEM-LCD}$ [MHz]	f_{SDRAM} [MHz]
0	99.5	50	99.5	99.5
1	199.1	50	99.5	99.5
2	298.6	50	99.5	99.5
3	132.7	66	132.7	66
4	199.1	99.5	99.5	99.5
5	298.6	99.5	99.5	99.5
6	398.1	99.5	99.5	99.5
7	265.4	132.7	132.7	66
8	331.8	165.9	165.9	83
9	398.1	196	99.5	99.5

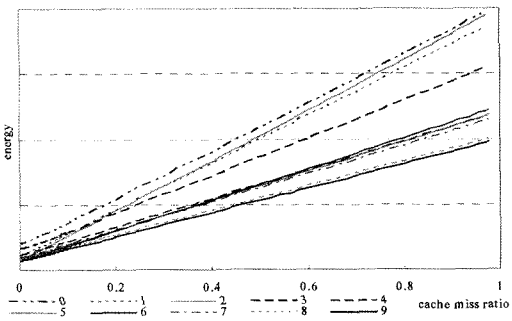


Figure 6 Energy consumption of the DVS modes for different cache miss ratios. Note that the fastest mode, setting 9, consumes the least amount of energy independent of the cache miss ratio

6.2 Exploiting Idle Time

Figure 7 shows the measured system power con-

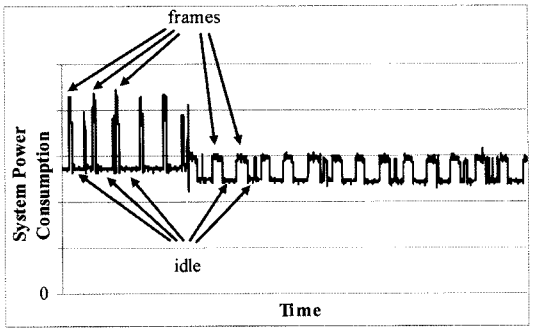


Figure 7 Exploiting idle time for MPEG decoding

sumption for MPEG. The CPU runs at the highest frequency when the benchmark starts running. The spikes represent the decoding of one frame every 1/24th of a second. The beds between the spikes occur because the CPU is put into a low-power sleep mode by Window CE's standard power manager (PM) whenever the CPU is idle. Spikes between frames (e.g. between frame 1 and 2) are caused by other tasks running simultaneously. The APM detects the periodic behavior and starts monitoring the performance counters for a couple of frames. After it has collected enough information, it computes the energy-optimal DVS setting and switches to that mode at frame 6. After switching to the slower frequency, decoding a frame takes longer as can be seen from the noticeably "wider" spikes. Even after switching to the slower frequency, there is still some idle time left. This is because the CPU is already running at the lowest speed possible. Windows CE's power manager continues to put the CPU into a low-power sleep mode during idle periods, but thanks to the reduced core voltage and the reduced memory bus frequency, the power consumption during idle periods is noticeably lower than before.

Table 2 shows the results for different video sizes. We compare the results to the default Windows CE power management policy of putting the CPU into a low-power sleep mode during idle periods. For a video size of 256×192 pixels, our APM achieves a reduction in energy consumption of 4%, for 160×120 pixels 7.3%, and for 80×60 pixels 8%. The energy savings increase as the video size decreases because the smaller the video

Table 2 Results for MPEG-2 Video Decoding

video size	Standard PM		APM		Power savings (%)
	avg.power [W]	avg.jitter [ms]	avg.power [W]	avg.jitter [ms]	
80×60	2.873	1.10	2.644	1.29	8.0
160×120	3.189	1.10	2.955	1.06	7.3
256×192	3.507	1.12	3.365	1.25	4.0

size the more idle time the APM can exploit. Table 2 also reveals that the average jitter slightly increases with our APM but is still within an unnoticeable range. The variance of the jitter is mainly caused by the frequency switch. The videos used for our benchmarks were rather short (about 50 frames each). For longer videos, the average jitter approaches the jitter value obtained with the default Windows CE power management policy.

Figure 8 shows the results for 3dview. 3dview renders a three dimensional landscape with 30 frames per second. The number of objects to be rendered depends on the position of the camera which follows a pre-recorded path through the virtual world. Figure 8(a) shows the complexity of the scene, i.e. the number of objects to be rendered. Figure 8(b) displays the idle time as reported by Windows CE. Finally, Figure 8(c) plots the CPU frequency chosen by the APM. The indices of the frequencies correspond to those in Table 1. At the beginning of the run, our APM immediately switches the CPU from the fastest to the slowest mode, and then settles on mode 1 (200MHz core clock). The idle time decreases as the complexity of the scene increases, and the APM switches to faster CPU frequencies. When the idle time starts increasing again, the APM selects a lower frequency to save as much energy as possible.

Applying our technique to the workload depicted in Figure 8(a) reduced the overall system energy consumption by 9% with an average jitter of 0.02ms per frame.

7. Conclusions

In this paper, we have introduced an application-specific and adaptive DVS algorithm that can exploit both memory-bound code regions as well as system idle time. Our adaptive power manager (APM)

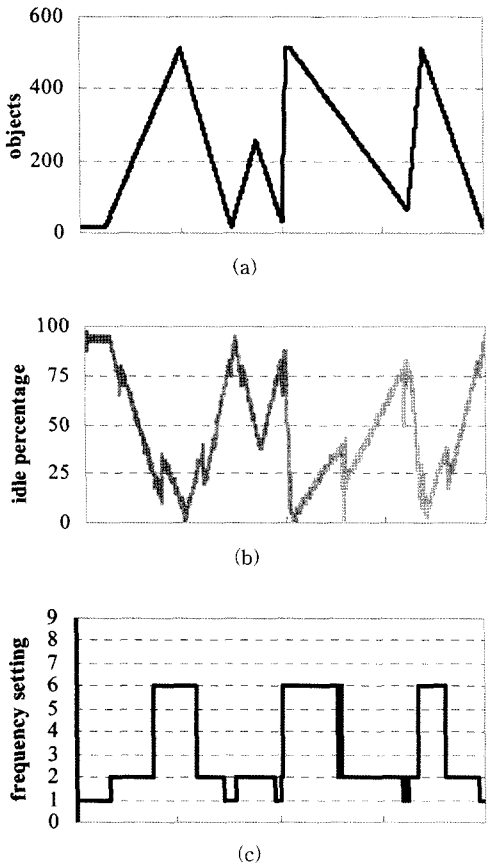


Figure 8 Results for 3dview. (a) shows the scene complexity (i.e., the number of objects to render), (b) the idle percentage, and (c) is CPU frequency mode selected by the APM

is linked to the existing operating system without the need of modifying the scheduler or the power managers of the OS. The APM contains linear models of a detailed power consumption and execution time profile of the system. The profile is obtained by running various synthetic instruction mixes with different cache miss ratios on the target platform while monitoring execution time and power consumption.

A post-pass optimizer inserts calls to the APM around potential candidate regions. At runtime, the APM determines the characteristics of each region by monitoring the CPU's performance counters. It then computes the energy-optimal DVS setting for each region and switches the CPU to that mode when entering the region.

Idle time is exploited by detecting periodic behavior of the APM regions and by monitoring system idle time. Depending on the characteristics of one period and the available idle time, the APM switches to the energy-optimal mode. If the idle percentage falls below a threshold, the APM switches to a faster mode to ensure that all deadlines can be met.

We have performed our experiments on an XScale development platform running Windows CE. Even though the development platform consumes a lot more energy compared to an energy-optimized handheld product, we achieve overall system energy savings of up to 9% compared to the default Windows CE power manager.

References

[1] N.AbouGhazaleh, *et al.* Collaborative Operating System and Compiler Power Management for Real-Time Applications. In ACM Transactions on Embedded Computing Systems (TECS), February 2006.

[2] Advanced Micro Devices. AMD Power Now Technology. 2000.

[3] H.Aydin, *et al.* Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In the 22nd IEEE Real-Time Systems Symposium, December 2001.

[4] T.Burd, R.Brodersen. Energy efficient CMOS microprocessor design. In the 28th Hawaii International Conference on System Sciences, 1995.

[5] A.Chandrakasan, and R.Brodersen. Low Power Digital CMOS Design. Kluver Academic Publishers, 1995.

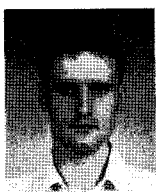
[6] K.Choi, R.Soma, M.Pedram. Off-chip Latency-Driven Dynamic Voltage and Frequency Scaling for an MPEG Decoding. In Design Automation Conference (DAC'04), June 2004.

[7] K.Choi, W.Lee, R.Soma, M.Pedram. Dynamic Voltage and Frequency Scaling Under a Precise Energy Model Considering Variable and Fixed Components of the System Power Dissipation. In International Conference on Computer-Aided Design (ICCAD'04), November 2004.

[8] B.Egger, J.Lee, H.Shin. An Application-Specific and Adaptive Power Management Technique. In First Int'l Workshop on Power-Aware Real-Time Computing (PARC'04), September 2004.

[9] B.Egger *et al.* A Dynamic Code Placement Technique for Scratchpad Memory using Postpass Optimization. In Proceedings of the 2006 international conference on Compilers, architecture and

synthesis for embedded systems (CASES'06), October 2006.
[10] B.Egger, J.Lee, H.Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT'06), October 2006.
[11] M. Fleischmann. Longrun Power Management. White Paper of Transmeta Corporation, January 2001.
[12] C-H. Hsu, U.Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In Programming Language Design and Implementation (PLDI'03), June 2003.
[13] C.Hughes, J.Srinivasan, S.Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In Proceedings of the 34th Annual International Symposium on Microarchitecture, December 2001.
[14] Intel PXA255 Processor. Developer's Manual, January 2004.
[15] Intel XScale Microarchitecture. http://developer.intel.com/design/intelxscale/
[16] T.Ishihara, H.Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), August 1998.
[17] C.Lee, M.Potkonjak, W.H.Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In 30th Annual International Symposium on Microarchitecture (Micro'97), December 1997.
[18] J.Lee, Y.Solihin, J.Torellas. Automatically Mapping Code on an Intelligent Memory Architecture. In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), January 2001.
[19] J.Lorch, A.Smith. Improving dynamic voltage algorithms with PACE. In Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2001), June 2001.
[20] Microsoft. Windows CE.NET. http://msdn.microsoft.com/embedded/ce.net/
[21] G.A.Paleologo, L.Benini, G.De Micheli. Policy Optimization for Dynamic Power Management. In Design Automation Conference (DAC'98), June 1998.
[22] H.Saputra, *et al.* Energy-Conscious Compilation Based on Voltage Scaling. In Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compiler for Embedded Systems (SCOPES'02), June 2002.
[23] D.Shin, J.Kim, and S.Lee. Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis. In Design Automation Conference (DAC'01), June 2001.
[24] V.Swaminathan, K.Chakrabarty. Investigating the effect of voltage switching on low-energy task scheduling in hard real-time systems. In Asia South Pacific Design Automation Conference (ASP-DAC'01), January/February 2001.
[25] A.Varma, *et al.* A Control-Theoretic Approach to Dynamic Voltage Scheduling. In International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'03), October 2003.
[26] M.Weiser, *et al.* Scheduling for Reduced CPU Energy. In Proceedings of the First Symposium on Operating Systems Design and Implementation, Usenix Association, November 1994.
[27] F.Xie, M.Martinosi, S.Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In Programming Language Design and Implementation (PLDI'03), June 2003.
[28] R.Xu, *et al.* Practical PACE for Embedded Systems. In Proceedings of the 4th ACM international conference on Embedded software (EMSOFT'04), October 2004.



이 강 웅
1999년 스위스 ETH Zurich, 컴퓨터공학부, 학사. 2001년 스위스 ETH Zurich, 컴퓨터공학부, 석사. 2003년~현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 Compilers, Computer Architecture, Programming Languages, Embedded Systems.

이 재 진
1991년 서울대학교 물리학과, 학사. 1995년 미국 Standford University, Computer Science, 석사. 1999년 미국 University of Illinois at Urbana-Champaign, Computer Science, 박사. 2000년~2002년 미국 Michigan State University, Department of Computer Science and Engineering, 조교수 2002년~현재 서울대학교 컴퓨터공학부 부교수. 관심분야는 Compilers, Computer Architecture, High Performance Computing Systems, Embedded Systems

신 현 식
1973년 서울대학교 응용물리학과, 학사 1978년 미국 서던일리노이대학교 생물학, 학사. 1980년 미국 텍사스대학교 의공학과, 석사. 1985년 미국 텍사스대학교 전기컴퓨터공학과, 박사. 1986년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 실시간 내장형 시스템, 모바일 컴퓨팅, 멀티미디어 컴퓨팅