PLDI 2003: The ACM SIGPLAN Conference on Programming Language Design and Implementation.
San Diego, California, June 2003. (This research was done at Rutgers University)

The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction

Chung-Hsing Hsu and Ulrich Kremer
Department of Computer Science
Rutgers, The State University of New Jersey
{chunghsu,uli}@cs.rutgers.edu

ABSTRACT

This paper presents the design and implementation of a compiler algorithm that effectively optimizes programs for energy usage using dynamic voltage scaling (DVS). The algorithm identifies program regions where the CPU can be slowed down with negligible performance loss. It is implemented as a source-to-source level transformation using the SUIF2 compiler infrastructure. Physical measurements on a high-performance laptop show that total *system* (i.e., laptop) energy savings of up to 28% can be achieved with performance degradation of less than 5% for the `SPECfp95` benchmarks. On average, the system energy and energy-delay product are reduced by 11% and 9%, respectively, with a performance slowdown of 2%. It was also discovered that the energy usage of the programs using our DVS algorithm is within 6% from the theoretical lower bound. To the best of our knowledge, this is one of the first work that evaluates DVS algorithms by physical measurements.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

Dynamic voltage scaling, energy savings

1. INTRODUCTION

Power dissipation has always been a crucial issue in the design of battery-powered computing systems. Projected improvements in the capacity of the batteries (5-10%) cannot keep pace with what is needed to support the increasing demands of new features and performance on mobile platforms [23]. This widening "battery gap" drives the research and development of low power electronics since battery lifetime correlates positively with power dissipation, i.e., reduced power dissipation leads to prolonged battery life. Yet, current state-of-the-art high-performance mobile microprocessors have battery lives of less than 4 hours for typical Windows applications [31].

More recently, power is becoming a first-class design constraint for high-performance computing systems [35]. It is projected that power dissipation of future microprocessor chips will increase from 100 W today to about 2,000 W in 2010 [6]. High power consumption raises temperature, deteriorates performance and reliability, and increases the costs of thermal packaging and power delivery. Power is also a big issue for server clusters. A server farm with 8,000 servers consumes 2 megawatts [35], and a future petaflop system will consume around 100 megawatts [4]. Heavy-duty air conditioning and backup cooling and power-generation equipment already constitute a significant portion of the total operation cost, presently around 25% [35, 39]. The environmental impact of increased power demands has also become a major concern. The recent trend towards ultra-dense clusters [40] will only worse the problem.

Dynamic voltage scaling (DVS) is recognized as one of the most effective power reduction techniques. It exploits the fact that a major portion of power of CMOS circuitry scales quadratically with the supply voltage [8]. As a result, lowering the supply voltage can significantly reduce power dissipation. For non-interactive applications such as movie playing, decompression, and encryption, fast processors reduce device idle times, which in turn reduce the opportunities for power savings through hibernation strategies. In contrast, DVS techniques are still beneficial in such cases, i.e., DVS reduces power even when these devices are active. However, DVS comes at the cost of performance degradation. An effective DVS algorithm is one that intelligently determines *when* to adjust the current frequency-voltage setting (*scaling points*) and to *which* frequency-voltage setting (*scaling factors*), so that considerable savings in energy can be achieved while the required performance is still delivered.

Designing a good DVS algorithm is not an easy task. First of all, the overheads of transitions to and from different frequency-voltage settings may wipe out the benefits of DVS. Currently, it takes hundreds of microseconds to switch from one setting to another, which may be translated into tens of thousands of instructions for a high-performance processor. As a result, proposals such as using cache misses to trigger DVS [29] may not be effective if the performance requirement is critical. Furthermore, the battery lifetime

PLDI'03, June 9–11, 2003, San Diego, California, USA.

does not have a simple linear relationship with the power consumption of the circuit. It has been shown that the maximum battery lifetime is achieved when the variance of the discharge current distribution is minimized [36]. Most prior DVS algorithms do not consider these transition overheads. Recent work (e.g., [1, 18, 50]) starts to pay attention to these costs.

Even if the transition overheads are taken into account, the design of a good DVS algorithm is still not easy. A simple approach is to identify regions that show energy reduction potential within the performance bounds, and to execute these regions at lower frequencies. This is exactly the philosophy behind many so-called interval-based DVS algorithms (e.g. [37, 10, 27, 13, 44, 43, 28]). Interval-based DVS algorithms adapt to the variations of a workload closely by calculating a scaling factor at the beginning of each fixed-length time interval. Unfortunately, in practice, a recent study by Grunwald et al. [15] shows noticeable performance loss in some of interval-based algorithms. Theoretically, it has been formally proved [51, 21] that the optimal DVS algorithm is to run each non-interactive task at a constant speed and complete it right at the deadline. Ishihara and Yasuura goes further by showing that, in a system where only a limited set of voltage-frequency settings are available, at most two of the settings are required to implement the optimal DVS algorithm. While this strategy seems simple and attractive, it is practically not implementable for tasks whose execution times are unknown in advance,

The impact of shutting down other computer components, such as the disk and the display, is another factor a good DVS algorithm needs to take into account. In this setting, running at a higher frequency and turning off components may result in lower energy usage than running at a slow speed and leaving them on (until the deadline) [27, 33]. In other words, running as slowly as possible will minimize the *CPU* energy consumption but increase the total *system* energy consumption, because the power-consuming non-CPU devices stay on longer. Similarly, the performance impact of computer components may need to be considered as well. For example, nonideal memory behavior are identified in [5, 30], which may affect the choice of the scaling factor at each scaling point.

This paper presents the design and implementation of a compiler algorithm that effectively addresses the aforementioned design issues. The idea is to identify the program regions in which the CPU is mostly idle due to memory stalls, and slow them down for energy reduction. On an architecture that allows the overlap of the CPU and memory activities, such slow-down will not result in serious performance degradation. As a result, it alleviates the situation where non-CPU system components become dominant in the total system energy usage. The algorithm takes transition overheads into account, and is evaluated on a real system with the total system energy as the comparison metric. More importantly, the algorithm is parametric. A set of parameter values is provided and shown to be appropriate for the effectiveness of the algorithm. Finally, we choose to use `SPECfp95` benchmark suite for experiments because of its variety of CPU-boundness, which will be explained later, and because we target both mobile computers and high-performance systems. The main contributions of this paper include

- the design and implementation of a compiler-directed DVS algorithm, and the evaluation of its effectiveness on a real high-performance laptop machine through physical measurements. To the best of our knowledge, this is one of the first implementation of a DVS algorithm on a *real* system.
- physical measurements showing that total *system* energy savings in the range of 0% to 28% can be achieved with performance degradation from 0% to 4.7% for the `SPECfp95` benchmarks. On average, the energy usage and energy-delay product are reduced by 11% and 9%, respectively, at a performance penalty of 2.15%. To the best of our knowledge, this is one of the first evaluation using physical measurements and addressing total system energy usage.
- further findings that our profile-driven algorithm is able to reduce energy usage within 6% from the theoretical lower bound, and that its effectiveness is not critically dependent on the training inputs.
- a study of two commercial laptops from the system design point of view. The experimental results show that in some cases the high-performance high-power system together with our DVS algorithm is more energy efficient than the low-performance low-power system. We are not aware of any similar study published in the literature.

The rest of the paper is organized as follows: Section 2 lists some of the previous work. Section 3 discusses in detail the design and implementation of our compiler-directed DVS algorithm. The experiment setup and results are discussed in Section 4, followed by conclusions and future work in Sections 5.

2. RELATED WORK

There are many proposed DVS algorithms in the literature. Due to the space limitation, we only discuss work related to intra-task DVS algorithms, as which our algorithm can be categorized. An intra-task algorithm allows the scaling points to be put in the middle of a task execution. The determination of scaling points and the calculation of scaling factors may be done off-line or on-line.

Interval-based DVS algorithms are one type of intra-task algorithms. They operate at fixed-length time intervals and rely solely on the state of the system and the trace history to determine the scaling factors. Examples in this category include [37, 10, 27, 13, 44, 43, 28]. Checkpoint-based algorithms are another type of intra-task algorithms. In this type of algorithm, the scaling points are identified off-line and the scaling factors are calculated on-line [25, 34, 42, 3]. In contrast to interval-based algorithms which place scaling points at the beginning of each fixed-length interval, checkpoint-based algorithms place scaling points at selected branches to exploit the slacks due to run-time variations. As a result, they require off-line program analysis to identify the candidate branches.

The algorithm we present in this paper identifies scaling points and determines scaling factors off-line, with the help of profile data. Our algorithm has many significant differences from the algorithms mentioned above. Our algorithm has a tighter performance constraint in mind. Many existing algorithms use the worst-case execution time or the execution time of the unoptimized program as the performance

constraint. In our experiments, 5% of the total execution time of the optimized program is all the slack time DVS can exploit. As we will show in Section 4.4, tighter performance constraint may lead to better system energy efficiency.

Our algorithm also takes the transition overheads into account and is evaluated on real systems with the measurements of the total system energy usage. Many of the previous results were based on simulation, using simulators such as Wattch [7] and SimplePower [52], and evaluated the proposed algorithms using the power model associated with the particular simulator. As a consequence, the quality of the comparison results relies on the accuracy of the power model in a simulator. To the best of our knowledge, our work is one of the first attempts in using the physical measurement for the evaluation of the DVS algorithms. Pillai and Shin [38] evaluated their inter-task algorithm on a laptop with a 550 MHz AMD K6-2+ chip. Flautner and Mudge [12] implemented their inter-task algorithm on a laptop with a 300-600 MHz Transmeta TM5600 processor.

Our work adopts a table-driven approach similar to the one presented by Saputra et al. [41]. However, the table entries in our algorithm only store the performance information, through profiling, and the energy information is derived from an analytical model. In contrast, their algorithm stores both performance and energy information in the table. The work in this paper is also different from our previous work in [20, 18]. In the previous work, an analytical performance prediction model was used. As a result, the effectiveness of the DVS algorithm depends on how well the performance model predicts the target architecture. In this work we treat the target architecture as a black box. In addition, this work presents a more general framework for our DVS algorithm and introduces an additional constraint for a better quality of the algorithm. Finally, the work of [20, 18, 41] was done using a simulator, while the work presented in this paper uses physical measurements for evaluation, and targets total system energy usage, not only energy usage of single system components.

3. THE ALGORITHM

We propose a DVS algorithm which can be summarized as solving the following minimization problem.

$$\min_{R,f} \; P_f \cdot T(R,f) + P_{f_{max}} \cdot T(P-R, f_{max}) + P_{trans} \cdot 2 \cdot N(R) \quad (1)$$

subject to

$$T(R,f) + T(P-R, f_{max}) + T_{trans} \cdot 2 \cdot N(R) \leq (1+r) \cdot T(P, f_{max}) \quad (2)$$

The problem simply states: given a program P, find a region R and a frequency f such that, if region R is executed at frequency f and the rest of the program $P - R$ is executed at the peak frequency f_{max}, the total execution time plus the switching overhead $T_{trans} \cdot 2 \cdot N(R)$ is increased no more than r percent of the original execution time $T(P, f_{max})$, while the total energy usage is minimized. Here $T(R, f)$ represents the total execution time of region R running at frequency f, $N(R)$ represents the number of times region R is executed, P_f represents the power dissipation of the system at frequency f, and T_{trans} and P_{trans} represent a single switching overhead in term of performance and power, respectively.

In our DVS algorithm, a program region R is assumed to be a single entry and single exit program structure. Examples of a region include a loop nest, a call site, a called procedure, a sequence of statements, or even the entire program. While this definition may sound too restrictive, it is able to guarantee that all the top-level statements inside a region are executed the same number of times. As a result, the algorithm is able to count the number of occurrences of DVS switchings as $2 \cdot N(R)$, since the algorithm assumes that DVS interface will only be called at the entry and the exit of the region. Experiments have shown that this definition works reasonably well in practice.

Besides the performance constraint in Equation (2), our DVS algorithm introduces an additional constraint on the size of the selected region, namely

$$T(R, f_{max}) / T(P, f_{max}) \geq \rho \quad (3)$$

Equation (3) enforces the size of the selected region R to be sufficiently large for two reasons. First, it makes sure that the region takes longer time to execute than a single execution of the DVS call. Furthermore, our past experience has suggested that executing a larger region (in time) at a higher frequency often has less performance impact than executing a smaller region at a lower frequency. The importance of introducing Equation (3) will be illustrated in Section 4.7.

In short, the DVS algorithm we propose in this paper is parameterized by four sets of factors. Tables $T(R, f)$ and $N(R)$ capture the behavior of the input program. Parameters P_f, T_{trans}, and P_{trans} model the underlying machine. Parameter r represents the user's specification, while parameter ρ is a design parameter for the compiler.

3.1 Implementation Details

The prototype for our DVS algorithm is implemented as a profile-driven source-to-source transformation in `SUIF2` [48], as shown in Figure 1. It starts by instrumenting the input program at selected program locations (the instrumentation phase). The instrumented code is then executed, filling a subset of entries in tables $T(R, f)$ and $N(R)$ (the profiling phase). Once the profiling is done, the rest of table entries are derived based on these filled entries. Then the minimization problem is solved by enumerating all possible regions and frequencies. Finally, the corresponding DVS system calls are inserted at the boundaries of the selected region (the selection phase).

Two kinds of program constructs are instrumented in our implementation, namely, all sites and explicit loop structures. Explicit loop structures include `for` and `while` loops. Currently, loops based on `goto`'s are not instrumented and will not be considered as candidate regions.

The profiling of the instrumented program only constructs part of tables $T(R, f)$ and $N(R)$. The rest of the table entries are derived using the rules shown in Figure 2. In order to do this, an interprocedural analysis pass is implemented. The pass traverses all procedures reachable from the main routine in reverse topological order, treating strongly connected components in the call graph as single nodes. For each visited procedure, the annotated abstract syntax tree (AST), embedded in the `SUIF2` intermediate format, is traversed in a bottom-up fashion. To improve the efficiency, only AST nodes representing `if` statements, explicit loop structures, and call sites are annotated with the appropriate $T(R, f)$ and $N(R)$ values. The corresponding values for

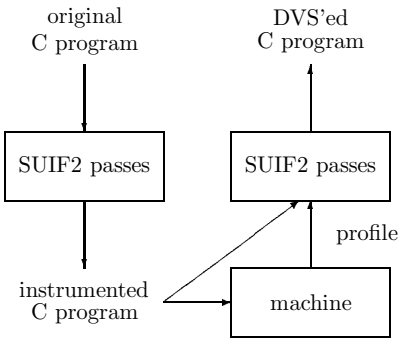


Figure 1: The flow diagram of the compiler implementation.

sequences of regions as a larger region are computed "on the fly" in the selection phase.

Note that only regions in a *forward* sequencing manner are taken into account. That is, for program construct

$$\textbf{loop}()\ \textbf{sequence}(R_1, \ldots, R_n)$$

the regions composed of $R_i \rightarrow R_{i+1} \rightarrow \ldots \rightarrow R_j$ for $j \geq i$ are considered, but not the "wrap-around" regions, $R_i \rightarrow \ldots \rightarrow R_n \rightarrow R_1 \rightarrow \ldots \rightarrow R_j$ for $j < i$.

3.2 An Illustrating Example

In this section, we illustrate our compiler algorithm using an example program shown on the left in Figure 3. The example code is presented in terms of control flow between what we call *basic* regions, where `L` and `C` stand for loop nests and call sites, respectively. The algorithm first identifies which regions to instrument. In our example, it is these basic regions our algorithm instruments. After the profiling phase, the entries of $T(R_i, f)$ and $N(R_i)$ for these regions are filled, as shown on the right in Figure 3 assuming only two CPU frequencies f_{max} and f_{min} are available. Then, the algorithm starts to consider other candidate regions which we call *combined* regions. For example, `if(L4,L5)` is a candidate region since it is an if-then-else construct that encloses regions `L4` and `L5`. Similarly, `seq(C2,C3)` is a candidate region that consists of two consecutive procedure calls. The region `seq(C2,C3,if(L4,L5))` is also considered as a candidate region. Even the entire program `foo` is treated as a candidate region in our implementation.

Not all combinations of regions are qualified as candidate regions. For example, `seq(C1,C2)` is not considered as a combined region since it has two entry points, one at the entry of `C1` and the other at the entry of `C2`. Similarly, `seq(C3,L4)` is not a candidate region because of the two exit points. Our current implementation does not consider region `seq(if(L4,L5),L2)` either. While it satisfies the single-entry-single-exit property, it does not satisfy our forward sequencing restriction. The particular reason for us to impose this restriction is to reduce the number of candidate regions that need to be examined in the region selection phase. For our example, there are nine regions total.

During the selection phase, the values of $T(R, f)$ and $N(R)$ for the combined regions are required. To derive the values, our implementation follows the rules in Figure 2. For example, the execution time of region `if(L4,L5)` running at

`if` statement:
R: **if** () **then** R_1 **else** R_2
$T(R, f) = T(R_1, f) + T(R_2, f)$
$N(R) = N(R_1) + N(R_2)$

explicit loop structure:
R: **loop** () R_1
$T(R, f) = T(R_1, f)$
$N(R)$ is profiled

call site:
R: **call** F()
$T(R, f) = T(F, f) \cdot N(R)/N(F)$
$N(R)$ is profiled

sequence of regions:
R: **sequence**(R_1, ... ,R_n)
$T(R, f) = \sum\{T(R_i, f) :\ 1 \leq i \leq n\}$
$N(R) = N(R_1) = \ldots = N(R_n)$

procedure:
F: **procedure** F() R
$T(F, f) = T(R, f)$
$N(F) = \sum\{N(R_i) :\ R_i$ is a call site to $F()\}$

Figure 2: The rules of deriving the table entries $T(R, f)$ in our DVS algorithm.

frequency f can be derived as the sum of execution times of regions `L4` and `L5` running at frequency f, i.e.,

$$T(\texttt{if}(\texttt{L4}, \texttt{L5}), f) = T(\texttt{L4}, f) + T(\texttt{L5}, f)$$

As a result, our implementation derives $T(\texttt{if}(\texttt{L4}, \texttt{L5}), f_{max})$ $= 8 + 2 = 10$ and $T(\texttt{if}(\texttt{L4}, \texttt{L5}), f_{min}) = 12 + 4 = 16$. The number of visits in region `if(L4,L5)`, $N(\texttt{if}(\texttt{L4}, \texttt{L5}))$ can be derived as ten. For the call sites, let us assume `C1` and `C3` call to the same procedure foo, which is only called by these two sites and contains basic regions. Our implementation attributes the time period from the entry of `C1` to the entry of the first encountered basic region in foo to $T(\texttt{C1}, f)$. This profiled time is usually very small; in our case it is zero. As a result, we need to "recover" the execution time for each call site. The rule in Figure 2 assumes the execution time of a call does not depend on the actual parameters. We can then estimate the execution time for call site `C1` as

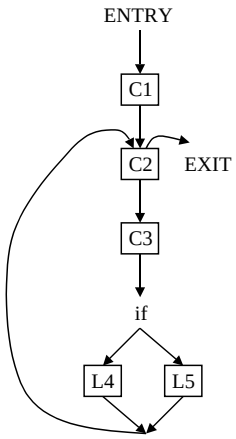
$$T(\texttt{C1}, f) = T(foo, f) \cdot 1/(1 + 10)$$

where $T(foo, f)$ is the estimated total execution time for procedure foo running at frequency f.

Finally, our implementation enumerates all candidate regions with respect to the minimization problem shown in Equations (1)–(3). In general, our implementation examines many more candidate regions. For example, it found 30 program locations to instrument in the `SPEC95 swim` benchmark and considered 2387 candidate regions.

3.3 Discussion

The proposed DVS algorithm is parameterized in terms of $T(R, f)$ without describing how to compute these values. Our current implementation uses profiling to get these



R	$N(R)$	$T(R,f)$ f_{max}	$T(R,f)$ f_{min}
C1	1	0	0
C2	10	10	12
C3	10	0	0
L4	8	8	12
L5	2	2	4

Figure 3: The example program is shown on the left in terms of control flow between regions. The table on the right represents the profiled data for the instrumented regions.

values. Profile-driven compiler optimization has its advantages and disadvantages. A program optimized with respect to one data input or machine configuration may not work well on another input or configuration. Profile-driven optimization also increases the compilation time. On the other hand, profiling captures more system effects which a compiler model may have difficulty to model, is more generally applicable, and allows more aggressive optimization. Our early work [19] proposed a compiler model that enables us to profile $T(R, f_{max})$ and estimate the rest of $T(R, f)$ analytically. While it may shorten the compilation time, this early work involves the computation of the memory stall time for each region and requires the help from performance counters in the system. Unfortunately, for the target system we experimented on (described later), we had a hard time relating the counted events to the actual performance. This is partially due to the lack of documentation and desired event types. Using a compile-time prediction model to compute all $T(R, f)$ entries sounds attractive since it is "portable" across different data inputs and machine configurations. However, the quality of the optimized code highly depends on the accuracy of the model which is hard to guarantee due to the complex interaction between all components in the system.

4. EXPERIMENTS

4.1 Hardware Platform

The hardware platform is a Compaq Presario 715US notebook computer. We chose this laptop as our hardware platform for at least three reasons. First of all, notebook computers are battery-powered and are therefore very sensitive to energy consumption. Secondly, the technology used in notebook computers today addresses more power issues and will soon be adapted to the high-performance systems for temperature control. Thirdly, this laptop is equipped with a high-performance microprocessor (mobile AMD Athlon 4) that allows DVS. Intel's Xscale-based processors, although they support DVS, do not have floating-point units. Transmeta's Crusoe processors do not provide enough memory

spec	Compaq Presario 715US
CPU	AMD mobile Athlon 4
f	600-1200 MHz
V	1.15-1.45 V
front side bus	DDR 100 MHz
memory	256 MB PC-133
graphics	VIA 16 MB
LCD display	14.1-inch 1024x768
disk	20 GB

f	V_f
600	1.15
700	1.20
800	1.25
900	1.30
1000	1.35
1100	1.40
1200	1.45

Table 1: The system configuration of the Compaq Presario 715US notebook computer.

level parallelism (i.e., allow multiple outstanding cache misses at the same time) which is a salient feature in many high-performance computers.

The Presario computer is equipped with a high-performance mobile AMD Athlon 4 microprocessor. The processor is a 3-way superscalar out-of-order decode and execution decoupled computing engine with dynamic branch prediction and speculative execution. It contains a 64KB instruction cache, a 64KB data cache, a full-speed on-die 256 KB level two exclusive cache, and a hardware data prefetching unit. The processor supports DVS under software control. For our experiments, it is able to operate from 600 MHz at 1.15 V to 1200 MHz at 1.45 V, wih 7 different frequency-voltage pairs. Table 1 summaries the configuration of the system.

4.2 Software Platform

The Linux 2.4.18 kernel was installed on the laptop. All the benchmarks were compiled by the GNU compilers using optimization level `-O2`. The DVS support is done through user-level system calls. The input of a DVS call is the desired frequency. The call will find the corresponding voltage and write both frequency and voltage encodings into machine-specific registers. The registers' values are then used by the regulator to adjust the CPU clock frequency and voltage level.

To profile the values of $T(R, f)$ and $N(R)$, we also implemented another user-level system call. The input of such a call is the region number. For $T(R, f)$, a high-resolution timer is needed to measure the elapsed time between two such system calls. We did this by reading out the current cycle counter value on a per-process basis. For $N(R)$, the system call implementation maintains a table indexed by the region number and increments the appropriate table entry.

4.3 Benchmark Choices

The `SPECfp95` benchmark suite was used for experiments because of its variety of CPU-boundness and because we target both mobile computers and high-performance systems. We found that the overall energy savings for a benchmark correlates negatively with the CPU boundness of the benchmark. In other words, less CPU-bound applications have potentially more energy reduction. Here we define the *CPU boundness* β_{cpu} as a ratio between 0 and 1, with 1 being extremely CPU-bound, using the least square fitting of $\{ T_f \}$ to the following linear model.

$$T_f / T_{f_{max}} = \beta_{cpu} \cdot (f_{max}/f) + c_0$$

SPECfp95	
benchmark	β_{cpu}
swim	0.04
tomcatv	0.06
hydro2d	0.13
su2cor	0.17
applu	0.30
apsi	0.37
mgrid	0.45
wave5	0.57
turb3d	0.75
fpppp	1.00

SPECint95	
benchmark	β_{cpu}
compress	0.47
vortex	0.70
gcc	0.83
ijpeg	0.95
li	1.00
perl	0.98
go	1.00
m88ksim	1.00

Table 2: The potential DVS energy savings and performance slopes of the SPEC95 benchmarks.

Table 2 shows the corresponding CPU-boundness for each benchmark in the entire SPEC95 benchmark suite. It can be seen that SPECfp95 benchmarks have a wider range of program behavior than SPECint95 benchmarks. Recent studies [45, 46] have shown that typical multimedia benchmarks are more CPU-bound than SPECint95 benchmarks. This is partially due to the fact that many of the popular multimedia benchmarks are really compression and decompression programs [24]. In addition, a recent study [22] observes that multimedia applications do not just contain fixed-point operations. We believe that more and more multimedia applications will be implemented as floating-point intensive computations, partly because of the current trend of physically-based modeling for virtual reality environments such as PC games, for example, [49].

4.4 Comparison Metrics

For comparison, we used total execution time T, total system energy usage E, and energy-delay product $E \cdot T$ [14]. As described in Section 1, power-performance trade-offs motivates the technique of dynamic voltage scaling. While an application can be executed with low power dissipation, its execution time may be unacceptably long. To seemlessly include the latency constraint into the picture, energy and energy-delay product are used by many as metrics for the comparison of different power-aware systems. The energy is equal to the product of the average power dissipation and the total execution time, i.e., $E = P \cdot T$. Energy-delay product is equal to the product of the energy usage and the total execution time, i.e., $E \cdot T = P \cdot T^2$. Energy translates directly to battery life, while the energy-delay product ensures a greater emphasis on performance.

4.5 Measurements

We performed several experiments with our DVS algorithm and measured the actual energy consumption of the system through a digital power meter. The power meter, a Yokogawa WT110 [32], sent power measurement data every 250 ms to another computer, which stores them in a log for later use. Each power measurement data point is the average power over 9500 samples in the period of 250 ms. That is, the power meter samples current and voltage at a rate of 26 μs. The comparisons were done by executing the benchmark with the reference data set. When profiling, the training data set (`train.in`) provided with the SPEC95 benchmark distribution was used. All the benchmarks were run to completion. During the measurements, the battery was removed. In addition, the power dissipation of the AC adapter was excluded. The monitor may be on or off during the measurements, depending on whether the benchmark needs it or not. Figure 4 shows the measurement setup.

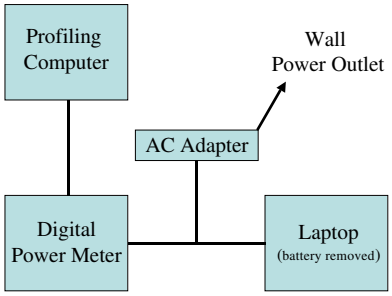


Figure 4: The experimental setup.

parameter	value
$T(R, f)$	profiled
$N(R)$	profiled
P_f	$V_f^2 \cdot f$
T_{trans}	20 μs
P_{trans}	0 W
r	5%
ρ	20%

Table 3: The input parameters for our algorithm.

The cost of each instrumentation call is about 50 ns. The cost of each DVS call is approximately 10 μs plus the transition time from one DVS level to another. We do not know the actual time required for voltage transition to occur. The white paper [2] suggests that it takes less than 100 μs but provides no specific information. A typical DC-DC converter is about 200μs/1V [42]. In the experiments we set the transition time to be 20 μs and the associated power dissipation to be 0 W. We found that $T_{trans} = 20$ μs works well in our experiments. As long as it is sufficiently large to prevent the transition overheads from being a dominant factor in the system performance, the accuracy of T_{trans} is not critical. In addition, the large value of T_{trans} allows us to ignore the cost of P_{trans} and simplify the algorithm. Table 3 lists the parameter settings of our DVS algorithm used in the experiments.

4.6 The Compilation Time

The compilation time of our algorithm is in the order of minutes. Table 4 lists the timing spent in each phase. The instrumentation phase takes 7–157 seconds which includes the times of converting to and from the SUIF2 intermediate representation plus the time of selecting program locations to instrument. The sub-phase of selecting program locations to instrument contributes 6%–13% of the total compilation time for the instrumentation phase, with 9% on average. In other words, the conversion between the input C program and the SUIF2 representation is very expensive. On the other hand, in the selection phase, the dominating sub-phase is the process of evaluating all candidate regions for the best region. It accounts for 74%–98% of the total compilation

	total compilation time	instru-mentation phase	profiling phase	selection phase
swim	34	7	8	19
tomcatv	173	4	158	11
hydro2d	340	44	173	123
su2cor	403	37	257	109
applu	284	83	13	188
apsi	1264	157	40	1067
mgrid	190	10	152	28
wave5	544	151	48	345
turb3d	1839	39	268	1532
fpppp	1628	82	11	1535

Table 4: The compilation time (in seconds) of our algorithm in various phases.

time for the phase, with the average 83%.

Benchmarks `apsi`, `turb3d` and `fpppp` took the longest compilation times among all benchmarks. The long compilation times can be attributed to the large number of candidate regions and the cost of finding these candidate regions. The current implementation enumerates all possible sequences of a statement list for finding candidate regions. As a result, the cost can be characterized by the number of statement sequences tried. For benchmarks `apsi` and `fpppp`, the selection phase evaluated 77,335 and 51,025 candidate regions respectively. In contrast, only 2,044–27,535 candidate regions were evaluated for other benchmarks. The selection phase looked 290,299–340,448 statement sequences for the three benchmarks. It only looked 3,955–118,775 combinations for other benchmarks. Clearly, there is room for improvement in our compiler algorithm.

4.7 Experimental Results

The experimental results are shown in Table 5. The execution time T_r and energy consumption E_r are all relative to the case in which the same program was run on the non-DVS system, i.e., the program is executed at the peak frequency and voltage. Note that the energy consumption is the overall system energy usage, not just the microprocessor's energy usage. It can be seen that program energy savings of 0% to 28% can be achieved with performance degradation of 0% to 4.7%. On average, the energy and energy-delay product are saved 11% and 9%, respectively, with a performance slowdown of 2.15%.

Furthermore, the energy savings of a benchmark using our compiler algorithm correlates negatively with its CPU boundness. Our algorithm is able to identify that benchmark `fpppp` is extremely CPU-bound (its $\beta_{cpu} = 1$) and therefore cannot be slowed down without significant performance penalties. There is a big gap in terms of energy usage at $\beta_{cpu} = 0.3$. It indicates that if an application is CPU-bound, our algorithm may not be able to reduce a considerable amount of energy without increase the performance tolerance. As we will see in Section 4.9, *none* of the DVS algorithm is able to do so if β_{cpu} is greater than 0.5 when only 5% of performance slow-down is allowed.

Our DVS algorithm has introduced a region size constraint, Equation (3), which prefers large region to be selected. In the experiments, this size constraint was set to be 20% or more of the total execution time. If this constraint is dropped, our algorithm will select a different but smaller region for benchmark `turb3d`. Specifically, with the constraint, 64% of time the benchmark is executed at 1100 MHz. In contrast, without the constraint, 31% of time the benchmark is executed at 700 MHz. Both selections are able to reduce the energy usage by 3%. However, the performance penalty becomes 10% if such a size constraint is not included. This particular case illustrates the importance of introducing Equation (3).

benchmark	selection	T_r (%)	E_r (%)
swim	R/600	102.93	76.88
tomcatv	R/800	101.18	72.05
hydro2d	R/900	102.21	78.70
su2cor	R/700	100.43	86.37
applu	R/900	104.72	87.52
apsi	R/1100	100.94	97.67
mgrid	R/1100	101.13	98.67
wave5	R/1100	104.32	94.83
turb3d	R/1100	103.65	97.19
fpppp	P/1200	100.0	100.0
average		102.15	88.99

Table 5: The relative execution time and system energy usage for the `SPECfp95` benchmarks using training input `train.in`.

benchmark	`std.in`	selection	T_r
swim	ref,900→45	P/700	101.10
tomcatv	ref,750→62	same	101.18
hydro2d	ref,200→6	same	102.21
su2cor	ref,40→5	R/800	107.31
applu	ref,300→5	R/900	103.61
apsi	ref,960→6	R/900	105.00
mgrid	test,40→4	R/1000	102.91
wave5	ref,40→10	R/800	102.33
turb3d	ref,111→2	R/1100	106.16
fpppp	train	same	100.00

Table 6: The relative execution time and system energy usage for the `SPECfp95` benchmarks using training input `std.in`.

4.8 Different Training Inputs

A common question for profile-based algorithms is how much the quality of the results is affected by the different training inputs. In this section, we evaluate such an impact using another training data set `std.in` developed by Burger [9]. For `SPECfp95` benchmarks, since they all have a common structure of an initialization phase followed by repetitions of a computation phase, in most cases data set `std.in` uses the same reference data set but reduces the number of repetitions. Table 6 column "`std.in`" gives the definition of data set `std.in` for `SPECfp95` benchmarks. For example, data set `std.in` for benchmark `swim` uses the same reference data set but reduces the repetitions from 900 down to 45. Note that benchmarks `mgrid` and `fpppp` do not use the reference data set as input.

The column "selection" in Table 6 lists the slow-down strategy our compiler finds if data set `std.in` is used as

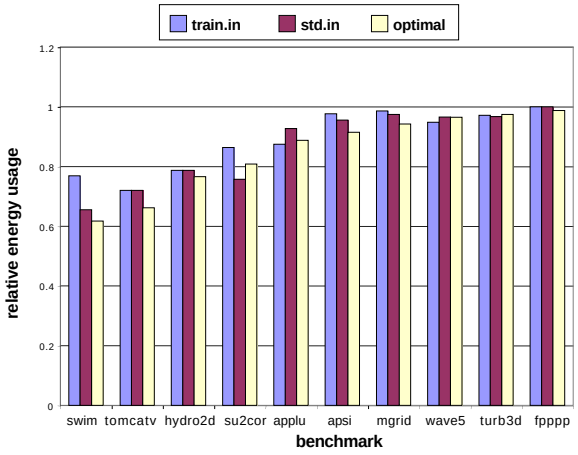


Figure 5: The relative energy usage of our compiler approach using training inputs `train.in` and `std.in` and the potential energy savings.

the training input. Except those benchmark rows marked "same", our compiler algorithm found different regions to slow down at different speed if using `std.in`. However, as shown in Figure 5, the quality of slow-down strategies using different training inputs is quite similar in terms of energy usage. Similar conclusion can be drawn for the energy-delay product as well.

For benchmarks `swim` and `su2cor`, using data set `std.in` as training input seems to produce better results. This is due to the different CPU boundness values provided by the two different data sets. For the reference data set of `swim`, the CPU boundness β_{cpu} is 0.04. In contrast, data set `std.in` provides β_{cpu} = 0.07 and data set `train.in` provides β_{cpu} = 0.19. Since data set `std.in` more closely models the CPU boundness of the reference data set, our algorithm was able to find a better slow-down strategy. Similarly, the reference data set of `su2cor` provides β_{cpu} = 0.17, while the data sets `std.in` and `train.in` provides β_{cpu} = 0.25 and 0.47, respectively. As a result, our algorithm was able to exploit more memory boundness of the benchmark and produced a better energy-delay product value.

4.9 Comparison with Other Algorithms

A natural follow-up question is how effective our compiler algorithm is as compared to the other proposed DVS algorithms. We would like to point out that many of the existing DVS algorithms have a set of parameters that can be tuned. While it is possible to manipulate these parameters to get better effectiveness of the algorithms, it is still an open question how to do this in a systematic fashion. For example, Grunwald et al. [15] studied several interval-based OS-directed algorithms and found that the effectiveness of the best algorithm they studied depends on the selected threshold values which are data sensitive.

To avoid the complications due to different settings of algorithm parameters, we have developed a methodology to compute the potential energy savings any DVS algorithm can generate [17]. Specifically, given the set of measured execution time and system power $\{ (T_f, P_f) \}$ at various frequencies f, the minimum energy usage E_r^* can be derived by solving the following linear programming problem.

$$E_r^* = \min_{t_f}(\sum_f P_f \cdot t_f)/(P_{f_{max}} \cdot T_{f_{max}})$$

subject to

$$\sum_f t_f \le (1+r) \cdot T_{f_{max}}, \quad \sum_f t_f/T_f = 1, \quad 0 \le t_f$$

The optimal DVS algorithm $\{ t_f \}$ determines the duration (in seconds) at each frequency f such that the relative energy usage E_r is minimized while the deadline is met and the required workload is performed.

Figure 5 compares the energy usage derived by our DVS algorithm and the theoretical lower bound. It can be seen that in many cases our algorithm has energy reduction very close to the lower bound, within 6%. In a few benchmarks, the lower bounds are larger than those using our algorithm. For example, the largest error occurs for benchmark `su2cor` using data set `std.in`, which is 5% more. It is because our algorithm does not guarantee that the performance constraint will always be satisfied. The main reason is that the training input and the reference input may have different behaviors in terms of CPU-boundness. In this case our algorithm generated the performance slow-down. For benchmark `su2cor`, since our algorithm generates performance slow-down of 7.3%, we can replace r = 5% by r = 7.3% when computing E_r^*. As a result, the difference drops down to 1.2%. We believe the inaccuracy of 1.2% comes from the measurement errors when acquiring T_f and P_f experimentally. Nevertheless, the difference in energy usage between our algorithm and the theoretical optimum is never greater than 2% for all tests we have done.

4.10 Multiple Region Extension

Our compiler algorithm assumes that only one region is allowed to be slowed down. This can certainly be relaxed to allow multiple regions to be slowed down. One way to do so is to formulate the algorithm as solving a zero-one integer linear programming problem (ZILP), as follows. Given a program $P = \bigcup_i R_i$, we are trying to find out the values for zero-one variables $\theta(R_i, f)$ that solves the following minimization problem:

$$\min_\theta \sum_{i,f} \theta(R_i, f) \cdot P_f \cdot T(R_i, f) + P_{trans} \cdot N_{trans}$$

subject to

$$\sum_{i,f} \theta(R_i, f) \cdot T(R_i, f) + T_{trans} \cdot N_{trans} \le (1+r) \cdot T(P, f_{max})$$
$$\sum_f \theta(R_i, f) = 1$$

where

$$N_{trans} = \sum_{i,j} N(R_i, R_j) \cdot \tfrac{1}{2} \cdot \sum_f |\theta(R_i, f) - \theta(R_j, f)|$$

Solution $\theta(R_i, f) = 1$ means that region R_i is executed at frequency f, 0 if not. Note that the total number of transitions N_{trans} is $2 \cdot N(R)$ in the single-region algorithm. In the multi-region algorithm, it is replaced by a more complex equation. The equation enumerates all pairs of regions R_i and R_j and accumulates the number of transitions from R_i to R_j, as indicated by $N(R_i, R_j)$, if the two regions are assigned with different frequencies, as indicated by $\frac{1}{2} \cdot \sum_f |\theta(R_i, f) - \theta(R_j, f)|$.

To estimate $N(R_i, R_j)$ from the profile $N(R)$, a transition graph is constructed through inter-procedural analysis. A transition graph is a directed graph whose nodes are regions R_i and edges are weighted by $N(R_i, R_j)$. The current prototype implements reaching definition analysis and approximates the number of transitions between regions $N(R_i, R_j)$. More details can be found in [18]. Note that, the problem formulation is new, which unfortunately introduces more variables to solve due to the discreteness of the DVS levels.

ZILP problems are in general considered hard problems due to the combinatorial aspect of integer programming. Experiences tell us that when the number of transitions exceeds over 50, the solver has a hard time to solve it efficiently, i.e., within a reasonable time. In our experiments, we set the reasonable time as an hour. Unfortunately, except benchmarks `swim` and `tomcatv`, the rest of `SPECfp95` benchmarks have the number of transitions more than 50 and cannot be solved efficiently. On one hand, techniques need to be invented to reduce the problem size or approximate the optimal solutions. On the other hand, as shown in Section 4.9, single-region algorithm is not far from the optimal DVS algorithm in terms of producing energy efficient programs. There is not much room for improvement because of multiple region flexibility. In general, this is good news since the multi-region algorithm is more complicated to implement and less efficient in terms of compilation time.

4.11 System Design Concerns

In this section we present an interesting comparison of two commercial laptop systems. One system has a high-performance processor with a more efficient memory subsystem, and the other system has a low-power processor with a less efficient memory subsystem. Both systems support DVS using different DVS algorithms. We compare the two systems for their energy efficiency.

The high-performance system is the Compaq Presario laptop, which is equipped with our DVS algorithm. The low-power system is the Fujitsu LifeBook P2040 laptop, which is operated by another DVS algorithm, the `LongRun` technology [11]. Table 7 compares the two systems, including their DVS support and memory system performance. It can be seen that the Presario computer is high-power and has a worse power-performance efficiency (MIPS/W). The ratios were computed by running the Dhrystone 2.1 benchmark on the two systems. On the other hand, its memory system performance is better. Both computers have similar memory latency, but the memory bandwidth (MB/s) of the Presario computer is larger. In addition, it provides more memory-level parallelism (MLP), i.e., it allows multiple outstanding cache misses. The memory bandwidth was computed using the `STREAM` benchmark [47]. The memory latency and the memory-level parallelism ratio were derived using the `LMbench` benchmark [26].

The comparison of the two systems is shown in Figure 6. The baseline is the energy-delay product on the Presario computer with our DVS algorithm disabled. The `LongRun` technology is set to the `economy` mode with all five performance levels available. It can be seen that in some benchmarks such as `swim` and `hydro2d`, applying our DVS algorithm on a less power-efficient system results in better energy efficiency. Furthermore, for CPU-bound applications such as `wave5` and `fpppp`, even the high-performance non-DVS system has much smaller energy-delay products. On average, the high-performance Presario computer together with our DVS algorithm reduces the energy-delay product by 9%. In contrast, the low-power LifeBook computer with the `LongRun` technology is 4% more than the Presario computer without any DVS algorithm enabled.

There are at least two reasons why the high-performance high-power system performs better in some cases. First of all, the high-performance features of the system pays off by reducing the total execution time significantly, though the power dissipation (i.e., energy usage per second) is higher due to the implementation of these features. Secondly, the energy usage of the processor in the Presario computer is significant. At the peak performance level, 64% of the total system power can be attributed to the processor for the Presario computer. In contrast, the processor of the LifeBook computer only contributes 27%.

Although AMD has an on-line DVS algorithm as part of

spec	Compaq Presario 715US	Fujitsu LifeBook P2040
CPU	AMD mobile Athlon 4	Transmeta Crusoe TM5800
cache	RISC out-of-order 384KB	VLIW in-order 512KB
f	600-1200 MHz	300-800 MHz
V	1.15-1.45 V	1.0-1.3 V
levels	7	5
P_{dhry}	52.79 W	13.65 W
MIPS/W	29.24	45.47
l_{mem} (ns)	210.6	195.7
MLP	2.5	1.0
MB/s	436.4	347.1

Table 7: The DVS support and power breakdown of the two laptops we tested.

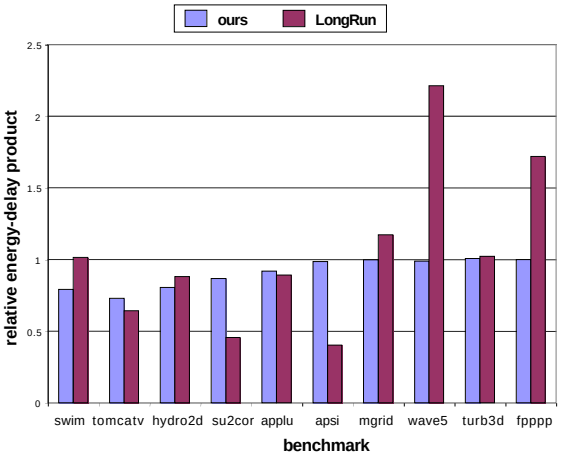


Figure 6: The relative energy-delay product of our compiler approach and the `LongRun` technology with respect to the Presario notebook without DVS. The energy is the entire system energy.

the *PowerNow!* technique we are not able to compare it against our DVS algorithm since the software is only executed on the Microsoft Windows system.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have discussed a novel compiler algorithm that effectively utilizes dynamic voltage scaling to save energy. The algorithm picks a single region to be executed at a lower performance level without introducing serious performance degradation. A prototype implementation based on the SUIF2 compiler infrastructure was used to evaluate the algorithm on the `SPECfp95` benchmarks. Physical measurements showed that significant energy savings in the range of 0% to 28% can be achieved with performance penalties between 0% and 4.7% for the `SPECfp95` benchmarks. On average, the energy and energy-delay product are reduced by 11% and 9%, respectively, with a performance slowdown of 2.15%. To the best of our knowledge, this work presents one of the first working implementation of DVS algorithms and one of the first to evaluate DVS algorithms through physical measurements.

We plan to study the impact of locality optimizations on the effectiveness of our DVS algorithm. Most advanced locality optimizations try to reduce the memory stalls to improve performance while our DVS algorithm exploits memory stalls for energy reduction. An early work [16] has shown that there are still plenty of opportunities to apply our DVS algorithm to the highly optimized codes. It is also observed that in some cases the less successful optimization lead to higher energy savings. We plan to perform a similar study on real systems to see whether these observations still hold.

6. ACKNOWLEDGEMENTS

This research was partially supported by NSF CAREER award CCR-9985050.

7. REFERENCES

[1] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[2] Advanced Micro Devices, Inc. Mobile AMD athlon 4 processor model 6 CPGA data sheet. Publication 24319, November 2001.

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints in the COPPER framework. In *Proceedings of Design, Automation and Test in Europe Conference*, March 2002.

[4] D. Bailey. 21st century high-end computing. In *Invited Talk, Applications, Algorithms, and Architectures Workshop for BlueGene/L*, August 2002.

[5] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of 9th ACM SIGOPS European Workshop*, September 2000.

[6] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July/August 1999.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture* , June 2000.

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

[9] D. Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1998.

[10] L. Chandrasena and M. Liebelt. A rate selection algorithm for quantized undithered dynamic supply voltage scaling. In *Proceedings of the International Symposium on Low-Power Electronics and Design* , August 2000.

[11] Transmeta Corporation. http://www.transmeta.com.

[12] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[13] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, July 2001.

[14] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[15] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating System Design and Implementation* , October 2000.

[16] C.-H. Hsu and U. Kremer. Dynamic voltage and frequency scaling for scientific applications. In *Proceedings of the 14th annual workshop on Languages and Compilers for Parallel Computing* , August 2001.

[17] C.-H. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling for memory-bound applications. Technical Report DCS-TR-498, Department of Computer Science, Rutgers University, August 2002.

[18] C.-H. Hsu and U. Kremer. Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Workshop on Power-Aware Computer Systems* , 2002.

[19] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems* , November 2000.

[20] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design* , August 2001.

[21] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design* , pages 197–202, August 1998.

[22] S. Kim and A. Somani. Characterization of an extended multimedia benchmark on a general purpose microprocessor architecture. Technical Report DCNL-CA-2000-002, Electrical and Computer Engineering Department, Iowa State University, 2000.

[23] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Battery-driven system design: A new frontier in low power design. In *Asia South Pacific Design Automation Conference / International Conference on VLSI Design*, January 2002.

[24] B. Lee and L. John. Implications of programmable general purpose processors for compression/encryption applications. In *IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, July 2002.

[25] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation* , pages 806–809, June 2000.

[26] The LMbench benchmark. http://www.bitmover.com/lmbench/.

[27] J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* , June 2001.

[28] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the 2002 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, October 2002.

[29] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*, June 2000.

[30] T. Martin and D. Siewiorek. Nonideal battery and main memory effects on cpu speed setting for low power. *IEEE Transactions on Very Large Scale Integration System*, 9(1):29–34, February 2001.

[31] T. L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1999.

[32] K. Masahiro, K. Kazuo, H. Kazuo, and O. Eiichi. WT110/WT130 digital power meters. Yokogawa Technical Report 22, 1996.

[33] A. Miyoshi, C. Lefurgy, E. Hensbergen, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing*, June 2002.

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

[35] T. Mudge. Power: A first class design constraint for future architectures. *IEEE Computer*, 34(4):52–58, April 2001.

[36] M. Pedram and Q. Wu. Design considerations for battery-powered electronics. In *Proceedings of the 36th Conference on Design Automation*, June 1999.

[37] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of 1998 International Symposium on Low Power Electronics and Design* , pages 76–81, August 1998.

[38] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th Symposium on Operating Systems Principles*, October 2001.

[39] R. Rajamony and R. Bianchini. Energy management for server clusters. In *Tutorial, 16th Annual ACM International Conference on Supercomputing*, June 2002.

[40] RLX Technologies. Serverblade. http://www.rlxtechnologies.com.

[41] H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, J. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems* , June 2002.

[42] D. Shin and J. Kim. A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications. In *Proceedings of the International Symposium on Low-Power Electronics and Design* , August 2001.

[43] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling for portable systems. In *Proceedings of the 38th Design Automation Conference* , June 2001.

[44] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the 14th International Conference on VLSI Design*, January 2001.

[45] N. Slingerland and A. Smith. Cache performance for multimedia applications. In *Proceedings of the 15th IEEE International Conference on Supercomputing*, June 2001.

[46] S. Sohoni, Z. Xu, R. Min, and Y. Hu. A study of memory system performance of multimedia applications. In *ACM Joint International Conference on Measurement & Modeling of Computer Systems* , June 2001.

[47] The STREAM benchmark. http://www.cs.virginia.edu/stream/.

[48] SUIF. Stanford University Intermediate Format.

[49] K. van den Doel, P. Kry, and D. Pai. FoleyAutomatic: Physically-based sound effects for interactive simulation and animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, August 2001.

[50] F. Xie and M. Martonosi and S. Malik. Compile time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation* , June 2003.

[51] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 374–382, October 1995.

[52] W. Ye, N. Vijaykrishna, M. Kandemir, and M.J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Design Automation Conference* , June 2000.