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 SPEC95 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 works 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 [30].

More recently, power is becoming a first-class design constraint for high-performance computing systems [34]. 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, degrades 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 [34], 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% [34, 38]. The environmental impact of increased power demands has also become a major concern. The recent trend towards ultra-dense clusters [46] will only worsen 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 [28] may not be effective if the performance requirement is critical. Furthermore, the battery lifetime...
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 [35]. Most prior DVS algorithms do not consider these transition overheads. Recent work (e.g., [1, 18, 48]) 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., [36, 10, 26, 13, 42, 41, 27]). 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 [49, 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) [36, 32]. 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 is identified in [5, 29], 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 SPECint95 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 SPECint95 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 [36, 10, 26, 13, 42, 41, 27]. 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, 33, 40, 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 slack 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 ac-
count and is evaluated on real systems with the measure-
ments of the total system energy usage. Many of the previ-
ous results were based on simulation, using simulators such
as Wattch [7] and SimplePower [50], and evaluated the pro-
posed 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
[37] 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 TM2600 processor.

Our work adopts a table-driven approach similar to the
one presented by Saputra et al. [39]. However, the table
entries in our algorithm only store the performance infor-
mation, through profiling, and the energy information is de-

erived from an analytical model. In contrast, their algorithm
stores both performance and energy information in the ta-
ble. The work in this paper is also different from our pre-
vious 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 ad-
dition, 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,
39] 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.

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

subject to

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

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 $f_{\text{max}}$, the total execution time plus
the switching overhead $T_{\text{trans}} \cdot 2 \cdot N(R)$ is increased no more
than $r$ percent of the original execution time $T(P, f_{\text{max}})$,
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 $P_{\text{trans}}$ and $P_{\text{trans}}$ 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. Ex-
amples 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

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

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 execu-
tion 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. Parame-
ters $P_f$, $T_{\text{trans}}$, and $P_{\text{trans}}$ model the underlying machine.
Parameter $r$ represents the user's specification, while pa-
parameter $\rho$ 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 SUP2 [45],
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 profil-
ing 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 possi-
ble 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 struc-
tures. 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 en-
tries 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 con-
ected components in the call graph as single nodes. For
each visited procedure, the annotated abstract syntax tree
(AST), embedded in the SUP2 intermediate format, is tra-
versed 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 appropri-
ate $T(R, f)$ and $N(R)$ values. The corresponding values for
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 $T(R, f)$ and $N(R)$ for these regions are filled, as shown on the right in Figure 3 assuming only two CPU frequencies $f_{\text{max}}$ and $f_{\text{min}}$ are available. Then, the algorithm starts to consider other candidate regions which we call combined regions. For example, $\text{if}(L4, L5)$ is a candidate region since it is an if-then-else construct that encloses regions L4 and L5. Similarly, $\text{seq}(C2, C3)$ is a candidate region that consists of two consecutive procedure calls. The region $\text{seq}(C2, C3, \text{if}(L4, L5))$ is also considered as a candidate region. Even the entire program $\text{foo}$ is treated as a candidate region in our implementation.

Not all combinations of regions are qualified as candidate regions. For example, $\text{seq}(C1, C2)$ 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, $\text{seq}(C3, L4)$ is not a candidate region because of the two exit points. Our current implementation does not consider region $\text{seq}(\text{if}(L4, L5), L2)$ 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.

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

$$
\text{if statement:} \\
R: \text{ if } \text{ then } R_1 \text{ else } R_2 \\
T(R,f) = T(R_1,f) + T(R_2,f) \\
N(R) = N(R_1) + N(R_2)
$$

explicit loop structure:

$$
R: \text{ loop } R_i \\
T(R,f) = T(R_i,f) \\
N(R) \text{ is profiled}
$$

call site:

$$
R: \text{call F()} \\
T(R,f) = T(F,f) \cdot N(R)/N(F) \\
N(R) \text{ is profiled}
$$

sequence of regions

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

procedure:

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

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

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

As a result, our implementation derives $T(\text{if}(L4, L5), f_{\text{max}}) = 8 + 2 = 10$ and $T(\text{if}(L4, L5), f_{\text{min}}) = 12 + 4 = 16$. The number of visits in region $\text{if}(L4, L5)$, $N(\text{if}(L4, L5))$ can be derived as ten. For the call sites, let us assume C1 and C3 call to the same procedure $\text{foo}$, 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 $\text{foo}$ to $T(C1, f)$. 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

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

where $T(\text{foo}, f)$ is the estimated total execution time for procedure $\text{foo}$ 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
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 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, with 7 different frequency-voltage pairs.

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

<table>
<thead>
<tr>
<th>spec</th>
<th>Compaq Presario 715US</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>mobile Athlon 4</td>
</tr>
<tr>
<td>$f$</td>
<td>600-1200 MHz</td>
</tr>
<tr>
<td>$V$</td>
<td>1.15-1.45 V</td>
</tr>
<tr>
<td>front side bus</td>
<td>DDR 100 MHz</td>
</tr>
<tr>
<td>memory</td>
<td>256 MB PC-133</td>
</tr>
<tr>
<td>graphics</td>
<td>VIA 16 MB</td>
</tr>
<tr>
<td>LCD display</td>
<td>14.1-inch 1024x768</td>
</tr>
<tr>
<td>disk</td>
<td>20 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f$</th>
<th>$V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>1.15</td>
</tr>
<tr>
<td>700</td>
<td>1.20</td>
</tr>
<tr>
<td>800</td>
<td>1.25</td>
</tr>
<tr>
<td>900</td>
<td>1.30</td>
</tr>
<tr>
<td>1000</td>
<td>1.35</td>
</tr>
<tr>
<td>1100</td>
<td>1.40</td>
</tr>
<tr>
<td>1200</td>
<td>1.45</td>
</tr>
</tbody>
</table>

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 SPECps6 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 \( \beta_{pu} \) 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.

\[
T_f / T_{f_{\text{max}}} = \beta_{pu} \left( f_{\text{max}} / f \right) + c_0
\]

Table 2 shows the corresponding CPU-boundness for each benchmark in the entire SPEC35 benchmark suite. It can be seen that SPECfp35 benchmarks have a wider range of program behavior than SPECint35 benchmarks. Recent studies [43, 44] have shown that typical multimedia benchmarks are less CPU-bound than SPECint35 benchmarks. The partiality 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, [47].

### 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 4, power-performance trade-offs motivate the technique of dynamic voltage scaling. While an application can be executed with low power dissipation, its execution time may be unacceptably long. To seemingly 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 [31], sent power measurement data every 250 ms to another computer, which stored them in a log for later use. Each power measurement data point is the

![Figure 4: The experimental setup.](image)

Table 3: The input parameters for our algorithm.

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(R, f) )</td>
<td>profiled</td>
</tr>
<tr>
<td>( N(R) )</td>
<td>profiled</td>
</tr>
<tr>
<td>( P_f )</td>
<td>( V_f^2 \cdot f )</td>
</tr>
<tr>
<td>( T_{\text{trans}} )</td>
<td>20 ( \mu )s</td>
</tr>
<tr>
<td>( r )</td>
<td>5%</td>
</tr>
<tr>
<td>( p )</td>
<td>20%</td>
</tr>
</tbody>
</table>

average power over 9500 samples in the period of 250 ms. That is, the power meter samples current and voltage at a rate of 20 \( \mu \)s. The comparisons were done by executing the benchmark with the reference data set. When profiling, the training data set (train.in) provided with the SPEC35 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.

The cost of each instrumentation call is about 50 ns. The cost of each DVS call is approximately 10 \( \mu \)s 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 \( \mu \)s but provides no specific information. A typical DC-DC converter is about 200\( \mu \)s/IV [40]. In the experiments we set the transition time to be 20 \( \mu \)s and the associated power dissipation to be 0 W. We found that \( T_{\text{trans}} = 20 \mu \)s 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 \( T_{\text{trans}} \) is not critical. In addition, the large value of \( T_{\text{trans}} \) allows us to ignore the cost of \( T_{\text{trans}} \) 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 SUP2 intermediate representation plus the time of selecting program locations to instrument. The sub-phase of selecting program locations to instrument contributes 5%–13% of the total compilation time for the instrumentation phase, with 9% on average. In
4.7 Experimental Results

In other words, the conversion between the input C program and the SUTP2 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 time for the phase, with the average 88%.

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

<table>
<thead>
<tr>
<th>benchmark</th>
<th>total compilation time</th>
<th>instrumentation phase</th>
<th>profiling phase</th>
<th>selection phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>34</td>
<td>8</td>
<td>7</td>
<td>193</td>
</tr>
<tr>
<td>tomcatv</td>
<td>173</td>
<td>3</td>
<td>173</td>
<td>11</td>
</tr>
<tr>
<td>hydro2d</td>
<td>310</td>
<td>44</td>
<td>173</td>
<td>123</td>
</tr>
<tr>
<td>sun2kco</td>
<td>403</td>
<td>37</td>
<td>257</td>
<td>109</td>
</tr>
<tr>
<td>applu</td>
<td>284</td>
<td>83</td>
<td>13</td>
<td>188</td>
</tr>
<tr>
<td>apsi</td>
<td>1264</td>
<td>157</td>
<td>40</td>
<td>1067</td>
</tr>
<tr>
<td>mgid</td>
<td>190</td>
<td>10</td>
<td>192</td>
<td>28</td>
</tr>
<tr>
<td>wave5</td>
<td>544</td>
<td>151</td>
<td>48</td>
<td>345</td>
</tr>
<tr>
<td>turbox</td>
<td>1500</td>
<td>33</td>
<td>208</td>
<td>1532</td>
</tr>
<tr>
<td>fpppp</td>
<td>1628</td>
<td>82</td>
<td>11</td>
<td>1535</td>
</tr>
</tbody>
</table>

4.8 Different Training Inputs

The experimental results are shown in Table 5. The execution time $T_e$ and energy consumption $E_e$ 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 47%. 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 $β_{cpu} = 1$) and therefore cannot be slowed down without significant performance penalties. There is a big gap in terms of energy usage at $β_{cpu} = 0$. 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.

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

<table>
<thead>
<tr>
<th>benchmark</th>
<th>selection</th>
<th>$T_e$ (%)</th>
<th>$E_e$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>R/600</td>
<td>102.93</td>
<td>76.88</td>
</tr>
<tr>
<td>tomcatv</td>
<td>R/800</td>
<td>101.48</td>
<td>72.65</td>
</tr>
<tr>
<td>hydro2d</td>
<td>R/900</td>
<td>102.21</td>
<td>78.70</td>
</tr>
<tr>
<td>sun2kco</td>
<td>R/700</td>
<td>100.43</td>
<td>86.37</td>
</tr>
<tr>
<td>applu</td>
<td>R/900</td>
<td>104.72</td>
<td>87.52</td>
</tr>
<tr>
<td>apsi</td>
<td>R/1100</td>
<td>100.94</td>
<td>97.76</td>
</tr>
<tr>
<td>mgid</td>
<td>R/1100</td>
<td>101.43</td>
<td>98.67</td>
</tr>
<tr>
<td>wave5</td>
<td>R/1100</td>
<td>104.32</td>
<td>94.83</td>
</tr>
<tr>
<td>turbox</td>
<td>R/1100</td>
<td>103.65</td>
<td>97.19</td>
</tr>
<tr>
<td>fpppp</td>
<td>R/1200</td>
<td>100.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

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 turbox. Specifically, with the constraint, 64% of the 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).
Table 6: The relative execution time and system energy usage for the SPECfp95 benchmarks using training input std.in.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>std.in</th>
<th>selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>train.in</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>ref.900--45</td>
<td>P/700</td>
<td>103.10</td>
</tr>
<tr>
<td>tomcatv</td>
<td>ref.750--62</td>
<td>same</td>
</tr>
<tr>
<td>hydro2d</td>
<td>ref.200--6</td>
<td>same</td>
</tr>
<tr>
<td>su2cor</td>
<td>ref.40--5</td>
<td>R/800</td>
</tr>
<tr>
<td>appli</td>
<td>ref.300--5</td>
<td>R/900</td>
</tr>
<tr>
<td>api</td>
<td>ref.900--6</td>
<td>R/900</td>
</tr>
<tr>
<td>mgrid</td>
<td>test.40--4</td>
<td>R/1000</td>
</tr>
<tr>
<td>wave5</td>
<td>ref.40--10</td>
<td>R/800</td>
</tr>
<tr>
<td>turb3d</td>
<td>ref.111--2</td>
<td>R/1100</td>
</tr>
<tr>
<td>fpppp</td>
<td>train</td>
<td>same</td>
</tr>
</tbody>
</table>

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

= 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 $\beta_{cpu} = 0.17$, while the data sets std.in and train.in provides $\beta_{cpu} = 0.25$ 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, Gruenwald 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 $E^*$ can be derived by solving the following linear programming problem:

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

subject to

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

The optimal DVS algorithm $\{ t_f \}$ determines the duration (in seconds) at each frequency $f$ such that the relative energy usage $E^*$ 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 slowdown. For benchmark su2cor, since our algorithm generates performance slowdown of 7.3%, we can replace $r = 5\%$ by $r = 7.3\%$ when computing $E^*$. 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 $P = \bigcup R_i$, we are trying to find out the values for zero-one variables $\theta(R_i,f)$ that solves the following minimization problem:

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

subject to

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

where

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

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
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 ratio was 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 [2]. The memory latency and the memory-level parallelism ratio were derived using the Dhrystone benchmark [2].

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 waven and fppp, 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 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 SUNF2 compiler infrastructure was used to evaluate the algorithm on the SPECint95 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 SPECint95 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.
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.

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-0985550.

7. REFERENCES


