A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance

Qiang Wu[1], V.J. Reddi[2], Youfeng Wu[3], Jin Lee[3], Dan Connors[2], David Brooks[4], Margaret Martonosi[1], Douglas W. Clark[1]

[1]Depts. of Computer Science, Electrical Engineering, Princeton University
[2]Dept. of Electrical and Computer Engineering, U. of Colorado at Boulder
[3]Programming Systems Lab, Corporate Tech. Group, Intel Corporation
[4]Division of Engineering and Applied Science, Harvard University

{jqwu, mrm, doug}@princeton.edu, {youfeng.wu, jin.lee}@intel.com
{vijay.janapareddi, dconnors}@colorado.edu, dbrooks@eecs.harvard.edu

ABSTRACT

Dynamic voltage and frequency scaling (DVFS) is an effective technique for controlling microprocessor energy and performance. Existing DVFS techniques are primarily based on hardware, OS time-interrupts, or static-compiler techniques. However, substantially greater gains can be realized when control opportunities are also explored in a dynamic compilation environment. There are several advantages to deploying DVFS and managing energy/performance tradeoffs through the use of a dynamic compiler. Most importantly, dynamic compiler driven DVFS is fine-grained, code-aware, and adaptive to the current microarchitecture environment.

This paper presents a design framework of the run-time DVFS optimizer in a general dynamic compilation system. A prototype of the DVFS optimizer is implemented and integrated into an industrial-strength dynamic compilation system. The obtained optimization system is deployed in a real hardware platform that directly measures CPU voltage and current for accurate power and energy readings. Experimental results, based on physical measurements for over 40 SPEC or Olden benchmarks, show that significant energy savings are achieved with little performance degradation. SPEC2K FP benchmarks benefit with energy savings of up to 70% (with 0.5% performance loss). In addition, SPEC2K INT show up to 44% energy savings (with 5% performance loss), SPEC95 FP save up to 64% (with 4.9% performance loss), and Olden save up to 61% (with 4.5% performance loss). On average, the technique leads to an energy delay product (EDP) improvement that is 3X-5X better than static voltage scaling, and is more than 2X (22% vs. 9%) better than the reported DVFS results of prior static compiler work. While the proposed technique is an effective method for microprocessor voltage and frequency control, the design framework and methodology described in this paper have broader potential to address other energy and power issues such as di/dt and thermal control.

1. Introduction

Dynamic voltage and frequency scaling (DVFS) is an effective technique for microprocessor energy and power control and has been implemented in many modern processors [4, 11]. Current practice for DVFS control is OS-based power scheduling and management (i.e., the OS selects a new voltage setting when a new application or task is scheduled or if the processor is switched between idle/active states) [4, 10]. In this work, we focus instead on more fine-grained intra-task DVFS, in which the voltage and frequency can be scaled during program execution to take advantage of the application phase changes.

While significant research efforts have been devoted to the intra-task DVFS control, most of them are based on hardware [16, 19, 22], OS time-interrupt [7, 23], or static compiler techniques [14, 25]. (A brief description of these existing techniques is available in Section 2.) Very little has been done to explore DVFS control opportunities in a dynamic compilation or optimization environment. In this paper, we consider dynamic compiler DVFS techniques, which optimize the application binary code and insert DVFS control instructions at program execution time.

A dynamic compiler is a run-time software system that compiles, modifies, and optimizes a program's instruction sequence as it runs. In recent years, dynamic compilation is becoming increasingly important as a foundation for run-time optimization, binary translation, and information security. Examples of dynamic compiler based infrastructures include HP Dynamo [2], IBM DAISY [8], Intel IA32EL [3], and Intel PIN [18]. Since most DVFS implementations allow direct software control via mode set instructions (by accessing special mode set registers), a dynamic compiler can be used to insert DVFS mode set instructions into application binary code at run time. If there exists CPU execution slack (i.e., CPU idle cycles waiting for memory), these instructions will scale down the CPU voltage and frequency to save energy with no or little impact on performance.

Using dynamic compiler driven DVFS offers some unique features and advantages not present in other approaches. Most importantly, it is more fine-grained and more code-aware than hardware or OS interrupt based schemes. Also it is more adaptive to the run-time environment than static compiler DVFS. In Section 2, we will give statistical results to further motivate dynamic compiler driven DVFS, and discuss its advantages and disadvantages.

This paper presents a design framework of the run-time DVFS optimizer (RDO) in a dynamic compilation environment. Key design issues that have been considered include code region selection, DVFS decision, and code insertion/transformation. In particular, we propose a new DVFS decision algorithm based on an analytic DVFS decision model. A prototype of the RDO is implemented and integrated into an industrial-strength dynamic optimization system (a variant of the Intel PIN system [18]). The obtained optimization system is deployed into a real hardware platform (an Intel development board with a Pentium-M processor), that allows us to directly measure CPU current and voltage for accurate power and energy readings. The evaluation is based on experiments with phys-

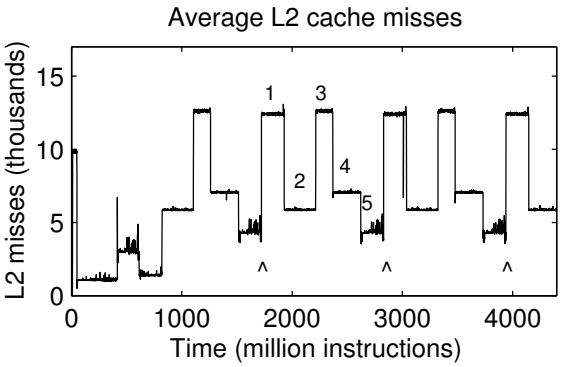

Figure 1: Number of L2 cache misses for every million instructions retired during the execution of SPEC2000 benchmark 173.applu. Numbers 1 to 5 mark five memory characteristic phases. The symbol ^ in the figure marks the recurring point of the program phases.

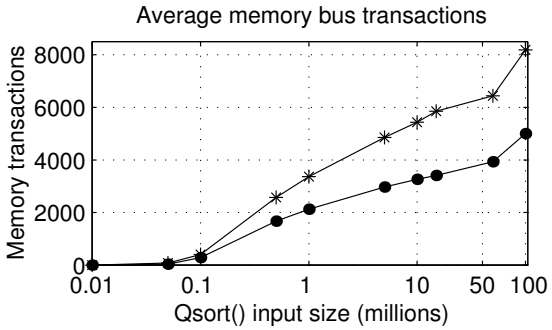

Figure 2: Average number of memory bus transactions (per 1M instructions) for the function qsort() (as in stdlib.h) with different input sizes and different input patterns (random input:• versus pre-sorted input: ∗).

ical measurements for over 40 SPEC or Olden benchmarks. Evaluation results show that significant energy efficiency is achieved. For example, up to 70% energy savings (with 0.5% performance loss) is accomplished for SPEC benchmarks. On average, the technique leads to energy delay product (EDP) improvements of 22.4% for SPEC95 FP, 21.5% for SPEC2K FP, 6.0% for SPEC2K INT, and 22.7% for Olden benchmarks. These average results are 3X-5X better than those from static voltage scaling (22.4% vs. 5.6%, 21.5% vs. 6.8%, 6.0% vs. -0.3%, 22.7% vs.6.3%), and are more than 2X better (21.5% vs. 9%) than those reported by a static compiler DVFS scheme [14].

Overall, the main contributions of this paper are twofold. First, we have designed and implemented a run-time DVFS optimizer, and deployed it on real hardware with physical power measurements. The optimization system is more effective in terms of energy and performance efficiency, as compared to existing approaches. Second, to our knowledge, this is one of the first efforts to develop dynamic compiler techniques for microprocessor voltage and frequency control. A previous work [12] provides a partial solution to the Java Method DVFS in a Java Virtual Machine, while we provide a complete design framework and apply DVFS in a more general dynamic compilation environment with general applications.

The structure for the rest of the paper is as follows. Section 2 further motivates dynamic compiler DVFS. Section 3 presents the design framework of the RDO. Section 4 describes the implementation and deployment of the RDO system. This is followed by the experimental results in Section 5. Section 6 highlights related work. Section 7 discusses future work. Finally, our conclusions are offered in Section 8.

2. Why Dynamic Compiler Driven DVFS?

In this section, we discuss in more detail the unique features, advantages, and disadvantages of dynamic compiler driven DVFS, as compared to existing DVFS techniques.

2.1 Advantages over hardware or OS DVFS

Existing hardware or OS time-interrupt based DVFS techniques typically monitor some system statistics (such as issue queue occupancy [22]) in fixed time intervals, and decide DVFS settings for future time intervals [7, 19, 22, 23]. Since the time intervals are pre-determined and independent of program structure, the DVFS control by these methods may not be efficient in adapting to program phase changes. One reason is that program phase changes are generally caused by the invocation of different code regions, as observed in [15]. Thus, the hardware or OS techniques may not be able to infer enough about the application code attributes and find the most effective adaptation points. Another reason is that program phase changes are often recurrent (i.e., loops). In this case, the hardware or OS schemes would need to detect and adapt to the recurring phase changes repeatedly.

To illustrate the above reasoning, Figure 1 shows an example trace of program phase changes for the SPEC2000 benchmark 173.applu. (The trace is from a part of the program with about 4500 million instructions.) The y-axis is the number of L2 cache misses for every 1M instructions during the program execution. (All results in the section were obtained using hardware performance counters in a processor setup described in Section 5.) From the figure, we see that there are about 5 distinct memory phases characterized by different L2 cache miss values and duration. As will be shown in Section 5, these phases actually correspond to 5 distinct code regions (functions) in the program. Also, we see the phase changes are recurrent, as shown by the marked points in the figure.

Compiler driven DVFS schemes (static or dynamic) can apply DVFS to fine-grained code regions so as to adapt naturally to program phase changes. Hardware or OS-based DVFS schemes with fixed intervals lack this fine-grained, code-aware adaptation.

2.2 Advantages over static compiler DVFS

Existing compiler DVFS work is primarily based on static compiler techniques [14, 25]. Typically profiling is used to learn about program behavior. Then some offline analysis techniques (such as linear programming [14]) are used to decide DVFS settings for some code regions.

One limitation to static compiler DVFS is that, due to different runtime environments for the profiler and the actual program, the DVFS setting obtained at static compile time may not be appropriate for the program at runtime. The reasoning is that DVFS decisions are dependent on the program's memory boundedness (i.e., the CPU can be slowed down if it is waiting for memory operation completion). Then, the program behavior in term of memory boundedness is in turn dependent on run-time system settings such as machine/architecture configuration, program input size and patterns. For example, machine/architecture settings such as cache configuration or memory bus speed may affect how much CPU slack or idle time exists. Also, different program input sizes or patterns may affect how much memory is to be used and how it is going to be used.

As an illustration, Figure 2 shows the average number of memory bus transactions (per 1M instructions) for the function qsort() (as

Table 1: Examples of different memory behavior for SPEC programs with reference and train inputs.
L2: Average Num of L2 cache misses per 1M instructions
Mem: Average Num of memory bus transactions per 1M inst
4L sweep(): means 4th loop in function sweep()

Benchmark	Code region	reference		train	
		L2	Mem	L2	Mem
103.su2cor	corr()	3.9K	13.6K	1.4K	5.7K
103.su2cor	4L sweep()	4.3K	14.4K	1.8K	6.1K
107.mgrid	mg3p()	2.6K	9.5K	0.6K	2.3K
189.lucas	fftSquare()	6.8K	18.1K	0.06K	0.1K
256.bzip2	DoRevers-Transform()	7.8K	11.9K	1.3K	3.1K

in the stdlib). The curve with • is for random input elements with different input sizes, while the curve with $*$ is for pre-sorted input elements. Figure 2 shows that the average numbers of memory bus transactions vary significantly for different input sizes and input patterns. (Not surprisingly, larger input sizes lead to more L2 cache misses and thus more memory bus transactions.)

While the above example is from a small program for illustration, Table 1 shows examples of different memory behavior from the SPEC programs with reference or train inputs. We show the average number of L2 cache misses and the average number of memory bus transactions (per 1M instructions) for some example code regions. Similarly, we see these numbers may become very different if input is changed from reference to train, or vice versa.

Based on the above observation, we see, for different input sizes or patterns, different DVFS might be needed to have the best energy/performance results. For the qsort() example in Figure 2, a more aggressive (i.e., lower) DVFS setting should be used for a large input size (like 100M) to take advantage of the memory boundedness in the program and save more energy. Conversely, a more conservative (i.e., higher) DVFS setting should be used for a small input size (like 10K) to avoid excessive performance losses. For the SPEC benchmark 103.su2cor in Table 1, the code region in the function sweep() might need different DVFS settings for different input sets. Our empirical experience on the Intel Pentium-M processor shows that, assuming a performance loss constraint of 4%, this code region can be clocked at 1.0Ghz with the reference input, while it has to be clocked at the maximum 1.6Ghz for the train input.

While it is inherently difficult for a static compiler to make DVFS decisions adaptive to the above factors, dynamic compiler DVFS can utilize run-time system information and make input-adaptive and architecture-adaptive decisions.

2.3 Disadvantages and challenges

Having discussed its advantages, we would like to point out that dynamic compiler DVFS also has its disadvantages. The most significant one is that, just as for any dynamic optimization technique, every cycle spent for optimization might be a cycle lost to execution (unless optimizations are performed as side-line optimizations on a chip multiple-processor [5]). Therefore, one challenge to dynamic compiler driven DVFS is to design simple and inexpensive analysis and decision algorithms in order to minimize the run-time optimization cost.

3. Design Framework and DVFS Decision Algorithms

In this section, we present a design framework for the run-time DVFS optimizer (RDO) in a dynamic compilation and optimization environment. We start by considering some key design issues in general. Then we give the detailed design of a new DVFS decision algorithm.

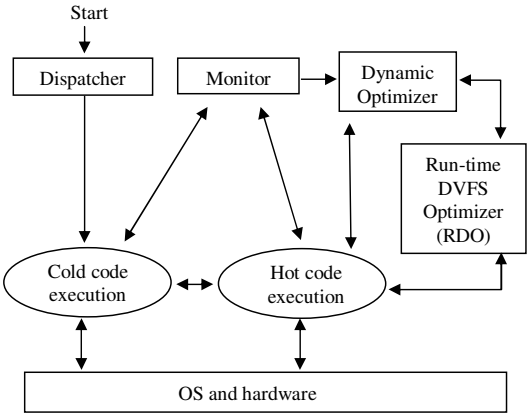

Figure 3: The overall block diagram showing the operation and interactions among different components of a dynamic compiler DVFS optimization system.

3.1 Key design issues

Candidate code region selection: Like other dynamic optimization techniques, we want only to optimize those frequently executed code regions (so-called hot code regions), in order to be cost effective. In addition, since DVFS is a relative slow process (the voltage transition rate is typically around $1mv/1\mu s$), we also want to only optimize long-running code regions. In our design, we choose functions and loops as candidate code regions. Since most dynamic optimization systems are already equipped with some light-weight profiling mechanism to identify hot code regions (for example DynamoRio profiles every possible loop target [5]), we will extend the existing profiling infrastructure to monitor and identify hot functions or loops.

DVFS decisions: For each candidate code region, an important step is to decide whether it is beneficial to apply DVFS to it (i.e., whether it can operate at a lower voltage and frequency to save energy without significant impact on the overall performance) and what the appropriate DVFS setting is. As we mentioned earlier, for a dynamic optimization system, the analysis or decision algorithm needs to be simple and fast to minimize overhead. Thus, the offline analysis techniques used by static compiler DVFS [14] are typically too time-consuming and are not appropriate here. For our work, we have designed a fast DVFS decision algorithm, which is based on an analytical decision model and uses hardware feedback information.

DVFS code insertion and transformation: If a candidate code region is found beneficial for DVFS, DVFS mode set instructions will be inserted at every entry point of the code region to start the DVFS, and at every exit point of the code region to restore the voltage level. One design question is how many adjusted regions we want to have in a program. Some existing static compiler algorithms choose only a single DVFS code region for a program [14] (to avoid the excessively long analysis time). In our design, we will allow/identify multiple DVFS regions to provide more energy saving opportunities. But things become complicated when two DVFS regions are nested. (If a child region is nested in a parent region, then a child may not know the DVFS setting for the parent at its exit points.) We provide two design solutions. One is to maintain a relation graph at run time, and only allow the parent region to be scaled. The other one is to have a DVFS-setting stacked so that both the parent and the child regions can be scaled. In addition

to the code insertion, the dynamic compiler can also perform code transformation to create more energy saving opportunities. One example is to merge two separate (small) memory bound code regions into one big one. Of course, we need to check that this code merging does not hurt the performance (or the correctness) of the program. So there will exist interactions among the DVFS optimizer and the conventional performance optimizer.

Overall operation block diagram: The block diagram in Figure 3 shows the overall operation and interactions between different components of a dynamic compiler DVFS optimization system. At the start, the dynamic optimizer dispatches or patches original binary code and delivers the code to execution by the hardware. At this moment, the system is in a *cold-code* execution mode. While the cold code is being executed, the dynamic optimization system monitors and identifies the frequently executed or hot code regions. Then, the RDO optimization is applied to the hot code regions, either before or after the conventional performance optimizations have been conducted. (For a hot code region, the RDO optimization can be applied once per program execution, or multiple times – which we call periodic re-optimization.) Lastly, if a code transformation is desirable, the RDO will query the regular performance optimizer to check the feasibility of the code transformation.

Next, we describe in detail a key design component: the DVFS decision algorithm.

3.2 DVFS decision algorithms

To make DVFS decisions, RDO first inserts some testing and decision code at the entry and exit points of a candidate code region. (A candidate region can be viewed as a single entry, multiple exits code region.) The testing and decision code collects some run-time information (such as number of cache misses or memory bus transactions for this code region). If enough information has been collected, RDO decides the appropriate DVFS setting for a candidate code region based on the collected information and the RDO setup. After a decision is made, RDO removes the testing and decision code and prepares for possible DVFS code insertion and transformation.

The above testing steps assume that a candidate code region has relatively stable or slowly-varying run-time characteristics for a given input. Therefore, the obtained decision based on the testing information will be valid for the rest of the program execution, or valid until the next re-optimization point if we choose periodic re-optimizations. Note that this assumption has been shown reasonable or valid in practice by studies such as [26].

The key testing step in the above is the DVFS decision making. As we mentioned earlier, in order to be beneficial for DVFS, a code region first needs to be long-running, which can be easily checked. The harder question is, for a long running code region, how to decide whether it is beneficial to have DVFS and what an appropriate DVFS setting is. To answer this question, we first look at an analytical decision model for DVFS.

3.2.1 An analytical decision model for DVFS

The discussion and analysis model in this section assume that the goal of our energy control is to minimize the energy consumption, *subject to some performance constraints*. (Note that the analytical model for a different objective, such as thermal control, might be different.)

In general, scaling down the CPU voltage and frequency will certainly reduce processor power consumption, but it will also slow down the CPU execution speed (and the resulting energy delay product improvement might be low or even negative). The key insight to a beneficial DVFS (which saves energy but with no or little performance impact) is that there exists an asynchronous memory system, which is independent of the CPU clock and is many times slower than the CPU. Therefore, if we can identify the CPU execution slack (i.e., CPU stall or idle cycles waiting for the completion of memory operations), we can scale down the CPU voltage and frequency to save energy without much performance impact.

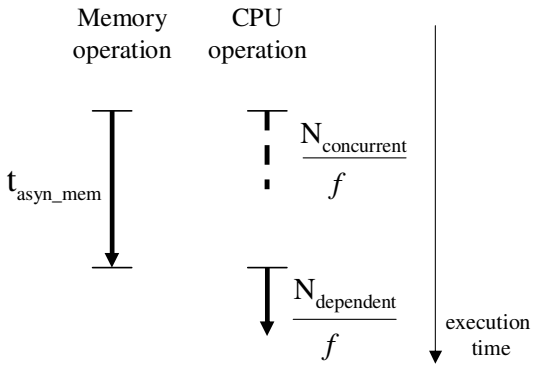

Figure 4: An analytical decision model for DVFS. t_{asyn_mem} is the asynchronous memory access time, $N_{concurrent}$ is the number of execution cycles for the concurrent CPU operation, $N_{dependent}$ is the number of cycles for the dependent CPU operation, f is the CPU frequency.

Based on the above rationale, Figure 4 shows our analytical decision model for DVFS, which is an extension of the analytical model proposed in [25]. As in Figure 4, the processor operations are categorized into two groups: memory operation and CPU operation. Since memory is asynchronous with respect to the CPU frequency f, we denote the time for memory operation as t_{asyn_mem}. The CPU operation time can be further separated into two parts: part 1 is those CPU operations that can run concurrently with memory operations; part 2 is those CPU operations that depend on the final results of the pending memory operations. Since the CPU operation time is dependent on the CPU frequency f, we denote the concurrent CPU operation time as $N_{concurrent}/f$, where $N_{concurrent}$ is the number of clock cycles for the concurrent CPU operation. Similarly, we denote the dependent CPU operation time as $N_{dependent}/f$. (Note, in actual program execution, the memory operation and the CPU operation, either concurrent or dependent, will be interleaved somehow. However, for an analytical model, we abstract the execution model by lumping all the occurrences of each category together. This is the same treatment as that in [25].)

From Figure 4, we see if the overlap period is memory bound, i.e $t_{asyn\_mem} > \frac{N_{concurrent}}{f}$, there exists a CPU slack time defined as

$$\text{CPU slack time } = t_{asyn\_mem} - \frac{N_{concurrent}}{f} \qquad (1)$$

Ideally, the concurrent CPU operation can be slowed down to consume the CPU slack time.

With the above model, we want to decide the frequency scaling factor β for a candidate code region. (So, if the original clock frequency is f, the new clock frequency will be $\beta\ f$; and the voltage will be scaled accordingly.) We assume the execution time for a candidate code region can be categorized according to Figure 4. So frequency scaling will have two effects on the CPU operation. First, it will increase the concurrent CPU operation time and reduce the CPU slack time (if any). Second, it will dilate the dependent CPU operation time, which will cause performance loss unless $N_{dependent} = 0$.

Next we will give a detailed method to select or compute the scaling factor β.

3.2.2 DVFS selection method

We introduce a new concept called relative CPU slack time. Based on the definition of CPU slack time in (1), we define

$$\text{relative CPU slack time} = \frac{t_{asyn\_mem} - N_{concurrent}/f}{total\_time} \quad (2)$$

where the $total\_time$ is the total execution time in Figure 4. For a memory bound case, $total\_time = t_{asyn\_mem} + N_{dependent}/f$. From Figure 4, we see the larger the relative CPU slack, the more frequency reduction the system can have without affecting the overall performance. So the frequency reduction (i.e., $1 - \beta$) is proportional to the relative CPU slack time. We have

$$(1 - \beta) = k_0 \left(\frac{t_{asyn\_mem}}{total\_time} - \frac{N_{concurrent}/f}{total\_time} \right) \quad (3)$$

where k_0 is a constant coefficient. Note that the value of k_0 can be chosen to be either relatively large to have more aggressive energy reduction, or relatively small to preserve performance more. Therefore, to take into account the effect of the maximum allowed performance loss P_{loss}, we replace k_0 in (3) by $k_0\ P_{loss}$, and we have

$$\beta = 1 - P_{loss}\ k_0 \frac{t_{asyn\_mem}}{total\_time} + P_{loss}\ k_0 \frac{N_{concurrent}/f}{total\_time} \quad (4)$$

Intuitively, the above equation means the scaling factor is negatively proportional to the memory intensity level (the term with t_{asyn_mem}), and positively proportional to the CPU intensity level (the term with $N_{concurrent}$). The time ratios in the above equation can be estimated using hardware feedback information such as hardware performance counter (HPC) events. For example, for an x86 processor, the two time ratios in the above equation can be estimated by ratios of some HPC events [11].

$$\frac{t_{asyn\_mem}}{total\_time} \simeq k_1 \frac{\text{Num\_of\_mem\_bus\_transactions}}{\text{Num\_of\_}\mu ops\text{\_retired}} \quad (5)$$

$$\frac{N_{concurrent}/f}{total\_time} \simeq k_2 \frac{\text{Num\_of\_FP\_INT\_instructions}}{\text{Num\_of\_}\mu ops\text{\_retired}} \quad (6)$$

where in (5) the first HPC event is the number of memory bus transaction, which is what we have used in Section 2 to measure memory busy-ness. The second HPC event is the total number of μops retired. The ratio of these two events is used to estimate the relative memory busy-ness. Similarly, in (6), the first HPC event is the number of FP/INT instruction retired (while there is an outstanding memory operation). The second event is also the number of μops retired. The ratio of these two HPC event is used to estimate the concurrent CPU busy-ness. Like k_0 in (3), k_1 and k_2 in the above are constant coefficients which depend on machine configurations and can be estimated empirically and reset at the installation time of a dynamic compiler.

Because the above method computes β directly from some run-time hardware information, it is simple and fast. The downside is that the formulation is relatively ad-hoc, especially the way it considers the constraint P_{loss}. We have also developed an alternative method which is more precise in handling the performance constraint P_{loss}, but is more complicated (it computes β using two separate sub-factors – details are omitted due to space limit). We see this alternative method as a complement to the above method.

4. Implementation and Deployment: Methodology and Experience

We have implemented a prototype of the proposed run-time DVFS optimizer (RDO), and integrated the RDO into a real dynamic compilation system. To evaluate it, our results present live-system physical power measurements.

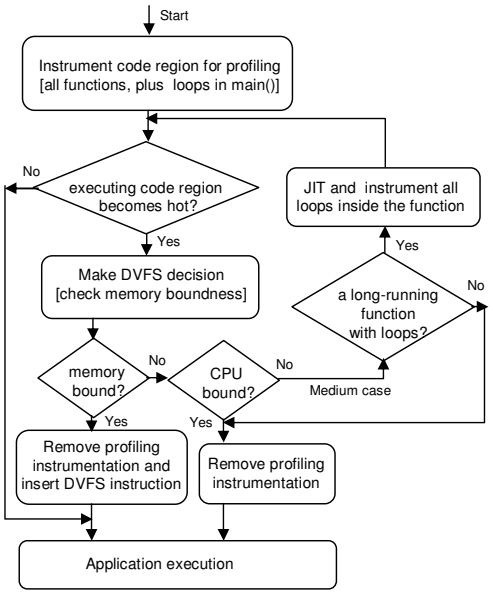

Figure 5: The operation flow diagram for our prototype implementation of the RDO.

4.1 Implementation

We use the Intel PIN system [18] as the basic software platform to implement our DVFS algorithm and develop the RDO. PIN is a dynamic instrumentation and compilation system developed at Intel and is publicly available. The PIN system which we use is based on the regular PIN but has been modified to be more suited and more convenient for dynamic optimizations. (For convenience, we refer it as O-PIN, i.e., Optimization PIN.) Compared to the standard PIN package, O-PIN has added more features to support dynamic optimizations, such as adaptive code replacement (i.e., the instrumented code can update and replace itself at run time) and customized trace or code region selection. In addition, unlike the PIN which is JIT-based and executes the generated code only [18], O-PIN takes a partial-JIT approach and executes a mix of the original code and the generated code. For example, O-PIN can be configured to first patch, instrument, and profile the original code at a coarse granularity (such as function calls only). Then, at run time, it selectively generates (JIT) code and does more fine-grained profiling and optimization of the dynamically compiled code (such as all loops inside a function). Therefore, O-PIN has less operation overhead, compared to regular PIN [1].

Figure 5 shows the operation flow graph for our prototype implementation of the RDO system. At the start, RDO instruments all function calls in the program, and all loops in the main() function, in order to monitor and identify the frequently executed code regions. (Strongly connected components in the call graph are treated as single nodes.) If a candidate code region is found hot (i.e. the execution count is greater than a *hot threshold*), DVFS testing and decision code will be started to collect run-time information and decide how memory bound the code region is. If the code region is found to be memory bound, RDO will remove the instrumentation code, insert DVFS mode set instructions, and resume the program execution. On the other hand, if a code region is found CPU bound, no DVFS instructions will be inserted. There is still a medium case where the candidate code region may exhibit mixed memory behavior (likely because it contains both memory-bound and CPU-bound sub-regions). For this case, RDO will check if it is a long-running function containing loops. If it is, a copy of this function will be

Running system (V/I measurement) (a) → Signal conditioning unit (noise reduction) (b) → Data acquisition unit (sampling and reading) (c) → Data logging unit (processing unit) (d)

Figure 6: Processor power measurement setup. This setup consists of four components.

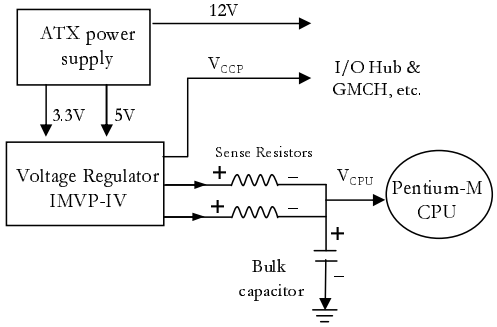

Figure 7: A system diagram showing the CPU voltage and current measurement points (marked by $+$ and $-$) on the development board.

dynamically generated and all loops inside this function will be identified[1] and instrumented. And the process continues.

The DVFS selection method in Section 3.2.2 is used to check the memory boundedness of a code region and select a DVFS setting. The required HPC events in equation (5), Number of memory bus transactions and Number of μops retired, are among the $\sim$ 100 countable HPC events provided by a Pentium-M processor [17]. (This will be the core of our hardware platform, as to be discussed in the next subsection.) However, the HPC event in equation (6) to estimate the time ratio $\frac{N_{concurrent}/f}{total\_time}$ is not available for Pentium-M processors. Instead, we approximate the estimation in (6) by a new ratio obtained from available but less-related HPC events. There are several ways to choose the HPC events for this. For our implementation, we used the ratio of Number of μops retired over Number of instructions retired. Based on our empirical experience, we found that the larger this ratio is, the more concurrent CPU instructions there are for a code region. (Note, in Section 5, we will actually show the inverse of this ratio, i.e., the average number of instructions per 1M μops retired.)

Based on the scaling factor β obtained from the DVFS decision algorithm, we choose the actual DVFS setting for a code region (i.e., new_f $=$ β old_f). Since most existing processors have only a limited number of DVFS setting points (e.g., six to eight), we pick the setting point close to the desired DVFS setting. (If the desired setting is between two available setting points, we pick the more conservative or higher one. Of course, the more fine-grained DVFS settings available, the better control effectiveness.)

[1]To identify loops, some linear-time loop analysis techniques such as that in [20] can be used. For our implementation, to reduce the run-time analysis overhead, a simple and fast loop-identification heuristic is used. A likely loop is identified if a conditional branch is going from a higher address to a lower address. Our experience shows this heuristic works quite well for most applications in practice.

4.2 Deployment in a real system

We have deployed our RDO system in a real running system. The hardware platform we use is an Intel development board with a Pentium-M processor (855GME, FW82801DB), which is shown in Figure 6a. The Pentium-M processor we use has a maximum clock frequency of 1.6GHz, two 32K L1 caches, and one unified 1M L2 cache. The board has a 400MHz FSB bus and 512M DDR RAM.

There are 6 DVFS settings or so-called SpeedSteps for Pentium-M (expressed in frequency/voltage pairs): 1.6GHz/1.48v, 1.4GHz/1.42v, 1.2GHz/1.27v, 1.0GHz/1.16v, 800MHz/1.04v, and 600MHz/ 0.96v. The voltage transition rate for DVFS is about $1mv/1\mu s$ (based on our own measurements).

The OS is Linux kernel 2.4.18 (with gcc updated to 3.3.2). We have implemented two loadable kernel modules (LKM) to provide user level support for DVFS control and HPC reading in the form of system calls.

The above system allows accurate power measurements. The overall procedure for power measurements is that, we first collect sampling points of CPU voltage and current values, then we compute the power trace and the total energy from these sampling points. Figure 6 shows the processor power measurement setup, which includes four components, as detailed below.

Running system voltage/current measurement unit: This unit isolates and measures CPU voltage and current signals. The reason for isolating and measuring the CPU power (instead of power for the whole board) is that we want to have more deterministic and accurate results, not affected by other random factors on the board. Figure 7 is a system diagram showing the CPU voltage and current measurement points (marked with $+$ and $-$) on the 855GME development board. As seen in the figure, we use the output sense resistors of the main voltage regulator (precision resistors of $2m\Omega$ each) to measure the current going to the CPU (i.e. measure the voltage drop, then use $I_{CPU} = V_{drop}/R_{sense}$), and use the bulk capacitor to measure the CPU voltage. Note that, as shown in Figure 7, if we simply measure the power supply line going to the voltage regulator, the obtained power reading will undesirably include power consumed by components other than the CPU (such as the I/O Hub).

Signal conditioning unit: This unit reduces the measurement noise to get more accurate readings. Measurement noise is inevitable because of the noise sources like the CPU board itself. In particular, since the voltage drop across the sense resistor in Figure

Table 2: Statistical results obtained for some SPEC benchmarks.
Average number is for per 1M μops; 9L name() means 9th loop in name()

Benchmark	total hot regions	total DVFS regions	region name	total μops	Average L2 cache misses	Average Memory trans	Average Inst	DVFS setting (Hz)
101.tomcatv	63	4	4th loop in main()	23M	3.9K	14.4K	0.99M	1.0G
			11th loop in main()	5M	9.1K	44.7K	0.99M	0.6G
			14th loop in main()	3M	10.0K	49.4K	0.98M	0.6G
			16th loop in main()	2M	12.7K	69.1K	0.97M	0.6G
104.hydro2d	184	3	tistep()	11M	4.5K	18.6K	0.91M	1.0G
			advnce()	284M	4.9K	21.5K	0.87M	1.2G
			check()	14M	5.3K	22.9K	0.85M	1.2G
173.applu	72	5	jacld()	208M	12.4K	24.8K	0.99M	0.8G
			blts()	286M	5.9K	11.5K	0.99M	1.2G
			jacu()	156M	12.7K	25.6K	0.99M	0.8G
			buts()	254M	7.0K	12.9K	0.99M	1.2G
			rhs()	188M	4.2K	8.2K	1.0M	1.4G
176.gcc	5673	0	Mem: 0.01K - 1.0K for candidate regions; No DVFS					
181.mcf	34	2	primal_net_simplex()	3644M	21.0K	83.0K	0.85M	1.0G
			flowcost()	20M	32.0K	112K	0.94M	0.6G
186.crafty	588	0	Mem: 0.00K - 0.01K for candidate regions; No DVFS					
187.facerec	207	2	9L gaborroutines_mp_gabortrato()	22M	3.4K	11.0K	0.95M	1.2G
			16L gaborroutines_mp_gabortrato()	11M	3.4K	10.5K	0.96M	1.2G
254.gap	823	1	collectGaib()	315M	6.6K	17.0K	0.86M	1.4G

7 is on the order of $1mv$ while the noise is on the order of $10mv$ in practice, the noise for our system is 10 times larger than the measured signal. Because noise typically has much higher frequency than the measured signals, we use a two-layer low-pass filter to reduce the measurement noise, which includes a National Instrument (NI) signal conditioning module AI05 and a simple RC filter as shown in Figure 6b. With these filters, we are able to reduce the relative noise error to less than 1%.

Data acquisition (DAQ) unit: This unit samples and reads the voltage and current signals. In order to capture the program behavior variations (especially with DVFS), a fast sampling rate is required. We use the NI data acquisition system DAQPad-6070E [21], which has a maximum sampling rate of 1.2M/s (aggregate), as shown in Figure 6c. Since three measurement channels are needed – two for the CPU current and one for the CPU voltage, we set a sampling rate of 200K/s for each channel (so a total of 600K/s is used). This gives a $5\mu s$ sample length for each channel. Given that the minimum voltage transition time is $20 - 100\mu s$ [11], the $5\mu s$ sampling length is adequate.

Data logging and processing unit: This is the host logging machine which processes the sampling data. Every 0.1 seconds, the DAQ unit sends collected data to the host logging machine via a high-speed fire-wire cable. (For each channel, 20K samples are first stored in an internal buffer in the DAQ unit before they are sent out.) The logging machine then processes the received data. We use a regular laptop running NI Labview DAQ software to process the data. We have configured the Labview for various tasks: monitoring, raw data recording, and power/energy computation.

5. Experimental Results

5.1 Experimental setup

For all experiments, we use the software and hardware platforms described in previous sections. Our run-time DVFS optimization system is set to have a performance loss constraint P_{loss} of 5%. (If a larger P_{loss} were used, the resulting frequency settings would be lower, allowing more aggressive energy savings. Conversely, a smaller P_{loss} would lead to larger and more conservative DVFS settings.) For a candidate code region, the *hot threshold* is chosen to be 4 (i.e., a code region is hot if it has executed at least 4 times). But we found our results are not sensitive to this value when it is varied from $3 - 20$. Since the voltage transition time between different SpeedSteps is about $100\mu s - 500\mu s$ for our machine [11], we set the long-running threshold for a code region (as described in our DVFS algorithm in Section 3) to be 1.5ms (or 2.4M cycles for a 1.6GHz processor) to make it as least 3X bigger than the voltage transition time. For nested functions, we handle them using a relation graph as described in Section 3

For evaluation, we use all SPEC2K FP and SPEC2K INT benchmarks. Since previous static compiler DVFS work in [14] used SPEC95 FP benchmarks, we also include them in our benchmark suites. In addition, we include some Olden benchmarks [6] as they are popular integer benchmarks to study program memory behavior.[2] For each benchmark, the Intel C++/Fortran compiler V8.1 is used to get the application binary (compiled with -O2). We test each benchmark with the largest ref input set (running to completion). The power and performance results reported here are average results obtained from three separate runs.

To illustrate and give insight for RDO operation, Table 2 shows some statistical results obtained from the RDO system for some SPEC benchmarks. In the table, we give total number of hot code regions in the program and total number of DVFS regions identified. For each DVFS code region, we show the total number of μops retired for the code region (in a single invocation), average L2 cache misses, average number of memory bus transactions, average number of instructions retired (per 1M μops), and the obtained DVFS settings. The DVFS settings are based on the average number of memory bus transactions and the average number of instructions retired. In general, the higher these two numbers are, the lower the DVFS setting. Taking the benchmark 104.hydro2d as an example, we see both numbers contributed to the final DVFS settings. The quantitative relationship between those numbers and the DVFS setting is based on the formulas in Section 3. Since there are only 6 available frequency/voltage settings for our system, the obtained β needs to be rounded up to an available frequency point. Overall, we see the number of DVFS opportunities identified by RDO ranges from large (e.g. as low as 0.6Ghz for 101.tomcatv) to small (e.g. no DVFS for 176.gcc).

In order to look more closely at RDO operation, we next exam-

[2] Olden manipulates 7 different kinds of data organizations and structures ranging from linked list to heterogeneous OcTree. We choose the first 7 benchmarks which cover all 7 kinds of data organizations being studied.

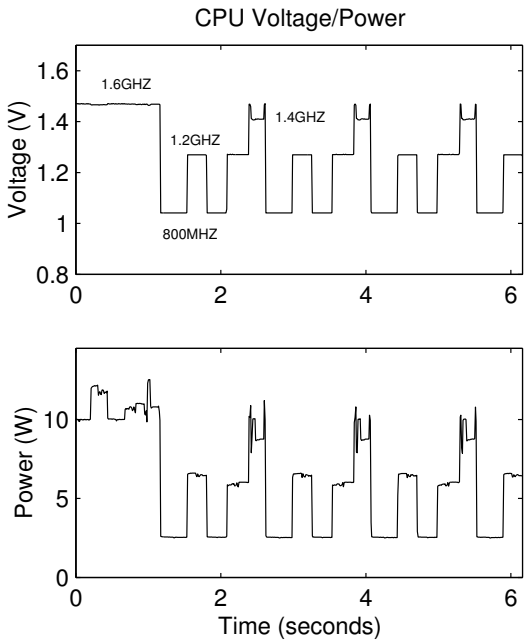

Figure 8: A partial trace of the CPU voltage and power for SPEC benchmark 173.applu running with the RDO

ine in detail one particular benchmark: 173.applu. Recall in Section 2 we observed recurring memory phase behavior for 173.applu. Analysis by RDO further reveals that those phase changes are mainly caused by invocations of the 5 different functions shown in Table 2. The 5 functions have different memory behavior in terms of average number of L2 cache misses and average number of memory bus transactions. Note this observation is consistent with the behavior shown in Figure 1. By inserting DVFS instruction directly in the code regions, RDO adjusts the CPU voltage and frequency to adapt to the program phase changes (with frequency settings of 0.8GHz for two regions, 1.2GHz for two other, and 1.4GHz for the last region). Figure 8 shows a part of the CPU voltage and power trace for 173.applu running with RDO. If we compare this figure with the figures in Section 2, it is interesting to see the CPU voltage/frequency are being adjusted to adapt to the recurring phase changes shown in Figure 1 (with lower clock frequencies corresponding to higher L2 cache misses). The power trace is also interesting. Initially it fluctuates around the value of $11W$ (due to different system switching activities). After the program execution enters into the DVFS code regions, the power drops dramatically to a level as low as $2.5W$. As will be shown by the experimental results in Section 5.2, the DVFS optimization applied to the code regions in 173.applu has led to considerable energy savings ($\sim$35%) with little performance loss ($\sim$5%).

5.2 Energy and performance results

We view the run-time DVFS optimizer (RDO) as an addition to the regular dynamic (performance) optimization system as shown in Figure 3. So, to isolate the contribution of the DVFS optimization, we will first report the energy and performance results relative to the O-PIN system *without* DVFS (i.e., we do not want to mix the effect of our DVFS optimization and that of the underlying dynamic compilation and optimization system, which is being developed heavily by researchers at Intel and U. of Colorado [1]). In addition, as a comparison, we will also report the energy results from a static voltage scaling, which simply scales the supply voltage and frequency statically for all benchmarks to get roughly the same amount of average performance loss as those in our results. (We chose $f = 1.4Ghz$ for static voltage scaling, which is the only voltage setting point in our system to get an average performance loss close to 5%.)

Table 3: Average results for each benchmark suite: *RDO* versus *Static*Scale.

Benchmark Suite	Performance degradation		Energy savings		Energy-Delay product improvement	
	RDO	*Static*	*RDO*	*Static*	*RDO*	*Static*
SPEC95 FP	2.1%	7.9%	24.1%	13.0%	22.4%	5.6%
SPEC2K FP	3.3%	7.0%	24.0%	13.5%	21.5%	6.8%
SPEC2K INT	0.7%	11.6%	6.5%	11.5%	6.0%	-0.3%
Olden	3.7%	7.8%	25.3%	13.7%	22.7%	6.3%

Figures 9 and 10 show the performance loss, energy savings, and energy delay product (EDP) improvement results for all SPEC95 FP, SPEC2K FP/INT, and Olden benchmarks. Note that these results have taken into account all DVFS optimization overhead, such as the time cost to check memory boundedness of a code region. For convenience, we refer to the result from our runtime DVFS optimizer as *RDO*, and refer to results by static voltage scaling as *StaticScale*. There are several interesting observations.

First, in terms of EDP improvement, *RDO* outperforms *StaticScale* by a big margin for nearly all benchmarks. This shows the efficiency of our design with fast and effective DVFS decisions.

Second, the energy and performance results for individual benchmarks in each benchmark suite vary significantly. On the high end, we have achieved up to 64% energy savings (4.9% performance loss) for SPEC95 FP (101.tomcatv), up to 70% energy savings (0.5% performance loss) for SPEC2K FP (171.swim), up to 44% energy savings (with 5% performance loss) for SPEC2K INT (181.mcf), and up to 61% energy savings (4.5% performance loss) for Olden benchmarks (Health). On the low end, we see close-to-zero (or even slightly negative) EDP improvement for some benchmarks in each benchmark suite. To understand the reasons, we see that the efficiency of a DVFS control is largely constrained by the memory boundedness of an application. The more memory bound an application is, the more opportunities and energy saving potentials there are for DVFS. Relative to our experimental system in Figure 6 (with a large 1M L2 cache), these benchmarks show a variety of memory boundedness which leads to a variety of the EDP results. Overall, we see the distribution of the SPEC2K-INT EDP results is concentrated and close to the low end, while the overall distribution of the Olden and SPEC-FP EDP results is very spread out between the high end and the low end.

The average results for each benchmark suite are summarized in Table 3. We show both the results from our techniques and the *StaticScale* results. On average, we have achieved an EDP improvement of 22.4% for SPEC95 FP, 21.5% for SPEC2K FP, 6.0% for SPEC2K INT, and 22.7% for Olden benchmarks. These represent $3-5$ fold better results as compared to the *StaticScale* EDP improvement: 5.6% for SPEC95 FP, 6.8% for SPEC2K FP, -0.3% for SPEC2K INT, and 6.3% for Olden benchmarks. (The average SPEC2K INT EDP result is relatively lower compared to the other three benchmark suites. This is because SPEC2K INT benchmarks are dominantly CPU bound as shown by previous studies [14]. There is nothing intrinsic about floating versus integer data. It is just about the amount of memory traffic.)

We also want to have a rough comparison with the static compiler DVFS results in [14] based on the reported energy performance numbers in that paper. (We were not able to re-implement their optimizer and replicate the experiments in [14].) Compared

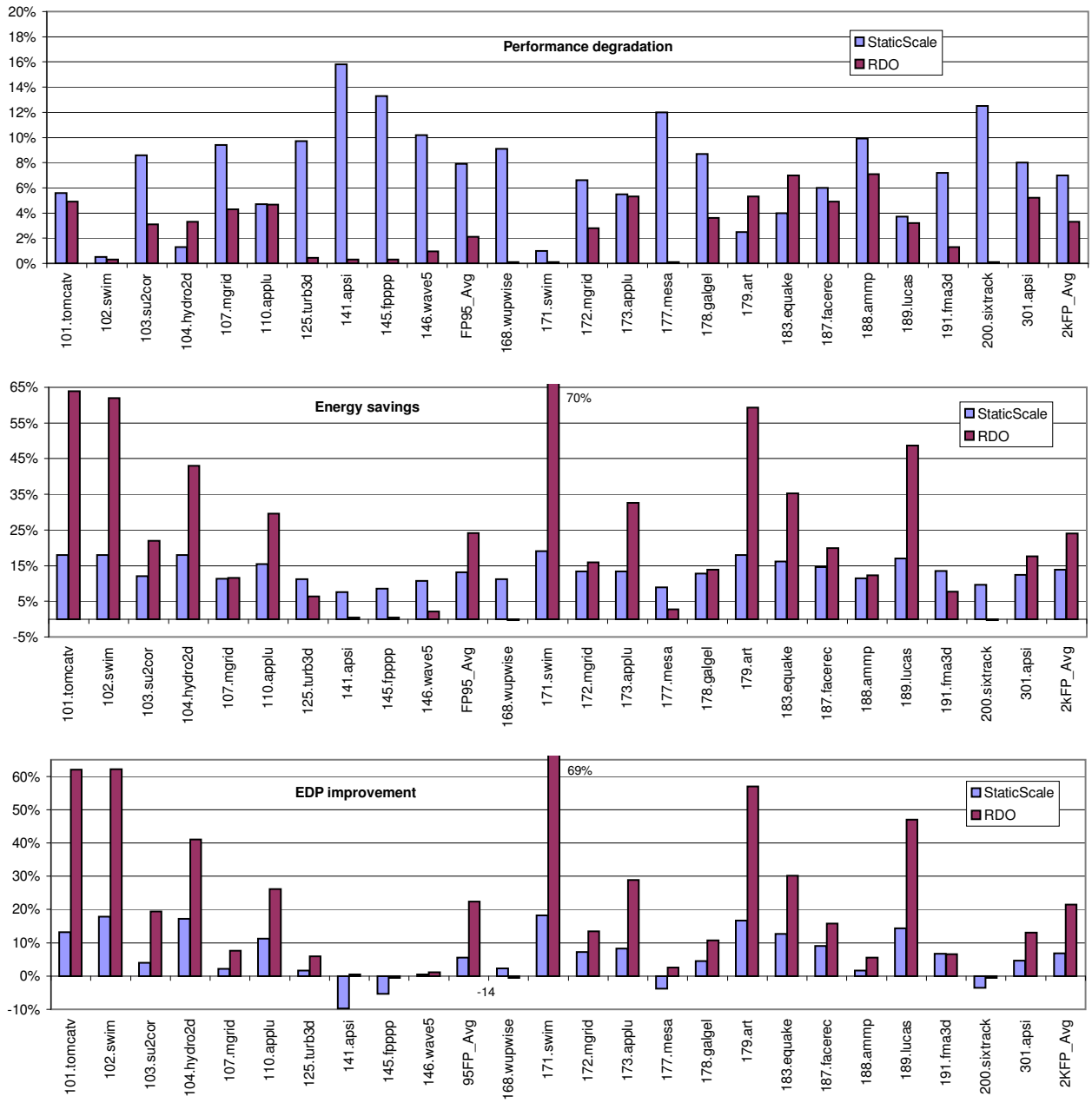

Figure 9: Performance degradation, energy savings, and energy-delay product (EDP) improvement for SPEC95 FP benchmarks (on the left) and SPEC2K FP benchmarks (on the right). We show results for both our runtime DVFS optimizer (*RDO*) and the *StaticScale* voltage scaling.

to the reported results for SPEC95 FP benchmarks in [14] (on average: 2.1% performance loss, 11.0% energy savings, 9.0% EDP improvement), we have achieved on average 2X as much energy savings for the same amount of performance loss. Apart from the dynamic versus static benefits described in Section 2, there are two other key factors contributing to the different results. First, the static compiler DVFS algorithm in [14] picks only a single DVFS code region for a program (to avoid the excessive offline analysis time), while our online DVFS design can identify multiple DVFS code regions in a program as long as they are beneficial, as illustrated by examples in Table 2. Second, the decision algorithm in [14] is based on (offline) timing profiling for each code region, while our algorithm is more microarchitecture oriented and directly uses information about run-time environment, such as hardware performance counts.

Overall, the results in Figures 9 and 10 and Table 3 show the proposed technique is promising in addressing the energy and performance control problem in microprocessors. We attribute the promising results to the efficiency of our design and to the advantages of the dynamic compiler driven approach.

5.3 Basic O-PIN overhead

A dynamic optimization system has basic setup/operation overhead (i.e., the time spent to do basic setup, to monitor/identify frequently executed code regions, etc). This overhead must be offset or amortized by the subsequent performance optimization gain before we can see any net performance improvement. It has been shown that a dynamic optimizer with aggressive performance

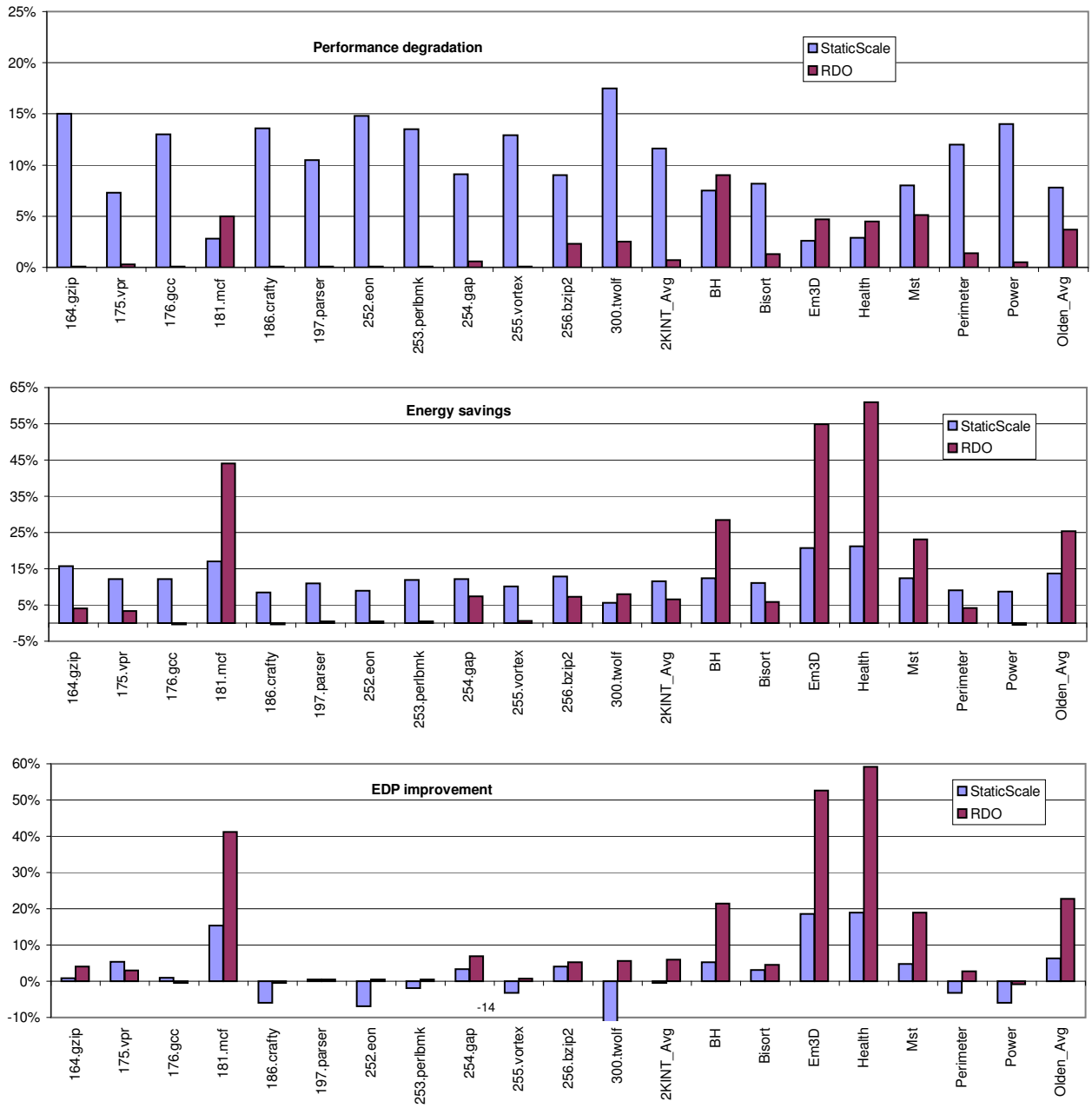

Figure 10: Performance degradation, energy savings, and energy-delay product (EDP) improvement for SPEC2K INT benchmarks (on the left) and Olden benchmarks (on the right). We show results for both our runtime DVFS optimizer (*RDO*) and the *StaticScale* voltage scaling.

optimizations will have significant performance gains for various benchmarks [2, 5].

The O-PIN system we use is a dynamic-optimization infrastructure and does not include implemented performance optimizations. (Users can use this infrastructure to implement their own performance optimizations like loop unrolling and data prefetching, but this is beyond the scope of this paper.) So compared to the native application, the basic O-PIN system has a negative performance gain. In other words, there is a performance and energy overhead associated with the basic O-PIN infrastructure. Next, to give a complete picture of this work, we will also show results for O-PIN and DVFS relative to the native.

Figure 11 shows the performance and energy overhead for the basic O-PIN infrastructure (computed as relative to the native). We see, for individual benchmarks, the performance overhead is as low as 0.5% for benchmarks like 164.gzip or 171.swim, and is as high as 15% for benchmarks like 176.gcc. On average, the performance overhead for O-PIN is about 3.3% for SPEC95 FP, 1.8% for SPEC2K FP, 3.7% for SPEC2K INT, and 0.5% for Olden benchmarks. The energy overhead values are similar. Note, these values are significantly lower than the basic overhead for a regular PIN system [18], because of the low-overhead implementation in O-PIN as described in Section 4.

If we look at the DVFS results from our scheme when computed with the inherited infrastructure overhead, the EDP numbers will be lower than those in the last subsection, as we expected. On average, the EDP improvement with the inherited overhead is about 16.7% for SPEC95 FP, 17.9% for SPEC2K FP, -1.4% for SPEC2K INT,

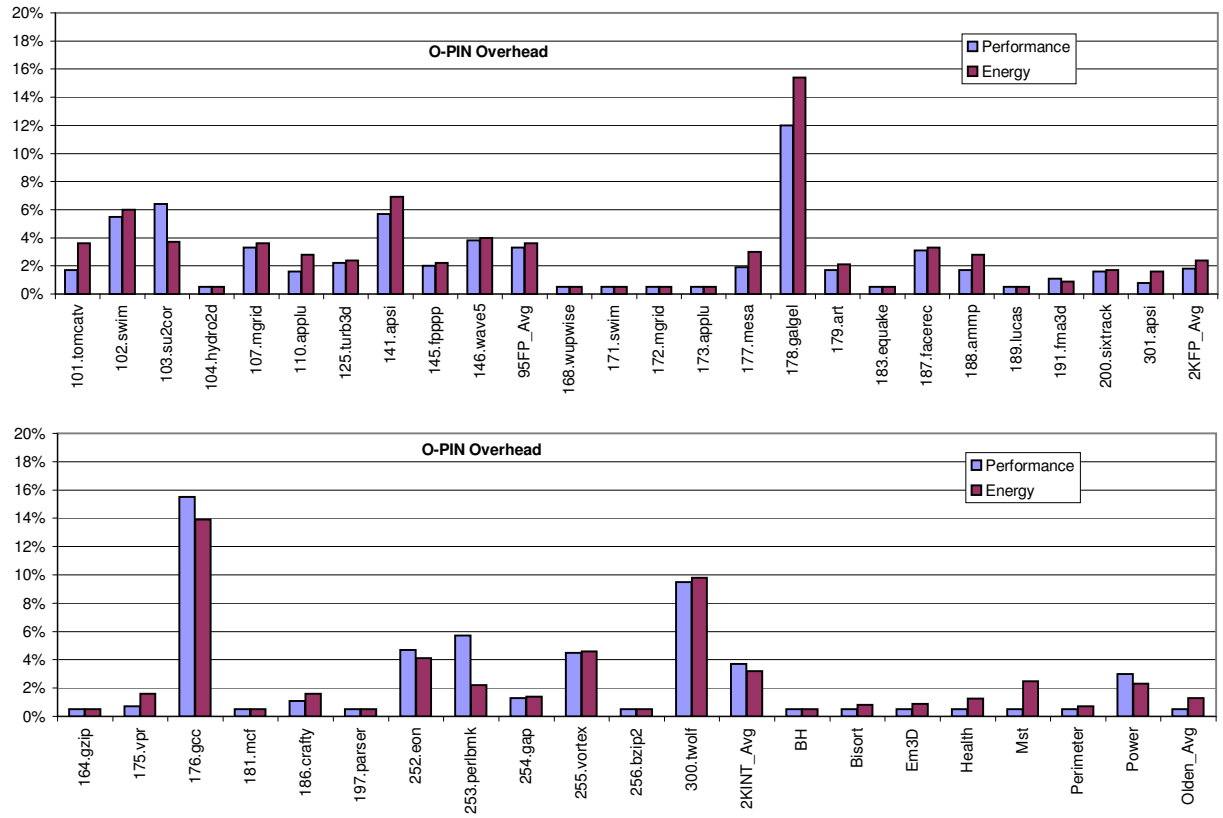

Figure 11: Performance and Energy overhead for the basic O-PIN infrastructure without applying optimizations.

and 20.9% for Olden benchmarks. In general, these are clearly still promising results.

5.4 Discussion and micro-architectural suggestions

For experimental results in this section, it would be desirable to have some potential or upper-bound DVFS numbers, and show how far our results are from these upper-bound numbers. However, it is still an open research question how to effectively and accurately compute the upper-bound DVFS results. One possible way to estimate the upper bound is to extend the mathematical formulation in [24] from optimizing multiple sequential scaling points to optimizing multiple DVFS code regions. We leave it for possible future exploration.

While the experimental results in this section are promising, they could be further improved if more micro-architectural support were available. One possible support could be some logic to identify and predict CPU execution slack such as that proposed in [9]. This would make the DVFS computation easier and more accurate. Another possible support could be some power-aware hardware monitoring counters and events to monitor the power consumption in a processor unit and the voltage variations. In addition, more fine-grained DVFS settings could make the intra-task DVFS design more effective. Our experience shows that, for many code regions in the benchmarks, RDO was forced to select an unnecessarily high voltage/frequency setting due to a lack of some middle steps between the current six SpeedSteps in Pentium-M processors.

6. Related Work

As we mentioned in the introduction, nearly all existing intra-task DVFS schemes are based on hardware [19, 22], OS time-interrupt [7, 23], or static compiler [14, 25] techniques. Very little has been done in the direction of dynamic compiler driven DVFS.

One piece of related work along that direction is the Java virtual machine DVFS presented in [12]. Their work is similar to ours in the sense that both use run-time software to decide DVFS settings for the application. However, their work differs from ours in the following aspect. First, they use the Java virtual machine to target Java applications at the granularity of Java method, while we use a general dynamic optimization system to target general applications at a more fine-grained granularity including code regions like loops. Second, their DVFS algorithm does not take into account the memory boundedness of a code region (they assume the execution time of a code region always scales linearly with the frequency, no matter how memory bound it is). Also, their DVFS algorithm assumes some sort of time budget (so called *projected time*), and compares the current execution time with the time-budget to decide how much to scale. This treatment might be suitable for MultiMedia applications which have a pre-determined time-budget for each frame, but might not be as suitable to general applications. In contrast, our DVFS algorithm considers the memory boundedness of a code region, and works well for general applications. Third, the power evaluation in [12] is based on simulation, while our evaluation is based on live-system physical power measurements.

7. Future Work

There are several possible avenues for future work. The focus of this work is on the new concept of dynamic compiler driven DVFS and the overall design framework. A direct follow-up work would look at specific design issues and techniques in more depth, such as code transformation and periodic re-optimization for DVFS. Also deeper analysis could be done for the experimental results, such as a breakdown of the results/benefits by regions or by different contributing factors. However, since we use a real system as opposed to simulation, it will be challenging to break down the results/benefits

in an effective way.

Another possible future work is to implement some conventional performance optimizations (like loop unrolling and data prefetching), and study the interactions between energy optimizations and performance optimizations in a run-time system. In addition, some new processors allow DVFS for the memory bus as well. A possible future direction is to generalize the analytical decision model and the DVFS algorithm in this paper for the case where both CPU and memory can have DVFS.

8. Conclusions

The work presented in this paper represents some of our most recent efforts in developing a dynamic compilation framework for microprocessor energy and performance control. The focus is on dynamic voltage and frequency scaling (DVFS).

Specifically, we have given reasoning and statistical results to highlight the unique features and advantages of dynamic compiler driven DVFS over existing techniques. We have presented a design framework of the run-time DVFS optimizer in a general dynamic compilation system. We have described the methodology and reported our experiences in implementing and deploying a run-time DVFS optimization system.

Experimental results based on physical measurements show that SPEC benchmarks benefit up to 70% energy savings (with about 0.5% performance loss). On average, results with over 40 SPEC or Olden benchmarks show that our technique leads to an energy delay product (EDP) improvement which is 3X-5X better than that from static voltage scaling, and is more than 2X better than that reported by a static compiler DVFS scheme. We attribute these promising results to the efficiency of our design, which makes fast/effective decisions for multiple code regions, and to the advantages of the dynamic compiler-driven approach in terms of fine-grained and code-aware phase adaptation, and the ability to utilize accurate run-time information.

While the proposed technique is an effective method for microprocessor voltage and frequency control, the design framework and methodology described in this paper can be generalized for other emerging microprocessor issues, such as di/dt and thermal control. Pilot studies in [13] have already shown that a dynamic compiler, with feedback information from a collaborating hardware control system, can provide a novel approach to tackling to the di/dt problem. Overall, we feel the proposed dynamic compilation framework has a great potential in addressing the energy, performance, and power control problem in modern processors.

Acknowledgments

We would like to thank Gilberto Contreras, Ulrich Kremer, Chung-Hsing Hsu, C-K Luk, Robert Cohn, and Kim Hazelwood for their helpful discussions during the development of this work, and the anonymous reviewers for their useful comments and suggestions on the paper. This work is supported in part by NSF grants CCR-0086031 (ITR), CNS-0410937, CCF-0429782, Intel, IBM, and SRC.

9. References

[1] PIN manuals and APIs. In *http://rogue.colorado.edu/Pin/index.html*, August 2005.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of PLDI*, June 2000.

[3] L. Baraz, T. Devor, O.Etzion, S.Gondenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *Proc. of the 36th Micro*, Dec 2003.

[4] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In *Proceedings of the IEEE SOC Conference*, Sep 2003.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, March 2003.

[6] M.C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[7] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proceedings of DATE*, Feb 2004.

[8] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of ISCA*, June 1997.

[9] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of ISCA*, May 2002.

[10] D. Genossar and N. Shamir. Intel pentium m processor power estimation, budgeting, optimization, and validation. *Intel Technology Journal*, 07(2), 2003.

[11] S. Gochman, R. Ronen, et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 07(2), 2003.

[12] V. Haldar, C. Probst, V. Venkatachalam, and M. Franz. Virtual machine driven dynamic voltage scaling. Technical Report CS-03-21, University of California, Irvine, CA, Oct 2003.

[13] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proceedings of ISLPED'04*, August 2004.

[14] C-H Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proc. of PLDI-2003*, pages 38–48, June 2003.

[15] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of ISCA*, June 2003.

[16] C. J. Hughes and S. V. Adve. A formal approach to frequent energy adaptations for multimedia applications. In *Proc. of 31st ISCA*, June 2004.

[17] Intel Corporation, Santa Clara, CA. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2005.

[18] C-K Luk, R. Cohn, R. Muth, R. Muth, H. Patil, A. Kaluser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI'05*, June 2005.

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

[20] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, 1997.

[21] National Instruments. *Data Acquisition (DAQ) Hardware, http://www.ni.com/dataacquisition*, 2005.

[22] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, and M.L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proc. of the 35th Micro*, pages 356–367, November 2002.

[23] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of CASE'02*, Oct 2002.

[24] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage/frequency scaling: An exact algorithm and a linear-time heuristic approximation. In *Proc. of ISLPED*, August 2005.

[25] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proc. of 2003 PLDI*, June 2003.

[26] Y.Wu, M. Breternitz, J.Quek, O. Etzion, and J. Fang. The accuracy of initial prediction in two-phase dynamic binary translators. In *Proceedings of CGO'04*, March 2004.