Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance

A general dynamic-compilation environment offers power and performance control opportunities for microprocessors. The authors propose a dynamic-compiler-driven runtime voltage and frequency optimizer. A prototype of their design, implemented and deployed in a real system, achieves energy savings of up to 70 percent.

Qiang Wu
Margaret Martonosi
Douglas W. Clark
Princeton University

Vijay Janapa Reddi
Dan Connors
University of Colorado at Boulder

Youfeng Wu
Jin Lee
Intel Corp.

David Brooks
Harvard University

Energy and power have become primary issues in modern processor design. Processor designers face increasingly vexing power and thermal challenges, such as reducing average and maximum power consumption, avoiding thermal hotspots, and maintaining voltage regulation quality. In addition to static design time power management techniques, dynamic adaptive techniques are becoming essential because of an increasing gap between worst-case and average-case demand.

In recent years, researchers have proposed and studied many low-level hardware techniques to address fine-grained dynamic power and performance control issues.[1-3] At a higher level, the compiler and the application can also take active roles in maximizing microprocessor power control effectiveness and actively managing power, performance, and thermal goals. In this article, we explore power control opportunities in a general dynamic-compilation environment for microprocessors. In particular, we look at one control mechanism: dynamic voltage and frequency scaling (DVFS). We expect our methods to apply to other control means as well.

A dynamic compiler is a runtime software system that compiles, modifies, and optimizes a program's instruction sequence as it runs. Examples of infrastructures based on dynamic compilers include the HP Dynamo,[4] the IBM DAISY (Dynamically Architected Instruction Set from Yorktown),[5] Intel IA32EL[6] and the Intel PIN.[7]

Figure 1 shows the architecture of a general dynamic-compiler system. A dynamic-compilation system serves as an extra execution layer between the application binary and the operating system and hardware. At runtime, a dynamic-compilation system interacts with application code execution and applies possible optimizations to it. Beyond regular performance optimizations, a dynamic compiler can also apply energy optimizations such as DVFS because most DVFS implementations allow direct software control through mode set instructions (by accessing special mode set registers). A dynamic compiler can simply insert DVFS mode set instructions into application binary code. If CPU execution slack exists (that is, CPU idle cycles are waiting for

0272-1732/06/$20.00 © 2006 IEEE Published by the IEEE computer Society

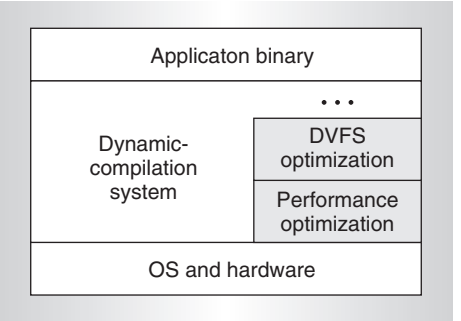

Figure 1. General dynamic-compiler system architecture, serving as an extra execution layer between application binary and OS and hardware.

memory), these instructions can scale down the CPU voltage and frequency to save energy with little or no performance impact.

Because a dynamic compiler has access to high-level information on program code structure as well as runtime system information, a dynamic-compiler-driven DVFS scheme offers some unique features and advantages not present in existing hardware-based or static-compiler-based DVFS approaches. (See the "Why dynamic-compiler-driven DVFS?" sidebar for detail.)

Our work is one of the first efforts to develop dynamic-compiler techniques for microprocessor voltage and frequency control.[8] We

Why dynamic-compiler-driven DVFS?

Most existing research efforts on fine-grained DVFS control fall into one of two categories: hardware or OS interrupt-based approaches or static-compiler-based approaches.

Hardware or OS time-interrupt-based DVFS techniques typically monitor particular system statistics, such as issue queue occupancy,[1] in fixed time intervals and choose DVFS settings for future time intervals.[1-3] Because the time intervals are predetermined and independent of program structure, DVFS control by these methods might 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.[4] Thus, hardware or OS techniques might not be able to infer enough about application code attributes and find the most effective adaptation points. Another reason is that program phase changes are often recurrent (that is, loops). In this case, the hardware or OS schemes would need to detect and adapt to the recurring phase changes repeatedly. A compiler-driven DVFS scheme can apply DVFS to fine-grained code regions, adapting naturally to program phase changes. Hardware- or OS-based DVFS schemes with fixed intervals lack this code-aware adaptation.

Existing compiler DVFS work focuses mainly on static-compiler techniques.[5,6] Typically, these techniques use profiling to learn about program behavior. Then, offline analysis techniques, such as linear programming,[5] decide on DVFS settings for some code regions. A limitation is that the DVFS setting obtained at static-compile time might not be appropriate for the program at runtime because the profiler and the actual program have different runtime environments. The reasoning behind the above statement is that DVFS decisions depend on the program's memory-boundedness. In turn, the program's behavior in terms of memory-boundedness depends on runtime system characteristics such as machine or architecture configuration or program input size and patterns. For example, machine or architecture settings such as cache configuration or memory bus speed can affect how much CPU slack or idle time exists. Also, different program input sizes or patterns can affect how much memory must be used and how it must be used. It is thus inherently difficult for a static compiler to make DVFS decisions that adapt to these factors. In contrast, dynamic-compiler DVFS can use runtime system information to make input-adaptive and architecture-adaptive decisions.

We must also point out that dynamic-compiler DVFS has disadvantages. The most significant is that, just as for any dynamic-optimization technique, every cycle spent for optimization is a cycle lost to execution. Therefore, our challenge is to design simple and inexpensive analysis and decision algorithms that minimize runtime optimization cost.

References

1. G. Semeraro et al., "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture," *Proc. 35th Ann. Symp. Microarchitecture* (Micro-35), IEEE Press, 2002, pp. 356-367.
2. D. Marculescu, "On the Use of Microarchitecture-Driven Dynamic Voltage Scaling," *Proc. Workshop on Complexity-Effective Design* (WCED 00), 2000, http://www.ece.rochester.edu/~albonesi/wced00/.
3. A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems* (CASES 02), ACM Press, 2002, pp. 238-246.
4. M.C. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to energy reduction," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE Press, 2003, pp. 157-168.
5. C.-H. Hsu and U. Kremer, "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 38-48.
6. F. Xie, M. Martonosi, and S. Malik, "Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 49-62.

have developed a design framework for a runtime DVFS optimizer (RDO) in a dynamic-compilation environment. We have implemented a prototype RDO and integrated it in an industrial-strength dynamic optimization system. We deployed the optimization system in a real hardware platform that allows us to directly measure CPU current and voltage for accurate power and energy readings. To evaluate the system, we experimented with physical measurements for more than 40 SPEC or Olden benchmarks. Evaluation results show that the optimizer achieves significant energy efficiency—for example, up to 70 percent energy savings (with 0.5 percent performance loss) for the SPEC benchmarks.

Design framework

There are several key design issues to consider for the RDO in a dynamic-compilation and optimization environment.

Candidate code region selection

For cost effectiveness, we want to optimize only frequently executed code regions (so-called hot code regions). In addition, because DVFS is a relatively slow process (with a voltage transition rate typically around 1 mv per 1 μs), we want to optimize only long-running code regions. Therefore, in our design, we chose functions and loops as candidate code regions. Most dynamic optimization systems are already equipped with a lightweight profiling mechanism to identify hot code regions (for example, DynamoRio profiles every possible loop target[9]). We extended 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 applying DVFS is beneficial (that is, whether the region can operate at a lower voltage and frequency to save energy without significant impact on overall performance) and to determine the appropriate DVFS setting. For a dynamic optimization system, the analysis or decision algorithm must be simple and fast to minimize overhead. The offline analysis techniques used by static-compiler DVFS[10] are typically too time consuming and are not appropriate here. For our work, we followed an analytical decision model to design a fast DVFS decision algorithm that uses hardware feedback information.

DVFS code insertion and transformation

If the decision algorithm finds DVFS beneficial for a candidate code region, we insert DVFS mode set instructions at every code region entry point to start DVFS and at every exit point 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 (to avoid an excessively long analysis time).[10] In our design, we identify multiple DVFS regions to provide more energy-saving opportunities. In addition to code insertion, the dynamic compiler can perform code transformation to create energy-saving opportunities—for example, merging two separate (small) memory-bound code regions into one big one. The DVFS optimizer and the conventional performance optimizer interact to check that this code merging doesn't harm the program's performance or correctness.

Operation block diagram

Figure 2 shows a block diagram of a dynamic-compiler DVFS optimization system's overall operation and the interactions between its components. At the start, the dynamic optimizer dispatches or patches original binary code and delivers it to the hardware for execution. The system is now in cold-code execution mode. While the cold code executes, the dynamic optimization system monitors and identifies the hot code regions. Then, the RDO applies optimization to the hot code regions, either before or after conventional performance optimizations have occurred. The system is now in hot-code execution mode, in which the now-optimized code is executed. Finally, if a code transformation is desirable, the RDO queries the regular performance optimizer to check the code transformation's feasibility.

DVFS decision algorithms

To make DVFS decisions, the RDO first inserts testing and decision code at a candidate code region's entry and exit points. The code collects runtime information such as the number of cache misses or memory bus transactions for this code region. When sufficient

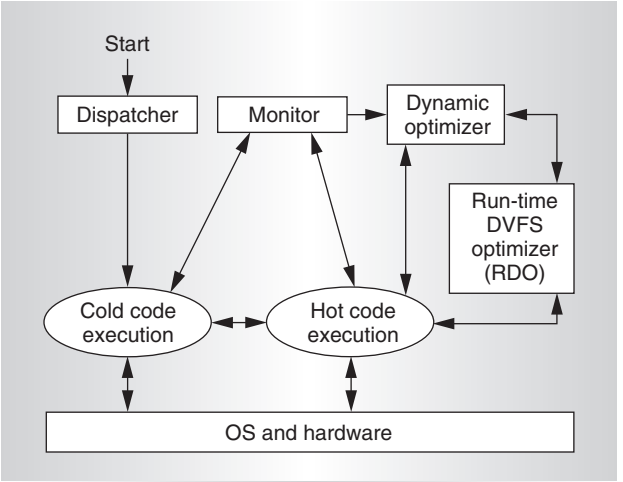

Figure 2. Operation and interaction block diagram of dynamic-compiler DVFS optimization system.

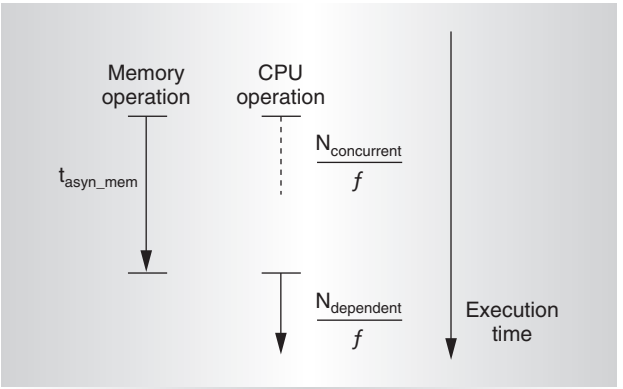

Figure 3. Analytical decision model for DVFS (t_{asyn_mem} is asynchronous memory access time, $N_{concurrent}$ is the number of execution cycles for concurrent CPU operation, $N_{dependent}$ is the number of cycles for dependent CPU operation, and f is CPU clock frequency).

information has been collected, the RDO tests whether the code region is long-running. Then it decides whether applying DVFS is beneficial (saving energy with little or no performance cost) and what the appropriate DVFS setting is for the candidate region.

The key observation supporting beneficial DVFS is the existence of an asynchronous memory system independent of the CPU clock and many times slower than the CPU. Therefore, if we can identify CPU execution slack (CPU stall or idle cycles waiting for the completion of memory operations), we can scale down CPU voltage and frequency to save energy without much performance impact.

Figure 3 shows our analytical decision model for DVFS based on this rationale; it is an extension of the analytical model proposed by Xie, Martonosi, and Malik.[11] It categorizes processor operations into memory operations and CPU operations. Because memory is asynchronous with respect to CPU frequency f, we denote memory operation time as t_{asyn_mem}. CPU operation time can be further divided: Part 1 is CPU operations that can run concurrently with memory operations; part 2 is CPU operations dependent on the final results of pending memory operations. Because CPU operation time depends on CPU frequency f, we denote concurrent CPU operation time $N_{concurrent}/f$, where $N_{concurrent}$ is the number of clock cycles for concurrent CPU operations. Similarly, we denote dependent CPU operation time $N_{dependent}/f$.

Figure 3 shows that if the overlap period is memory-bound—that is, $t_{asyn\_mem} > N_{concurrent}/f$—there is a CPU slack time defined as

$$\text{CPU slack time} = t_{asyn\_mem} - N_{concurrent}/f$$

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

Following this model, we want to compute frequency-scaling factor β for a candidate code region. (Thus, if the original clock frequency is f, the new clock frequency will be βf, and the voltage will be scaled accordingly.) We introduce a new concept called relative CPU slack time, which we define as CPU slack time over total execution time. For a memory-bound case, total_time = $t_{asyn\_mem} + N_{dependent}/f$. From Figure 3, we see that the larger the relative CPU slack, the more frequency reduction the system can have without affecting overall performance. So frequency reduction $(1 - \beta)$ is proportional to relative CPU slack time.

A derivation we have detailed elsewhere[8] gives the following equation for β:

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

where P_{loss} is the maximum allowed performance loss expressed in percentage, and k_0 is a constant coefficient depending on machine

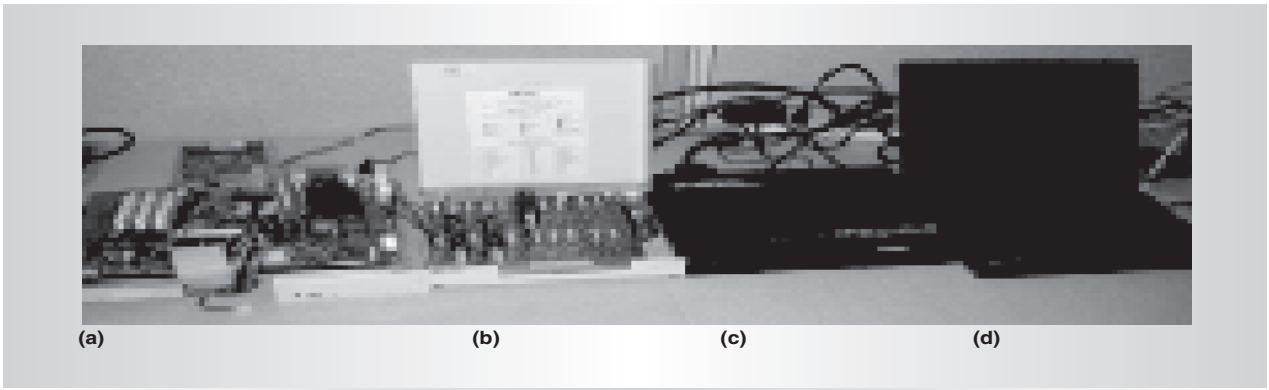

Figure 4. Processor power measurement setup: running system (a); signal-conditioning unit (b); data acquisition unit (c); data-logging unit (d).

configurations. Intuitively, this equation means that the scaling factor is negatively proportional to the memory intensity level (the term including t_{asyn_mem}) and positively proportional to the CPU intensity level (the term including $N_{concurrent}$).

We can estimate the time ratios in this equation by using hardware feedback information such as hardware performance counter (HPC) events. For example, for an x86 processor, we can estimate the first time ratio in the equation by using the ratio of two HPC events: the number of memory-busy transactions and the number of micro-operations retired.[8]

Implementation and deployment methods and experience

We have implemented a prototype of the proposed RDO and integrated it into a real dynamic-compilation system. To evaluate it, we conducted live-system physical power measurements.

Implementation

To implement our DVFS algorithm and develop the RDO, we used the Intel Pin system as the basic software platform. Pin is a dynamic instrumentation and compilation system and is publicly available (http://rogue.colorado.edu/Pin/index.html). We modified the regular Pin system to make it more suitable and more convenient for dynamic optimizations. We refer to the modified system as O-Pin (Optimization Pin). Compared with the standard Pin package, O-Pin has more features supporting dynamic optimizations, such as adaptive code replacement (the instrumented code can update and replace itself at runtime) and customized trace or code region selection. In addition, unlike the standard Pin, which is JIT (just-in-time compiler) based and executes generated code only, O-Pin takes a partial-JIT approach and executes a mix of original and generated code. For example, we can configure O-Pin to first patch, instrument, and profile the original code at a coarse granularity (such as function calls only). Then, at runtime, it selectively generates JIT code and performs 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 than regular Pin.

We outline the prototype RDO system's operation as follows:

1. The RDO instruments all function calls in the program and all loops in the main function to monitor and identify frequently executed code regions.
2. If the RDO finds that a candidate code region is hot (that is, the execution count is greater than a certain threshold), the DVFS testing and decision code starts to collect runtime information and decide how memory-bound the code region is.
3. If the code region is memory-bound, the RDO removes the instrumentation code, inserts DVFS mode set instructions, and resumes program execution. On the other hand, if a code region is CPU-bound, no DVFS instructions are inserted.
4. For the medium case, in which the candidate code region exhibits mixed memory behavior (probably because it

contains both memory-bound and CPU-bound subregions). the RDO checks whether it is a long-running function containing loops. If it is, the RDO dynamically generates a copy of this function and identifies and instruments all loops inside the function.
5. The process repeats.[8]

Deployment in a real system

We deployed our RDO system in a real running system. Figure 4a shows the hardware platform, an Intel development board containing a Pentium M processor. The Pentium M we used has a maximum clock frequency of 1.6 GHz, two 32-Kbyte L1 caches, and one unified 1-Mbyte L2 cache. The board has a 400-MHz front-side bus and a 512-Mbyte double-data-rate RAM.

The Pentium M has six DVFS settings, or "SpeedSteps" (expressed in frequency/voltage pairs): 1.6 GHz/1.48 V, 1.4 GHz/1.42 V, 1.2 GHz/1.27 V, 1.0 GHz/1.16 V, 800 MHz/1.04 V, and 600 MHz/0.96 V. The DVFS voltage transition rate is about 1 mv per 1 μs (according to our own measurements).

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

The running system allows accurate power measurements. Figure 4 shows the entire processor power measurement setup, which includes the following four components.

- *Running system.* This unit (Figure 4a) isolates and measures CPU voltage and current signals. We isolate and measure CPU power rather than the entire board's power because we want more deterministic and accurate results, unaffected by random factors on the board. We use the main voltage regulator's output sense resistors to measure current going to the CPU, and the bulk capacitor to measure CPU voltage.
- *Signal-conditioning unit.* This unit (Figure 4b) reduces measurement noise for more accurate readings. Measurement noise from sources such as the CPU board is inevitable. Because noise typically has far higher frequency than measured signals, we use a two-layer low-pass filter to reduce measurement noise.
- *Data acquisition (DAQ) unit.* This unit (Figure 4c) samples and reads voltage and current signals. Capturing program behavior variations (especially with DVFS) requires a fast sampling rate. We use National Instruments' data acquisition system DAQPad-6070E, which has a maximum aggregate sampling rate of 1.2M/s (http://www.ni.com/dataacquisition). We set a sampling rate of 200,000 per second for each channel (5-μs sample length), which is more than adequate for our reads.
- *Data-logging and -processing unit.* This unit (Figure 4d) is the host logging machine, which processes sampling data. Every 0.1 seconds, the DAQ unit sends collected data to the host logging machine via a high-speed fire-wire cable. The logging machine then processes the received data. We use a regular laptop running National Instruments' Labview DAQ software to process the data. We configured Labview for various tasks: monitoring, raw data recording, and power and energy computation.

Experimental results

For all experiments, we chose a performance loss target P_{loss} of 5 percent. (With a larger P_{loss}, the resulting frequency settings would be lower, allowing more aggressive energy savings. Conversely, a smaller P_{loss} would lead to higher and more conservative DVFS settings.) Because the voltage transition time between SpeedSteps is about 100 μs to 500 μs for our machine,[12] we set the long-running threshold for a code region to 1.5 ms (or 2.4 million cycles for a 1.6-GHz processor) to make it at least three times the voltage transition time.

For evaluation, we used all the SPECfp2000 and SPECint2000 benchmarks. Because previous static-compiler DVFS work[10] used SPECfp95 benchmarks, we also included them in our benchmark suites. In addition, we included several Olden benchmarks,[13] which are popular integer benchmarks for studying program memory behavior. For each benchmark, we used the Intel C++/Fortran compiler V8.1 to obtain the application binary (compiled with optimization level 2). We tested each benchmark with the largest reference input set running to completion. The power

Table 1. Information and DVFS settings obtained by RDO for SPEC benchmark 173.applu. Average numbers are per million micro-operations retired.

Total hot regions	Total DVFS regions	Region name	Total micro-operations (millions)	Average L2 cache misses (thousands)	Average memory transactions (thousands)	Average instructions retired (millions)	DVFS setting (GHz)
72	5	jacld()	208	12.4	24.8	0.99	0.8
		blts()	286	5.9	11.5	0.99	1.2
		jacu()	156	12.7	25.6	0.99	0.8
		buts()	254	7.0	12.9	0.99	1.2
		rhs()	188	4.2	8.2	1.0	1.4

and performance results reported here are average results obtained from three separate runs.

Table 1 shows program information and results obtained by the RDO system for SPEC benchmark 173.applu, a program for elliptic partial differential equations. The table lists the total number of hot code regions in the program and the total number of DVFS regions identified. For each DVFS code region, it shows total number of micro-operations retired for the code region (in a single invocation), average number of L2 cache misses, average number of memory bus transactions, average number of instructions retired, and obtained DVFS settings.

Figure 5 shows part of the CPU voltage and power traces for 173.applu running with RDO. By inserting DVFS instructions directly into the five code regions indicated in the table, RDO adjusts the CPU voltage and frequency to adapt to program phase changes. Specifically, as program execution enters code region jacld(), RDO scales CPU clock frequency (and voltage) from the default 1.6 GHz to the selected DVFS setting 0.8 GHz. When the program enters the next code region, blts(), the clock frequency switches to 1.2 GHz; and so on. The power trace is also interesting. Initially it fluctuates around the value of 11 W (because of various system-switching activities). After program execution enters the DVFS code regions, power drops dramatically to a level as low as 2.5 W. As experimental results will show, DVFS optimization applied to the code regions in 173.applu led to considerable energy savings (~35 percent) with little performance loss (~5 percent).

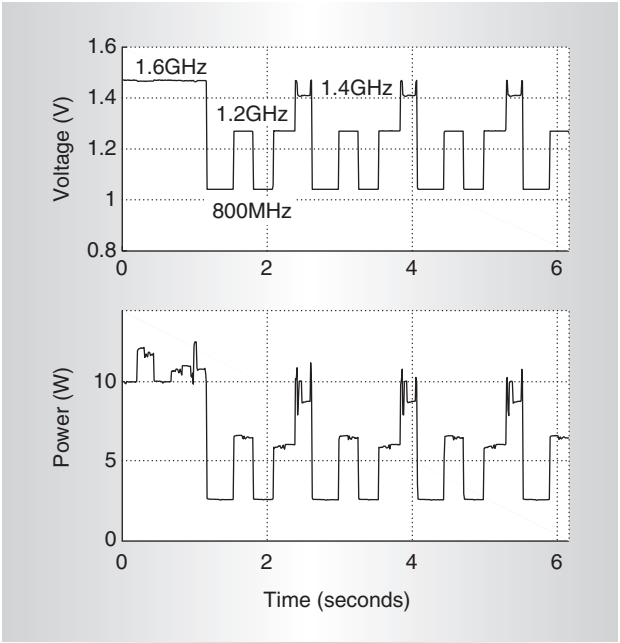

Figure 5. Partial traces of CPU voltage and power for SPEC benchmark 173.applu running with RDO.

Energy and performance results

As Figure 2 shows, we view the RDO as an addition to the regular dynamic (performance) optimization system. So, to isolate the contribution of the DVFS optimization, we report energy and performance results relative to the O-Pin system without DVFS (we don't want to mix the effect of our DVFS optimization and that of the underlying dynamic compilation and optimization system being developed by researchers at Intel and the University of Colorado). In addition, as a comparison, we report the energy results obtained from static voltage scaling, which scales supply voltage and frequency statically for all

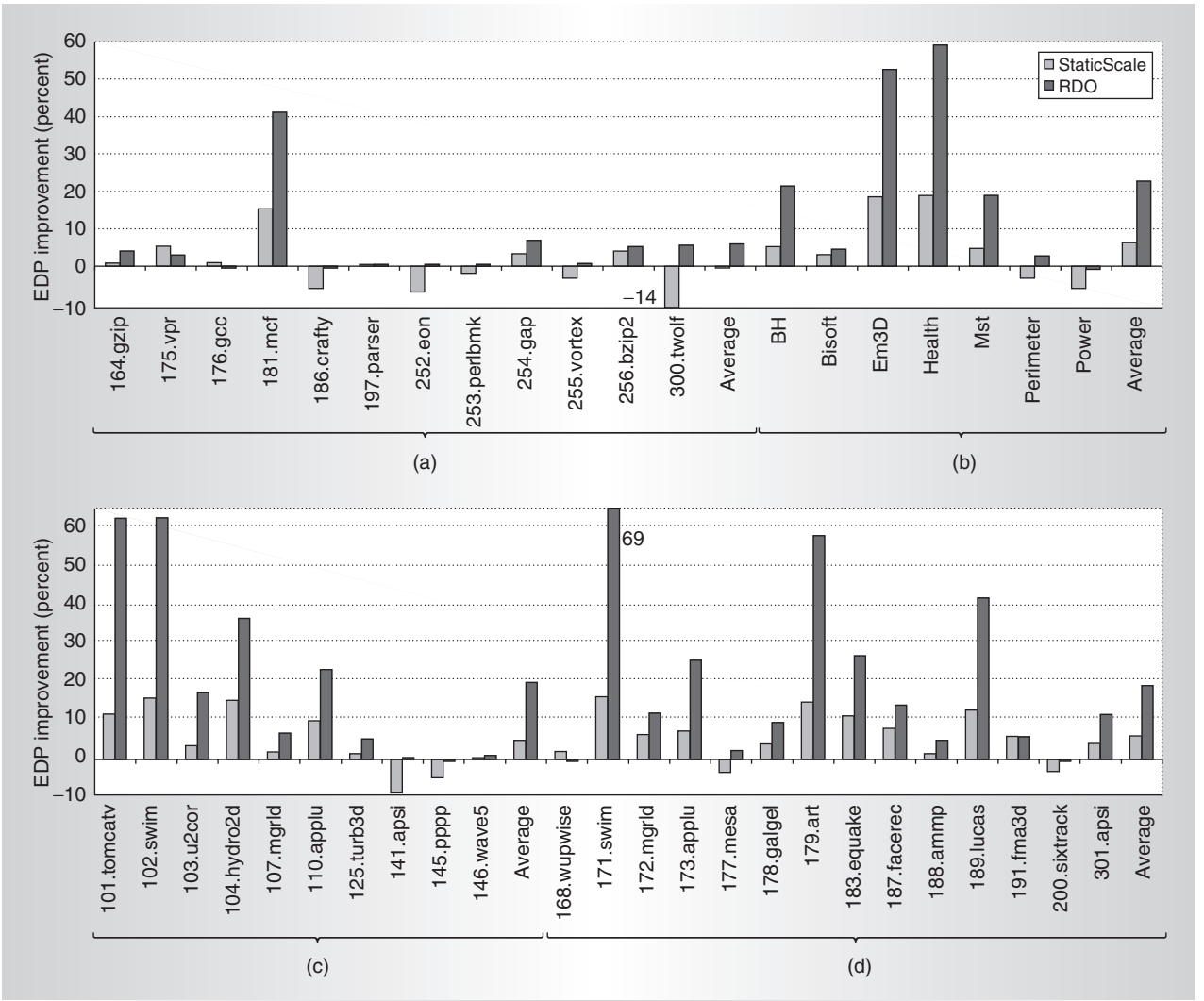

Figure 6. Energy delay product (EDP) improvement for SPECint2000 (a), Olden (b), SPECfp95 (c), and SPECfp2000 (d) benchmarks, obtained from RDO and static voltage scaling (StaticScale).

benchmarks to get roughly the same amount of average performance loss as that in our results. (We chose f = 1.4 GHz for static voltage scaling, which is the only voltage setting point in our system that produces an average performance loss close to 5 percent.)

Figure 6 shows energy delay product (EDP) improvement results obtained for all our benchmarks from RDO and static voltage scaling (StaticScale). These results take into account all DVFS optimization overhead, such as time required for checking a code region's memory-boundedness.

These results lead to several interesting observations. First, in terms of EDP improvement, RDO outperforms StaticScale by a wide margin for nearly all benchmarks. Second, energy and performance results for individual benchmarks in each benchmark suite vary significantly (from -1 percent to 70 percent for RDO; from -14 percent to 20 percent for StaticScale). This is because individual applications vary widely in their proportion of memory-boundedness. In particular, the SPECint2000 programs are almost all CPU-bound (except for 181.mcf) and hence are nearly immune to our attempted optimizations.

Table 2 summarizes the average results for each benchmark suite. It shows the results from our techniques and the StaticScale results. The EDP improvement results for RDO represent a three- to fivefold better

Table 2. Average results for each benchmark suite: RDO versus StaticScale.

Benchmark suite	Performance degradation (%)		Energy savings (%)		Energy delay product improvement (%)	
	RDO	StaticScale	RDO	StaticScale	RDO	StaticScale
SPECfp95	2.1	7.9	24.1	13.0	22.4	5.6
SPECfp2000	3.3	7.0	24.0	13.5	21.5	6.8
SPECint2000	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

improvement than the StaticScale results. (We present additional experimental results, including those for basic O-Pin overhead, in another publication.[8])

Overall, the results in Figures 6 and Table 2 show that the proposed technique addresses the microprocessor energy and performance control problem well. We attribute these promising results to our design's efficiency and to the advantages of the dynamic-compiler-driven approach.

Microarchitectural suggestions

The experimental results are promising, but we could improve them if more microarchitectural support were available. For example, logic to identify and predict CPU execution slack, such as proposed by Fields, Bodik, and Hill,[14] would make DVFS computation easier and more accurate. Another supporting mechanism would be power-aware hardware-monitoring counters and events to keep track of a processor unit's power consumption and voltage variations. In addition, additional fine-grained DVFS settings would make the intratask DVFS design more effective. In our experiments, RDO was forced to select an unnecessarily high voltage or frequency setting for many code regions in the benchmarks because the Pentium M processors lack intermediate steps between the six SpeedSteps.

The proposed technique's orthogonal approach and advantages make it an effective complement to existing techniques and will help us work toward a multilayer (software and hardware), collaborative power management scheme. In addition, the design framework and methodology described here are applicable to other emerging microprocessor problems, such as di/dt and thermal control. Such techniques and framework offer great promise for adaptive power and performance management in future microprocessors. MICRO

Acknowledgments

We thank Gilberto Contreras, Ulrich Kremer, Chung-Hsing Hsu, C.-K. Luk, Robert Cohn, and Kim Hazelwood for their helpful discussions during the development of our design framework and methodology. We also thank the anonymous reviewers for their useful comments and suggestions about this article. Qiang Wu was supported by an Intel Foundation Graduate Fellowship. Our work was also supported in part by NSF grants CCR-0086031 (ITR), CNS-0410937, CCF-0429782, Intel, IBM, and SRC.

References

1. D. Marculescu, "On the Use of Microarchitecture-Driven Dynamic Voltage Scaling," *Proc. Workshop on Complexity-Effective Design* (WCED 00), 2000, http://www.ece.rochester.edu/~albonesi/wced00/.
2. G. Semeraro et al., "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture," *Proc. 35th Ann. Symp. Microarchitecture* (Micro-35), IEEE Press, 2002, pp, 356-367.
3. A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management," *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems* (CASES 02), ACM Press, 2002, pp. 238-246.
4. V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. Conf. Programming Language Design and Implementation* (PLDI 00), ACM Press, 2000, pp. 1-12.
5. K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural

Compatibility," *Proc. 24th Ann. Int'l Symp. Computer Architecture* (ISCA 97), IEEE Press, pp. 26-37.
6. L. Baraz et al., "IA-32 Execution Layer: A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems," *Proc. 36th Ann. Symp. Microarchitecture* (Micro-36), ACM Press, 2003, pp. 191-204.
7. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. Conf. Programming Language Design and Implementation* (PLDI 05), ACM Press, 2005, pp. 190-200.
8. Q. Wu, et al., "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," *Proc. 38th Ann. Symp. Microarchitecture* (Micro-38), ACM Press, 2005, pp. 271-282.
9. D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," *Proc. Int'l Symp. Code Generation and Optimization* (CGO 03), ACM Press, 2003, pp. 265-275.
10. C.-H. Hsu and U. Kremer, "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 38-48.
11. F. Xie, M. Martonosi, and S. Malik, "Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits," *Proc. Conf. Programming Language Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 49-62.
12. S. R. Gochman et al., "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology J.*, vol. 7, no. 2, May 2003, pp. 21-36.
13. M.C. Carlisle et al., "Early Experiences with Olden," *Proc. 6th Int'l Workshop on Languages and Compilers for Parallel Computing* (LCPC 93), 1993, http://www.cs.princeton.edu/~mcc/olden.html.
14. B. Fields, R. Bodik, and M.D. Hill, "Slack: Maximizing Performance under Technological Constraints," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE Press, 2002, pp. 47-58.

Qiang Wu is a PhD candidate in Princeton University's Computer Science Department. His research interests include power-aware microprocessor design and control. Wu has an MSc in electrical engineering from the University of Sydney. He is a student member of the IEEE and the ACM.

Margaret Martonosi is a professor of electrical engineering at Princeton University. Her research interests include computer architecture and hardware-software interfaces, with a focus on power-efficient systems and mobile computing. Martonosi has a BS from Cornell University and an MS and a PhD from Stanford University, all in electrical engineering. She is a senior member of the IEEE and a member of the ACM.

Douglas W. Clark is a professor of computer science at Princeton University. His research interests include computer architecture, low-power techniques, and clocking and timing in digital systems. Clark has a BS in engineering and applied science from Yale University and a PhD in computer science from Carnegie-Mellon University.

Vijay Janapa Reddi is pursuing a PhD in the Department of Electrical and Computer Engineering at the University of Colorado at Boulder. His research interests include the design of dynamic code transformation systems for deployment in everyday computing environments. Reddi has a BSc in computer engineering from Santa Clara University.

Dan Connors is an assistant professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Colorado at Boulder and the leader of the DRACO research group on runtime code transformation. His research interests include modern compiler optimization and multithreaded multicore architecture design. Connors has an MS and PhD in electrical engineering from the University of Illinois at Urbana-Champaign.

Youfeng Wu is a principal engineer in Intel's Corporate Technology Group and leads a research team on multiprocessor compilation and dynamic binary optimizations. His research interests include codesigned power-efficient computer systems, binary and

dynamic optimizations, and security and safety enhancement via compiler and binary tools. Wu has an MS and PhD in computer science from Oregon State University. He is a member of the ACM and the IEEE.

Jin Lee is the president and CEO of Amicus Wireless Technology. He previously worked at Intel, where he participated in the work described in this article. His research interests include compiler technology, network processors, and semiconductor fabrication technologies. Lee has an MS and PhD in mechanical engineering from Stanford University.

David Brooks is an assistant professor of computer science at Harvard University. His research interests include architectural-level power modeling and power-efficient design of hardware and software for embedded and high-performance computer systems. Brooks has a BS from the University of Southern California and an MA and PhD from Princeton University, all in electrical engineering. He is a member of the IEEE.

Direct questions and comments about this article to Qiang Wu, Dept. of Computer Science, Princeton University, Princeton, NJ 08544; jqwu@princeton.edu.