

# Dynamic Compiler Driven Control for Microprocessor Energy and Performance

Qiang Wu<sup>1</sup>, V.J. Reddi<sup>2</sup>, Youfeng Wu<sup>3</sup>, Jin Lee<sup>3</sup>, Dan Connors<sup>2</sup>,  
David Brooks<sup>4</sup>, Margaret Martonosi<sup>1</sup>, Douglas W. Clark<sup>1</sup>

<sup>1</sup>Depts. of Computer Science, Electrical Engineering, Princeton University

<sup>2</sup>Dept. of Electrical and Computer Engineering, U. of Colorado at Boulder

<sup>3</sup>Programming Systems Lab, Corporate Tech. Group, Intel Corporation

<sup>4</sup>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

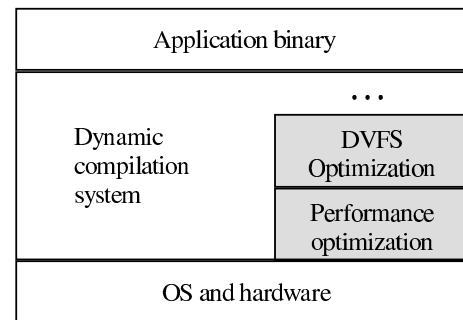
This work explores power and performance control opportunities in a general dynamic compilation environment for microprocessors. A compiler driven run-time voltage and frequency optimizer is proposed and designed. A prototype of the design is implemented and deployed in a real system with physical power measurements.

## 1. Introduction

Energy and power have become first-class issues in modern processor design. Designers of current or future generation processors face increasingly vexing power/thermal challenges such as reducing average and maximum power consumption, avoiding thermal hotspots, and maintaining voltage regulation quality. In addition to static design-time techniques for power management, dynamic adaptive techniques are becoming crucial due to the fact that there is an increasing gap between the worst-case and average-case demand.

In recent years, numerous low-level hardware techniques have been proposed and studied to address fine-grained dynamic power and performance control issues, for example [11, 13, 14]. At a higher level, the compiler and the application can also take an active role in microprocessor power control, in order to maximize the control effectiveness and actively manage its power, performance, and thermal goals. In this work, 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). Our methods should be more broadly applicable to other control means as well.

A dynamic compiler is a run-time software system that compiles, modifies, and optimizes a program's instruction sequence as it runs. Examples of dynamic compiler based infrastructures include HP Dynamo [2], IBM DAISY [5], and Intel PIN [10]. Figure 1 shows the system architecture for a general dynamic compiler. A dynamic compilation system serves as an extra execution layer between the application binary and the OS/hardware. At run time, a dynamic compilation system interacts with the application code execution and applies possible optimizations to it. Beyond regular performance optimizations, a dynamic compiler can also apply energy optimizations like DVFS. This is because most DVFS implementations allow direct software control via mode set instructions (by accessing special mode set registers). A dynamic compiler can just insert DVFS mode set instructions into application binary code. If there exists CPU execution slack (i.e., CPU idle cycles waiting for



**Figure 1: System architecture for a general dynamic compiler, which is seen as an extra execution layer between the application binary and the OS/hardware.**

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

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

Our work [15] is one of the first efforts to develop dynamic compiler techniques for microprocessor voltage and frequency control. We have proposed a design framework for a run-time DVFS optimizer (RDO) in a dynamic compilation environment. In addition, we have implemented a prototype of the RDO and integrated it into an industrial-strength dynamic optimization system (a variant of the Intel PIN system [10]). 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 physical 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.

## 2. Why Dynamic Compiler Driven DVFS?

(Note to the editor: This section will be placed as a sidebar)

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

Existing hardware or OS time-interrupt based DVFS techniques typically monitor some system statistics (such as issue queue occupancy [13]) in fixed time intervals, and decide DVFS settings for future time intervals [11, 13, 14]. Since the time intervals are pre-determined and independent of program structure, 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 [9]. Thus, 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. Thus one advantage for a compiler driven DVFS scheme is that it can apply DVFS to fine-grained code regions so as to adapt naturally to program phase changes, while hardware or OS-based DVFS schemes with fixed intervals lack this code-aware adaptation.

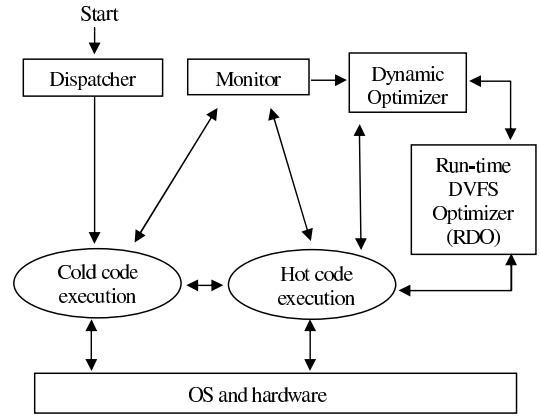
Existing compiler DVFS work is primarily based on static compiler techniques [8, 16]. Typically profiling is used to learn about program behavior. Then some offline analysis techniques (such as linear programming [8]) are used to decide DVFS settings for some code regions. One limitation to static compiler 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. Then, the program behavior in term of memory boundedness is in turn dependent on run-time system characteristics 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. While it is thus inherently difficult for a static compiler to make DVFS decisions adaptive to the above factors, dynamic compiler DVFS has the advantage of being able to utilize run-time system information and make input-adaptive and architecture-adaptive decisions.

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. Therefore, one challenge 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

There are several key design issues to consider for the run-time DVFS optimizer (RDO) in a dynamic compilation and optimization environment.

**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 tar-



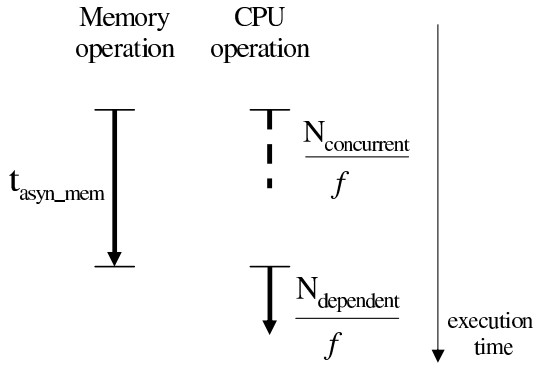
**Figure 2: The overall block diagram showing the operation and interactions among different components of a dynamic compiler DVFS optimization system.**

get [3]), 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 [8] are typically too time-consuming and are not appropriate here. For our work, we have designed a fast DVFS decision algorithm that 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 [8] (to avoid the excessively long analysis time). In our design, we will identify multiple DVFS regions to provide more energy saving opportunities. 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 2 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, RDO optimization is applied to the hot code regions, either before or after the conventional performance optimizations have been conducted. The system is now in *hot-code* execution mode with possible performance optimizations or DVFS. Lastly, if a code transformation is desirable, the RDO will query the regular performance optimizer to check the feasibility of the code transformation.



**Figure 3: 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.

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

### 3.1 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. 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 first tests whether the code region is long-running. Then it decides whether it is beneficial to apply DVFS and what the appropriate DVFS setting is for a candidate code region.

The key observation supporting beneficial DVFS (which saves energy but costs no or little performance) 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 (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.

Based on this rationale, Figure 3 shows our analytical decision model for DVFS, which is an extension of the analytical model proposed in [16]. Processor operations are categorized into two groups: memory operations and CPU operations. 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 for CPU operations that can run concurrently with memory operations; part 2 is for CPU operations dependent on the final results of pending memory operations. Since the CPU operation time is dependent on the CPU frequency  $f$ , we denote the concurrent CPU operation time by  $N_{concurrent}/f$ , where  $N_{concurrent}$  is the number of clock cycles for the concurrent CPU operations. Similarly, we denote the dependent CPU operation time by  $N_{dependent}/f$ .

From Figure 3, 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

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

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

With the above model, we want to compute the frequency scaling factor  $\beta$  for a candidate code region. (So, if the original clock frequency is  $f$ , the new clock frequency will be  $\beta \cdot f$ ; and the voltage

will be scaled accordingly.) We introduce a new concept called relative CPU slack time, which is defined as the 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 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. A derivation detailed in [15] gives the following equation for  $\beta$ .

$$\beta = 1 - P_{loss} k_0 \frac{t_{asyn\_mem}}{total\_time} + P_{loss} k_0 \frac{N_{concurrent}/f}{total\_time} \quad (2)$$

where  $P_{loss}$  is the maximum allowed performance loss expressed in percentage;  $k_0$  is a constant coefficient depending on machine configurations. 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 first time ratio in the above equation can be estimated by the ratio of two HPC events: number of memory busy transactions and number of  $\mu ops$  retired. More details on this can be found in [15].

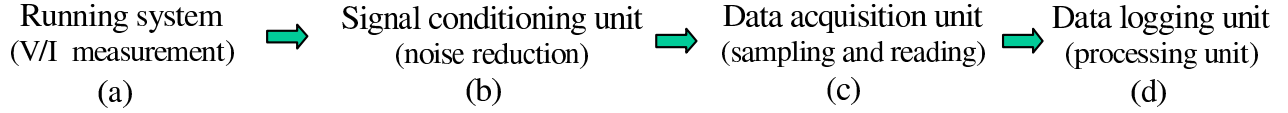
## 4. Implementation and Deployment: Methods and Experience

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, we conducted live-system physical power measurements.

### 4.1 Implementation

We use the Intel PIN system [10] 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, 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 [10], 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].

The implementation of our prototype RDO system can be highlighted as follows. At the start, RDO instruments all function calls in the program, and all loops in the main() function, in order to monitor and identify frequently executed code regions. 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 program execution. On the other hand, if a code region is found to be CPU bound, no DVFS instructions will



**Figure 4: Processor power measurement setup. This setup consists of four components.**

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 dynamically generated and all loops inside this function will be identified and instrumented. And the process continues [15].

## 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, which is shown in Figure 4a. 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. Figure 4 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. We use the output sense resistors of the main voltage regulator to measure the current going to the CPU, and use the bulk capacitor to measure the CPU voltage.

**Signal conditioning unit:** This unit shown in Figure 4b reduces the measurement noise to get more accurate readings. Measurement noise is inevitable because of the noise sources like the CPU board itself. Because noise typically has much higher frequency than the measured signals, we use a two-layer low-pass filter to reduce the measurement noise.

**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 [12], which has a maximum sampling rate of 1.2M/s (aggregate),

as shown in Figure 4c. We set a sampling rate of 200K/s for each channel ( $5\mu s$  sample length), which is more than adequate for our reads.

**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. 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

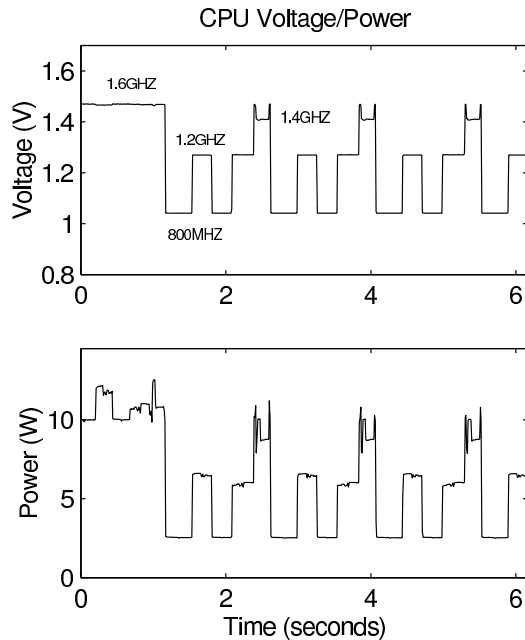
For all experiments, we chose a performance loss target  $P_{loss}$  of 5%. (If a larger  $P_{loss}$  were used, the resulting frequency settings would be lower, allowing more aggressive energy savings. Conversely, a smaller  $P_{loss}$  would lead to higher and more conservative DVFS settings.) Since the voltage transition time between different SpeedSteps is about  $100\mu s - 500\mu s$  for our machine [7], we set the long-running threshold for a code region to be 1.5ms (or 2.4M cycles for a 1.6GHz processor) to make it at least 3 times bigger than the voltage transition time.

For evaluation, we use all SPEC2K FP and SPEC2K INT benchmarks. Since previous static compiler DVFS work in [8] used SPEC95 FP benchmarks, we also include them in our benchmark suites. In addition, we include some Olden benchmarks [4] as they are popular integer benchmarks to study program memory behavior. 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.

As an illustrative example, Table 1 and Figure 5 show some program information and results obtained from the RDO system for one SPEC benchmark 173.applu, a program for elliptic partial differential equations. In the table, we give the total number of hot code regions in the program and the total number of DVFS regions identified. For each DVFS code region, we show the total number of  $\mu ops$  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  $\mu ops$ ), and the obtained DVFS settings. In Figure 5, we show a part of the CPU voltage and power trace 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

**Table 1: Information and DVFS settings for SPEC benchmark 173.applu.**Average number is for per 1M  $\mu ops$  retired

total hot regions	total DVFS regions	region name	total $\mu ops$	Average L2 cache misses	Average Memory trans	Average Instruction retired	DVFS setting (Hz)
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

**Figure 5: A partial trace of the CPU voltage and power for SPEC benchmark 173.applu running with the RDO**

the program phase changes. Specifically, as the program execution enters into the code region *jaclD()*, the CPU clock frequency (and voltage) is scaled from the default 1.6GHz to the selected DVFS setting 0.8GHz. Then it switches to 1.2GHz when the program enters into the next code region *blts()*; and so on. 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 later, the DVFS optimization applied to the code regions in 173.applu has led to considerable energy savings ( $\sim 35\%$ ) with little performance loss ( $\sim 5\%$ ).

## 5.1 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 2. So, to isolate the contribution of the DVFS optimization, we will 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 volt-

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

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%

age 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%.)

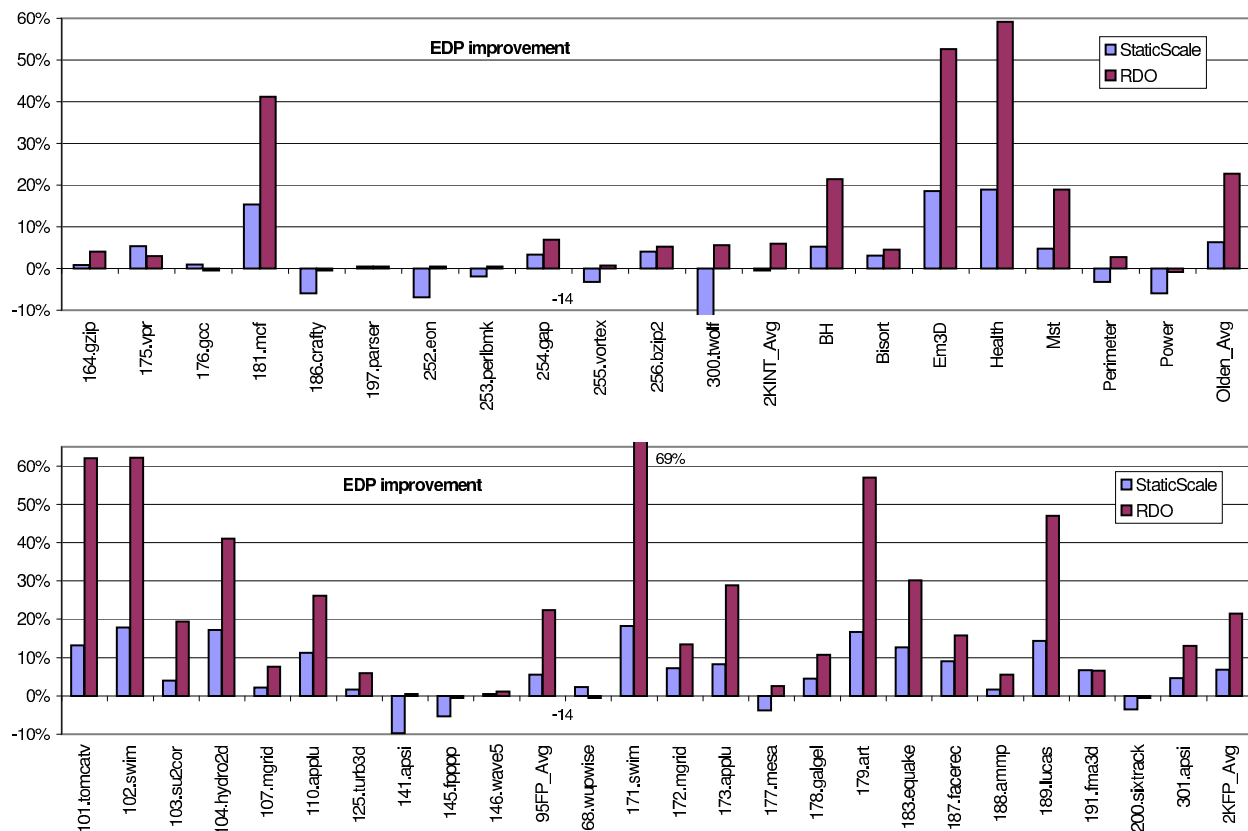
Figure 6 shows the energy delay product (EDP) improvement results for all our benchmarks. Note that these results have taken into account all DVFS optimization overhead, such as the time cost to check memory boundedness of a code region. For convenience, we refer to the result from our runtime DVFS optimizer 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. Second, the energy and performance results for individual benchmarks in each benchmark suite vary significantly (from  $-1\%$  to 70%). This is of course due to the fact that individual applications vary widely in their proportion of memory boundedness. In particular, the SPEC2K integer programs are almost all CPU-bound (except for 181.mcf) and hence nearly immune to our attempted optimizations.

The average results for each benchmark suite are summarized in Table 2. 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 than the *StaticScale* EDP improvement. (For those interested, more experimental results, including those for the basic O-PIN overhead, can be found in [15].)

Overall, the results in Figures 6 and Table 2 show the proposed technique does quite well in addressing the energy and performance control problem in microprocessors. We attribute these promising results to the efficiency of our design and to the advantages of the dynamic compiler driven approach.

## 5.2 Micro-architectural suggestions

While the experimental results 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 [6]. This would make the DVFS computation easier and more accurate. Another



**Figure 6: Energy-delay product (EDP) improvement for SPEC2K INT benchmarks (top row, left side), Olden benchmarks (top row, right side), SPEC95 FP benchmarks (bottom row, left side) and SPEC2K FP benchmarks (bottom row, right side). We show results for both our runtime DVFS optimizer (*RDO*) and the *StaticScale* voltage scaling.**

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. Conclusions

Our work has found that dynamic compiler driven voltage and frequency control is very effective for high-end microprocessors. In addition, the design framework and methodology described in this paper is generalizable for other emerging microprocessor issues (such as di/dt and thermal control) and other environments (such as Java Virtual Machine systems).

Overall, this work shows that high-level dynamic compiler does have an important role to play in microprocessor power and performance management. In particular, the orthogonal approach and advantages of the proposed techniques make them an effective complement to existing techniques, and allow us to work toward a multi-layer (software and hardware) collaborative power management scheme in the future. We feel these techniques and schemes offer great promise for adaptive run-time power and performance management in future microprocessors.

## 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.

## 7. 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] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, March 2003.
- [4] 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.
- [5] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of ISCA*, June 1997.
- [6] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of ISCA*, May 2002.
- [7] S. Gochman, R. Ronen, et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 07(2), 2003.
- [8] 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.
- [9] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of ISCA*, June 2003.
- [10] 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.
- [11] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *In Workshop on Complexity Effective Design, Vancouver, Canada, June 2000.*, June 2000.
- [12] National Instruments. *Data Acquisition (DAQ) Hardware*, <http://www.ni.com/dataacquisition>, 2005.
- [13] G. Semeraro, D.H. Albonese, 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.
- [14] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of CASE'02*, Oct 2002.
- [15] Q. Wu, V.J. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D.W. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th MICRO*, Nov 2005.
- [16] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proc. of 2003 PLDI*, June 2003.