A New Way Of Estimating Compute Boundedness And Its Application To Dynamic Voltage Scaling

Vasanth Venkatachalam*, Michael Franz
Donald Bren School of Information and Computer Science,
University of California, Irvine, Irvine, CA, U.S.A.
E-mail: {vvenkata,franz}@uci.edu
*Corresponding author

Christian W. Probst
Informatics and Mathematical Modelling
Technical University of Denmark, 2800 Kongens Lyngby, Denmark
E-mail: probst@imm.dtu.dk

Abstract: Many recent dynamic voltage-scaling (DVS) algorithms use hardware events (such as cache misses, memory bus transactions, or instruction execution rates) as the basis for deciding how much a program region can be slowed down with acceptable performance loss. Although these approaches result in power savings, the hardware events measured are at best *indirectly* related to execution time and clock frequency. We propose a new metric for evaluating the performance loss caused by DVS, a metric that is *logically* related to clock frequency and execution time, namely the *percentage drop in cycles*. Further, we show that we can predict with high accuracy the execution time of a code region at *any* clock frequency after measuring the total number of cycles spent in that region for two clock frequencies—the maximum and the second highest clock frequency. Measurements using several real-world applications show that this "two-point" model predicts execution times with an accuracy that is greater than 95% in many cases. This result can be used to develop low-overhead DVS algorithms that are more system-aware than many of the current algorithms, which rely on measuring indirect effects.

Keywords: dynamic voltage scaling; performance estimation; virtual machines.

Reference to this paper should be made as follows: Venkatachalam, V., Franz, M. and Probst, C.W. (2006) 'A New Way Of Estimating Compute Boundedness And Its Application To Dynamic Voltage Scaling', Int. J. Embedded Systems, Vol. 1, Nos. 1/2/3, pp.64–74.

Biographical notes: Vasanth Venkatachalam is a PhD candidate in the Donald Bren School of Information and Computer Science at the University of California, Irvine. His research focuses on compilers and operating systems for low power computing. He has bachelors degrees in mathematics and philosophy from Wesleyan University, and a masters degree in Computer Science from the University of California, Irvine.

Michael Franz is an Associate Professor in the Donald Bren School of Information and Computer Science at the University of California, Irvine. His research focuses on security and efficiency aspects of mobile-code systems, on virtual machine technology in general, compiling for low-power usage, code compression, and programming languages and architectures for component-based software construction. He received a Dr. sc. techn. degree in Computer Science and a Dipl. Informatik-Ing. ETH degree, both from the Swiss Federal Institute of Technology, ETH Zurich.

Christian W. Probst is an Assistant Professor in the Department for Informatics and Mathematical Modelling at the Technical University of Denmark. His research interests include language-based security, virtual execution environments, static analyses and optimising compilers, and distributed systems. He received Dr. Ing. and Diploma degrees in Computer Science from Saarland University, Germany.

1 Introduction

Dynamic voltage scaling (DVS) remains one of the most popular and effective techniques for reducing the power consumption of microprocessors. This technique, which can be controlled from either of the hardware, operating system or compiler levels, works by reducing a processor's clock frequency and voltage in lockstep (Venkatachalam and Franz, 2005). Because CPU power dissipation is quadratic with respect to supply voltage and linear with respect to clock frequency, DVS should, under ideal conditions, reduce a processor's instantaneous power dissipation cubically.

However, when the processor is slowed down, program execution will slow down as well. Thus DVS is only effective in code regions that can be slowed down without unacceptably increasing a program's total execution time (or a user's experience of it). Thus the correct question to ask is: "What is the relationship between a program region's execution time and the clock frequency at which it is run"?

Due to the effects of processor stalls, memory or disk intensive programs tend to have less performance loss than compute-intensive programs when run at a lower clock frequency. This makes them ideal candidates for DVS algorithms. Thus several recent DVS approaches (Choi et al., 2004a,b,c,d; Li et al., 2003; Marculescu, 2000; Poellabauer et al., 2005; Singleton et al., 2005; Weissel and Bellosa, 2002; Wu et al., 2005) aim to detect memory boundedness, or how often the processor is stalling due to memory requests. Similarly, some approaches (Hsu and Feng, 2004, 2005) aim to infer memory boundedness from CPU-boundedness. Although each of these approaches may have good results, the key challenge they face is how to detect processor stalls. Many of the architectures that are widely used (e.g., Pentium M, Pentium III) lack hardware counter events for measuring cycles spent during processor stalls. Thus these approaches rely on *indirect* information that is provided by the hardware-event counters available, information about hardware events such as cache misses (Choi et al., 2004b,c,d; Kondo and Nakamura, 2004; Li et al., 2003; Marculescu, 2000; Poellabauer et al., 2005; Singleton et al., 2005), memory bus transactions (Wu et al., 2005), memory requests per cycle (Weissel and Bellosa, 2002), or instruction execution rates (Hsu and Feng, 2004, 2005).

The problem is that the relationship between the events measured and the execution time of a program at a given clock frequency is at best indirect. Although the measurements may be *statistically* related to processor stalls, they do not *imply* processor stalls and it is difficult to draw a correlation between them and program execution time, which depends on *everything* that is happening in a system. Moreover, not all processors provide the means to measure these events, and devices lacking this ability can not be supported by the approaches mentioned above.

To overcome these difficulties, we present a new methodology to understand the relationship between performance loss and clock frequency. The key observation is that the execution time of a code region under a fixed clock frequency is the ratio of (a) the total number of elapsed clock cycles during the region's execution, and (b) the clock frequency (cycles/s) at which the region was executed. Thus if a region is run at a lower clock frequency and its execution time does not increase as much as one would expect—which is the case for memory bounded code regions—then executing the region takes fewer clock cycles at the lower frequency than at the higher clock frequency.

Thus we propose to use the *percentage decrease in clock cycles* as the measure of how compute intensive a code region is. Further, we show that just by knowing the total cycles it takes to run a program at the maximum clock frequency, and the total cycles it takes to run it at *a single notch* below the maximum clock frequency, we can predict with high accuracy the program's execution time at *any* lower clock frequency. This result can be used to develop low-overhead DVS algorithms, which do not depend on platform-specific event counters.

In sum, we make the following contributions:

- We introduce a new measure of compute-boundedness which is based on *logical* foundations, namely the percentage decrease in cycles.
- We show that we can estimate, with high accuracy, the execution time of a code region at an arbitrary clock frequency simply by running the region at the maximum clock frequency and at one notch below the maximum clock frequency, and extrapolating from the difference in execution cycles.
- We show how this result can be used to develop low-overhead DVS algorithms that do not rely on hardware event counters. We evaluate our technique on a wide variety of real-world applications and show what percentage of their total runtime is spent in processor stalls. This can, among other things, be used to focus the development of power-management techniques toward those benchmarks that are most amenable to DVS.

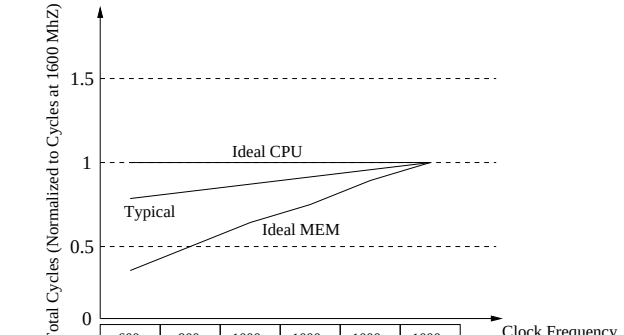
The rest of this paper is structured as follows. Section 2 provides the theoretical framework of our model by formalising the decrease in clock cycles for an arbitrary code region that is slowed down. Section 3, Section 4 and Section 5 discuss the implementation of our model, the methodology used to validate our model, and provide results attesting to the accuracy of this model for a wide variety of applications, respectively. Section 6 discusses how this model can be applied to dynamic voltage scaling. Section 7 discusses the difference between our model and related work, and Section 8 summarises our conclusions and describes avenues for future work.

2 Theoretical Foundation

In this section, we provide a formal model for understanding how much a program's total clock cycles decrease when the program is run at a lower clock frequency. Using this result, we show how we can estimate the CPU (or memory or I/O) boundedness of a program region by simply slowing the region down to *any* lower clock frequency and extrapolating from the difference in total execution cycles.

To put all these ideas in context, we first describe two theoretical extremes, an ideal, CPU-bound program that contains only computations (with no memory accesses), and an ideal, memory-bound program that contains only memory accesses (no computations). As Figure 1 shows, slowing down an ideal, CPU-bound program (top curve) does not change the total number of clock cycles needed for executing the program. Instead, the total cycles will remain the same because computations (e.g., addition or multiplication) generally require a fixed number of clock cycles. As a result, the execution time of the CPU-bound program increases inversely with respect to the decreasing clock frequency. (Cutting the clock frequency in half causes the program to run twice as long.)

In contrast, when an ideal, memory-bound program (bottom curve) is slowed down, the total number of cycles needed for executing the program decreases in direct proportion to the decrease in clock frequency. This is because the CPU is not performing any useful work during this ideal memory-bounded program, but waits. As a result, slowing down the CPU does not affect the amount of time it takes to run the program. This is why the ideal memory-bounded program does not suffer any performance loss when slowed down, in contrast to the ideal, CPU-bounded program, which does suffer performance loss.



	600 MhZ	800 MhZ	1000 MhZ	1000 MhZ	1000 MhZ	1000 MhZ
Ideal CPU	1	1	1	1	1	1
Ideal MEM	0.375	0.5	0.625	0.75	0.875	1
Clock Cycles Normalized To Cycles At 1600 MhZ						

Figure 1: Execution cycles as a function of the clock frequency for three programs, an ideal memory bounded program, and ideal CPU bounded program, and a typical program in between these two theoretical extremes.

Most programs (middle curve) will exhibit behaviour in between these two theoretical extremes. They may contain regions where the processor is performing computations as well as regions where the processor is idle, waiting for memory, disk, or network accesses to complete. When such a program is run at a lower clock frequency, its total clock cycles will decrease because the regions where the processor is idle will need fewer clock cycles for execution. *This percentage decrease in total clock cycles is the primary (and most accurate) indicator of how CPU (memory) bounded a program region is.*

To quantify this decrease for an arbitrary program region, we consider two scenarios of program region execution that exhaust all possible cases. In the first scenario, computations do not overlap with memory accesses, while in the second scenario, they do.

2.1 Scenario 1: No Overlap Between Computations and Memory Accesses.

Scenario 1 assumes that either the processor or the main memory is executing at any given time. Figure 2 shows a simplified version of this scenario. The processor first performs some computations, then requests data from main memory, idles while waiting for the memory transaction to complete, and then continues performing more computations. Let *C1* mark the first computation phase, *I1* mark the idle phase, and *C2* mark the second computation phase. Then the total processor clock cycles for the execution of this program region is:

$$Cycles(total) = Cycles(C1) + Cycles(I1) + Cycles(C2) \quad (1)$$

Moreover, the total clock cycles that the processor spends idling (during *I1*) is the product of the clock frequency and the duration of phase *I1*. That is,

$$Cycles(I1) = T_{I1} f \quad (2)$$

Because memory is asynchronous with respect to the processor (under our assumptions), the duration of the idle period, T_{I1}, is constant across all clock frequencies. Thus the total idle cycles ($Cycles(I1)$) decrease as the frequency f is lowered. On the other hand, the total computation cycles ($Cycles(C1) + Cycles(C2)$) do not decrease since *C1* and *C2* are computational phases. As a consequence, the decrease in total execution cycles needed for executing the program region at a lower clock frequency f_{new} instead of the current clock frequency f_{old} is:

$$Cycles_{old} - Cycles_{new} = T_{I1}(f_{old} - f_{new}) \quad (3)$$

2.2 Scenario 2: Possible Overlap Between Computations And Memory Accesses

We now consider a slightly more complex scenario (Figure 3). As before, the processor first performs some computations and then issues a memory request. But this

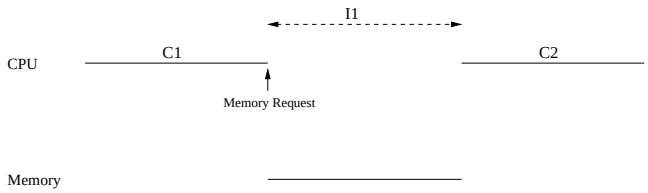


Figure 2: Processor and memory workload in scenario 1.

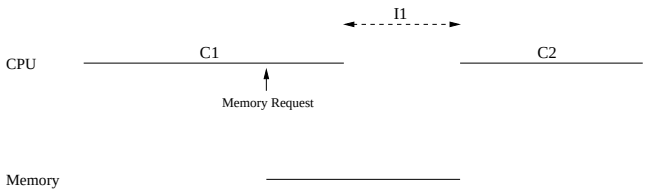


Figure 3: Processor and memory workload in scenario 2.

time, instead of idling during the entire memory transaction, the processor performs some computations and then idles. Finally, when the memory transaction is complete, the processor continues performing more computations.

As before, let *C1* stand for the first computation phase (which includes computations performed during the memory transaction), *I1* stand for the phase in which the processor is idle, and *C2* stand for the second computation phase (which is dependent on the memory transaction completing).

Now suppose the code region is run at a lower clock frequency than the current clock frequency. Then, just as in scenario 1, the total clock cycles spent in computation ($Cycles(C1) + Cycles(C2)$) will remain the same since computations require a fixed number of cycles independent of the clock frequency. And just as before, the total clock cycles spent idling will decrease. However, the amount by which the idle cycles decrease will be *greater* in this scenario than it was in the previous scenario. This is because the time that the processor spends idling (T_{I1}) is not fixed, but rather decreases as the clock frequency is lowered, since the processor spends more time executing the (slowed down) computations in *C1* while the memory is accessed. To formalise this reduction in idle time, let T_{I1}^{old} be the processor's idle time at clock frequency f_{old} and T_{I1}^{new} be the processor's idle time at clock frequency f_{new}. Then, the the difference in total clock cycles for executing the code region at the lower frequency f_{new} instead of the frequency f_{old} is:

$$Cycles_{old} - Cycles_{new} = T_{idle}^{old}(f_{old}) - T_{idle}^{new}(f_{new}) \quad (4)$$

Notice that the only difference between this and the previous scenario is that T_{idle}^{new} is less than T_{idle}^{old}. If these times were equal, we would be back in scenario 1.

Applying this scenario to architectures such as the Pentium 6 is problematic, since they are pipelined and allow multiple loads and stores to overlap. The performance counters on these architectures do not provide sufficient information to quantify exactly how large this overlap between CPU and memory activity is; that is, there is insufficient information to determine exactly how much T_{idle}^{new} differs from T_{idle}^{old}.

Therefore, we will spend the rest of this section discussing the implications of scenario 1 (Figure 2). We will show that scenario 1 implies a very powerful result for DVS algorithms. This result, and the assumptions underlying scenario 1, will be substantiated by our measurements in Section 5.

2.3 The Implications of Scenario 1

In scenario 1 (Section 2.1), processor and memory are asynchronous, and memory stall time is roughly independent of the processor clock frequency.[1]

Let T_{idle} be the total amount of time that the processor stalls, waiting for memory requests to complete. (This is a summation of all the idle periods that are interspersed with computational periods in the entire program region.) Then according to equations 4, the total decrease in cycles when executing the code region at a lower clock frequency f_{new} instead of the current clock frequency f_{old} is:

$$Cycles_{old} - Cycles_{new} = T_{idle} \cdot (f_{old} - f_{new}) \quad (5)$$

As a result, we can deduce the idle period T_{Idle} to be:

$$T_{idle} = \frac{Cycles_{old} - Cycles_{new}}{(f_{old} - f_{new})} \quad (6)$$

This means that the time that the processor spends idling can be estimated by running the entire program region at a slower clock frequency and recording the difference in execution cycles. Since the idle period T_{idle} is constant across all clock frequencies, it only needs to be computed once, and it does not matter how much we slowed down the original program region in order to compute this.

Now we use T_{idle} to estimate the execution time of the program at any clock frequency. As before, let f_{old} be the old clock frequency at which we ran the program region and f_{new} be the new, slowed down clock frequency. Let C_{old} and C_{new} be the total execution cycles at the old frequency f_{old} and the new frequency f_{new}, respectively. Then, by equations 6 and 5 the execution time of the program region under the new clock frequency f_{new} is:

$$\begin{aligned} ExecutionTime_{New} &= \frac{C_{new}}{f_{new}} \\ &= \frac{C_{old} - (C_{old} - C_{new})}{f_{new}} \\ &= \frac{C_{old} - T_{idle} \cdot (f_{old} - f_{new})}{f_{new}} \end{aligned} \quad (7)$$

This equation implies that the execution time under the new clock frequency f_{new} is a function of the cycles C_{old} under the old clock frequency and the idle time T_{Idle}, which

[1] We are well aware that in some architectures the memory bus clock frequency depends on the processor clock frequency. However, if we know the ratio between the two clock frequencies (which should be given in the processor manuals), our approach can be adapted.

Frequency [MHz]	600	800	1000	1200	1400	1600
Voltage [mV]	956	1036	1164	1276	1420	1484

Table 1: Frequencies and associated voltage levels of the 1.6 GHz Pentium M processor.

we computed knowing merely the total cycles C_{old} under the old clock frequency f_{old} and the total cycles C_{new} under the new clock frequency f_{new}. In other words, the total number of cycles it takes to run a program at two arbitrarily chosen clock frequencies is enough information to estimate the program's execution time for any other clock frequency.

In particular, running a program region at a clock frequency that is "one notch" lower than the clock frequency at which the region originally ran, can provide enough information (i.e., the cycle counts) to estimate the execution time of the region at *any* clock frequency..

3 Implementation

This section gives an overview of the experimental platform we used to validate our model.

3.1 Experimental Platform

Our experimental platform is a Dell Latitude D600 laptop featuring a 1.6 GHz Pentium M processor. The processor supports Intel Enhanced Speedstep Technology, which allows switching between multiple frequency settings on the fly and automatically adjusts the voltage with respect to the frequency. The supported frequencies and their accompanying voltage levels are listed in Table 1. We are using the Linux CPUFreq driver to adjust the frequency settings. However, we have written a system call that directly invokes this driver's internal frequency setting method so that we can avoid the overhead of its default mechanism, a communication interface via the `proc` file system.

3.2 Changes to Linux Kernel

We have extended the Linux 2.6 kernel with a high-level interface for monitoring hardware events including clock cycles with minimal invasiveness and distinguishing events occurring in userspace from events occurring in kernelspace, in addition to retrieving process-specific event counts.

To distinguish kernel mode events from user mode events, we have modified the Linux kernel to save and restore the event counts whenever a task switches to and from kernel mode, and to also update the total number of kernel events, context switches, and interrupts for the current process. Similarly, to retrieve process-specific event counts, we have modified the scheduler so that it saves the event counts for each task being switched out and restores the original event counts for each task being switched in.

3.3 Changes to the Java Virtual Machine

We decided to implement our model on top of the Java virtual machine because mobile code such as Java and .NET bytecode is already ubiquitous in a variety of devices including laptops, PDAs, and set-top boxes.

We have implemented our model inside version 1.6.0 of the Sun Hotspot Client Virtual Machine (SUN HotSpot, 2006). We previously implemented the same mechanism inside the Jikes Research Virtual Machine (Arnold et al., 2000), but migrated it to Hotspot for two reasons. First, Hotspot is a production-quality, industrial VM, whereas Jikes is a prototype, research VM with many missing features. By using Hotspot, we obtain results that are more realistic and have broader industrial applicability. Second, Hotspot can run programs using any of the Sun Java libraries, allowing us to experiment with a broader range of real-world applications than any other VM. This again enhances the applicability of our approach.

We have extended the Hotspot VM, allowing the interpreter and just-in-time compiler to instrument the entries and exits of methods. Our instrumentation can be used to start and stop hardware performance counters, sample any hardware counter events (including clock cycles), and switch the processor clock frequencies for different method invocations. All of the instrumentation can be turned on and off dynamically for specific methods, independent of any other methods.

We ran all benchmarks in Hotspot's default mixed-mode execution framework, because the interpreter-only option is prohibitively slow. We can safely run our experiments this way because Hotspot's re-compilation heuristic is not sensitive to the processor clock frequency, but only depends on method invocation counts and backward branch counts, meaning that the heuristic makes the same decisions regardless of the clock frequency.

4 Experiment Methodology

4.1 The Two Point Hypothesis

Recall our model of execution time in equation 7 from Section 2.3:

$$ExecutionTime_{New} = \frac{C_{old} - T_{idle} \cdot (f_{old} - f_{new})}{f_{new}}$$

Our hypothesis is that if we know the execution cycles for a program at the maximum clock frequency and at the clock frequency one notch below the maximum clock frequency, this information allows us to predict, with reasonable ac-

Benchmark	Description
SpecJVM '98	
_201_compress	Compression algorithm using modified Lempel-Ziv method.
_202_jess	An Expert Shell System.
_209_db	Performs multiple database functions on memory resident database.
_213_javac	Java compiler from JDK 1.2.
_222_mpegaudio	Decompresses MPEG-3 audio files.
_227_mtrt	A multithreaded raytracer that works on a scene depicting a dinosaur.
_228_jack	A Java parser generator based on the Purdue Compiler Construction Tool Set.
DaCapo	
antlr	Parses one or more grammar files and generates a parser and lexical analyzer for each.
bloat	Performs a number of optimisations and analyses on Java bytecode files.
chart	Uses JFreeChart to plot a number of complex line graphs and renders them as PDF.
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
hsqldb	Executes a JDBC-like in-memory benchmark against a model of a banking application.
jython	Inteprets a series of Python programs.
pmd	Analyzes a set of Java classes for a range of source code problems.
ps	Reads and interprets a PostScript file.
xalan	Transforms XML documents into HTML.
JavaGrande	
Search	Solves a game of connect-4 on a 6 x 7 board using a alpha-beta pruning technique.
Euler	Solves the time-dependent Euler equations.
Mol. Dynamics	A simulation of N particles interacting under a Lennard-Jones potential.
MonteCarlo	A financial simulation, using Monte Carlo techniques.
LUFact	Solves an N x N linear system using LU factorisation followed by a triangular solve.
SOR	Performs 100 iterations of successive over-relaxation on a NxN grid.
HeapSort	Sorts an array of N integers using heap sort.
Crypt	Performs IDEA encryption and decryption on an array of N bytes.
FFT	Performs a one-dimensional forward transform of N complex numbers.
Sparse	Sparse matrix multiplication, using an unstructured sparse matrix.

Table 2: A Description of the benchmarks used in this study.

curacy, the execution time of the program at every other clock frequency.

We tested this "two point hypothesis" on 26 real-world Java applications from the SpecJVM '98, Dacapo, and JavaGrande benchmark suites, taking as our two points 1.6 GhZ (the maximum clock frequency) and 1.4 GhZ (one notch below the maximum clock frequency). The benchmarks used are described in Table 2. We ran each benchmark to completion ten times over every supported clock frequency on our platform, flushing the hardware caches in between each run to ensure that all the runs begin from the same state. For each benchmark we obtained for each clock frequency

- The total execution cycles for that clock frequency.
- The percentage decrease in clock cycles compared to running at the maximum clock frequency.
- The actual execution time for that clock frequency.
- The estimated execution time for that clock frequency, according to our model under the two-point hypothesis. This estimated time was computed by plugging into equations 6 and 7 the total cycles for 1.6 GHz ($Cycles_{1.6GHz}$) and for 1.4 GHz ($Cycles_{1.4GHz}$).
- The estimation error, given as a percentage. The errors were originally positive or negative depending on whether our model overestimated or underestimated the actual execution time. However, we have displayed here the absolute values of those errors.

5 Results

We illustrate our results with two different types of plots. First, Figure 4 is a histogram of the estimation errors for all benchmarks over all clock frequencies. On the x-axis is the clock frequency in GHz, and on the y-axis is the estimation error, given as a percentage. The datapoints correspond to the estimation errors for different benchmarks. The errors displayed for the reference points 1.4GhZ and 1.6GhZ will always be zero since these two points were initially chosen to extrapolate the execution times for all other clock frequencies. We are only interested in the datapoints plotted for the other clock frequencies (0.6GhZ up to 1.2GhZ). Of

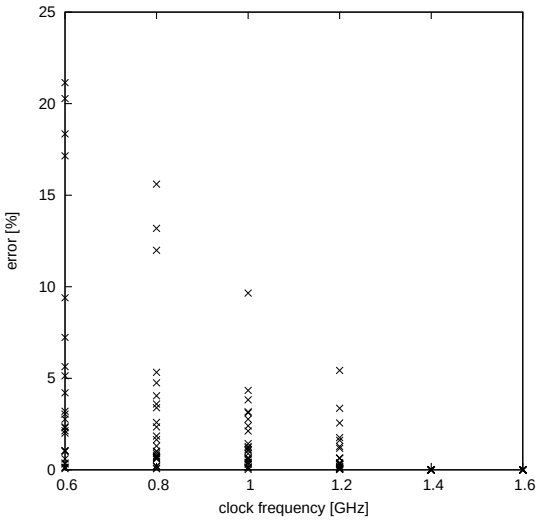


Figure 4: A histogram of the estimation error of our model for all benchmarks over all clock frequencies.

these 104 points, the majority of them (all but 8) lie below the 10% error mark. In fact, most of the points (all but 13) lie at or below the 5% error mark, and there are many of the remaining that are close to 1% error. A small number of points are higher on the error axis: two are between 10% and 15% error, three are between 15% and 20% error, and two are slightly above 20% error, but none of the points exceeds 25% error. This suggests that our model has an estimation accuracy of above 90-95% for most of the datapoints, and 75-80% accuracy for a small number of datapoints.

To more closely see what gives rise to these results, we present a second set of plots. Figure 5, Figure 6, and Figure 7 display the detailed results of running each of the benchmarks over all possible clock frequencies.[2] In each figure, the x-axis is the clock frequency in GhZ, the y-axis on the left denotes the execution time in seconds, and the y-axis on the right denotes the percentage decrease in clock cycles. Each figure displays three different plots as function of the clock frequency, namely the actual execution time, the estimated execution time, and the percentage drop in cycles.

There are several conclusions that can be drawn from the benchmarks. First, there is a wide variation in how much performance loss different benchmarks exhibit when run at lower clock frequencies, and there is also a correlation between this performance loss and the decrease in total cycles. On the one hand, the benchmarks *jython*, *ps*, *mpegaudio*, *crypto*, and *moldyn* are highly CPU-intensive. The percentage drop in clock cycles for these benchmarks is very small (less than 5%) and the overhead of slowing these benchmarks down is very large. For example, for *ps*, the execution time increases by 163% when the benchmark is run at the lowest clock frequency. Thus these benchmarks should not be considered for DVS. On the other hand, the benchmarks *antlr*, *db*, *jack*, *mtrt*, *euler*, *fft*, *flufact*, *heapsort*, *montecarlo*, and *sparsematmult* fall into the category of being memory or disk bound. They exhibit significantly less performance loss and a higher decrease in cycles when run at lower clock frequencies. For example, the *sparsematmult* benchmark has roughly a 3.5% performance loss and a 61% drop in cycles when run at the lowest clock frequency.

Second, in most of the graphs, the percentage drop in cycles increases almost linearly as the clock frequency is lowered from 1.6 GhZ to 0.6 GhZ. This suggests that the total processor stall time is roughly the same regardless of clock frequency, implying that scenario 1 (Figure 2), rather than scenario 2 (Figure 3), holds for most of these benchmarks. This supports our model. For two of the graphs (*DaCapo jython* and *Javagrande moldyn*) the drop in clock cycles appears erratic, but the drop is so small (less than 5%) that we can safely ignore it. (The fluctuations are most likely caused by measurement noise.)

Third, for most of the benchmarks, our estimate of execution time is very close to the actual execution time for the different clock frequencies. However, there are a few cases where the error is 15-20%, namely for the benchmarks *flufact*, *montecarlo*, *sor*, *fft*, and *mtrt*. For the first four of these benchmarks there is a pattern, which is most easily observable in the graph of *sor*. The drop in cycles first increases linearly as the clock frequency is lowered, and our estimated execution time remains close to the actual execution time at these points. However, at a certain point, the slope of this curve abruptly changes, becoming less steep, and exactly at that point, our estimated execution time diverges noticeably from the actual execution time. This happens because the number of clock cycles spent in memory stalls decreases as the clock frequency is decreased. However, there will be a point of diminishing returns where the clock cycle length is so long that decreasing the clock frequency will not decrease the total cycles any further because all of the memory stall slack has been used up. Although we may not be able to observe this point—there are limits to how low we can set the clock frequency—the trend is for applications to gradually converge to that point. We see this in the graph by noticing the "drop in cycles" curve becoming less and less steep. However, because our model is based on a simple linear extrapolation, it is not sensitive to this. This explains the larger error numbers (15%-20%) we see for the four benchmarks above (*flufact*, *sor*, *montecarlo*, *fft*).

The *mtrt* benchmark displays a different anomaly—the drop in clock cycles is initially very low as the processor clock frequency is switched from 1.6GhZ to 1.4GhZ, but below 1.4GhZ it suddenly becomes higher, causing our model to overestimate the execution time by around 20%. Nevertheless, the curve remains linear for frequencies below 1.4GhZ. We are currently studying this benchmark further to see what could be causing this early change in slope.

Appendix A contains a table of the numbers used to gen-

[2]To save space, we only show 6 plots for each benchmark suite. However, the appendix contains data for all benchmarks.

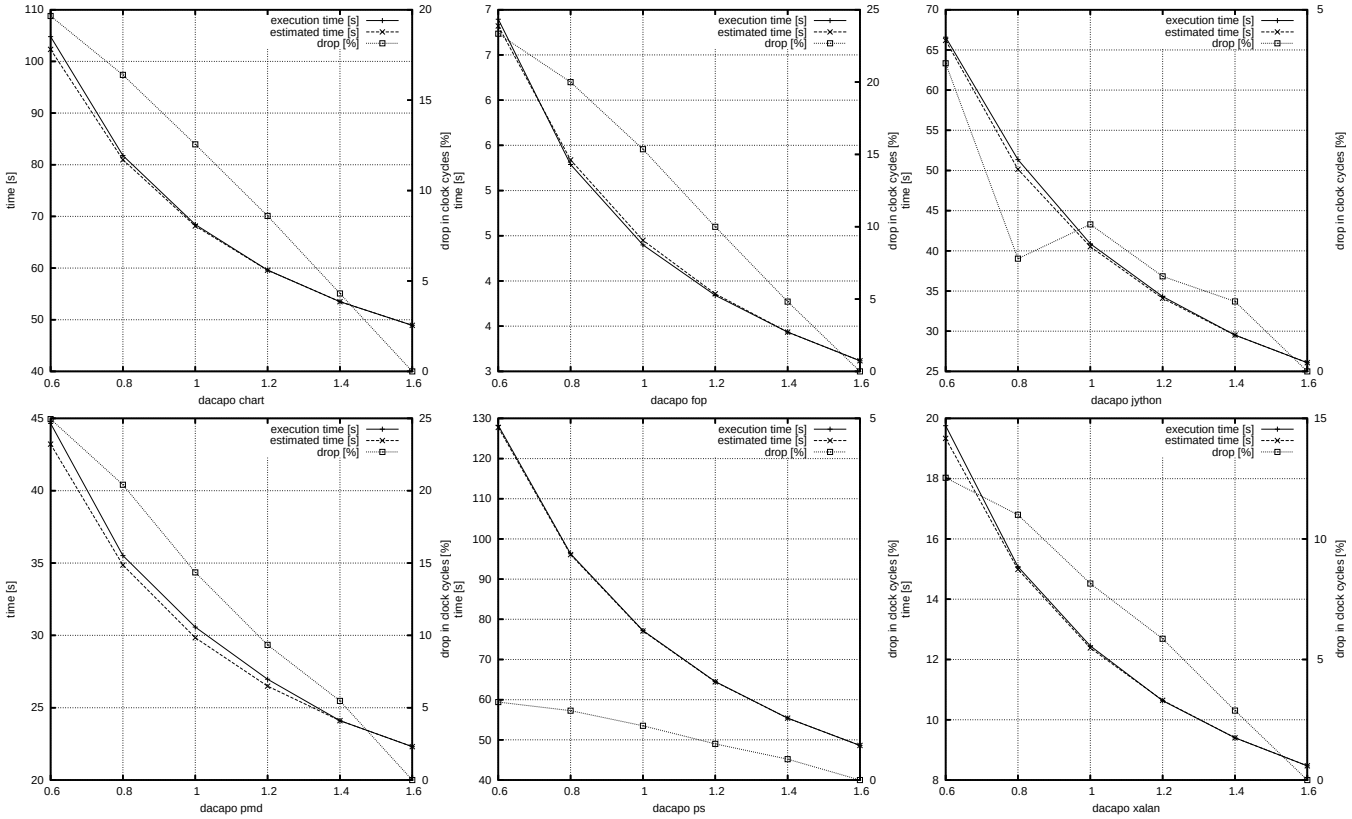


Figure 5: Results for representative benchmarks from the DaCapo suite.

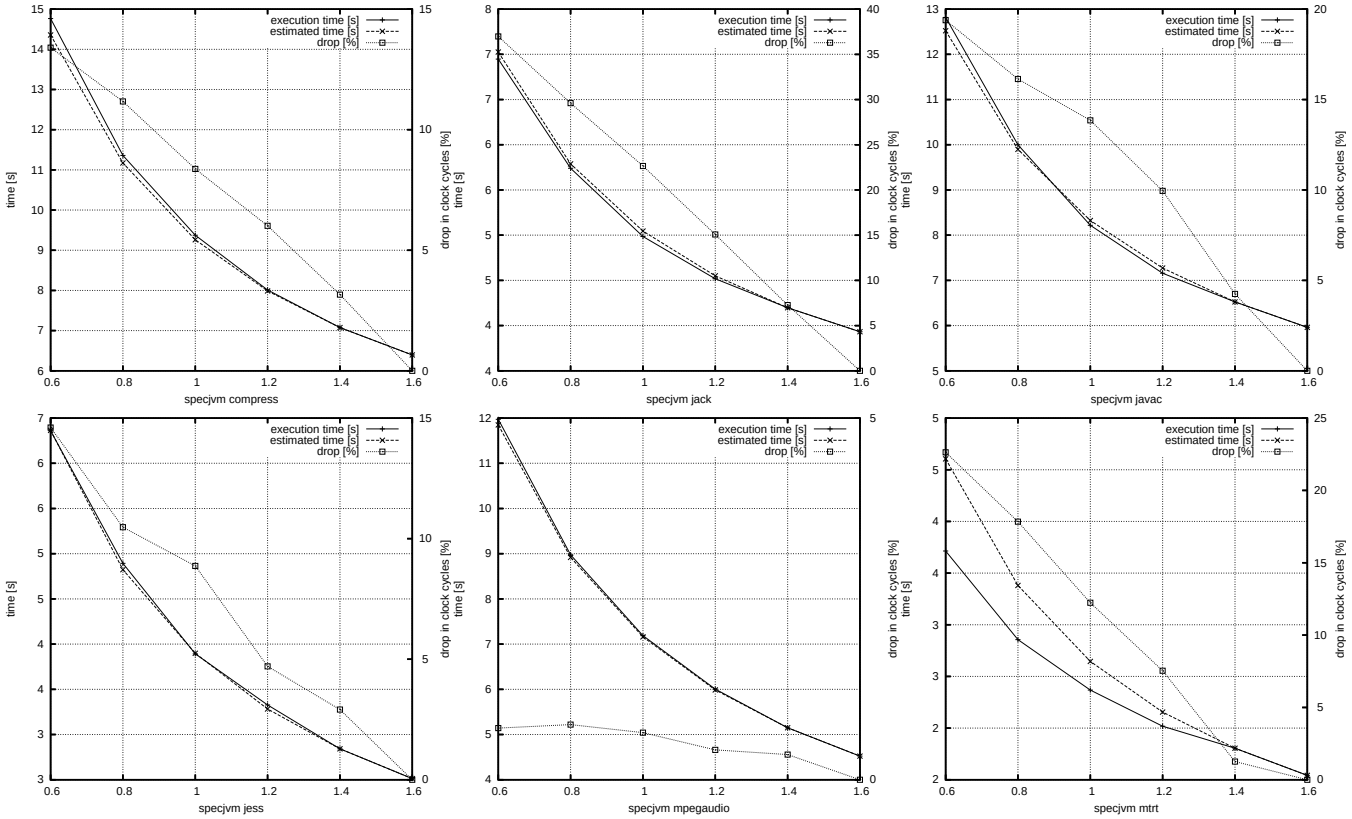


Figure 6: Results for representative benchmarks from the Spec JVM 98 suite.

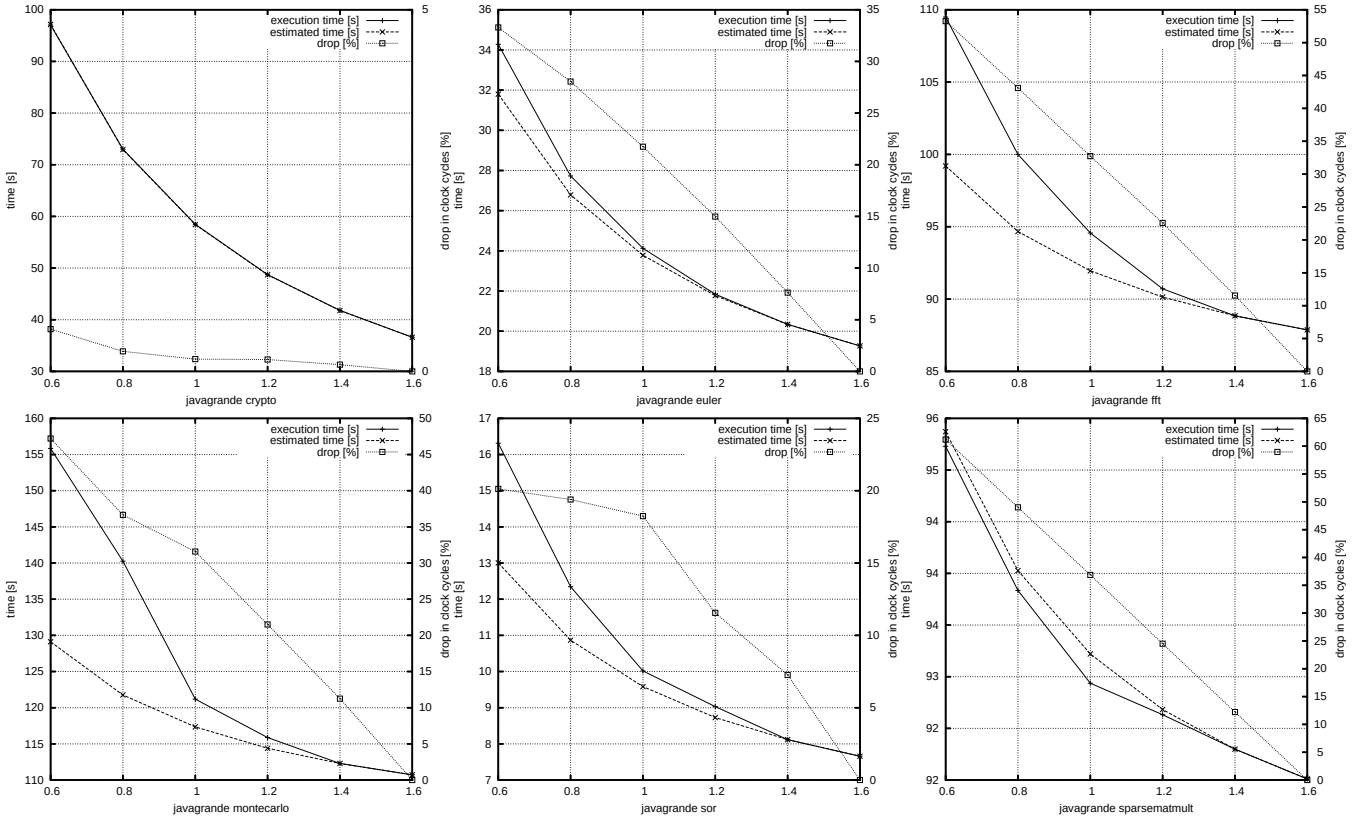


Figure 7: Results for representative benchmarks from the JavaGrande suite.

erate these graphs, as well as the numbers for the benchmarks not shown as graphs.

6 Applications Of Our Model

There are many possible applications for this mechanism. For example, embedded systems can use this model to speed up the offline experiments used to predict the execution times of specialised programs. Another application would be to use this model at runtime in a DVS algorithm. One of the goals of a DVS algorithm is to find the optimal clock frequency f_{new} that keeps the execution time of the program region from increasing more than, say N% of the original execution time. Using equation 7, we can model this constraint as

$$\begin{aligned} ExecutionTime_{New} &= \frac{C_{old} - T_{idle}(f_{old} - f_{new})}{f_{new}} \\ &< (\frac{C_{old}}{f_{old}}) + (\frac{N}{100})(\frac{C_{old}}{f_{old}}) \end{aligned} \quad (8)$$

Solving this equation for F_{new} we get:

$$f_{new} >= \frac{C_{old} - T_{idle} f_{old}}{\frac{(100+N)\cdot C_{old}}{100\cdot f_{old}} - T_{idle}} \quad (9)$$

Then the goal of a DVS heuristic would be to find the lowest clock frequency f_{new} that meets the above constraint. The different parameters in equation 9 can be estimated at runtime (by an OS or dynamic compiler) using the equations just provided. For example, at method level one would measure a method's execution cycles C_{old} under the current clock frequency f_{old}. Once a stable reading has been obtained, the clock frequency for the method is changed for the next invocations. To avoid performance loss, the frequency is reduced by just one notch below the previous clock frequency. After the method has run enough times to obtain stable readings, we measure the difference in execution cycles with respect to the previous invocation, and estimate T_{idle} using equation 6. This provides all the parameters (C_{old}, f_{old}, T_{idle}) needed to use equation 9 to determine the appropriate clock frequency for the method. Note that this approach is not restricted to method-level, or compiler-based DVS. It can be applied at other granularities, e.g., at the OS-level, where entire tasks could be slowed down using this approach.

The advantages of this approach are:

- There is a direct relationship between clock cycles, clock frequency, and execution time. This allows for a more accurate measure of how compute-intensive a code region is, and a more accurate computation of the correct clock frequency.

- The cycle counts encapsulate all the "hard-to-measure" system effects (i.e., disk accesses, network accesses, multiple cache misses) that could give rise to processor stalls, thus providing more complete in-

formation for making DVS decisions than any single hardware counter event.

- Because the clock frequency is only lowered a single notch below the top clock frequency, and this is only done once for the sake of measurement, there will be little performance loss.

- Clock cycles can be easily measured on most devices, allowing for portability even if hardware event counters fall out of fashion.

7 Related Work

There is a wealth of literature on DVS algorithms DVS (Azevedo et al., 2002; Choi et al., 2004b; Dudani et al., 2002; Flautner et al., 2001; Govil et al., 1995; Gruian, 2001; Hsu and Feng, 2004; Hsu and Kremer, 2003; Kondo and Nakamura, 2004; Pillai and Shin, 2001; Saputra et al., 2002; Stanley-Marbell et al., 2002; Weiser et al., 1994). Due to limited space, we will only give a few representative examples of the kinds of approaches that our mechanism can be used to extend, namely those approaches that attempt to extrapolate the memory boundedness of programs from information provided by hardware event counters (Choi et al., 2004b,c,d; Kondo and Nakamura, 2004; Li et al., 2003; Marculescu, 2000; Poellabauer et al., 2005; Singleton et al., 2005; Stanley-Marbell et al., 2002; Wu et al., 2005).

Marculescu (2000) was one of the earliest to propose using cache misses to drive dynamic voltage scaling. The main idea is that between the time when a cache miss is detected and the time it is resolved, the CPU activity can be divided into an independent and a dependent phase. The independent phase consists of instructions that can be executed while the miss is still being resolved. The dependent phase consists of instructions that have to wait until the miss is resolved. The CPU is slowed down immediately after the miss is detected, so that its workload during the independent phase finishes exactly when the miss is resolved.

Kondo and Nakamura (2004) propose an interval-based approach driven by cache misses. The heuristic periodically calculates the number of outstanding cache misses and increments one of three counters depending on whether the number of outstanding misses is zero, one, or greater than one. The cost function expresses the memory boundedness of the code as a weighted sum of the three counters. In particular, the third counter, which is incremented each time there are multiple outstanding misses, receives the heaviest weight. At fixed length intervals, the heuristic compares the memory boundedness so computed, with respect to an upper and lower threshold. If the memory boundedness is greater than the upper threshold, then the heuristic decreases the frequency and voltage by one setting; otherwise, if the memory boundedness is below the lower threshold, then it increases the frequency and voltage settings by one unit.

Weissel and Bellosa (2002) propose a heuristic that monitors the rates of different hardware events (i.e., cache misses) and attributes these rates to different processes that are executing. At each context switch of the OS scheduler, this heuristic sets the clock frequency for the process being switched in based on its previously measured event rates. To do this, it refers to a table that assigns clock frequencies based on event rates and performance loss thresholds. Weissel and Bellosa construct this table using exhaustive offline experiments where they measure the lowest clock frequency that will allow a program to satisfy a performance loss threshold under different combinations of event rates.

Wu et al. (2005) have developed a memory-aware DVS algorithm that can be used inside a dynamic compiler. Their cost model is based on the analytical model described in (Xie et al., 2003, 2004). The main idea is to estimate the scaling factor based on the fraction of time that the processor is stalled. The model extrapolates this information from three hardware counter events, memory-bus transactions, FP/INT instructions retired, and micro-ops retired. The values of the three main terms in their cost model (the optimal scaling factor, the total time that the memory is busy, the total time that CPU and memory are both busy) depend on platform-specific coefficients that are estimated using offline simulations.

Poellabauer et al. (2005) attempt to divide a program's execution time into two parts—the time spent in computations and the time spent in memory accesses. To estimate memory boundedness they propose a new metric, which they call *memory access rate*, which quantifies the average rate at which cache misses are occurring per instruction executed. They use performance counters to measure cache misses and instructions executed. To determine how to slow down the processor on the basis of these measurements, they construct a table that maps these memory access rates to matrices of scaling factors. Their heuristic consults this precomputed table at runtime.

Choi et al. (2004b) use the Intel XScale processor's performance counters to determine how much of a program's execution time is spent on-chip versus off-chip. Under this cost model, this reduces to estimating the average cycles for an on-chip instruction. For specific benchmarks, they find that this latter quantity is linear with respect to the average CPU stall cycles per instruction. Reasoning that stall cycles increase with respect to cache misses, they construct a table that associates cache miss ranges with stall cycles. Their DVS heuristic consults this table to choose the correct clock frequency.

Hsu and Feng (2004, 2005) propose an interval-based DVS algorithm that measures CPU-boundedness. According to their model, the execution time for running a program at a given clock frequency is proportional to a constant β, which is a measure of how computation-intensive the program is. Their DVS heuristic estimates β at runtime, using a regression method over past instruction exe-

cution rates.

All of the above works attempt to use specific hardware events as indicators for estimating a program's execution time at a given clock frequency. In the case of Kondo and Nakamura (2004), Poellabauer et al. (2005), Marculescu (2000), and Choi et al. (2004b) the events are cache misses. In Weissel and Bellosa (2002) the events are memory requests per cycles and instructions per cycle. In Wu et al. (2005), the events are memory-bus transactions and micro-ops retired. In Hsu and Feng (2004, 2005), the events are instrution execution rates. These approaches involve a level of indirection, because they are based on statistical correlations between the events in question (e.g., cache misses, instruction execution rates) and execution time and clock frequency. On the other hand, execution time, clock frequency and clock cycles are *logically* related. Thus our metric, *the percentage drop in cycles*, is a more appropriate metric for assessing how much a code region will slow down when it is run at a lower clock frequency.

8 Conclusions and Future Work

We have presented a new way of understanding and predicting how compute-intensive a code region is for the purposes of DVS. Namely, the *percentage decrease in cycles* is the primary indicator of how much the execution time will increase when a code region is run at a lower clock frequency. It is the most fundamental measure of compute-boundedness because of the logical relationship between clock cycles, clock frequency, and execution time. We can estimate with high precision the execution time of a code region at *any* clock frequency just by running the region at the two highest clock frequencies, and extrapolating from the decrease in cycles. This result can be used to develop low-overhead DVS algorithms that are more system-aware than current approaches (Choi et al., 2004b,c,d; Hsu and Feng, 2004, 2005; Kondo and Nakamura, 2004; Li et al., 2003; Marculescu, 2000; Poellabauer et al., 2005; Singleton et al., 2005; Weissel and Bellosa, 2002; Wu et al., 2005), approaches that choose clock frequencies based on hardware events that may be at best indirectly related to execution time.

We are integrating our model into a DVS algorithm for the Sun Hotspot VM. So far we have collected detailed method-level execution traces of the benchmarks in this paper. While many of these benchmarks spend a significant amount of time in memory stalls, we find that most of these stalls are occurring at the loop granularity rather than the method granularity. Moreover, in the 3 or 4 benchmarks that contain opportunities for DVS on method level, we find that switching the clock frequency on every method entry and exit amounts to a lot of overhead. We are still investigating the right granularity at which to apply DVS.

Another avenue for future work involves extending this model. Although we can predict execution times with reasonable accuracy using a "two-point" approach, we can increase the accuracy by considering more points. This will allow us to better deal with special cases.

9 Acknowledgements

Parts of this effort have been sponsored by the National Science Foundation (NSF) under ITR grant CCR-0205712 and by the Office of Naval Research (ONR) under grant N00014-01-1-0854. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science Foundation, the Office of Naval Research, or any other agency of the U.S. Government. The authors also gratefully acknowledge gifts from Intel, Microsoft Research, and Sun Microsystems that partially supported this work.

The authors further thank Vivek Haldar and Andreas Gal for commenting on an early draft of this paper, and Andreas Gal also for helping us implement a lightweight interface to the Pentium 6 performance counters, which we used in our early investigations.

REFERENCES

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2000). Adaptive Optimization in the Jalapeno JVM: The Controller's Analytical Model. In *The 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*.

Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., and Nicolau, A. (2002). Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 168. IEEE Computer Society.

Choi, K., Lee, W., Soma, R., and Pedram, M. (2004a). Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *Proceedings of the International Conference on Computer Aided Design*.

Choi, K., Soma, R., and Pedram, M. (2004b). Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 174–179. ACM Press.

Choi, K., Soma, R., and Pedram, M. (2004c). 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 *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10004, Washington, DC, USA. IEEE Computer Society.

Choi, K., Soma, R., and Pedram, M. (2004d). Off-chip latency-driven dynamic voltage and frequency scaling for an mpeg decoding. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 544–549, New York, NY, USA. ACM Press.

Dudani, A., Mueller, F., and Zhu, Y. (2002). Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 213–222. ACM Press.

Flautner, K., Reinhardt, S., and Mudge, T. (2001). Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 260–271. ACM Press.

Govil, K., Chan, E., and Wasserman, H. (1995). Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, pages 13–25. ACM Press.

Gruian, F. (2001). Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 46–51. ACM Press.

Hsu, C. and Feng, W. (2004). Effective dynamic voltage scaling through cpu-boundedness detection. In *Workshop on Power Aware Computing Systems*.

Hsu, C.-H. and Feng, W.-C. (2005). A power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA.

Hsu, C.-H. and Kremer, U. (2003). The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–48. ACM Press.

Kondo, M. and Nakamura, H. (2004). Dynamic processor throttling for power efficient computations. In *Workshop on Power Aware Computing Systems*.

Li, H., Cher, C.-Y., Vijaykumar, T. N., and Roy, K. (2003). Vsv: L2-miss-driven variable supply-voltage scaling for low power. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 19, Washington, DC, USA.

Marculescu, D. (2000). On the use of microarchitecture-driven dynamic voltage scaling. In *Proceedings of the Workship on Complexity-Effective Design*.

Pillai, P. and Shin, K. G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18^{t}h ACM Symposium on Operating Systems Principles*, pages 89–102. ACM Press.

Poellabauer, C., Singleton, L., and Schwan, K. (2005). Feedback based dynamic voltage and frequency scaling for memory-bound real-time applications. In *IEEE Real Time and Embedded Technology and Applications Symposium*, pages 234–243.

Saputra, H., Kandemir, M., Vijaykrishnan, N., Irwin, M. J., Hu, J. S., Hsu, C.-H., and Kremer, U. (2002). Energy-conscious compilation based on voltage scaling. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 2–11.

Singleton, L., Poellabauer, C., and Schwan, K. (2005). Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling. In *Proceedings of the 12th Annual Multimedia Computing and Networking Conference*.

Stanley-Marbell, P., Hsiao, M., and Kremer, U. (2002). A Hardware Architecture for Dynamic Performance and Energy Adaptation. In *Proceedings of the Workshop on Power-Aware Computer Systems*, pages 33–52.

SUN HotSpot (2006). SUN HotSpot's Homepage `http://java.sun.com/products/hotspot/`, last visited 2/22/2006.

Venkatachalam, V. and Franz, M. (2005). Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3):195–237.

Weiser, M., Welch, B., Demers, A. J., and Shenker, S. (1994). Scheduling for reduced CPU energy. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23.

Weissel, A. and Bellosa, F. (2002). Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 238–246. ACM Press.

Wu, Q., Martonosi, M., Clark, D. W., Reddi, V. J., Connors, D., Wu, Y., Lee, J., and Brooks, D. (2005). A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282, Washington, DC, USA. IEEE Computer Society.

Xie, F., Martonosi, M., and Malik, S. (2003). Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62, New York, NY, USA. ACM Press.

Xie, F., Martonosi, M., and Malik, S. (2004). Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Trans. Archit. Code Optim.*, 1(3):323–367.

A Benchmark Data

freq [MHz]	clock cycles	actual runtime [ms]	estimated runtime [ms]	error [%]	cycle drop [%]	perf. loss [%]	clock cycles	actual runtime [ms]	estimated runtime [ms]	error [%]	cycle drop [%]	perf. loss [%]
data/dacapo-antlr.data							data/dacapo-bloat.data					
600	3.05e10	50769.1	48160.8	5.1	37.1	67.5	4.60e10	76603.4	75066.2	2.0	19.4	114.7
800	3.42e10	42747.0	41017.0	4.0	29.4	41.0	4.78e10	59711.3	59309.9	0.7	16.3	67.3
1000	3.75e10	37526.4	36730.8	2.1	22.6	23.8	5.02e10	50162.3	49856.1	0.6	12.1	40.6
1200	4.07e10	33887.1	33873.3	0.0	16.1	11.8	5.24e10	43704.9	43553.6	0.3	8.1	22.5
1400	4.46e10	31832.2	31832.2	0.0	8.0	5.0	5.47e10	39051.8	39051.8	0.0	4.2	9.4
1600	4.85e10	30301.4	30301.4	0.0	0.0	0.0	5.71e10	35675.4	35675.4	0.0	0.0	0.0
data/dacapo-chart.data							data/dacapo-fop.data					
600	6.29e10	104790.0	102330.0	2.4	19.6	114.2	4.44e09	7394.6	7320.1	1.0	23.3	104.4
800	6.54e10	81777.0	80961.1	1.0	16.4	67.2	4.63e09	5788.1	5839.0	-0.9	19.9	60.0
1000	6.84e10	68425.0	68139.7	0.4	12.5	39.9	4.90e09	4899.0	4950.3	-1.1	15.3	35.4
1200	7.15e10	59602.8	59592.1	0.0	8.6	21.8	5.21e09	4341.0	4357.9	-0.4	10.0	20.0
1400	7.49e10	53486.7	53486.7	0.0	4.3	9.3	5.51e09	3934.7	3934.7	0.0	4.8	8.7
1600	7.83e10	48907.6	48907.6	0.0	0.0	0.0	5.79e09	3617.3	3617.3	0.0	0.0	0.0
data/dacapo-jython.data							data/dacapo-pmd.data					
600	4.00e10	66585.7	66186.4	0.6	4.2	155.3	2.68e10	44650.6	43222.2	3.2	24.9	100.1
800	4.11e10	51348.1	50144.1	2.3	1.5	96.8	2.84e10	35513.6	34857.5	1.9	20.4	59.1
1000	4.09e10	40880.7	40518.7	0.9	2.0	56.7	3.06e10	30573.3	29838.7	2.4	14.3	37.0
1200	4.12e10	34317.2	34101.8	0.6	1.3	31.5	3.24e10	26968.1	26492.8	1.8	9.3	20.8
1400	4.13e10	29518.3	29518.3	0.0	0.9	13.1	3.37e10	24102.9	24102.9	0.0	5.4	8.0
1600	4.17e10	26080.6	26080.6	0.0	0.0	0.0	3.57e10	22310.4	22310.4	0.0	0.0	0.0
data/dacapo-ps.data							data/dacapo-xalan.data					
600	7.69e10	128234.0	127759.0	0.4	1.0	163.7	1.19e10	19757.6	19335.7	2.1	12.5	133.2
800	7.70e10	96290.6	96100.2	0.2	0.9	98.0	1.21e10	15078.7	14989.9	0.6	11.0	78.0
1000	7.72e10	77195.5	77104.8	0.1	0.7	58.8	1.24e10	12449.2	12382.4	0.5	8.1	46.9
1200	7.74e10	64491.3	64441.2	0.1	0.5	32.6	1.28e10	10633.3	10644.1	-0.1	5.8	25.5
1400	7.76e10	55395.7	55395.7	0.0	0.2	13.9	1.32e10	9402.5	9402.5	0.0	2.8	10.9
1600	7.78e10	48611.6	48611.6	0.0	0.0	0.0	1.36e10	8471.2	8471.2	0.0	0.0	0.0
data/javagrande-crypto.data							data/javagrande-euler.data					
600	5.82e10	97044.9	97160.2	-0.1	0.5	165.1	2.06e10	34276.8	31797.8	7.2	33.2	77.9
800	5.84e10	73008.5	72938.4	0.1	0.2	99.4	2.22e10	27726.7	26785.9	3.4	28.0	43.9
1000	5.85e10	58470.0	58405.3	0.1	0.1	59.7	2.41e10	24123.9	23778.7	1.4	21.7	25.2
1200	5.85e10	48727.9	48716.6	0.0	0.1	33.1	2.62e10	21841.2	21773.9	0.3	14.9	13.3
1400	5.85e10	41796.1	41796.1	0.0	0.0	14.1	2.85e10	20341.9	20341.9	0.0	7.6	5.5
1600	5.86e10	36605.7	36605.7	0.0	0.0	0.0	3.08e10	19267.9	19267.9	0.0	0.0	0.0
data/javagrande-fft.data							data/javagrande-flufact.data					
600	6.57e10	109492.0	99203.9	9.4	53.2	24.6	3.28e10	54735.0	44692.3	18.4	49.7	34.1
800	8.00e10	100000.0	94670.8	5.3	43.1	13.8	3.62e10	45291.3	43139.6	4.8	44.5	10.9
1000	9.46e10	94568.8	91951.0	2.8	32.7	7.6	4.36e10	43564.8	42207.9	3.1	33.2	6.7
1200	1.09e11	90713.4	90137.8	0.6	22.5	3.2	4.99e10	41623.3	41586.8	0.1	23.5	1.9
1400	1.24e11	88842.6	88842.6	0.0	11.5	1.1	5.76e10	41143.2	41143.2	0.0	11.7	0.8
1600	1.41e11	87871.2	87871.2	0.0	0.0	0.0	6.53e10	40810.5	40810.5	0.0	0.0	0.0
data/javagrande-heapsort.data							data/javagrande-moldyn.data					
600	3.54e10	58939.4	57144.3	3.0	41.4	56.2	1.29e11	214407.0	223422.0	-4.2	4.7	153.9
800	3.95e10	49408.5	49379.0	0.1	34.5	30.9	1.30e11	161997.0	167821.0	-3.6	4.0	91.8
1000	4.47e10	44726.9	44719.9	0.0	25.9	18.5	1.34e11	133587.0	134461.0	-0.7	1.1	58.2
1200	4.99e10	41568.3	41613.7	-0.1	17.3	10.1	1.34e11	112035.0	112221.0	-0.2	0.4	32.7
1400	5.52e10	39395.1	39395.1	0.0	8.6	4.4	1.35e11	96335.3	96335.3	0.0	0.1	14.1
1600	6.04e10	37731.1	37731.1	0.0	0.0	0.0	1.35e11	84421.0	84421.0	0.0	0.0	0.0

continued on next page

continued from previous page

freq [MHz]	clock cycles	actual runtime [ms]	estimated runtime [ms]	error [%]	cycle drop [%]	perf. loss [%]	clock cycles	actual runtime [ms]	estimated runtime [ms]	error [%]	cycle drop [%]	perf. loss [%]
data/javagrande-montecarlo.data							data/javagrande-search.data					
600	9.35e10	155848.0	129119.0	17.1	47.2	40.7	3.32e10	55259.4	55434.1	-0.3	21.6	109.0
800	1.12e11	140254.0	121757.0	13.2	36.6	26.6	3.48e10	43515.2	43834.3	-0.7	17.6	64.6
1000	1.21e11	121183.0	117341.0	3.2	31.5	9.4	3.66e10	36635.6	36874.4	-0.7	13.3	38.5
1200	1.39e11	115875.0	114396.0	1.3	21.5	4.6	3.86e10	32171.6	32234.4	-0.2	8.7	21.7
1400	1.57e11	112293.0	112293.0	0.0	11.2	1.4	4.05e10	28920.2	28920.2	0.0	4.2	9.4
1600	1.77e11	110716.0	110716.0	0.0	0.0	0.0	4.23e10	26434.5	26434.5	0.0	0.0	0.0
data/javagrande-sor.data							data/javagrande-sparsematmult.data					
600	9.79e09	16312.5	13004.9	20.3	20.1	113.0	5.71e10	95229.3	95370.8	-0.1	61.1	3.5
800	9.88e09	12346.6	10866.3	12.0	19.3	61.2	7.51e10	93835.0	94026.7	-0.2	49.0	1.9
1000	1.00e10	10017.6	9583.1	4.3	18.2	30.8	9.29e10	92937.1	93220.2	-0.3	36.8	1.0
1200	1.08e10	9031.3	8727.7	3.4	11.5	17.9	1.11e11	92631.7	92682.6	-0.1	24.4	0.6
1400	1.14e10	8116.7	8116.7	0.0	7.2	5.9	1.29e11	92298.5	92298.5	0.0	12.2	0.3
1600	1.23e10	7658.4	7658.4	0.0	0.0	0.0	1.47e11	92010.5	92010.5	0.0	0.0	0.0
data/specjvm-compress.data							data/specjvm-db.data					
600	8.86e09	14764.9	14356.3	2.8	13.4	130.9	1.14e10	19033.6	17960.9	5.6	44.1	49.0
800	9.09e09	11358.8	11171.3	1.6	11.1	77.6	1.30e10	16308.5	15886.0	2.6	36.1	27.6
1000	9.37e09	9373.3	9260.3	1.2	8.3	46.6	1.52e10	15222.7	14641.1	3.8	25.5	19.1
1200	9.61e09	8012.2	7986.2	0.3	6.0	25.3	1.70e10	14174.0	13811.1	2.6	16.7	10.9
1400	9.91e09	7076.2	7076.2	0.0	3.1	10.6	1.85e10	13218.3	13218.3	0.0	9.4	3.4
1600	1.02e10	6393.7	6393.7	0.0	0.0	0.0	2.04e10	12773.7	12773.7	0.0	0.0	0.0
data/specjvm-jack.data							data/specjvm-javac.data					
600	4.47e09	7447.3	7525.8	-1.1	36.9	68.0	7.69e09	12816.3	12521.4	2.3	19.3	114.9
800	4.99e09	6239.8	6288.2	-0.8	29.6	40.7	8.00e09	9999.8	9897.7	1.0	16.1	67.7
1000	5.48e09	5484.9	5545.7	-1.1	22.6	23.7	8.22e09	8218.4	8323.5	-1.3	13.8	37.8
1200	6.02e09	5018.3	5050.7	-0.7	15.0	13.2	8.59e09	7158.6	7274.0	-1.6	9.9	20.0
1400	6.58e09	4697.1	4697.1	0.0	7.2	5.9	9.13e09	6524.3	6524.3	0.0	4.2	9.4
1600	7.09e09	4431.9	4431.9	0.0	0.0	0.0	9.54e09	5962.1	5962.1	0.0	0.0	0.0
data/specjvm-jess.data							data/specjvm-mpegaudio.data					
600	4.12e09	6859.4	6864.1	-0.1	14.6	127.7	7.18e09	11973.3	11849.5	1.0	0.7	164.7
800	4.31e09	5393.6	5323.4	1.3	10.4	79.0	7.18e09	8975.6	8918.6	0.6	0.7	98.4
1000	4.39e09	4392.4	4399.0	-0.1	8.8	45.8	7.19e09	7188.6	7160.1	0.4	0.6	58.9
1200	4.59e09	3827.6	3782.7	1.2	4.7	27.0	7.21e09	6004.8	5987.7	0.3	0.4	32.7
1400	4.68e09	3342.5	3342.5	0.0	2.9	10.9	7.21e09	5150.3	5150.3	0.0	0.3	13.8
1600	4.82e09	3012.3	3012.3	0.0	0.0	0.0	7.24e09	4522.3	4522.3	0.0	0.0	0.0
data/specjvm-mtrt.data												
600	2.53e09	4214.2	5105.3	-21.1	22.6	106.3						
800	2.69e09	3356.5	3880.3	-15.6	17.8	64.3						
1000	2.87e09	2868.3	3145.2	-9.7	12.2	40.4						
1200	3.02e09	2518.4	2655.2	-5.4	7.5	23.2						
1400	3.23e09	2305.2	2305.2	0.0	1.2	12.8						
1600	3.27e09	2042.7	2042.7	0.0	0.0	0.00						