

Self-Calibrating Processor Speed A New Feedback Loop for Dynamic Voltage Scaling Control

November 16, 2006

Abstract

The benefit of dynamic voltage scaling (DVS) is related to how compute-bound a workload is. The more time a processor stalls, the more the workload can be slowed down without incurring a proportional performance loss. However, determining a measure of “compute-boundedness” is not trivial; this property has often been inferred from secondary effects, such as cache miss rates.

We propose a new mechanism for extrapolating compute-boundedness from minute variations of processor speed. Using simulations, we show that our technique can determine the compute-boundedness of a workload, and thereby the optimal processor speed, by adjusting the clock frequency by as little as tens of megahertz during program execution. We also show that at the larger frequency step interval of hundreds of megahertz that is offered by today’s DVS-enabled processors, our simulation is consistent with actual hardware behavior. We are therefore confident that our results will translate directly into a hardware implementation and that they point the way to future processors with self-calibrating DVS feedback loops.

1 Introduction

Due to the effects of processor stalls, memory or disk bound programs can often be slowed down without a proportional performance loss. Thus many of the recent processor clock scaling algorithms [2, 3, 4, 5, 15, 16, 18, 20, 24, 25] slow down the processor during code regions that are memory-bound, to save energy with minimal performance loss. They extrapolate memory-boundedness from hardware events such as cache misses [3, 4, 5, 14, 15, 16, 18, 20], memory bus transactions [25] or memory requests per cycle [24]. The drawback is that these events only give a piece of the whole picture, since program execution time depends on all the complex interactions in a system, including the state of the cache, disk and network interfaces, as well as the interaction of multiple programs executing concurrently.

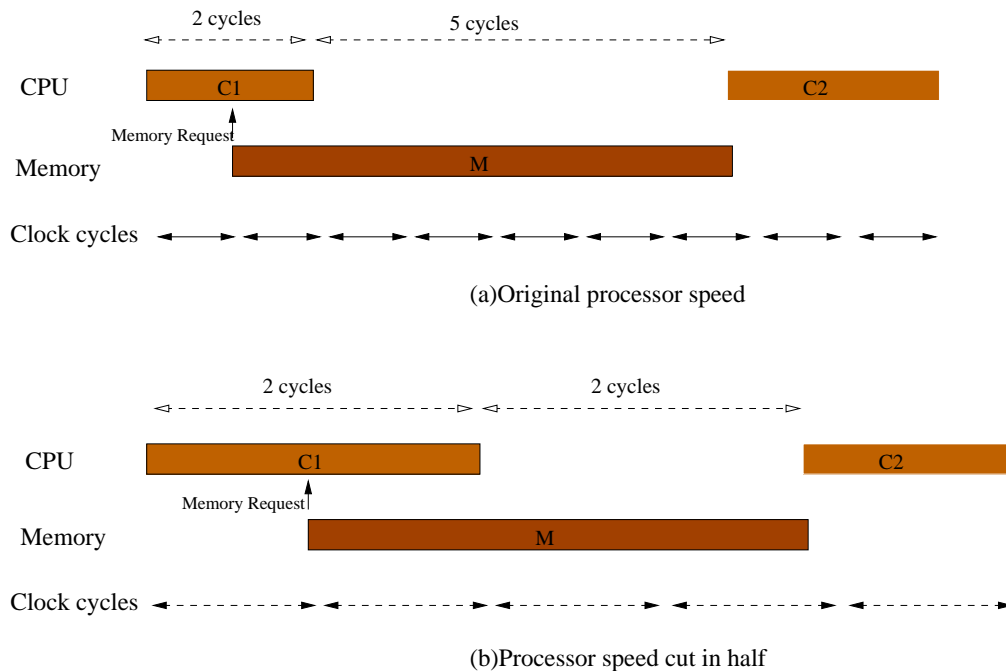


Figure 1: How a processor stall affects the total number of cycles. In (a), the processor performs some computations (C1), stalls until the data it requested earlier is brought in from memory, and then performs some more computations (C2) that depend on the data. In (b) the whole example is run at half the processor clock rate. The number of required cycles for computations (C1 and C2) is unaffected, but the cycles for the processor stall decreases from 5 to 2. Thus the percentage decrease in cycles indicates how much the processor is stalling.

Instead of measuring memory-boundedness, the DVS algorithm of Hsu and Feng [10, 11] attempts to estimate compute-boundedness using a regression method over past instruction execution rates. This approach accounts for more system-effects than approaches that only measure, for example, cache misses.

Venkatachalam et al. [22] have developed a new technique for assessing how compute-bound a workload is. The key idea is that if a workload is run at a lower clock frequency and its execution time does not increase as much as one would expect—which may be the case if the processor stalls due to excessive memory, disk or network accesses—then executing the workload takes fewer processor clock cycles at the lower clock frequency than at the higher clock frequency.

Figure 1 gives an example of how this decrease in cycles can come about. In (a), the processor performs some computations, and then requests data that is not present in its cache. For a while it may perform some more computations

that do not depend on the data, but eventually has to wait for the data to be brought in before performing the computations (C2) that require the data.

When the clock frequency is cut in half (b), the computational phases (C1 and C2) take the same number of cycles. The only difference is in the number of cycles that the processor stalls. If the memory subsystem is asynchronous, which is the case in many systems, or if data has to be transferred from the disk or downloaded from another computer through the network, then the total time to satisfy the data request will be independent of the processor clock frequency. Thus the total cycles that the processor stalls will decrease. Because of this decrease in cycles, the increase in overall runtime in part (b) will not be proportional to the decrease in clock frequency, but will be less than expected.

The main idea is that when a workload is run at a lower clock frequency, the *percentage decrease in cycles* indicates how much the execution time of the workload will increase, and therefore how compute-bounded the workload is.

However, to determine a workload’s compute-boundedness using this metric, it is not necessary to run the workload at every clock frequency. In fact, it has been shown [22] that by running a workload at *any* two clock frequencies, and in particular, the maximum and second highest clock frequencies, one can predict with high accuracy the workload’s execution times at all other clock frequencies.

These ideas can be applied to develop a highly lightweight approach to CPU clock scaling, the *self-calibrating feedback loop*. Namely, while a workload is running, slow down the processor just a little, to the next lower clock frequency, and use the limited information from these two points (the current and next lower clock frequencies) to extrapolate how the workload will behave at any clock frequency and thereby determine the optimal clock frequency.

This paper takes these ideas a step further by exploring how minutely we can adjust the processor clock speed and still allow this “two-point” approach to be effective. On typical laptops that support processor clock scaling, the difference between consecutive frequency steps is on the order of hundreds of megahertz. Using simulations, we will show that there is a one-to-one correspondence between how a workload behaves when the clock frequency is changed by hundreds of megahertz, and how it behaves when the frequency is changed by tens of megahertz. Thus by changing the clock frequency by as little as *tens* of megahertz, we can learn enough about the execution behavior to reliably predict what the execution times would be when the frequency is changed by hundreds of megahertz. Moreover, the simulation behavior at the level of hundreds of megahertz (which is offered by typical laptops) is similar to the behavior on actual hardware. Thus we are confident that our simulation results will carry over to real hardware.

The rest of this paper is organized as follows. Section 2 describes our methodology and results. Section 3 explores the implication of these results for future hardware. Section 4 summarizes related work, and Section 5 discusses future work and concludes.

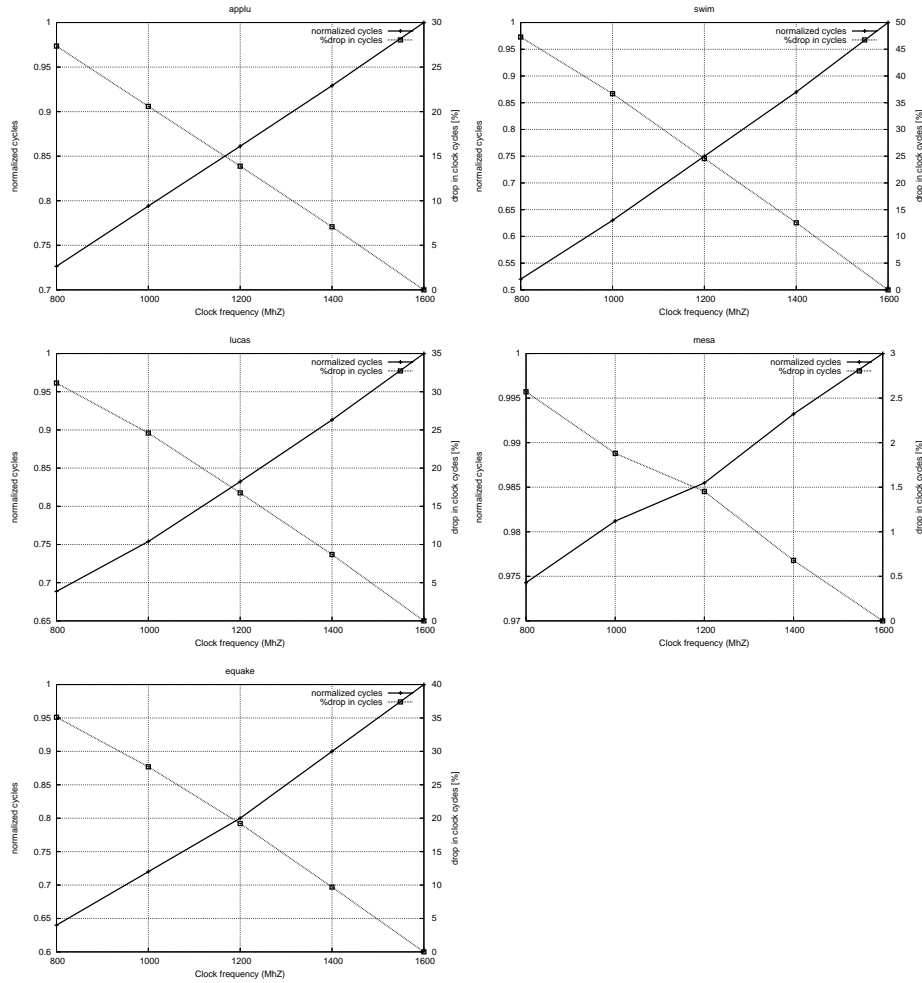


Figure 2: Clock cycles and percentage drop in clock cycles as a function of clock frequency on real hardware.

2 Methodology and Results

On many of the current DVS-enabled laptops, the “distance” between consecutive clock frequency steps is on the order of hundreds of megahertz. The *big-step hypothesis*, which has been validated on real hardware [22], is that by knowing only the clock cycles at the maximum and second highest clock frequencies one can extrapolate the execution times of a workload at any other clock frequency.

Our hypothesis, which we call the *small-step hypothesis* is that even if the distance between consecutive frequency steps were considerably smaller, such as on the order of tens of megahertz, we can use a simple linear extrapolation

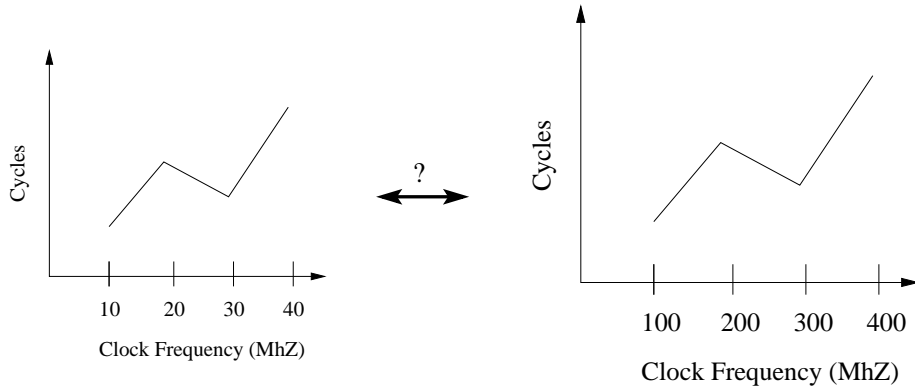


Figure 3: Our goal is to determine whether there is an isomorphism between program behavior—in terms of the relation between clock cycles and clock frequency – when clock frequency is changed by tens of megahertz and when it is changed by hundreds of megahertz.

to predict what the execution times would be if the processor were slowed down by hundreds of megahertz.

We validate our hypothesis in two stages. First we show that at the granularity of hundreds of megahertz, which is supported by typical laptops, the behavior of workloads in our simulations (in terms of the relation between clock cycles and clock frequency) is comparable to that on a real laptop. This ensures that we have set up our simulation parameters in a way that reflects actual hardware. Second, we show that in our simulations, there is a 1-1 correspondence (Figure 3) between workload behavior at the granularity of hundreds of megahertz and the granularity of tens of megahertz. These two results lead us

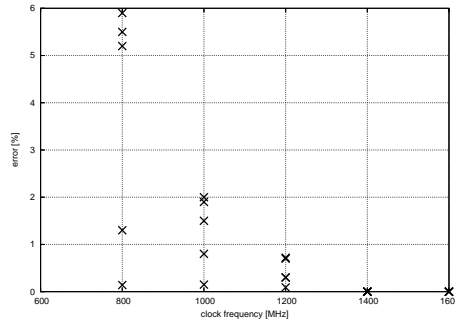


Figure 4: Validating the big-step model on real hardware. This is a histogram of the error in estimating execution times based on the two points 1600 MHz and 1400 MHz.

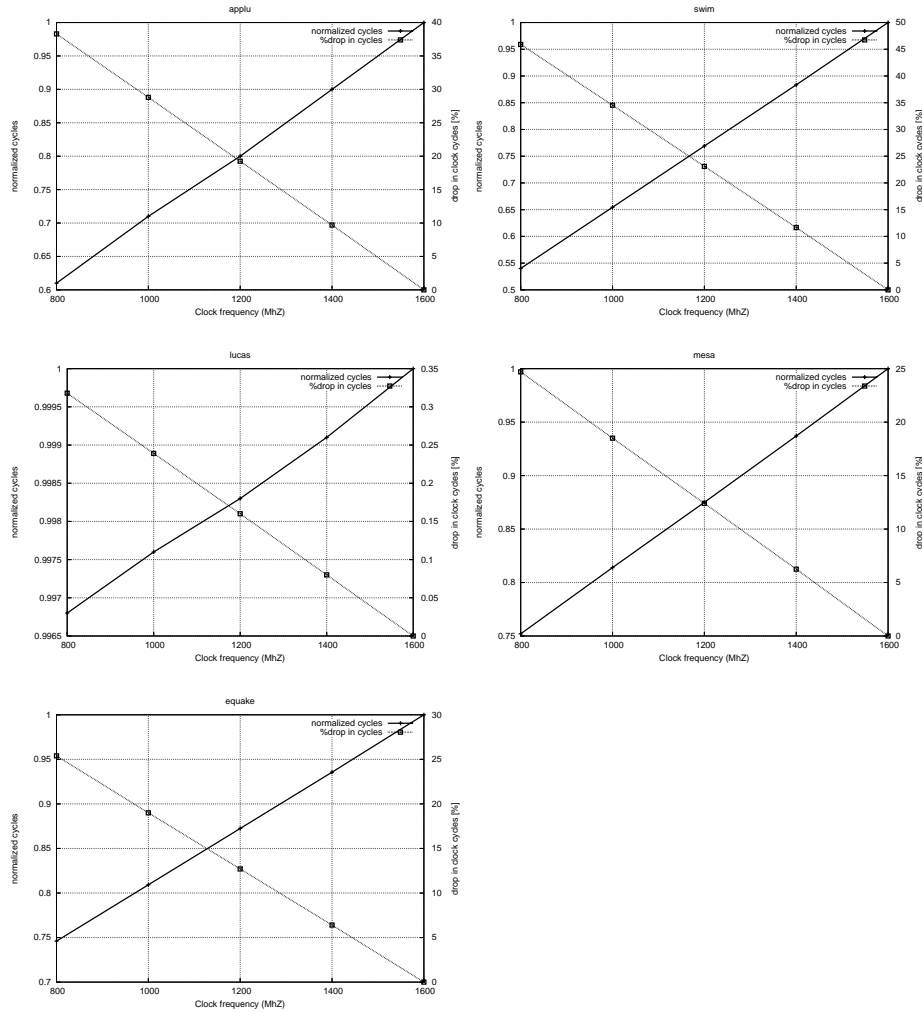


Figure 5: Clock cycles and percentage drop in clock cycles as a function of clock frequency on the simulator.

to believe that by transitivity, there would be a 1-1 correspondence between the program behavior at these two granularities on real hardware.

We chose five benchmarks from the Spec 2000 suite, which are described in Table 1. For our runs on actual hardware, we used the reference-size inputs. For our simulations, we used the Minnespec versions [13] of these inputs, because otherwise the simulations would take prohibitively long—on the order of months—to run.

We modified SimpleScalar to simulate a DVS-enabled out-of-order processor that supports the same frequency steps as the laptop we use in our experiments,

which is a Dell Latitude D600. This laptop has a 1.6 GHz Pentium M processor that supports switching between multiple frequency settings on the fly and automatically adjusts the voltage with respect to the frequency. The frequencies range from 800 MhZ to 1600 MhZ and can be incremented in steps of 200 MhZ. 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.

2.1 Isomorphism Between Simulation Behavior and Actual Hardware

First we ran each benchmark over each of the supported clock frequencies on our laptop. We then recorded for each clock frequency the total execution cycles for running at that frequency, as well as the percentage drop in execution cycles compared to running at the maximum clock frequency. We repeated this process with our simulator, which was configured with the same frequency settings as our laptop.

In Figure 2, we have plotted both the execution cycles and the percentage drop in cycles as a function of clock frequency, for each of the benchmark runs on our laptop. In each plot the y-axis on the left corresponds to the cycles, while that on the right corresponds to the drop percentage. Notice that the normalized execution cycles in each case is nearly linear with respect to clock frequency, and consequently so is the drop percentage. Because of this linear relationship, one can estimate execution times at any clock frequency on the basis of two points alone, the maximum and second highest clock frequencies.

In fact, Figure 4 plots the error in estimating the execution times based on a simple linear extrapolation from the two highest clock frequencies. On the x-axis is the clock frequency in MhZ, 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 1600 MhZ and 1400 MhZ will always be zero since these two points were chosen to extrapolate the execution times for all other clock frequencies. Notice that all the other datapoints lie below the 6% error mark, indicating that a linear extrapolation is very accurate on real hardware.

171.swim	Shallow water equations
183.quake	Seismic wave propagation simulation
189.lucas	Number theory / primality testing.
176.gcc	C programming language compiler
177.mesa	3D graphics library
173.applu	Parabolic/Elliptic Partial Differential Equations

Table 1: A description of the benchmarks from the SPEC 2000 suite used in this study.

Similarly, in Figure 5 we have plotted execution cycles as a function of clock frequency for each of the simulator runs. These plots are similar to the plots in Figure 2 in that the normalized execution cycles and the percentage drop in cycles are linear with respect to the clock frequency. This shows that the behavior we see in simulations is consistent with the behavior on actual hardware.

2.2 Isomorphism Between Simulation Behavior at Different Granularities

Next we ran each benchmark in our simulator 10 megahertz below the top clock frequency of 1600 MhZ and recorded the simulated execution cycles. Knowing the simulated clock cycles at the two points 1600 MhZ and 1590 MhZ, we linearly extrapolated execution times for all other clock frequencies, assuming that there is a 1-1 correspondence between the 10 MhZ case and the 200 MhZ case. We then calculated the error in doing so, by comparing the extrapolated execution time to the actual simulated execution time.

The top-left plot in Figure 7 illustrates the overall error of our approach. It is a histogram of the estimation errors for all benchmarks over all clock frequencies. On the x-axis is the clock frequency in MhZ, 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 highest clock frequency will always be zero since this was one of the two points chosen to extrapolate the execution times for all other clock frequencies. We are only interested in the datapoints plotted for the other clock frequencies. All of these 20 points lie below the 7% error range. In fact, nine of them lie below the 1% error range and four lie right on the 0% mark since the errors were so small as to be negligible.

Figure 6 plots the estimated and actual simulated execution times as a function of clock frequency for each of these benchmarks. In all these plots our estimate of execution time is nearly identical to the actual simulated execution time, further supporting our small-step hypothesis.

We ran two more experiments in the simulator to test whether these results would be affected by a change in either the memory access time or the range of clock frequencies supported. In the first test we changed the memory access time to a quarter of the original time and reran the above experiments, keeping all other simulation parameters the same. In the second test we changed the supported clock frequencies to vary from 100 MhZ to 600 MhZ in steps of 100 MhZ.

We found that decreasing the time for a memory access had the effect of making all the benchmarks more compute-intensive. However, even in this case, a minute change in processor speed (this time 30 MhZ) was sufficient to accurately predict execution times for all other clock frequencies. The top-right plot in Figure 7 is a histogram of the error in our approach, which is less than 8%.

Changing the supported clock frequencies to vary from 100 MhZ to 600 MhZ in steps of 100 MhZ had little effect on the results, as the bottom histogram

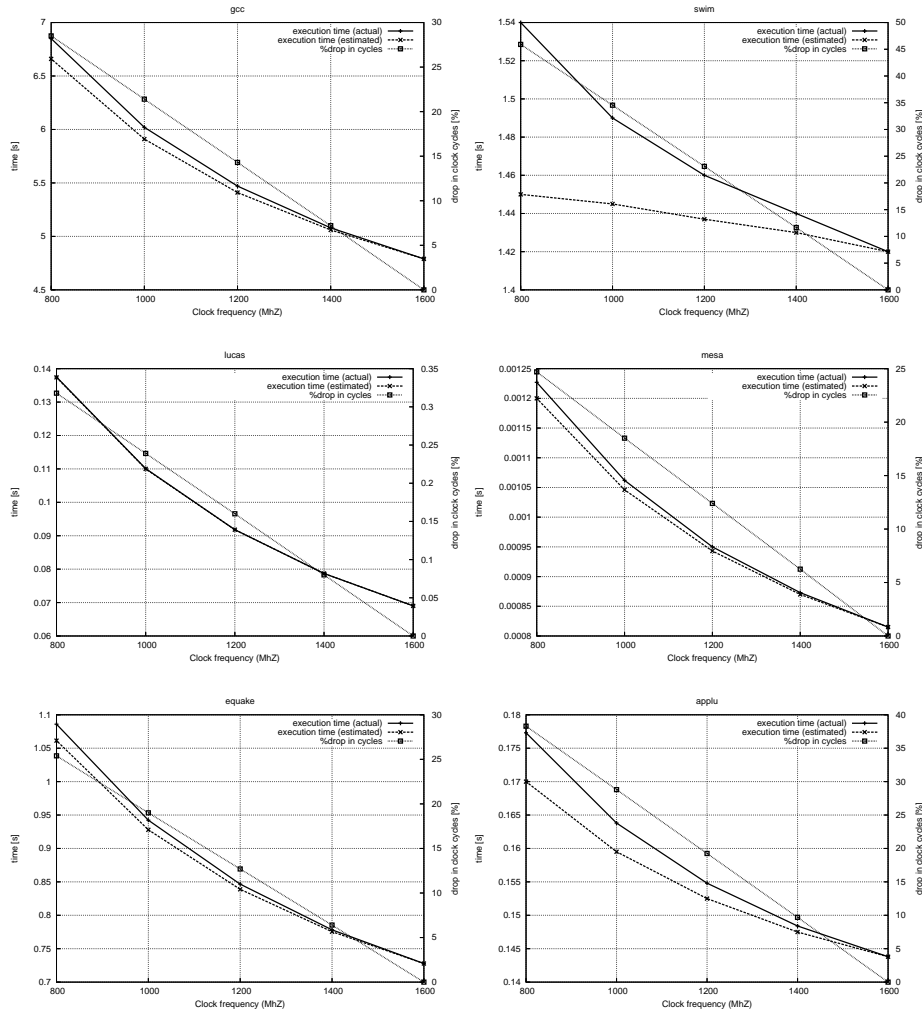


Figure 6: Validating the small-step hypothesis. Our estimate of execution time versus the actual simulated execution time as a function of clock frequency.

in Figure 7 shows. Most of the datapoints (13 out of 14) lie below the 10% error range. Only one point lies at the 17% error mark and was due to the swim benchmark. Here, our approach underestimated the idle time, which led to a more conservative estimate of the execution time.

3 Implications for Future Hardware

These results lead us to believe that on actual hardware, the behavior of a workload when varying the processor speed by just tens of megahertz would strongly

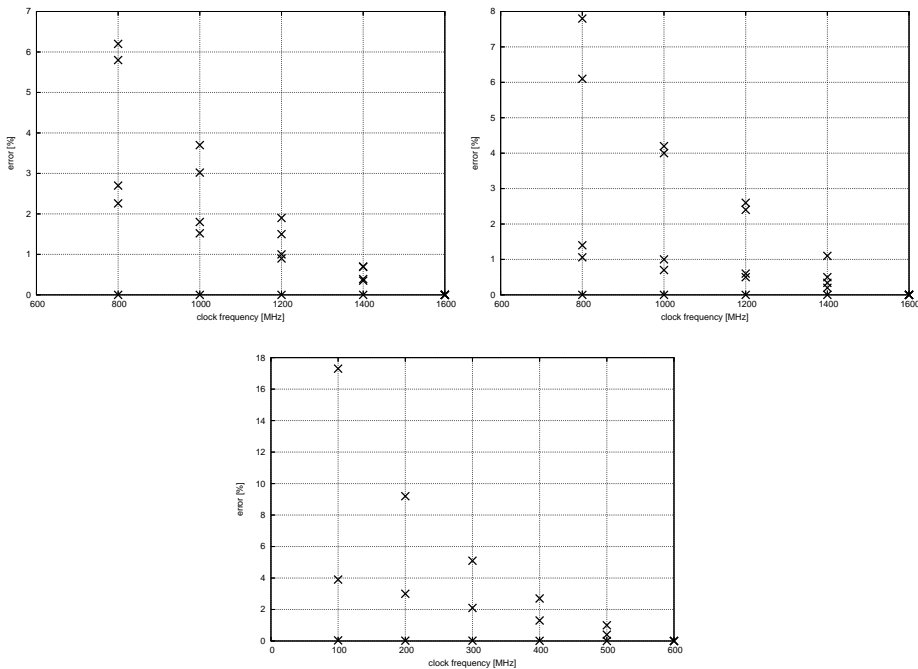


Figure 7: Histograms of the estimation error of our model for all benchmarks over all clock frequencies. The graphs show the results for the original Simplescalar configuration (top-left), a second configuration in which the memory access time is smaller (top-right), and a third configuration (bottom) in which the range of supported clock frequencies is different.

correspond to the behavior when varying the speed by hundreds of megahertz. As a result, we can develop highly efficient CPU clock scaling algorithms with minimal changes to the existing hardware for dynamic voltage scaling. As Figure 8 shows, the idea is to have a discrete number of settings for the clock frequency, just as in a typical DVS-enabled processor. (In this example, the settings are labelled f_0 to f_3). The “step size” or distance between consecutive clock frequencies may be on the order of hundreds of megahertz. However, surrounding each setting is window within which the clock frequency can be adjusted by a far smaller amount, which is on the order of tens of megahertz.

Figure 8 illustrates how this modified hardware would be used. Initially a workload would run at the maximum clock frequency and the hardware would profile its execution. After enough samples have been taken, the clock frequency would be lowered by just a little, within the current “frequency window”, and the resulting difference in execution cycles would be used to determine how compute-bound the workload is. This essential information, possibly combined with other information about power-performance tradeoffs, could be used to determine an optimal frequency setting. As the figure shows, the process of

resampling and readjusting the processor clock frequency in response to changes in the workload’s behavior, can be applied continuously during the program execution.

The end-user will not notice the minute changes in clock frequency that are used to measure the workload’s compute-boundedness. All the user would notice are the frequency changes at the scaling points, where the processor speed is adjusted by hundreds of megahertz.

There are three advantages to such an approach:

- 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 information for making DVS decisions than any single hardware counter event.
- Because the clock frequency is only lowered by tens of megahertz below the top clock frequency, and this is only done once for the sake of measurement, the performance loss would be imperceptible.

4 Related Work

There is a wealth of literature on DVS algorithms DVS [1, 3, 6, 7, 8, 9, 10, 12, 14, 17, 19, 21, 23]. 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 [3, 4, 5, 14, 15, 16, 18, 20, 21, 25].

Marculescu [16] 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 [14] 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

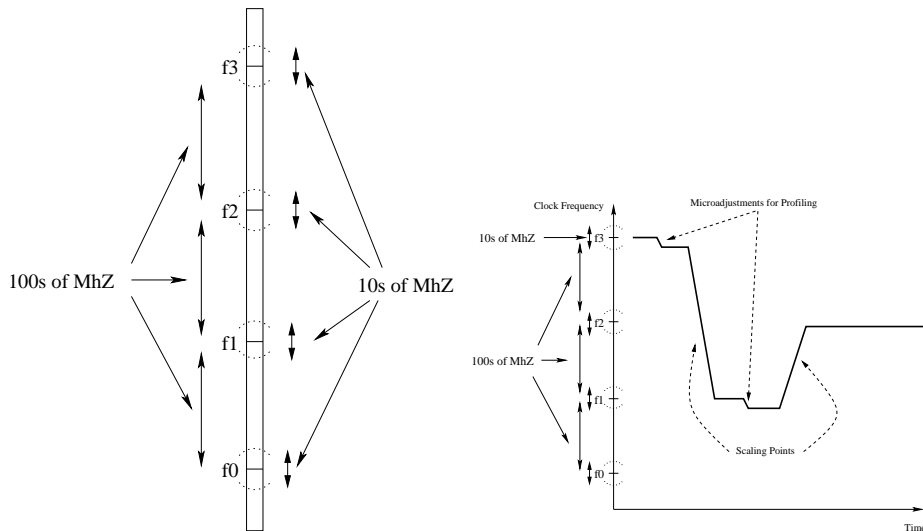


Figure 8: Self-calibrating feedback loop. On the left is the modification we propose to existing DVS hardware. As in a typical DVS-enabled processor, there are a discrete number of frequency steps (f_0 to f_3). However, surrounding each step is a window within which the clock frequency may be adjusted by a far smaller granularity. On the right we show how this modified hardware would be used.

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 [24] 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. [25] 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 [26, 27]. 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. [18] 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. [3] 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.

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 [14], [18], [16], and [3] the events are cache misses. In [24] the events are memory requests per cycles and instructions per cycle. In [25], the events are memory-bus transactions and micro-ops retired. These approaches rely on an imperfect statistical correlation between the events in question (e.g., cache misses, instructions per cycle) and execution time and clock frequency. On the other hand, execution time, clock frequency and clock cycles are directly related, making our metric, *the percentage drop in cycles*, a more appropriate metric for assessing how much a code region will slow down when it is run at a lower clock frequency.

5 Conclusions and Future Work

In simulations, we found a one-to-one correspondence between how a workload behaves when the clock frequency is changed by tens of megahertz, and how it behaves when the clock frequency is changed by hundreds of megahertz. By changing the frequency by as little as ten megahertz and measuring the difference in execution cycles, one can predict with high accuracy what would be simulated execution times if the frequency were adjusted by hundreds of megahertz. Moreover, we found that the underlying behavior that gives rise to these results — the linear relationship between clock cycles and clock frequency — is the same in both our simulations as well as on actual hardware. This gives us

good reason to believe that on actual hardware, a variation in clock frequency by as little as tens of megahertz would be enough to predict execution times of a workload at frequency steps that differ by hundreds of megahertz.

We have also explained how these ideas can be applied to develop a lightweight approach to CPU clock scaling, which we call the *self-calibrating feedback loop*. Namely, while a workload is running, slow down the processor just a little and use the limited information from these two points (the current and next lower clock frequencies) to extrapolate how the workload will behave at any clock frequency, and thereby determine the optimal clock frequency. To assist this approach, we propose a small modification to existing clock scaling hardware, namely to have a micro-window surrounding each frequency setting, within which the clock frequency can be adjusted in far smaller increments than the currently supported hundreds of megahertz.

One of the areas for future work is to implement this approach in a commercial operating system, possibly relying on timer-interrupts to decide to transition between the different control states of the algorithm. The challenge is that the OS lacks information about what part of the program is being executed at any given time. When our heuristic decreases the clock frequency by one notch to decide how compute intensive a code region is, it needs to be certain that any change in execution cycles is due to having lowered the clock frequency and not due to something else (e.g, a change in the program phase behavior). We may be able to ensure this by using hints from some additional hardware events (e.g., instruction execution rates). We are currently experimenting to find the right events for this purpose.

References

- [1] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. 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, 2002.
- [2] K. Choi, W. Lee, R. Soma, and M. Pedram. 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*, 2004.
- [3] K. Choi, R. Soma, and M. Pedram. 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, 2004.
- [4] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *DATE '04*:

Proceedings of the conference on Design, automation and test in Europe, page 10004, Washington, DC, USA, 2004. IEEE Computer Society.

- [5] K. Choi, R. Soma, and M. Pedram. 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, 2004. ACM Press.
- [6] A. Dudani, F. Mueller, and Y. Zhu. 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, 2002.
- [7] K. Flautner, S. Reinhardt, and T. Mudge. 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, 2001.
- [8] K. Govil, E. Chan, and H. Wasserman. 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, 1995.
- [9] F. Gruian. 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, 2001.
- [10] C. Hsu and W. Feng. Effective dynamic voltage scaling through cpu-boundedness detection. In *Workshop on Power Aware Computing Systems*, 2004.
- [11] C.-H. Hsu and W.-C. Feng. 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, 2005.
- [12] C.-H. Hsu and U. Kremer. 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, 2003.
- [13] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2006.
- [14] M. Kondo and H. Nakamura. Dynamic processor throttling for power efficient computations. In *Workshop on Power Aware Computing Systems*, 2004.

- [15] H. Li, C.-Y. Cher, T. N. Vijaykumar, and K. Roy. 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, 2003.
- [16] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *Proceedings of the Workshop on Complexity-Effective Design*, 2000.
- [17] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102. ACM Press, 2001.
- [18] C. Poellabauer, L. Singleton, and K. Schwan. 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, 2005.
- [19] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 2–11, 2002.
- [20] L. Singleton, C. Poellabauer, and K. Schwan. Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling. In *Proceedings of the 12th Annual Multimedia Computing and Networking Conference*, January 2005.
- [21] P. Stanley-Marbell, M. Hsiao, and U. Kremer. A Hardware Architecture for Dynamic Performance and Energy Adaptation. In *Proceedings of the Workshop on Power-Aware Computer Systems*, pages 33–52, 2002.
- [22] V. Venkatachalam, C. W. Probst, and M. Franz. A new way of estimating compute boundedness and its application to dynamic voltage scaling. *International Journal of Embedded Systems*, 2006.
- [23] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.
- [24] A. Weissel and F. Bellosa. 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, 2002.
- [25] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. 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, 2005. IEEE Computer Society.

- [26] F. Xie, M. Martonosi, and S. Malik. 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, 2003. ACM Press.
- [27] F. Xie, M. Martonosi, and S. Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Trans. Archit. Code Optim.*, 1(3):323–367, 2004.