PACS 2004: The Fourth Workshop on Power-Aware Computer Systems.
Portland, Oregon, December 2004, LA-UR 04-7195.

# Effective Dynamic Voltage Scaling through CPU-Boundedness Detection *

Chung-Hsing Hsu and Wu-chun Feng
{chunghsu,feng}@lanl.gov
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545

*Keywords*: Power-aware computing, dynamic voltage scaling, interval-based voltage scheduling, performance modeling, power-performance tradeoff.

## Abstract

Dynamic voltage scaling (DVS) allows a program to execute at a non-peak CPU frequency in order to reduce CPU power, and hence, energy consumption; however, it is done at the cost of performance degradation. For a program whose execution time is bounded by peripherals' performance rather than the CPU speed, applying DVS to the program will result in negligible performance penalty. Unfortunately, existing DVS-based power-management algorithms are *conservative* in the sense that they overly exaggerate the impact that the CPU speed has on the execution time, e.g., they assume that the execution time will double if the CPU speed is halved. Based on a new single-coefficient performance model, we propose a DVS algorithm that detects the CPU-boundedness of a program on the fly (via a regression method on the past MIPS rate) and then adjusts the CPU frequency accordingly. To illustrate its effectiveness, we compare our algorithm with other DVS algorithms on real systems via physical measurements.

## 1 Introduction

Dynamic voltage and frequency scaling (DVS) is a mechanism whereby software can dynamically adjust CPU voltage and frequency. This mechanism allows systems to address the problem of ever-increasing CPU power dissipation and energy consumption, as they are both quadratically proportional to the CPU voltage. However, reducing the CPU voltage may also require the CPU frequency to be reduced and results in degraded CPU performance with respect to execution time. In other words, DVS trades off performance for power and energy reduction.

The performance loss due to running at a lower CPU frequency raises several issues. First, a user who pays to upgrade his/her computer system does not want to see performance degradation. Second, running programs at a low CPU frequency may end up increasing total system energy usage [1, 2, 3, 4]. In order to control (or constrain) the performance loss effectively, a model that relates performance to the CPU frequency is essential for any DVS-based power-management algorithm (shortened as DVS algorithm hereafter).

A typical model used by many DVS algorithms predicts that the execution time will double if the CPU speed is cut in half. Unfortunately, this model overly exaggerates the impact that the CPU speed has on the execution time. It is only in the worst case that the execution time doubles when the CPU speed is halved; in general, the actual execution time is less than double. For example, in programs with a high cache miss ratio, performance can be limited by memory bandwidth rather than CPU speed. Since memory performance is not affected by a change in CPU speed, increasing or decreasing the CPU frequency will have little effect on the performance of these programs. We call this phenomenon — *sublinear performance slowdown*. Consequently, researchers have been trying to exploit this program behavior in order to achieve better power and energy reduction [5, 6, 7, 8]. One common technique decomposes program workload into regions based on their CPU-boundedness. The decomposition can be done statically using profiling information [5, 6] or dynamically through an auxiliary circuit [9, 10, 11] or through a built-in performance monitoring unit (PMU) [7, 8]. In this paper, we propose a new PMU-assisted, on-line DVS algorithm called *beta* that provides fine-grained, tight control over performance loss as well as takes the advantage of the sublinear performance slowdown. The new *beta* algorithm is based on an extension of the the-

*This work was supported by the DOE ASC Program through Los Alamos National Laboratory contract W-7405-ENG-36.

oretical work developed by Yao et al. [12] and by Ishihara and Yasuura [13]. Via physical measurements, we will demonstrate the effectiveness of the *beta* algorithm when compared to several existing DVS algorithms for a number of applications.

The rest of the paper is organized as follows: Section 2 characterizes how current DVS algorithms relate performance to CPU frequency. With this characterization as a backdrop, we present a new DVS algorithm (Section 3) along with its theoretical foundation (Section 4). Then, Section 5 describes the experimental set-up, the implemented DVS algorithms, and the experimental results. Finally, Section 6 concludes and presents some future directions.

## 2 Related Work

CPU utilization is often used to relate performance to the CPU frequency. While it is generally defined as the fraction of time that the CPU spends non-idle, CPU utilization may also be interpreted as the normalized workload (e.g., [14, 15, 16]). This particular interpretation has a nice property that there is a one-to-one correspondence between the desired normalized CPU speed and CPU idle time. Thus, if CPU utilization is 0.5 on a 2-GHz machine, then setting the CPU frequency to 1 GHz is predicted to eliminate all CPU idle time. Clearly, CPU utilization (or the normalized workload) follows the assumption that the execution time doubles when the CPU speed is halved. This type of model is popular because the metric is easy to derive at run time and it does not require application-specific information.

However, CPU utilization by itself does not provide enough information about system timing requirements, and DVS algorithms based on such information can only provide loose control over performance loss [17, 18, 19]. Thus, DVS algorithms with application-specific information have been proposed in order to provide tighter control over performance loss. For example, an application (or task or thread) can be associated with a deadline, in terms of seconds, as well as a CPU work requirement, in terms of CPU cycles. In this setting, performance is usually formulated as a linear function of the CPU speed. This type of performance model predicts that the execution time doubles when the CPU speed is halved. Other approaches use a target IPC (instructions per cycle) rate as the system timing requirements [20, 21]. Their performance model falls into the same category too.

There have been some attempts to exploit the sublinear performance slowdown to achieve more power and energy reduction. For example, Marculescu [9] proposed to set the CPU to a low speed whenever an L2 cache miss occurs. Li et al. [11] improved the algorithm by taking into account the transition overhead and scaling the CPU frequency and voltage according to the level of parallelism between the CPU and the memory subsystem. Stanley-Marbell et al. [10] designed an auxiliary hardware unit to detect loop-based, memory-bound execution phases.

The PMU-assisted "process cruise control" developed by Weissel and Bellosa [5] relies on a pre-computed table of optimal CPU speeds to direct the CPU speed change. The table is indexed by the run-time instruction counts per cycle and memory requests per cycle. Although the algorithm requires neither source code nor compiler support, it is inflexible in the sense that the table is obtained through extensive experiments of micro-benchmarks for a given performance loss (e.g., 10% in [5]). In other words, the algorithm does not allow for dynamic, application-specific control over performance loss.

Hsu and Kremer [6] use off-line profiling to identify memory-bound program regions, coupled with compiler transformations, to facilitate the setting of the CPU frequency. However, the need for source code and compiler support makes this approach more difficult to implement in practice. In general, compiler-directed DVS algorithms have the benefit of only requiring the host processor to export a DVS interface and does not require support from the OS scheduler. They also allow DVS scheduling decisions to be made in a global manner and to be in combination with performance-oriented optimization. On the other hand, savings are limited as speed-set instructions are inserted statically, and thus, apply to all execution of a specific memory reference, both cache misses as well as hits [22]. Moreover, input data sets may change program behavior that makes profile-based DVS algorithms less attractive.

Our work is closest to Choi et al.'s recent work [7, 8]. Both use a regression method and PMU support to perform on-line DVS scheduling through CPU-boundedness detection. However, the two works differ in their definition of CPU-boundedness, and thus, the detection mechanism. Choi et al.'s work is based on the ratio of the on-chip computation time to the off-chip access time. In contrast, our algorithm defines CPU-boundedness as the fraction of program workload that is CPU-bound. Because of the different definitions, the set of events monitored by the PMU for each algorithm is different. In Section 5.5, we argue that our DVS algorithm is equally effective but has a simpler implementation. Moreover, in contrast to [7, 8], we provide a theoretical foundation of why our DVS algorithm is effective in achieving energy optimality. The same theoretical result can be applied to their work as well.

In general, PMU-assisted DVS algorithms will encounter a couple of challenges. The PMU is notorious

for its incomplete set of event counting, inconsistency across generations of the CPU, and counters do not function as advertised. For example, Choi et al. presented two platform-dependent implementations [7, 8] of the same DVS algorithm because the PMUs of these two platforms count different sets of events. In addition, the correlation of event counts to power and performance is not yet clear and has been an ongoing research focus (e.g., [23, 24]).

# 3 A New DVS Algorithm

Here we describe a new interval-based PMU-assisted DVS algorithm that provides fine-grained, tight control over performance loss as well as exploits the sublinear performance scaling in memory-bound and I/O-bound programs. The theoretically-based heuristic algorithm is based on an extension of the theoretical work developed by [12] and [13] (details in Section 4):

> If the CPU power dissipation is a convex function of the CPU frequency, then for any program whose performance is an *affine* function of the CPU frequency, running at a constant CPU speed and meeting the deadline just in time will minimize the energy usage of executing the program. If the desired CPU frequency is not directly supported by the system, the two immediately-neighboring CPU frequencies can be used to emulate the desired CPU frequency and result in an energy-optimal DVS schedule.

To account for the sublinear performance slowdown, the following model that relates performance to the CPU frequency is often used [25, 7, 8]:

$$T(f) = W_{cpu} \cdot \frac{1}{f} + T_{mem} \tag{1}$$

The total execution time $T(f)$ at frequency $f$ is decomposed into two parts. The first part models on-chip workload in terms of CPU cycles. Its value is affected by the CPU speed change. The second part models the time due to off-chip accesses and is invariant to changes in the CPU speed. Note that this breakdown of the total execution time is inexact when the target processor supports out-of-order execution because on-chip execution may overlap with off-chip accesses [26, 22]. However, in practice, the error tends to be quite small [7, 8].

The model $T(f)$ treats program performance as an affine function of the CPU frequency $f$ and thus allows us to apply the aforementioned theoretical result. We simply execute a program at CPU frequency $f^*$ such that $D = T(f^*)$ where $D$ is the deadline of the program. However, there are two challenges in using the theorem this way. First, in many cases there is no consensus on how to assign a deadline to a program, e.g., scientific computation. Second, in order to use the model $T(f)$, we need to know the values of the coefficients, $W_{cpu}$ and $T_{mem}$. These coefficients are oftentimes determined by the hardware platform, program source code, and data input. Thus, calculating these coefficients statically is very difficult.

We address these challenges by defining a deadline as the relative performance slowdown and by estimating the model's coefficients on the fly (without any off-line profiling nor compiler support). The relative performance slowdown $\delta$

$$\delta = \frac{T(f)}{T(f_{max})} - 1 \tag{2}$$

where $f_{max}$ is the peak CPU frequency, as has been used in previous work [26, 7]. It is widely accepted in programs that are difficult to assign deadlines in terms of absolute execution time. It also carries more timing requirement information than CPU utilization and IPC rate. Providing this user-tunable parameter $\delta$ in our DVS algorithm allows fine-grained, tight control over performance loss.

To estimate the coefficients, we first re-formulate the original two-coefficient model in Equation (1) as a single-coefficient model:

$$\frac{T(f)}{T(f_{max})} = \beta \cdot \frac{f_{max}}{f} + (1 - \beta) \tag{3}$$

with

$$\beta = \frac{W_{cpu}}{W_{cpu} + T_{mem} \cdot f_{max}} \tag{4}$$

The coefficient $\beta$ is, by definition, a value between 0 and 1. It was introduced by one of the authors in [6] to quantify the CPU-boundedness of a program and its performance impact to the CPU speed change. The metric represents the fraction of the program workload that scales linearly with the CPU frequency. If a program has $\beta = 1$, it means the execution time of the program will double when the CPU speed is halved. In contrast, memory-bound and I/O-bound programs have their $\beta$ values close to zero, indicating that their execution time will remain the same even running at the slowest CPU speed. The single-coefficient model instead of the original two-coefficient model facilitates the calculation of the coefficient values in an efficient manner.

The coefficient $\beta$ is computed at run time using a regression method on the past MIPS rates reported from the PMU. Specifically, our DVS algorithm keeps track of the average MIPS rate for each executed CPU frequency and applies the least-square fitting at each interval to

For every $I$ seconds, doing the following:

1. Use Equation (5) to compute $\beta$.
2. Compute the ideal frequency $f^*$.

$$f^* = \begin{cases} f_{min} & \text{if } \beta \le \delta \\ f_{max}/(1+\delta/\beta) & \text{otherwise} \end{cases}$$

3. Figure out $f_j$ and $f_{j+1}$.

$$f_j \le f^* < f_{j+1}$$

4. Compute the ratio $r$

$$r = \frac{(1+\delta/\beta)/f_{max} - 1/f_{j+1}}{1/f_j - 1/f_{j+1}}$$

5. Run $r \cdot I$ seconds at frequency $f_j$.
6. Run $(1-r)\cdot I$ seconds at frequency $f_{j+1}$.
7. Update $\texttt{mips}(f_j)$ and $\texttt{mips}(f_{j+1})$.

Figure 1: Algorithm *beta*. Parameter $\delta$ is the relative performance slowdown and and parameter $I$ is the length of an interval in seconds.

dynamically re-compute the new $\beta$ value:

$$\beta = \frac{\sum_i (\frac{f_{max}}{f_i} - 1)(\frac{\texttt{mips}(f_{max})}{\texttt{mips}(f_i)} - 1)}{\sum_i (\frac{f_{max}}{f_i} - 1)^2} \quad (5)$$

where $\texttt{mips}(f)$ is the average MIPS rate for CPU frequency $f$. Note that our mechanism assumes a constant number of total instructions in a program, regardless of the running CPU frequency. This assumption has been verified through extensive experiments. In practice, the value of $\beta$ converges very quickly for the benchmarks we tested.

The rest of the algorithm simply applies the theoretical result to compute the desired CPU frequency $f^*$ for each interval, once the coefficient $\beta$ is updated, plus some bookkeeping on $\texttt{mips}(f)$. The derivation of $f^*$ comes by equating Equation (2) with Equation (3). Figure 1 outlines the entire algorithm.

Finally, we note that Choi et al.'s recent work on DVS algorithms [7, 8] is based on the on-line calculation of ratios $\alpha_f$, one for each frequency $f$, that are also derived from Equation (1). There, $\alpha_f$ is defined as the ratio of on-chip computation time to off-chip access times

$$\alpha_f = f \cdot \frac{T_{mem}}{W_{cpu}} \quad (6)$$

Using this $\alpha_f$, the desired CPU frequency for the next interval can be computed. The detailed comparison of both works is presented in Section 5.5.

# 4 Theoretical Foundation

In the previous section, we claim a theoretical result for energy-optimal DVS scheduling which extends both Yao et al.'s work in [12] and Ishihara and Yasuura's work in [13]. In this section we provide evidence to support our claim. However, due to the limit of paper length, all the proofs are left in the appendix.

The energy-optimal DVS scheduling problem considered here is taken from [6]. That previous work only provides a problem formulation. In this paper we provide two new theorems that characterize the energy-optimal DVS schedule for the problem. The two theorems are also closely related to some previous work such as Miyoshi et al.'s "critical power slope" [3].

A DVS system is assumed to export $n$ settings $\{(f_i, P_i)\}$, where $P_i$ is the CPU power dissipation (in watts) at CPU frequency $f_i$. Without loss of generality, we assume $0 < f_1 < \cdots < f_n$. We also denote the total execution time of a program running at setting $i$ as $T_i$. Finally, to facilitate discussion, we define $E_i = P_i \cdot T_i$.

The DVS scheduling problem is formulated as follows: given a program and a deadline $D$ (in seconds), find a DVS schedule $(t_1^*, \cdots, t_n^*)$ such that if the program is executed for $t_i^*$ seconds at setting $i$, the total energy usage $E$ is minimized, the deadline $D$ is met, and the required work is completed. Mathematically speaking,

$$\min E = \textstyle\sum_i P_i \cdot t_i \quad (7)$$

subject to

$$\textstyle\sum_i t_i \le D \quad (8)$$

$$\textstyle\sum_i t_i/T_i = 1 \quad (9)$$

$$t_i \ge 0 \quad (10)$$

To simplify the discussion of the main theorems, we handle a few corner cases first. The condition $D \ge \min_i T_i$ has to be satisfied so that the problem is feasible. If the condition $D \ge \max_i T_i$, the problem becomes the classical fractional Knapsack problem [27] because Equation (8) can be removed. In this case, the energy-optimal DVS schedule will execute the entire program at setting $i^*$ where $i^* = \arg_i \min\{E_i\}$. Similarly, for the case of $T_1 = \cdots = T_n$, the above DVS schedule is also energy-optimal. In the following, we will focus on cases where $T_1 \ne \cdots \ne T_n$ and $\min_i T_i < D < \max_i T_i$.

**Theorem 1** *If*

$$T_1 > T_2 > \cdots > T_n$$

*and*

$$0 \geq \frac{E_2 - E_1}{T_2 - T_1} \geq \frac{E_3 - E_2}{T_3 - T_2} \geq \cdots \geq \frac{E_n - E_{n-1}}{T_n - T_{n-1}}$$

*then*

$$t_i^* = \begin{cases} \frac{D - T_{j+1}}{T_j - T_{j+1}} \cdot T_j & i = j \\ D - t_j^* & i = j+1 \\ 0 & \text{otherwise} \end{cases}$$

*where*

$$T_{j+1} < D \leq T_j$$

Theorem 1 says that if the piecewise-linear function that connects points $\{(T_i, E_i)\}$ is convex and non-increasing on $[T_n, T_1]$, then running at a CPU frequency that finishes the execution right at the deadline is the most energy-efficient. If the desired CPU frequency is not directly supported, it can be emulated by the two immediately-neighboring CPU frequencies and result in the energy-optimal DVS schedule.

**Theorem 2** *If*

$$T_i = T(f_i) = \frac{c_1}{f} + c_0, \ c_1 \neq 0$$

*and*

$$\frac{P_1 - 0}{f_1 - 0} \leq \frac{P_2 - P_1}{f_2 - f_1} \leq \cdots \leq \frac{P_n - P_{n-1}}{f_n - f_{n-1}}$$

*then*

$$0 \geq \frac{E_2 - E_1}{T_2 - T_1} \geq \frac{E_3 - E_2}{T_3 - T_2} \geq \cdots \geq \frac{E_n - E_{n-1}}{T_n - T_{n-1}}$$

Theorem 2 says that, for any program whose execution time is an *affine* function of the CPU frequency, if the DVS settings in a CPU are *well-assigned*, then we can apply Theorem 1 to derive the energy-optimal DVS schedule. Theorem 2 apparently builds a bridge in using Theorem 1.

The DVS settings are considered well-designed if for any setting, it has the *lowest* power dissipation compared to the best possible combination of all other settings that emulates its frequency [19]. Equivalently, the DVS settings are well-designed if the CPU power dissipation is a convex function of the CPU frequency on $[0, f_{max}]$ (in contrast to convex on $[f_{min}, f_{max}]$). This is why Miyoshi et al. [3] found that in a few realistic CPUs, completing a task far before its deadline and putting the CPU into sleep mode is more energy-efficient than running the task as slow as possible to barely make the deadline. In these realistic CPUs, the CPU frequency $f_1$ can be emulated by the combination of CPU frequency 0 (i.e., the CPU in sleep mode) and a higher frequency with a lower power dissipation.

Finally, Theorem 2 extends the work presented by Yao et al. [12] and by Ishihara and Yasuura [13]. First, both works assume that $c_0 = 0$. Second, Ishihara and Yasuura's work assumes a fixed relationship between $f$ and $V$ in a DVS setting; namely,

$$f = k \cdot (V - V_T)^\alpha / V$$

where $k$, $V_T$, $\alpha$ are positive constants. Unfortunately, today's DVS processors may not be able to support such an assumption. This is because these processors only provide a discrete set of CPU frequencies and voltages, whereas the above equation requires a continuous range of CPU frequency i to be supported for a discrete set of voltages. Theorem 2 loosens these assumptions to facilitate DVS algorithms on realistic processors.

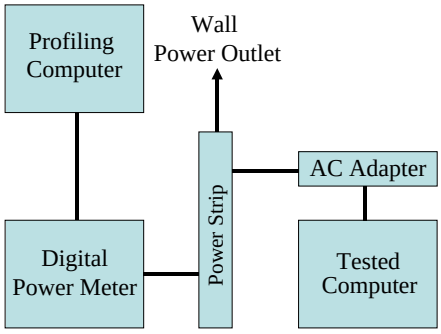


Figure 2: The experimental setup.

# 5 Experiments

In this section, we describe our experimental environment in which we evaluate and compare algorithm *beta* with several other DVS algorithms. We also discuss in depth the experimental results.

## 5.1 Experimental Setup

In order to get the high-fidelity experimental data, we set up our experiments using physical measurements, as shown in Figure 2. The experimental results were collected through a Yokogawa WT210 digital power meter [28]. The power meter continuously samples the instantaneous wattage at every 20 $\mu$s. The computer runs the Linux 2.4.18 kernel. All the benchmarks were compiled by GNU compilers with optimization level `-O2`. All the benchmarks were run to completion; each run took over a minute.

The benchmarks are taken from `SPEC`'s CPU95 benchmark suites. The `SPEC` benchmarks [29] emphasize the performance of the CPU and memory, but not other computer components such as I/O (disk drives), networking or graphics. We chose to use `SPEC` benchmarks

| $f$ (MHz) | $V$ |
|---|---|
| 1067 | 1.15 |
| 1333 | 1.25 |
| 1467 | 1.30 |
| 1600 | 1.35 |
| 1800 | 1.45 |

Table 1: The five settings on AMD's mobile Athlon XP.

because they demonstrate a range of performance sensitivity to the CPU frequency change, i.e., they have a wide range of $\beta$ values [6]. The experimental data are collected by running these SPEC benchmarks with the reference data input.

The hardware platform in our experiments is an HP NX9005 notebook computer. This computer includes a mobile AMD Athlon XP 2200+ processor, 256-MB DDR SDRAM, 266-MHz front-side bus, a 30-GB hard disk, and a 15-inch TFT LCD display. The mobile AMD Athlon XP processor has been used in Sun's Fire B100x blade servers [30]. It has a 128-KB L1 cache and a 256-KB L2 exclusive cache, making a total of 384-KB cache space. The processor exports two registers that the software can write the target frequency and voltage values into. In our experiments, we restrict the processor to have five settings as shown in Table 1. The transition time from one setting to another is 100 microseconds. During the measurements, the battery was removed and the monitor was turned off.

Finally, when presenting the experimental results, we associate with each application its $\beta$ value. Recall that the metric $\beta$ represents the fraction of the program workload that is very sensitive to the CPU speed change. That is, the higher the $\beta$ of a program, the more CPU-bound its performance. The $\beta$ value for each benchmark was derived by profiling total execution times for all settings and then applying the least-square fitting on Equation (3).

## 5.2 Implemented DVS Algorithms

To evaluate the effectiveness of our DVS algorithm *beta*, we have implemented a number of other DVS algorithms. The experiments by no means represent a comprehensive comparison among all existing approaches. Nevertheless, we feel that the range is wide enough to evaluate the effectiveness of our algorithm and to gain new insights from the experimental results. The following is a brief description of each algorithm we implemented.

*2step* This algorithm assumes dual CPU speeds in the processor and monitors the CPU utilization percentage periodically. If the percentage is higher than a predefined threshold, the algorithm will set the CPU to the fast speed; if it is lower than another pre-defined threshold, the algorithm will set the CPU to the low speed. This DVS algorithm is considered the best algorithm in Grunwald et al.'s empirical study on several interval-based algorithms using CPU utilization [18]. In our implementation, the two thresholds are 50% and 10% and the two speeds are the maximum CPU speed and the minimum CPU speed in the processor, respectively.

*nqPID* This algorithm was proposed by Varma et al. [15] as a refinement of the *2step* algorithm. Recognizing the similarity of the DVS scheduling and a classical control-systems problem, the authors took the equation describing a PID controller (Proportional-Integral-Derivative) and modified it to suit the DVS scheduling problem. This algorithm improved a lot on the control over performance loss that the *2step* algorithm lacks of. In addition, the authors found out that the algorithm's effectiveness does not depend on careful tuning of parameters, which is a nice feature given that *2step*'s effectiveness is critically dependent on the choice of application-specific threshold values [18].

*freq* This algorithm is similar to strategies that reclaim the slack time between the actual processing time and the worst-case execution time (e.g., [31, 32, 33, 34]). Specifically, the algorithm keeps track of the amount of remaining CPU work $W_{left}$ and the amount of remaining time before the deadline $T_{left}$. The desired CPU frequency $f^{new}$ at each interval is simply

$$f_{new} = \frac{W_{left}}{T_{left}}.$$

The algorithm assumes that the total amount of work in CPU cycles is known a priori, which, in practice, is often unpredictable [2] and not always a constant across frequencies [25].

*mips* This algorithm is taken from [21] that represents the DVS strategy guided by an externally specified performance metric. Specifically, the new frequency $f_{new}$ at each interval is computed by

$$f_{new} = f_{prev} \cdot \frac{\texttt{MIPS}_{target}}{\texttt{MIPS}_{observed}}$$

where $f_{prev}$ is the frequency for the previous interval, $\texttt{MIPS}_{target}$ is the externally specified performance requirement, and $\texttt{MIPS}_{observed}$ is the real MIPS rate observed in the previous interval. In our experiments, each benchmark has its own $\texttt{MIPS}_{target}$, which is derived by measuring the MIPS rate for the entire application and then dividing it by $(1 + \delta)$.

| **program** | $\beta$ | *2step* | *nqPID* | *freq* | *mips* | *beta* |
|---|---|---|---|---|---|---|
| swim | 0.02 | 1.00/1.00 | 1.04/0.70 | 1.00/0.96 | 1.00/1.00 | 1.04/0.61 |
| tomcatv | 0.24 | 1.00/1.00 | 1.03/0.69 | 1.00/0.97 | 1.03/0.83 | 1.00/0.85 |
| su2cor | 0.27 | 0.99/0.99 | 1.05/0.70 | 1.00/0.95 | 1.01/0.96 | 1.03/0.85 |
| compress | 0.37 | 1.02/1.02 | 1.13/0.75 | 1.02/0.97 | 1.05/0.92 | 1.01/0.95 |
| mgrid | 0.51 | 1.00/1.00 | 1.18/0.77 | 1.01/0.97 | 1.00/1.00 | 1.03/0.89 |
| vortex | 0.65 | 1.01/1.00 | 1.25/0.81 | 1.01/0.97 | 1.07/0.94 | 1.05/0.90 |
| turb3d | 0.79 | 1.00/1.00 | 1.29/0.83 | 1.03/0.97 | 1.01/1.00 | 1.05/0.94 |
| go | 1.00 | 1.00/1.00 | 1.37/0.88 | 1.02/0.99 | 0.99/0.99 | 1.06/0.96 |

Table 2: The effectiveness of 5 different DVS algorithms. Each table entry is in the format of *relative-time/relative-energy* with respect to the total execution time and system energy usage when running the application at the highest setting throughout the entire execution.

### 5.3 Experimental Results

Table 2 presents the experimental results for the 5 interval-based DVS algorithms. It can be seen that when a program's performance is toward memory-bound or I/O-bound ($\beta$ close to zero), there is a great potential in reducing a significant amount of CPU energy with negligible performance loss. In contrast, when a program is CPU-bound, there is little opportunity in reducing CPU power and energy within a tight performance loss bound of 5%. Moreover, none of the algorithms we tested was able to produce a DVS schedule that has the exact performance degradation of 5%. The actual performance loss varies from one benchmark to another.

Among the 5 tested interval-based DVS algorithms, algorithm *beta* outperforms others. In a sense, it verifies that our mechanism for computing CPU boundedness on the fly is of low overhead and that the algorithm is effective in providing tight control over performance loss due to DVS as well as exploiting the sublinear performance slowdown for a significant more CPU power and energy savings. Algorithms *mips* and *nqPID* are arguably ranked the second. Algorithm *mips* has better control over performance loss for all 8 benchmarks we tested, whereas algorithm *nqPID* has more power and energy reduction but at the cost of loose control over performance loss. This is especially obvious for CPU-bound benchmarks. Algorithms *freq* and *2step* are ranked the last.

So, what have we learned from this experiment? First, the number of instructions is a better metric for specifying the CPU work requirement than the number of CPU cycles. For the benchmarks we tested, we found that the number of instructions tends to remain constant across all settings. In contrast, the number of CPU cycles varies significantly depending on the executed DVS schedule. For example, the `swim` benchmark, when running at the lowest setting, has only 60% of the CPU execution cycles running at the highest setting. Typically, algorithm *freq* uses the worst-case execution cycles which in our case is the number of CPU cycles at the highest setting. This approach exaggerates the amount of the CPU work to be done and results in less effective energy reduction. This explains why algorithm *mips* performs better than algorithm *freq*.

The second point we have learned from the experiment is that a large window size of past PMU reports is better than a small window size of past PMU reports. In the experiments we found that the MIPS rate varies significantly from interval to interval, especially for CPU-intensive applications. However, the accumulated MIPS rate converges quickly, Thus, the use of the MIPS rate in a global manner seems to be more effective than the use of the rate in a local manner. This partially explains the effectiveness of algorithm *beta* comparing to algorithm *mips*. One concern for using a large window size is that the DVS algorithm may be less responsive for programs that exposes multiple execution phases of various degree of CPU-boundedness. For SPEC benchmarks, which are known to have the aforementioned behavior, this does not seem to be a problem. More details can be found in Section 5.4.

Finally, it is re-confirmed that CPU utilization by itself does not provide enough information about system timing requirements. As a result, the control over performance loss is unsatisfactory. This can be seen from the experimental results of algorithm *2step* and algorithm *nqPID*. Algorithm *2step* does not seem to perform any DVS scheduling. This is because the CPU for SPEC benchmarks is active almost all the time; That is, its CPU utilization is always full. In this case, there exists no optimal threshold values for *2step* to make it more effective. Algorithm *nqPID* refines algorithm *2step* by removing the threshold mechanism from the end-user. While it is more effective than algorithm *2step* in terms of CPU power and energy reduction, the lack of enough information about deadline makes it impossible to pro-

vide tight control over performance loss.

### 5.4 Discussion

To better address the impact of multiple-phase programs to the DVS algorithm *beta*, we compare it with a profile-based, off-line DVS algorithm called *hsu* [6]. The algorithm *hsu* uses PMU-assisted off-line profiling and source code analysis to identify the most energy-profitable region in a program to slow down without causing the performance loss to surpass a pre-defined level. Off-line profiling is performed on a section-by-section basis while the DVS scheduling decisions are made in a global manner, competitively comparing the different sections. This global view of the impact of DVS on different code sections allows more effective DVS scheduling, especially for multiple-phase programs such as SPEC benchmarks.

Algorithm *hsu* also uses the relative performance slow-down $\delta$ to specify the control over performance loss. Thus, it allows us to compare the two algorithms on a fair basis. In the experiments we executed the profile-based algorithm *hsu* with two different training inputs, denoted as *hsu(train)* and *hsu(ref)* respectively. The two set of training inputs are provided along with the SPEC benchmark codes. Table 3 shows the experimental results of both algorithms for the CFP95 benchmark suite.

We conclude that the effectiveness of algorithm *beta* is comparable to that of algorithm *hsu*. Both algorithms achieve a significant amount of CPU power and energy reduction with tightness of performance loss control. It is interesting to note that the two algorithms seem to complement each other. Algorithm *beta* performs better in CPU-bound benchmarks from mgrid to fpppp, whereas algorithm *hsu* performs better in memory-bound benchmarks from swim to hydro2d. We are in the process of investigating the causes for this phenomenon.

As mentioned in Section 2, the effectiveness of profile-based DVS algorithms is highly determined by its training data input. In our experiments, we found out that algorithm *hsu* chose different program regions to slow down in 7 of the 10 benchmarks. Running the reference data input as the training input does not necessarily yield a better result, for example, apsi. We suspect that the instrumented program for profiling has somewhat altered the instruction access pattern and is considerably different from the original code. According to Hsu's dissertation [35], the `SUIF2` compiler infrastructure, on which algorithm *hsu* was built, also has a big impact on the experimental results.

### 5.5 Further Discussion

In this section, we compare and contrast our work with Choi et al.'s work in [7, 8]. Recall that both works are based on the same Equation (1). The difference is in the calculation of equation coefficients. Our work calculates $\beta$ defined in Equation (4), whereas Choi et al.'s work calculates $\alpha_f$ defined in Equation (6).

Analytically, the two metrics are equivalent:

$$\beta = \frac{1}{1 + \alpha_f \cdot f_{max}/f}$$

However, there are several major differences in terms of implementation. First, metric $\beta$ is invariant to the CPU speed change, whereas metric $\alpha_f$ is defined with respect to a particular CPU frequency $f$. Thus, the number of coefficients calculated in Choi et al.'s work is more than the number of coefficients calculated in algorithm *beta*. Second, the formula in calculating $\alpha_f$ is more complex. This is due to the two-coefficient model they use, in contrast to the one-coefficient model we use. Finally, the number of PMU event counts needed for calculating $\beta$ is smaller than that for calculating $\alpha_f$. Since a CPU can simultaneously count a finite number of events, counting too many events may introduce a larger time overhead.

We feel that our new DVS algorithm has a simpler implementation than Choi et al.'s work. However, we cannot do an empirical comparison given the current setting we have. Choi et al. implemented their DVS algorithms on Intel Xscale-based processors which does not provide counting for the number of *retired* instructions. On the other hand, our hardware platform, Athlon XP processor, does not provide counting for the number of *executed* instructions. In fact, this is one of the big issues in using PMU to assist DVS scheduling — the CPU events may not be compatible nor consistent across different hardware platforms. This is also why Choi et al. presented two platform-dependent implementations [7, 8] of the same DVS algorithm [7].

## 6 Conclusions and Future Work

In this paper we have proposed a new PMU-assisted interval-based DVS algorithm that detects the CPU-boundedness of a program on the fly and adjust the CPU speed accordingly. The algorithm is no arbitrary heuristic. It is based on an extension of the previous theoretical work for energy-optimal DVS scheduling problem. The algorithm has also been proved to be effective in comparison with a number of DVS algorithms through physical measurements. That is, the new algorithm provides fine-grained, tight control over performance loss as well as exploits the sublinear per-

| **program** | $\beta$ | *hsu(train)* | *hsu(ref)* | *beta* |
|---|---|---|---|---|
| swim | 0.02 | 1.01/0.75 | 1.04/0.59 | 1.04/0.61 |
| tomcatv | 0.24 | 1.03/0.70 | 1.06/0.60 | 1.00/0.85 |
| hydro2d | 0.19 | 1.03/0.75 | 1.03/0.79 | 1.02/0.84 |
| su2cor | 0.27 | 1.01/0.88 | 1.02/0.83 | 1.03/0.85 |
| applu | 0.34 | 1.03/0.87 | 1.03/0.87 | 1.04/0.85 |
| apsi | 0.37 | 1.03/0.85 | 1.04/0.91 | 1.05/0.83 |
| mgrid | 0.51 | 1.01/1.00 | 1.01/1.00 | 1.03/0.89 |
| wave5 | 0.52 | 1.00/1.00 | 1.00/1.00 | 1.04/0.87 |
| turb3d | 0.79 | 1.04/0.95 | 1.04/0.95 | 1.05/0.94 |
| fpppp | 1.00 | 1.00/1.00 | 1.00/1.00 | 1.06/0.95 |

Table 3: The comparison of our new on-line DVS algorithm *beta* with an off-line DVS algorithm *hsu*. Each table entry is in the format of *relative-time/relative-energy* with respect to the total execution time and system energy usage when running the application at the highest setting throughout the entire execution.

formance slowdown. Finally, the algorithm is simple to implement.

Our new DVS algorithm can be refined in various ways. One particular direction is to use compiler hints as additional scheduling support. While this idea is not new (e.g., [36, 34, 37]), the type of hint that the compiler should provide so that the overall DVS algorithm is effective is still a research topic for general-purpose systems. To relieve the compiler from the difficulty of giving exact timing information off line, we could have the compiler to simply identify and distinguish execution phases of a program in terms of CPU-boundedness in an approximate manner. Algorithm *beta* can then be refined to compute the $\beta$ value for each of these phases in the hope to further improve its effectiveness for memory-bound programs.

# References


[1] T. Martin and D. Siewiorek. Nonideal battery and main memory effects on cpu speed setting for low power. *IEEE Transactions on Very Large Scale Integration (VLSI) System*, 9(1):29–34, February 2001.

[2] J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2001.

[3] A. Miyoshi, C. Lefurgy, E. Hensbergen, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS)*, June 2002.

[4] W. Kim, J. Kim, and S. Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.

[5] Andreas 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 (CASES)*, August 2002.

[6] 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 Conference on Programming Languages Design and Implementation (PLDI)*, June 2003.

[7] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ration of off-chip access to on-chip computation time. In *Design Automation and Test in Europe (DATE)*, February 2004.

[8] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.

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

[10] P. Stanley-Marbell, M. Hsiao, and U. Kremer. A hardware architecture for dynamic performance and energy adaptation. In *Workshop on Power-Aware Computer Systems (PACS'02)*, 2002.

[11] H. Li, C.-Y. Cher, T. Vijaykumar, and K. Roy. VSV: L2-miss-driven variable supply-voltage scaling for low power. In *The 36th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2003.

[12] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *IEEE Annual Symposium on Foundations of Computer Science*, October 1995.

[13] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 1998.

[14] A. Sinha and A. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the 14th International Conference on VLSI Design*, January 2001.

[15] A. Varma, B. Ganesh, M. Sen, S. Choudhary, L. Srinivasan, and B. Jacob. A control-theoretic approach to dynamic voltage scaling. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, October 2003.

[16] K.-Y. Mun, D.-W. Kim, D.-H. Kim, and C.-I. Park. dDVS: An efficient dynamic voltage scaling algorithm based on the

differential of CPU utilization. In *The 9th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, September 2004.

[17] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of 1998 International Symposium on Low Power Electronics and Design (ISLPED)*, August 1998.

[18] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[19] J. Lorch and A. Smith. Operating system modifications for task-based speed and voltage scheduling. In *The First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[20] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design*, June 2000.

[21] B. Childers, H. Tang, and R. Melhem. Adapting processor supply voltage to instruction-level parallelism. In *Kool Chips Workshop*, December 2000.

[22] F. Xie, M. Martonosi, and S. Malik. Compile time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, June 2003.

[23] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *The 36th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2003.

[24] C. Gniady, Y. Hu, and Y.-H. Lu. Program counter based techniques for dynamic power management. In *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2004.

[25] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. FAST: Frequency-aware static timing analysis. In *The 24th IEEE International Real-Time Systems Symposium (RTSS)*, December 2003.

[26] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS)*, November 2000.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[28] N. Hirofumi, N. Naoya, and T. Katsuya. WT210/WT230 digital power meters. Yokogawa Technical Report 35, 2003.

[29] The Standard Performance Evaluation Corporation. http://www.spec.org.

[30] Sun Fire B100x Blade Server. http://www.sun.com/servers/entry/b100x/.

[31] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC)*, June 2000.

[32] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compiler and Operating Systems for Low Power (COLP)*, October 2000.

[33] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward the placement of power management points in real time applications. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP)*, September 2001.

[34] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints in the COPPER framework. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2002.

[35] C.-H. Hsu. *Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction*. PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, June 2003.

[36] Cooperative voltage scaling (CVS) between OS and applicationsfor low-power real-time systems. In *IEEE Custom Integrated Circuits Conference (CICC)*, May 2001.

[37] N. AbouGhazaleh, D. Mossé, B. Childers, R. Melhem, and M. Craven. Collaborative operating system and compiler power management for real-time applications. May 2003.

# Appendix

**Theorem 3** *If* $T_1 > T_2 > \cdots > T_n$ *and* $0 \geq \frac{E_2-E_1}{T_2-T_1} \geq \frac{E_3-E_2}{T_3-T_2} \geq \cdots \geq \frac{E_n-E_{n-1}}{T_n-T_{n-1}}$, *then*

$$t_i^* = \begin{cases} \frac{D-T_{j+1}}{T_j-T_{j+1}} \cdot T_j & i = j \\ D - t_j^* & i = j+1 \\ 0 & \text{otherwise} \end{cases}$$

*where*

$$T_{j+1} < D \leq T_j$$

**Proof** (Sketch) To facilitate the proof, we define $r_i = t_i/T_i$ and introduce a new function $E_{min}(d)$ as follows.

$$E_{min}(d) = \min\{\sum_i r_i \cdot E_i : \sum_i r_i \cdot T_i = d, \sum_i r_i = 1, r_i \geq 0\}$$

If sequence $\{\frac{E_{i+1}-E_i}{T_{i+1}-T_i}\}$ is non-increasing, then function $E_{min}(d)$ is equivalent to the piecewise-linear function that connects points $\{(T_i, E_i)\}$. Since the slopes of chords in this piecewise-linear function are all non-positive, $E_{min}(d)$ is non-increasing. Thus, we seek for the solution $\{r_i\}$ of $E_{min}(D)$ as $E_{min}(D) = \min\{E_{min}(d) : d \leq D\}$. For $T_{j+1} < D \leq T_j$, $E_{min}(D)$ is the function value at $D$ in the chord connecting points $(T_j, E_j)$ and $(T_{j+1}, E_{j+1})$. The proof is completed by solving the linear system of $t_j^* + t_{j+1}^* = D$ and $t_j^*/T_j + t_{j+1}^*/T_{j+1} = 1$.

**Theorem 4** *If* $T_i = T(f_i) = \frac{c_1}{f} + c_0$, $c_1 \neq 0$ *and* $\frac{P_1-0}{f_1-0} \leq \frac{P_2-P_1}{f_2-f_1} \leq \frac{P_3-P_2}{f_3-f_2} \leq \cdots \leq \frac{P_n-P_{n-1}}{f_n-f_{n-1}}$, *then*

$$0 \geq \frac{E_2-E_1}{T_2-T_1} \geq \frac{E_3-E_2}{T_3-T_2} \geq \cdots \geq \frac{E_n-E_{n-1}}{T_n-T_{n-1}}$$

**Proof**

$$\frac{E_i-E_{i-1}}{T_i-T_{i-1}} - \frac{E_{i+1}-E_i}{T_{i+1}-T_i} = f_i \cdot \left(\frac{P_{i+1}-P_i}{f_{i+1}-f_i} - \frac{P_i-P_{i-1}}{f_i-f_{i-1}}\right)$$

$$+ f_i \cdot \frac{c_0}{c_1} \cdot \left(\frac{P_{i+1}-P_i}{f_{i+1}-f_i} \cdot f_{i+1} - \frac{P_i-P_{i-1}}{f_i-f_{i-1}} \cdot f_{i-1}\right) \geq 0$$

and

$$\frac{E_{i+1}-E_i}{T_{i+1}-T_i} = \frac{f_i f_{i+1}}{f_i - f_{i+1}} \cdot \left[\left(\frac{P_{i+1}}{f_{i+1}} - \frac{P_i}{f_i}\right) + \frac{c_0}{c_1}(P_{i+1}-P_i)\right] \leq 0$$