In the Proceedings of ACM/IEEE SC|05: Int'l Conf. on High-Performance Computing, Networking, and Storage
Seattle, WA, November 2005.

A Power-Aware Run-Time System for High-Performance Computing*

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

ABSTRACT

For decades, the high-performance computing (HPC) community has focused on performance, where performance is defined as speed. To achieve better performance per compute node, microprocessor vendors have not only doubled the number of transistors (and speed) every 18-24 months, but they have also doubled the power densities. Consequently, keeping a large-scale HPC system functioning properly requires continual cooling in a large machine room, thus resulting in substantial operational costs. Furthermore, the increase in power densities has led (in part) to a decrease in system reliability, thus leading to lost productivity.

To address these problems, we propose a power-aware algorithm that automatically and transparently adapts its voltage and frequency settings to achieve significant power reduction and energy savings with minimal impact on performance. Specifically, we leverage a commodity technology called "dynamic voltage and frequency scaling" to implement our power-aware algorithm in the run-time system of commodity HPC systems.

1. MOTIVATION

The notion of power-aware (or low-power) computing is *not* new, particularly in the areas of embedded systems and mobile computing [3, 4, 8, 9, 10, 13, 15, 17, 18, 19, 20, 23, 24, 26, 27, 28] where reducing energy consumption is critical in extending battery life. Laptops, for example, use simple power-aware algorithms that are based only on CPU (i.e., processor) utilization [10], making them ideal for interactive use. That is, if a laptop user is reading a document for an extended period of time while running on battery power, the laptop would automatically scale down the frequency and supply voltage of the CPU in order to reduce power consumption, as power consumption is proportional to the CPU frequency and to the square of the CPU supply voltage. The commodity technology that enables the above scaling of frequency and voltage for CPUs is called *dynamic voltage and frequency scaling (DVFS)*.[1] In contrast, the notion of power awareness (or low power) *is* new to the high-performance computing (HPC) community.

Why the above distinction? First, the computational characteristics found in embedded systems and mobile computing, e.g., laptop, differ markedly from those found in HPC. As a result, the power-aware algorithms that work well for the interactive use found in laptops fail miserably with respect to scientific applications [11]. Second, power awareness is needed for different reasons. In embedded and mobile computing, power awareness is needed to extend battery life; whereas in HPC, it is needed to reduce the operational costs of powering and cooling HPC systems as well as to improve reliability.

The issue of reliability in large-scale HPC systems is particularly insidious. For example, Table 1 shows the current reliability of leading-edge supercomputers [21]. With power densities doubling every 18-24 months (Figure 1) and large-scale HPC systems con-

System	CPUs	Reliability
ASCI Q	8,192	**MTBI: 6.5 hrs.** HW outage sources: storage, CPU , memory.
ASCI White	8,192	**MTBF: 5 hrs ('01) and 40 hrs ('03).** HW outage sources: storage, CPU, 3rd-party HW.
PSC Lemieux	3,016	**MTBI: 9.7 hours.**
Google	15,000	**20 reboots/day; 2-3% machines replaced/year.** HW outage sources: storage, memory.

MTBF/I: mean time between failures/interrupts

Table 1: Reliability of Leading-Edge Supercomputers.

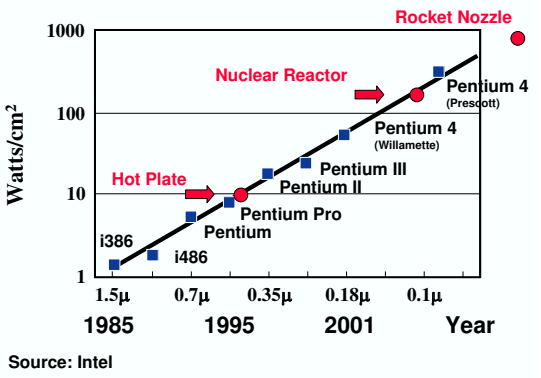


Figure 1: Moore's Law for Power Consumption.

*This work was supported by the DOE LDRD Exploratory Research Program through Los Alamos National Laboratory contract W-7405-ENG-36 and by Advanced Micro Devices, Inc. Available as LANL technical report: LA-UR 05-5650.

[1] AMD refers to their DVFS mechanism as PowerNow! while Intel refers to theirs as SpeedStep.

SC|05 November 12–18, 2005, Seattle, Washington, USA.

Service	Cost of One Hour of Downtime
Brokerage Operations	$6,450,000
Credit Card Authorization	$2,600,000
eBay	$225,000
Amazon.com	$180,000
Package Shipping Services	$150,000
Home Shopping Channel	$113,000
Catalog Sales Center	$90,000

Table 2: Estimated Costs of an Hour of System Downtime.

tinuing to increase in size, the amount of heat generated (and hence, temperature) continues to rise. And as a rule of thumb, Arrenhius' equation as applied to microelectronics notes that for every 10°C (18°F) increase in temperature, the failure rate of a system doubles.

Our own informal empirical data, taken from late 2000 to early 2002, supports Arrenhius' equation. In the winter, when the temperature inside our warehouse-based work environment was around 70-75°F, our traditional cluster — *Little Blue Penguin (LBP)* — failed approximately once a week; in the summer, when the temperature increased to 85-90°F, the cluster failed twice a week.

Even more worrisome is how our computing environment affected the results of the Linpack benchmark running on a very dense, 18-node Beowulf cluster. After ten minutes of execution, the cluster produced an answer outside the residual (i.e., a silent error) when running in our dusty 85°F warehouse but produced the correct answer when running in a 65°F machine-cooled room. Clearly, the HPC community must worry about power and its effect on reliability.

Furthermore, every hour that an HPC system is unavailable translates to lost business or lost productivity. This issue is of extraordinary importance for companies that rely on parallel-computing resources for their business, as noted in Table 2 [1].

Therefore, to address the above issues, we started the *Supercomputing in Small Spaces (SSS)* project (http://sss.lanl.gov/) in 2001. The first major instantiation of the SSS project was a 240-CPU energy-efficient cluster called *Green Destiny*. This Linux-based cluster possessed a footprint of only five square feet and sipped as little as 3.2 kW of power (i.e., two hairdryers). It produced 101 Gflops on the Linpack benchmark, which was as fast as a 256-CPU SGI Origin 2000, as shown at http://www.top500.org/list/2001/11/. Despite its admirable performance at the time, many still felt that Green Destiny sacrificed too much performance to achieve low power consumption and high reliability, i.e., no unscheduled downtime in its 24-month lifetime while running at 7,400 feet above sea level in a dusty 85°F warehouse without any cooling, air filtration, or air humidification.

To simultaneously address the performance issue as well as create a general power-aware solution that works on any commodity platform that supports DVFS, we propose a power-aware algorithm called the β-adaptation algorithm, implement the algorithm in the run-time system, and evaluate its performance on commodity HPC platforms, both uniprocessor and multiprocessor. The end result is a power-aware run-time (PART) system that transparently and automatically adapts CPU voltage and frequency in order to reduce power consumption (and energy usage) while minimizing impact on performance.

2. RELATED WORK

At the present time, there exists a handful of insightful case studies about the feasibility of employing DVFS to reduce the thermal power envelope of a high-performance computing (HPC) node, and thus, improve reliability, while minimizing impact on performance [2, 6]. With this knowledge that DVFS can indeed be effective in HPC, the next step is to develop various approaches that leverage DVFS. Such approaches include manual DVFS tuning [5, 7], compiler analysis with profiling [12], MPI library-based extensions, or an adaptive run-time system [11].

Manual DVFS tuning often involves profiling of the execution behavior of a program (or its structures) at all possible frequency-voltage settings. It can be as simple as recording the execution time of the program at each available CPU frequency, and then using the profile to select the lowest frequency that satisfies the performance constraint to execute the program. The feasibility studies of [2, 6] fall into this category.

However, this approach is very coarse-grained. For example, Figure 2 shows the profile for three frequency-voltage combinations on SPEC `tomcatv` benchmark. With a 5% performance-slowdown constraint in place, the figure indicates that there does not exist any DVFS setting that simultaneously reduces energy consumption and meets the 5% slowdown constraint. The 1.6GHz/1.3V and 1.2GHz/1.1V settings produce 6% and 22% performance slowdowns, respectively.

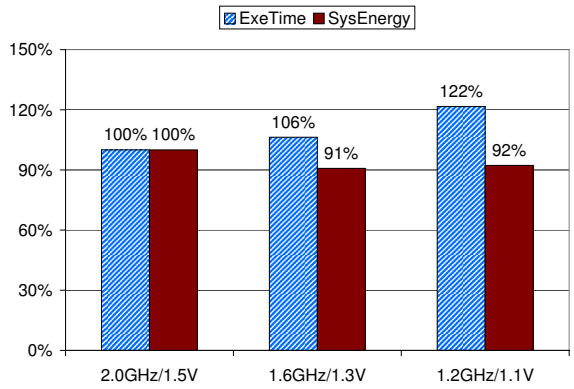


Figure 2: The performance-power profiles of `tomcatv`.

A more sophisticated approach to manual DVFS tuning is to look into the program structure of the code and profile each interesting program sub-structure for its execution behavior. For `tomcatv` whose program structure is shown in Figure 3, this means profiling the execution times of loop nests L1 to L9 at each CPU frequency as `tomcatv` executes a sequence of nested loops. The `tomcatv` benchmark spends most of its execution time executing loops L2 to L8 iteratively, with the number of iterations controlled by the variable ITACT in the code. In [5], Freeh et al. used this approach to select the CPU frequency to run for each loop nest and MPI call.

Figure 4 shows the execution times of the most time-consuming loops (i.e., L2, L5, L7, and L8) in `tomcatv`. For readability, we normalize all loop execution times with respect to the execution time of the `entire` benchmark running at 2.0 GHz.[2] The figure indicates that with a 5% performance-slowdown constraint, there exist many scheduling options. For example, we can execute loop L2 at 1.6 GHz, resulting in a 3% slowdown; or we can execute loops L5, L7, and L8 at 1.2GHz.

[2]That is, at the 2.0GHz/1.5V setting, 32% of the execution time is spent in loop L2, 24% in L5, 18% each in L7 and L8, and 8% in the remaining loops in total.

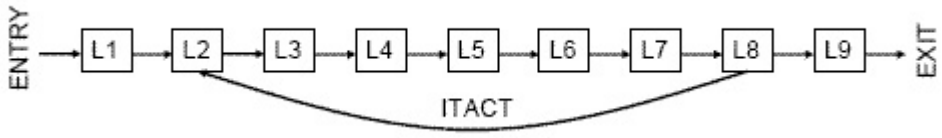


Figure 3: The program structure of SPEC benchmark `tomcatv`.

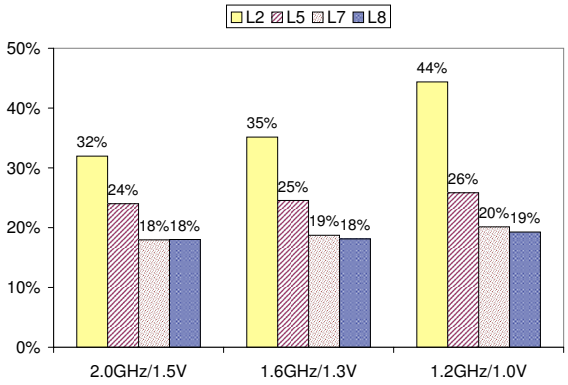


Figure 4: The execution-time profile of `tomcatv`.

Though the above approach for manual DVFS tuning is straightforward, it can be quite tedious, especially as the number of valid frequency-voltage settings increases and the program structure becomes more complex. Consequently, due to the complexity of manual DVFS tuning for large-scale applications such as climate modeling, automated profiling and subsequent profile analysis is often desired. In [12], Hsu and Kremer propose such an implementation based on compiler techniques. In their implementation, compiler techniques such as control-flow-graph analysis are used to deal with the time overheads caused by setting the CPU frequency and voltage as each such setting takes on the order of milliseconds. For `tomcatv`, their software chooses to slow down loops L6, L7, and L8 to 1.2 GHz.

The problems with the aforementioned approaches are threefold. First, they are all essentially profile-based and generally require the source code to be modified. As a result, these approaches are *not* completely transparent to the end user. Second, because the profile information can be influenced by program input, these approaches are input-dependent. Third, as noted in [12], the instrumentation of source code may alter the instruction access pattern, and therefore, may produce profiles that are considerably different from the execution behavior of the original code. So, in theory, while these approaches might provide maximal benefit relative to performance and power, they are of little use to end-user applications. Therefore, we believe that there exists a need for a transparent and self-adapting run-time system for power awareness.

The current approach towards an adaptive run-time system for power awareness is based primarily on CPU utilization, e.g., cpuspeed on laptops. For cpuspeed, when CPU utilization is below some threshold, the CPU voltage and frequency are lowered to conserve energy; when the CPU utilization exceeds some threshold, the CPU voltage and frequency are raised to improve performance. Although this simple approach is both application- and input-independent, it is only effective for interactive use, e.g., laptop usage of Microsoft Office, and depends critically upon the choice of the threshold values [10]. For scientific applications, its effectiveness is abysmal as such applications do not have an abundance of CPU idle time [11].

Other more sophisticated approaches based on CPU utilization such as those in [25] only provide loose control over DVFS-induced performance slowdown, e.g., 37% slowdown with only 12% system energy savings for the SPEC `go` benchmark, because the CPU utilization ratio by itself does not provide enough timing information. Therefore, we conclude that there exists a need for a power-aware run-time system that has tight performance-slowdown control and can deliver considerable energy savings.

3. β-ADAPTATION ALGORITHM FOR A POWER-AWARE RUN-TIME SYSTEM

Leveraging the DVFS mechanism, we propose an automatically-adapting, power-aware algorithm that is transparent to end-user applications and can deliver considerable energy savings with tight control over DVFS-induced performance slowdown. Performance slowdown in this paper is defined as the increase in relative execution time with respect to the execution time when the program is running at the peak CPU speed. A user can specify the maximum allowed performance slowdown δ (e.g., δ = 5%), and our algorithm will schedule CPU frequencies and voltages in such a way that the actual performance slowdown does not exceed δ.

Our power-aware algorithm, which we call the *β-adaptation algorithm* for reasons that will become apparent later, is an interval-based scheduling algorithm, i.e., scheduling decisions are made at the beginning of time intervals of the same length (e.g., every second). Interval-based algorithms are generally easy to implement because they make use of existing "alarm clock" functionality found in the operating system. By default, our power-aware algorithm (and its software realization as part of the run-time system) sets the interval length to be one second. However, the algorithm allows a user to change this value per program execution. The value is denoted as I hereafter.

In contrast to previous approaches, we want to ensure that our power-aware algorithm does not require any application-specific information a priori, e.g., profiling information, and more generally, that it is transparent to end-user applications. Therefore, it must implicitly gather such information, for example, by monitoring the intensity level of off-chip accesses during each interval I in order to make smart scheduling decisions. Intuitively, when the intensity level of off-chip accesses is high, it indicates that program execution is in a non-CPU-intensive phase, hence indicating that this phase can execute at a lower CPU frequency (and voltage) without affecting its performance.

While conceptually simple, this type of algorithm must overcome the following obstacle in order to be effective: The quantification of the intensity level of off-chip accesses needs to have a direct correlation between CPU frequency changes and execution-

time impact; otherwise, the tight control of DVFS-induced performance slowdown will be difficult to achieve. For example, one might think that the high cache-miss rate is a suitable indicator that program execution is in a non-CPU-intensive phase. But unless we can predict how the execution time will be lengthened for every lower CPU frequency that may be executed in this non-CPU-intensive phase, the information of the high cache-miss rate will *not* help in the selection of the appropriate CPU frequency to maintain tight control of DVFS-induced performance slowdown. *Therefore, we need a model that associates the intensity level of off-chip accesses with respect to total execution time.*

To overcome the above problem, we propose a model that is based on the MIPS rate (i.e., millions of instructions per second) which can correlate the execution-time impact with CPU frequency changes:

$$\frac{T(f)}{T(f_{max})} \approx \frac{\texttt{mips}(f_{max})}{\texttt{mips}(f)} \approx \beta\left(\frac{f_{max}}{f} - 1\right) + 1. \qquad (1)$$

The leftmost term $\frac{T(f)}{T(f_{max})}$ represents the execution-time impact of running at CPU frequency f in terms of the relative execution time with respect to running at the peak CPU frequency f_{max}. The rightmost term $\beta\left(\frac{f_{max}}{f} - 1\right) + 1$ introduces a parameter, called β, that quantifies the intensity level of on-chip accesses (and indirectly, off-chip accesses). By definition, $\beta = 1$ indicates that execution time doubles when the CPU speed is halved, whereas $\beta = 0$ means that execution time remains unchanged no matter what CPU speed will be used. Finally, the middle term $\frac{\texttt{mips}(f_{max})}{\texttt{mips}(f)}$ provides a way to describe the observed execution-time impact and will be used to adjust the value of β.

Ideally, if we knew the value of β a priori, we could use Equation (1) to select an appropriate CPU frequency to execute in the current interval such that the DVFS-induced performance slowdown is tightly constrained. (The selection of this CPU frequency will be presented later.) But because we want ensure that our power-aware algorithm does not require any application-specific information a priori, β is *not* known a priori. Therefore, the challenge for our automatically-adapting, power-aware algorithm lies in the "on-the-fly" estimation of β at run time, and hence, leads us to name our power-aware algorithm as the β-adaptation algorithm.

To estimate β at run time, we use a regression method over Equation (1) and leverage the fact that most DVFS-enabled microprocessors support a limited set of CPU frequencies to perform the regression. That is, given n CPU frequencies $\{f_1, \cdots, f_n\}$, we derive a particular β value that will minimize the least-squared error:

$$\min \sum_{i=1}^{n} \left\| \frac{\texttt{mips}(f_{max})}{\texttt{mips}(f_i)} - \beta\left(\frac{f_{max}}{f_i} - 1\right) - 1 \right\|^2 \qquad (2)$$

By equating the first differential of (2) to zero, we can derive β as a function of the MIPS rates and CPU frequencies, as follows:

$$\beta = \frac{\sum_{i=1}^{n} (\frac{f_{max}}{f_i} - 1)(\frac{\texttt{mips}(f_{max})}{\texttt{mips}(f_i)} - 1)}{\sum_{i=1}^{n} (\frac{f_{max}}{f_i} - 1)^2} \qquad (3)$$

Once we calculate the value of β using Equation (3), we can plug the value into Equation (1) and calculate the lowest CPU frequency f whose predicted performance slowdown $\beta\left(\frac{f_{max}}{f} - 1\right)$ does not exceed the maximum possible performance slowdown δ. Mathematically, this establishes the following relationship: $\delta = \beta\left(\frac{f_{max}}{f} - 1\right)$. By solving this equation for f, we determine the desired frequency f^* that the CPU should run at:

$$f^* = \max\left(f_{min}, \frac{f_{max}}{1 + \delta/\beta}\right) \qquad (4)$$

Conglomerating the aforesaid theory results in the β-adaptation algorithm shown in Figure 5. In essence, this power-aware algorithm wakes up every I seconds. The algorithm then calculates the value of β using the most up-to-date information on the MIPS rate based on Equation (3). Once β is derived, the algorithm computes the CPU frequency f^* for the interval based on Equation (4). Since a DVFS-enabled microprocessor only supports a limited set of frequencies, the computed frequency f^* may need to be emulated in some cases. (The emulation scheme is shown in Figure 6. The ratio r denotes the percentage of time to execute at frequency f_j.) This sequence of steps is repeated at the beginning of each subsequent interval until the program executes to completion.

Hardware:

n frequencies $\{f_1, \cdots, f_n\}$.

Parameters:

I: the time-interval size (default 1 sec).

δ: slowdown constraint (default 5%).

Algorithm:

Initialize $\texttt{mips}(f_i)$, $i = 1, \cdots, n$, by executing the program at f_i for I seconds.

Repeat

1. Compute coefficient β.
$$\beta = \frac{\sum_i (\frac{f_{max}}{f_i} - 1)(\frac{\texttt{mips}(f_{max})}{\texttt{mips}(f_i)} - 1)}{\sum_i (\frac{f_{max}}{f_i} - 1)^2}$$
2. Compute the desired frequency f^*.
$$f^* = \max\left(f_{min}, \frac{f_{max}}{1 + \delta/\beta}\right)$$
3. Execute the current interval at f^*.
4. Update $\texttt{mips}(f^*)$.

Until the program is completed.

Figure 5: β-Adaptation Algorithm for a Power-Aware Run-Time System.

To extend the β-adaptation algorithm from the uniprocessor environment that is implicitly assumed above to a multiprocessor environment, we simply replicate the algorithm onto each processor and run each local copy *asynchronously*. We adopt this strategy for the following reasons. First, the intensity level of off-chip accesses is a per-processor metric. Second, a coordination-based power-aware algorithm would need extra communication, and likely, synchronization — both of which add to the overhead costs (in terms of performance and energy) of running the power-aware algorithms. And as we will see in Section 5.2, the β-adaptation algorithm running asynchronously on each processor is quite effective in saving energy while minimizing impact on performance.

In summary, our β-adaptation algorithm is a power-aware and interval-based algorithm that is parameterized by two user-tunable variables: the maximum performance-slowdown constraint δ and the interval length I. The default values of which are 5% and one

3. Perform the following steps:

   (a) Figure out f_j and f_{j+1}.

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

   (b) Compute the ratio r.

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

   (c) Run $r \cdot I$ seconds at frequency f_j.

   (d) Run $(1 - r) \cdot I$ seconds at frequency f_{j+1}.

Figure 6: Step 3 of β-Adaptation Algorithm.

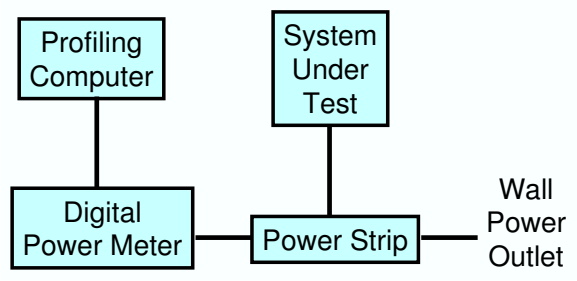


Figure 7: The Experimental Setup.

second, respectively. To facilitate an empirical evaluation of the effectiveness of this algorithm, we implement it in the run-time system, thus creating a power-aware run-time (PART) system. We then test the PART system on uniprocessor and multiprocessor platforms using appropriate benchmark suites, as discussed in Section 4.

4. EVALUATION METHODOLOGY AND EXPERIMENTAL SETUP

In this section, we describe the evaluation methodology and experimental setup that we use to evaluate the effectiveness of our β-adaptation scheduling algorithm.

4.1 Evaluation Methodology

To measure the execution time of a program, we use the global-time query functions provided by the operating system. In this paper, the execution time is referred to as the wall clock time of program execution.

The energy consumption of a program execution is often measured via a power meter. In our experiments, the power meter is connected to a power strip that passes electrical energy from the wall power outlet to the system under test, as shown in Figure 7. The power meter periodically samples the instantaneous system wattage, and the *total system energy consumption* is then calculated as the integration of these wattages over time. Specifically, we use a Yokogawa WT210 power meter whose sampling rate is 20 μs per sample and note that the power meter is also capable of performing the aforementioned integration internally.

Unfortunately, evaluating DVFS scheduling algorithms based on total system energy savings can be misleading since DVFS only affects CPU energy consumption. Because the percentage of CPU energy consumption, relative to the total system energy usage, can vary widely from platform to platform, the evaluation results become platform-dependent.

For example, consider two DVFS scheduling algorithms, each of which is able to reduce the total system energy by 9%. Intuitively, the two algorithms might be considered equally effective. However, what if one algorithm was evaluated on a HPC server where the CPU accounts for 30% of the total system energy usage while the other algorithm was evaluated on a high-performance laptop computer where the percentage increases to 60%? Back of the envelope calculations[3] show that the former algorithm reduces CPU energy by 30% and the latter algorithm reduces CPU energy by 15%. Clearly, the former algorithm is more effective than the latter algorithm. This example illustrates that using the platform-dependent, total system energy savings prohibits us from comparing DVFS algorithms evaluated on different platforms. Therefore, in this paper, we evaluate the effectiveness of our β-adaptation algorithm based on CPU energy savings.

Unfortunately, direct measurements of CPU energy consumption present technical challenges. A common but obtrusive method is to place a shunt resistor in series with the microprocessor chip and its input power supply. The power meter is connected to this shunt resistor in order to measure the energy used by the microprocessor [22]. However, obtrusive methods based on shunt resistors are argued to be less appropriate because shunt resistors interfere with operation of the system under test and unsuitable when there are large variations in current [14].

In this paper, we use an unobtrusive method to estimate the CPU energy consumption. We leverage a first-order power model for the CPU [16] and divide the sampled system wattage from the power meter into two parts:

$$P_{sys}(f, V) = \underbrace{C \cdot V^2 \cdot f}_{\text{the CPU power}} + P_{base} \quad (5)$$

The first term in the system wattage (P_{sys}) equation represents the CPU power consumption and depends on the current voltage V and CPU frequency f.[4] The second term (P_{base}) is independent of voltage and frequency and captures the power consumption of system components that are *not* driven by CPU clocks.

To estimate the CPU energy consumption for a given application-input pair, we perform a least-squared regression on Equation (5) with observation data derived from executing the application-input pair at each possible frequency-voltage combination. This simple approach turns out to be quite accurate when using the R-squared metric. The R-squared metric indicates the relative predictive power of a model; its range is between zero and one, inclusive. The closer the R-squared metric is to one, the more predictive that Equation (5) is. For all the benchmarks that we ran in this paper, R-squared is very close to one. Therefore, we adopt this unobtrusive approach to estimate CPU energy consumption in order to derive CPU energy savings that the β-adaptation algorithm can deliver.

4.2 Systems Under Test

In this section, we detail the hardware and software that we used for the performance evaluation of the β-adaptation algorithm in our power-aware run-time (PART) system. We begin by presenting the configurations of the uniprocessor and multiprocessor hardware platforms under test. Then, we describe the systems software on these platforms, followed by information about our implementation of the β-adaptation algorithm. Finally, we list the set of se-

[3]For the first algorithm, the CPU energy savings is calculated as $\frac{9\%}{30\%} = 30\%$; for the second algorithm, the savings is $\frac{9\%}{60\%} = 15\%$.
[4]The constant C in $C \cdot V^2 \cdot f$ denotes the switched capacitance which caused the energy to be consumed. C is application- and input-dependent.

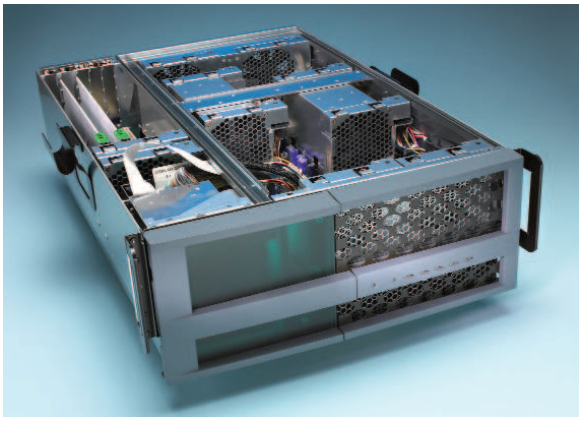


Figure 8: Celestica A8440.

f (GHz)	V
0.8	0.9
1.6	1.3
1.8	1.4
2.0	1.5

Table 3: The Operating Points of Our Tested Computer Systems.

quential and parallel benchmarks that we used for the evaluation of our β-adaptation algorithm.

The tested uniprocessor platform is based on an Asus K8V Deluxe motherboard that is bundled with an AMD Athlon64 3200+ processor (with 1-MB L2 cache) and 1-GB DDR-400 main memory. The tested multiprocessor platforms include a cluster of four of the above Athlon64-based compute nodes connected via Gigabit Ethernet and another four-node quad-CPU cluster based on the Celestica A8440 server. As shown in Figure 8, the Celestica A8440 server is a 4U server with four AMD Opteron 846 processors (and also 1-MB L2 cache per processor) and 4-GB DDR-333 main memory. This Opteron-based cluster is also connected via Gigabit Ethernet.

In our experiments, both Athlon64 3200+ and Opteron 846 processors can execute from 800 MHz at 0.9 V to 2 GHz at 1.5 V. Table 3 lists the four valid operating points (i.e., frequency-voltage pairs) that our β-adaptation algorithm can set during program execution. In theory, an Athlon64 3200+ processor can support clock frequencies from 800 MHz to 2 GHz at an increment of 200 MHz. So, why are only four operating points used? In practice, the set of CPU frequencies that can transition to each other directly (i.e., without intermediate frequencies) in an Athlon64 3200+ processor is restricted. Since the time overhead for a direct frequency-voltage transition is already on the order of milliseconds (as shown in Figure 9), we restrict ourselves to use only a subset of supported CPU frequencies that have direct transitions to each other. For other frequencies, we emulate them using the algorithm in Figure 6.

The operating system on the tested hardware platforms is SuSE Linux 2.6.7. This Linux distribution comes with GNU compilers 3.3.3, a DVFS interface called `cpufreq`, and a DVFS kernel module called `powernow-k8`. The `cpufreq` interface allows our β-adaptation algorithm to set a desired CPU frequency by writing the frequency to a particular `/sys` file. We did not use the `powernow-k8` kernel module in the distribution; instead, we use a version of `powernow-k8` that is freely downloadable from

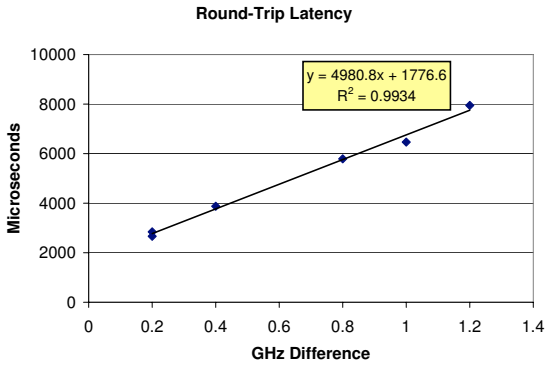


Figure 9: The Latency of Each Operating-Point Change.

the AMD website and allows us to specify Table 3 in the module. The end result is that whenever we write the target CPU frequency through the `cpufreq` interface, the CPU voltage associated with this frequency, as specified in Table 3, will also be set automatically in kernel space.

Our prototype implementation of the β-adaptation algorithm is has less than 500 lines of C code. The use of the implementation is similar to the use of a Unix `time` command. The implementation will fork two threads, one for the execution of the target program (as specified on the command line) and the other for the execution of Figure 5. Thus, our performance evaluation includes the time and energy overheads of running the β-adaptation algorithm on top of the normal program execution.

With respect to the benchmarks, we used the SPEC CFP95 and CPU2000 benchmarks for the uniprocessor platform and the latest NAS-MPI benchmarks, version 3.2, for the multiprocessor platforms. With the exception of SPEC CPU2000 benchmarks, all the other benchmarks were compiled using the GNU compiler 3.3.3 with optimization level `-O3`. The CPU2000 benchmarks were compiled using the Intel compiler 8.1 with the optimization level `-xW -ip -O3`. We used the Intel compiler, instead of the GNU compiler, because CPU2000 contains several FORTRAN-90 codes that the GNU compiler does not yet support. For the MPI benchmarks, LAM/MPI version 7.0.6 was used to run the benchmarks.

5. EXPERIMENTAL RESULTS

This section presents a performance evaluation of our β-adaptation algorithm as it is implemented in our power-aware run-time (PART) system. As implicitly noted earlier, we evaluate the PART system in both uniprocessor and multiprocessor environments.

5.1 Uniprocessor Platform

We first compare the performance of our automatically-adapting β-adaptation algorithm (that is running in our power-aware run-time system) to the compiler-based approach presented in [12] when running the SPEC CFP95 benchmarks. Although the CFP95 benchmarks have been retired for five years, they allow us to compare the results from our β-adaptation algorithm to previous case studies [11, 12]. We then evaluate the effectiveness of our power-aware run-time (PART) system when running the SPEC CPU200 benchmark suite.

5.1.1 SPEC CFP95 Benchmarks

Figure 10 shows a comparison of the actual performance slowdown between the run-time approach (denoted as `beta`) and the

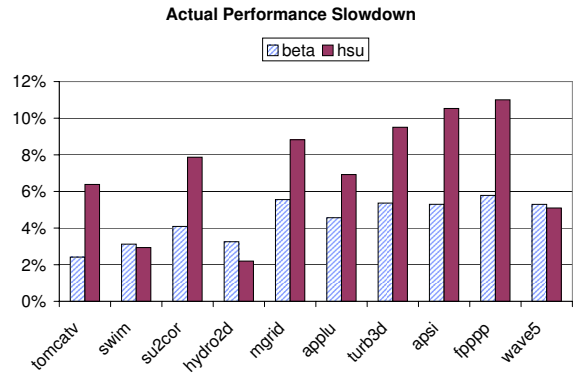


Figure 10: The Actual Performance Slowdown of the β-Adaptive Run-Time Approach versus the Compiler-Based Approach of Hsu et al.

compiler approach (denoted as `hsu`). Here we see that the actual performance slowdown induced by the compiler algorithm is poorly regulated given that the maximum performance-slowdown constraint was specified as 5%. In contrast, the β-adaptation algorithm, which is the foundation of our power-aware run-time (PART) system, regulates the actual performance slowdown much better.

Further investigation reveals that the benchmarks that cause the compiler approach to induce unacceptable performance slowdown (i.e., `mgrid`, `turb3d`, and `apsi`) have CPU-bound execution behavior. This implies that the β-adaptation algorithm for our PART system will perform more effectively on CPU-bound programs than the compiler approach will. Empirical results from a laptop computer [11] corroborate the above conclusion.

The effectiveness of our PART system is due to the validity of Equation (1). If we apply the least-squared regression on the equation using the *overall* execution time at various CPU frequencies for CFP95, we will see that R-squared is close to one for each CFP95 benchmark. In other words, Equation (1) is a good performance-prediction model for CFP95.

Relative to CPU energy reduction, previous studies such as [11, 12] report an average CPU energy reduction of 20% using the compiler approach on a laptop computer. In contrast, the average CPU energy savings for our uniprocessor HPC server platform using our automatically-adapting software is about 11%. The gap between the two energy-saving values is due to the difference in L2 cache size. For the laptop, the L2 cache size is only 256 KB whereas the L2 cache size for the HPC server is four times larger at 1 MB. Consequently, the intensity of *off-chip* accesses for the laptop is significantly higher than for the HPC server, thus providing substantially more opportunities for energy savings for the laptop.

5.1.2 SPEC CPU2000 Benchmarks

Here we evaluate the effectiveness of our PART system across the entire SPEC CPU200 benchmark suite. Figure 11 shows the actual performance slowdown and the CPU energy savings delivered by the PART system. The transparent and automatically-adapting β-adaptation algorithm in the PART system reduces the CPU energy consumption by 12% (on average) with only a 4% actual performance slowdown for SPEC CFP2000; for SPEC CINT2000, the two numbers are 9.5% and 4.8%, respectively. Because the average β values for CFP2000 and CINT2000 are 0.66 and 0.83, respectively, this means that CINT2000 is more CPU-bound than CFP2000, and therefore, has fewer opportunities for energy savings due to the correspondingly fewer off-chip accesses.

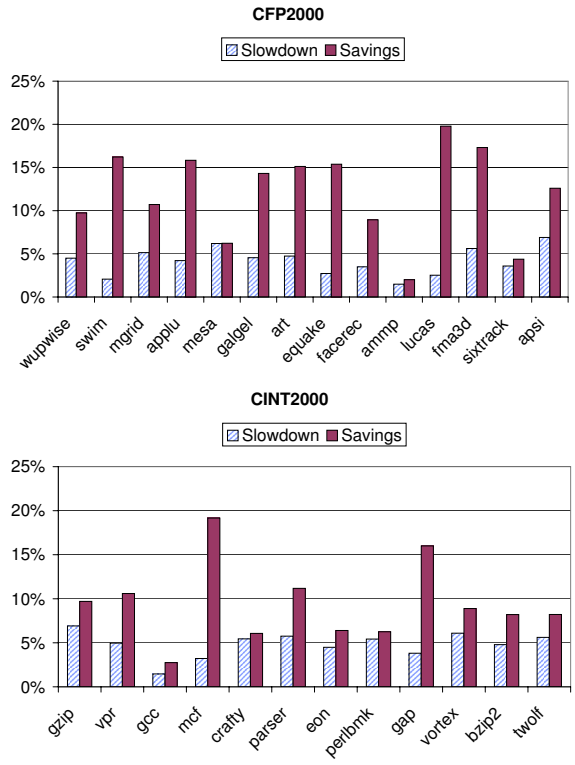


Figure 11: The Actual Performance Slowdown and CPU Energy Savings of CPU2000 Benchmarks Using Our PART System.

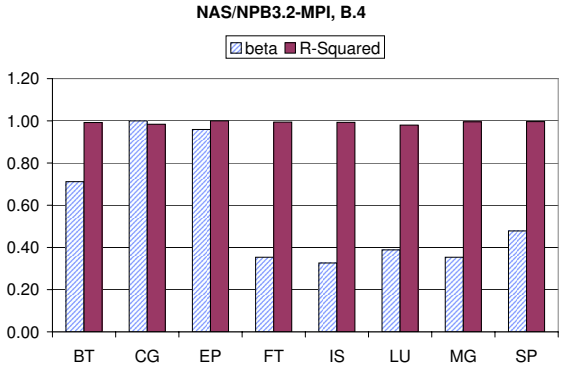
5.2 Cluster Platform

In this section, we present experimental results on an Athlon64-base cluster with four single-CPU nodes connected via Gigabit Ethernet as well as an Opteron-based cluster with four quad-CPU nodes connected via Gigabit Ethernet. For both clusters, the MPI implementation is LAM/MPI, version 7.0.6, and the benchmarks of choice is the latest NAS-MPI benchmarks, version 3.2.

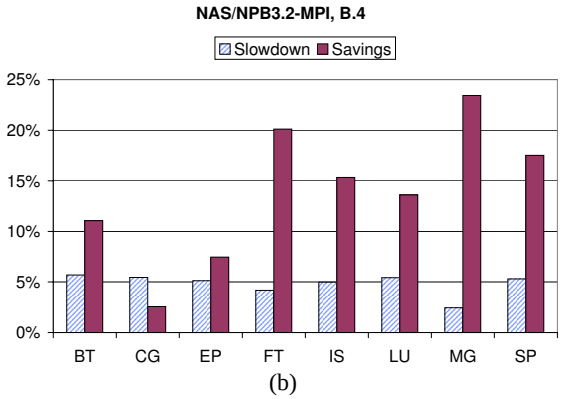
For our four-node Athlon64 cluster, Figure 12(a) shows the average β value for each of the eight NAS-MPI benchmarks as well as the associated R-squared metric for the class B workload. (Recall that the larger the β, the more CPU-bound the benchmark.) The β value of the benchmarks spans from 0.33 (IS benchmark) to 1.00 (CG and EP benchmarks) with an average value around 0.57. Compared to the SPEC CPU2000 benchmarks, the NAS-MPI benchmarks are generally less CPU-bound, which means more opportunities that can be exploited by the PART system for CPU energy reduction under the same performance-slowdown constraint δ.

Figure 12(b) shows the actual performance slowdown and CPU energy savings of NAS-MPI for the class B workload. On average, our PART system saves 14% CPU energy at 5% actual performance slowdown. For the class C workload, the average savings is about 12% at the cost of 4% actual performance slowdown, as shown in Figure 13.

For the Opteron-based cluster, Figure 14 shows that our PART system was able to save CPU energy ranging from 8% to 25%, with an average savings of 18%. The average actual performance slowdown is 3%.



(a)



(b)

Figure 12: NAS-MPI for Class B Workload on the Athlon64-Based Cluster.

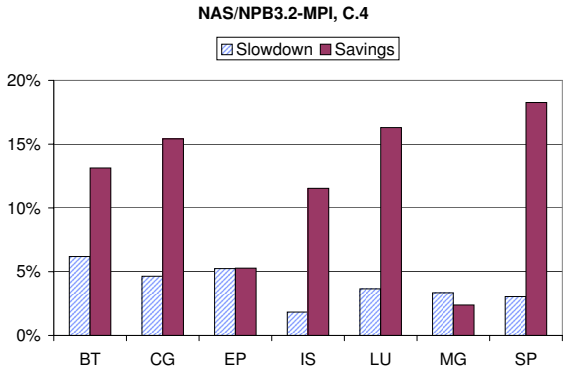


Figure 13: NAS-MPI for Class C Workload on the Athlon64-Based Cluster.

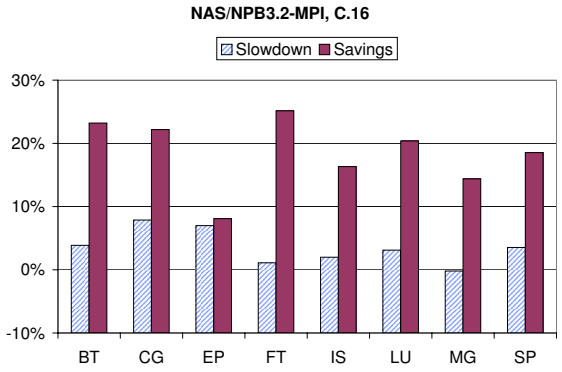


Figure 14: NAS-MPI for Class C Workload on the Opteron-based Cluster.

6. CONCLUSION

Power awareness has increasingly become an important issue in high-performance computing (HPC). In HPC, ignoring power consumption as a design constraint results in a system with high operational costs for power and cooling and can detrimentally impact reliability, which translates into lost productivity.

To address the above issues, we present a power-aware solution that works on any commodity platform that supports dynamic voltage and frequency scaling (DVFS). Specifically, we propose a power-aware algorithm called the β-adaptation algorithm and prototype an implementation of the algorithm as a power-aware runtime (PART) system. The PART system transparently and automatically adapts CPU voltage and frequency so as to reduce power consumption (and energy usage) while minimizing impact on performance. The performance evaluation on both uniprocessor and multiprocessor platforms show that the system achieves its design goal. That is, the system can save CPU energy consumption by as much as 20% for sequential benchmarks and 25% for parallel benchmarks that we tested, at a cost of 3-5% performance degradation. Moreover, the performance degradation was tightly controlled by our PART system for all the benchmarks.

7. ACKNOWLEDGEMENTS

First and foremost, we would like to thank Douglas O'Flaherty of AMD for his tremendous support of our research efforts. His contributions to the project were invaluable. We are also indebted to Paul Devriendt and Mark Langsdorf for providing technical details about PowerNow! on Opteron. Next, we acknowledge Western Scientific for building our Opteron-based cluster nodes and for providing technical support. Finally, we wish to recognize Jeremy S. Archuleta for his tireless efforts in building, configuring, and administering all the computing platforms that were used in this paper.

8. REFERENCES

[1] W. Feng. Making a case for efficient supercomputing. *ACM Queue*, 1(7):54–64, Oct. 2003.

[2] X. Feng, R. Ge, and K. Cameron. Power and energy profiling of scientific applications on distributed systems. *Proc. 19th Int'l Parallel & Distributed Processing Symp. (IPDPS 05)*, Apr. 2005.

[3] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. *Proc.7th*

Int'l Conf. Mobile Computing and Networking (MobiCom 01), July 2001.

[4] J. Flinn and M. Satyabarayanan. Energy-aware adaptation for mobile applications. *Proc. 17th Symp. Operating Systems Principles (SOSP 99)*, Dec. 1999.

[5] V. Freeh, D. Lowenthal, F. Pan, and N. Kappiah. Using multiple energy gears in MPI programs on a power-scalable cluster. *Proc. Symp. Principles and Practices of Parallel Programming (PPoPP 05)*, June 2005.

[6] V. Freeh, D. Lowenthal, R. Springer, F. Pan, and N. Kappiah. Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster. *Proc. 19th Int'l Parallel & Distributed Processing Symp. (IPDPS 05)*, Apr. 2005.

[7] R. Ge, X. Feng, and K. Cameron. Improvement of power-performance efficiency for high-end computing. *Proc. 1st Workshop High-Performance, Power-Aware Computing (HP-PAC 05)*, Apr. 2005.

[8] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. *Proc. 1st Int'l Conf. Mobile Computing and Networking (MobiCom 95)*, Nov. 1995.

[9] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and DVS processors. *Proc. Int'l Symp. Low-Power Electronics and Design (ISLPED 01)*, Aug. 2001.

[10] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. *Proc. 4th Symp. Operating System Design and Implementation (OSDI 00)*, Oct. 2000.

[11] C. Hsu and W. Feng. Effective dynamic voltage scaling through CPU-boundedness detection. *Proc. 4th Workshop Power-Aware Computer Systems (PACS 04)*, Dec. 2004.

[12] C. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *Proc. 2003 Conf. Programming Languages Design and Implementation (PLDI 03)*, June 2003.

[13] J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. *Proc. Int'l Conf. Measurement & Modeling of Computer Systems (SIGMETRICS 01)*, June 2001.

[14] A. Milenkovic, M. Milenkovic, E. Jovanov, and D. Hite. An environment for runtime power monitoring of wireless sensor network platforms. *Proc. 37th Southeastern Symp. System Theory (SSST 05)*, Mar. 2005.

[15] B. Mochocki, X. Hu, and G. Quan. A unified approach to variable voltage scheduling for nonideal DVS processors. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1370–1377, Sept. 2004.

[16] T. Mudge. Power: A first class design constraint for future architectures. *IEEE Computer*, 34(4):52–58, Apr. 2001.

[17] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED 00)*, July 2000.

[18] N. Pettis, L. Cai, and Y.-H. Lu. Dynamic power management for streaming data. *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED 04)*, Aug. 2004.

[19] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *Proc. 18th Symp. Operating Systems Principles (SOSP 01)*, Oct. 2001.

[20] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. *Proc. 7th Int'l Conf. Mobile Computing and Networking (MobiCom 01)*, July 2001.

[21] D. Reed. High-end computing: The challenge of scale. Director's Colloquium, Los Alamos National Laboratory, May 2004.

[22] J. Seng and D. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. *Proc. 7th Workshop Interaction between Compilers and Computer Architectures (INTERACT-9)*, Feb. 2003.

[23] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. *Proc. Int'l Conf. Computer-Aided Design (ICCAD 00)*, Nov. 2000.

[24] T. Simunic, L. Benini, and G. Micheli. Dynamic power management of portable systems. *Proc. 6th Int'l Conf. Mobile Computing and Networking (MobiCom 00)*, Aug. 2000.

[25] A. Varma, B. Ganesh, M. Sen, S. Choudhary, L. Srinivasan, and B. Jacob. A control-theoretic approach to dynamic voltage scaling. *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES 03)*, Oct. 2003.

[26] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. *Proc. 1st Symp. Operating Systems Design and Implementation (OSDI 94)*, Nov. 1994.

[27] R. Xu, C. Xi, R. Melhem, and D. Mossé. Practical PACE for embedded systems. *Proc. 4th Int'l Conf. Embedded Software (EMSOFT 04)*, Sept. 2004.

[28] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.