The Sixth LCI International Conference on Linux Clusters: The HPC Revolution 2005.
Chapel Hill, North Carolina, April 2005, LA-UR 05-1581.

Reducing Overheating-Induced Failures Via Performance-Aware CPU Power Management *

Chung-Hsing Hsu and Wu-Chun Feng

Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, U.S.A.
{chunghsu, feng}@lanl.gov

Abstract. Cluster end-users and administrators have become more cognizant of the fact that large-scale commodity clusters fail quite frequently, and the main source of these failures is hardware (e.g., processors) with the primary cause being heat. This situation is expected to worsen with even larger-scale clusters powered by faster (and/or multi-core) processors. In this paper, we propose a power-management algorithm that addresses heat-related reliability for processors by controlling their clock speeds in a performance-aware manner. This approach is complementary to existing approaches such as exotic cooling and fault-tolerant technologies in that it *proactively* deals with power and cooling issues *before* they become a problem. Our preliminary experimental work demonstrates that our approach can easily be applied commodity processors and can reduce heat generation by 30% on average with minimal effect on performance when running the SPEC benchmarks.

1 Introduction

Though the high-performance computing (HPC) community continues to provide better and better support for Linux-based commodity clusters, cluster end-users and administrators have become more cognizant of the fact that large-scale commodity clusters fail quite frequently. For example, the mean time between failure for the ASCI Q constellation at Los Alamos National Laboratory (LANL) is less than 6.5 hours, i.e., 114 unplanned outages per month [15]. The low reliability is largely due to hardware failures, and we argue that one of the main reasons for these hardware failures is due to heat. As noted by the CEO of Google, Eric Schmidt, the primary contributor to unreliability in their data centers is heat [12]. The empirical data from our 128-processor Little Blue Penguin (LBP) Beowulf cluster, located in a dusty and hot warehouse, also confirms that heat has an adverse effect on cluster reliability. In the winter when the temperature inside the warehouse was around 70-75°F, the cluster failed once a week; in

* This work was supported by DOE Laboratory-Directed Research & Development and by Advanced Micro Devices, Inc. through LANL contract W-7405-ENG-36. Available as LANL report: LA-UR 05-1581.

the summer when the temperature was 85-90°F, the cluster failed twice a week. Thus, reducing heat-related failures will improve system reliability. This paper shows how heat reduction can be achieved by decreasing the power draw of the processor while simultaneously minimizing impact on performance.

Because of the dual goals of improving performance and reducing power draw in order to enhance system reliability, the focus of this paper is on improving the performance-power ratio of processors (that are used in commodity clusters) in a way that is transparent to HPC applications. Specifically, we propose an automatically-adapting, power-management algorithm that intelligently sets the frequency and voltage of a processor at run time in order to minimize processor's power draw with little effect to system performance. The novelty of our approach is in its *proactive* nature, i,e., we address the likely power and cooling issues *before* they become a problem. This is in contrast with (and complementary to) the common approaches that reactively handle thermal emergencies with on-chip thermal protection and software-based fault tolerance.

The algorithm is based on the observation that a processor oftentimes cannot execute instructions at its maximum clock rate due to performance bottlenecks in the memory, I/O, or network subsystems; thus, its clock speed can be reduced to save power without having a major effect on the overall speed of the computation. The algorithm uses a mechanism called dynamic voltage and frequency scaling (DVS) that has been supported by commodity processors such as Intel's Nocona, AMD's Opteron, and IBM's PowerPC-970, and by the Linux community through a standardized kernel interface called `cpufreq`. Via real experimental measurements, preliminary research and evaluation shows that our software-based heat-reduction approach can improve the performance-power ratio of commodity processors by as much as 35% on SPEC benchmarks.

The rest of the paper is organized as follows. Section 2 provides background information on current approaches towards reducing heat-related failures. Then we present a software-based algorithm that can reduce heat generation in a performance-controlled manner in Section 3. Section 4 describes the experimental set-up, the implemented DVS algorithms, and the experimental results. Finally, Section 5 concludes the paper.

2 Background

Overheating-induced system failures can be reduced *reactively* through effective cooling or *proactively* through less heat generation. The most common *reactive* approach to reduce overheating-induced failures is to install extra cooling capacity. Unfortunately, this approach can be disastrous because if the cooling pushes the air under the floors too quickly, a Venturi effect can occur. This causes air to be sucked down into the floor rather than pushing it up into the "cold" aisle, and air from the "hot" aisle then flows over the top of the racks and re-circulates through the equipment, thus resulting in overheating. In other words, certain layouts of computing resources and cooling equipment may lessen power demand and keep key areas cool, while other layouts have the opposite effect. The "op-

timal" layout of computing resources and cooling equipment is a complex task; it has to consider all the variables in a machine room, model how the air moves, and determine what changes can be made to optimize the use of cooling capacity and reduce operating costs. Because of this level of the complexity, HP Labs, for example, resorts to computational fluid dynamics to create a three-dimensional airflow simulation model [13]. Furthermore, the power required to operate heat removal and/or cooling facilities increases the total cost of ownership for HPC systems. And in many cases, institutions that utilize HPC must spend millions to tens of millions of dollars simply to build the appropriate infrastructure to house their HPC systems. For the above reasons, this paper examines approaches for reducing the amount of heat generated by a HPC system.

A *proactive* approach to prevent overheating is to *not* generate heat to begin with (or at least decrease its generation). The reduction of heat generation can be achieved in two ways: via hardware/architecture design or via software-based optimization. IBM's Blue Gene/L [2, 4] and LANL's Green Destiny [5, 6, 17] are two representative examples of reducing heat via hardware/architecture design. These systems employ low-power, low-speed processors for heat reduction and oftentimes rely on customized components to achieve the dual goals of improving performance and reducing power draw. For example, Blue Gene/L is based on a stripped-down version of the 700-MHz PowerPC 440 embedded processor. In contrast, Green Destiny is based on Transmeta's 1-GHz commodity mobile processors powered by a high-performance version of Transmeta's proprietary code morphing software. However, we argue that the above approaches are not commodity-oriented, thus forcing one to select a particular hardware platform or a specific vendor.

Another way to reduce heat generation is via software-based optimization. Over the past few years, there has emerged an effective power-reduction mechanism called dynamic voltage and frequency scaling (DVS). At a high level, a DVS system exports a set of operating points that provide various performance-power tradeoffs and allows the software to switch between these operating points in pseudo-real-time (10-100 μs) during program execution. In general, the faster a CPU runs, the better performance it can deliver but the more power it will draw. Despite the fact that the DVS mechanism was first introduced for mobile systems and many DVS-based power-management algorithms (e.g., [7, 10, 11, 16]) were proposed to prolong the battery life of these systems, the use of DVS in HPC is virtually non-existent. Furthermore, the existing DVS-based power-management algorithms for mobile systems are ineffective on HPC systems because user interaction (e.g., editing a Microsoft Word document on a laptop) is assumed in these algorithms in order to determine system power-usage patterns, something that is non-existent in HPC.

For example, Intel has developed a DVS-based power-management algorithm for their Pentium-M mobile processors, part of the so-called *enhanced Intel Speed-Step technique*. The algorithm uses Linux's kernel interface `cpufreq` to switch between operating points and uses CPU utilization to guide the selection. Specifically, the algorithm periodically polls the current CPU utilization, defined as

the fraction of time that the CPU spends non-idle. If the utilization passes above a threshold value, the algorithm will select the operating point with the maximum clock speed. On the other hand, if the utilization falls below a threshold value, the algorithm will scale down the clock speed to the next available level.

The above algorithm has two problems for HPC. First, CPU utilization by itself does not provide enough information about system timing requirements; DVS-based algorithms based on such information can only provide loose control over DVS-induced performance loss. Second, the CPU utilization for HPC applications tends to be high during the entirety of their executions; as a result, Intel's SpeedStep power-management algorithm will always set the CPU to run at the maximum speed and does not use DVS at all.

Consequently, Hsu and Kremer recently demonstrated how to apply DVS to scientific codes [9]. Unfortunately, this earlier solution is based on profile-driven, off-line, compiler analysis. In this paper, we seek a run-time solution that can automatically adapt to various situations without any prior knowledge of a scientific application or its run-time behavior.

3 Lightweight Adaptive DVS

We propose a power-management algorithm that maximizes CPU power reduction while minimizing performance impact and does so by exploiting sub-linear performance scaling in memory-bound and I/O-bound programs. To explain sub-linear performance scaling, consider a program with a high cache-miss ratio. Such a program will likely be limited by memory bandwidth rather than CPU speed. Since memory performance is not affected by CPU speed, increasing or decreasing the CPU frequency will have little effect on the performance of the program. We can take advantage of this type of sub-linear performance scaling and lower the CPU speed with negligible impact on performance, thereby reducing the CPU power requirements significantly. For a DVS-enabled Intel Pentium-M processor, this can translate into a 76% reduction in CPU power draw.

Though sub-linear performance scaling occurs in memory-bound or I/O-bound programs, simply slowing down the CPU speed during *any* memory or I/O operation will result in both significant performance slowdown as well as increased energy consumption. (This is due in large part to the non-negligible amount of time needed to scale to a new voltage and frequency.) Therefore, we first develop an accurate model that relates a program's performance to CPU frequency in order to minimize DVS-induced performance loss when switching to a slower CPU speed. This model is critical to a power-management algorithm as different memory-bound programs will experience different performance impacts when running at the slowest CPU speed. Next, we must then be able to construct the model at run-time because the model's parameter values will be affected by the hardware platform, the program source code, and the data input.

Many workloads spend considerable time on memory accesses, e.g., SPEC CPU2000, BAPCo SysMark2000, and Quake3, and their performance can be

accurately predicted using the following simple model [3, 14]:

$$T(f) = i \cdot \frac{1}{f} + m \cdot l \tag{1}$$

This model decomposes the total execution time $T(f)$ of a program running entirely at frequency f into two parts. The first part models on-chip workload in terms of CPU cycles, where i captures the ideal number of cycles required (assuming perfect caches). The second part models off-chip workload, where m counts the number of off-chip accesses and l represents the memory-access latency in seconds. Because f only appears in the first part, changing the CPU speed only affects the performance of executing the on-chip workload and results in a sub-linear performance impact.

Once we know of the values of i, m, and l, we can construct a power-optimal CPU speed schedule with respect to a deadline D — running the entire program at a CPU speed f^* which solves the following optimization problem:

$$\begin{aligned} & \min\{P(f) : \ T(f) \leq D\} \\ = & \min\left\{P(f) : \ i \cdot \frac{1}{f} + m \cdot l \leq D\right\} \\ = & \min\left\{P(f) : \ f \geq \frac{i}{D - m \cdot l}\right\} \end{aligned} \tag{2}$$

where $P(f)$ is the power consumption of a DVS-enabled processor running at the clock frequency f. If $P(f)$ is an increasing function, then Equation (2) simply says that running at the lowest possible CPU speed without violating the performance constraint will minimize the CPU power consumption. In general, the power function $P(f)$ is an increasing function since the faster a CPU runs, the more power it consumes.

However, for this scheduling technique to be effective, it requires knowing the values of i, m, and l which are difficult to derive since the values, in particular i and m, vary with different data inputs. Hence, in order to automatically adapt to various data inputs without any prior knowledge, the values of i and m should not be explicitly used. Knowing that the values of i and m are fixed for a particular data input as is their impact to program performance, we can simply concentrate on the performance sensitivity to changes in CPU speed, thus minimizing the dependence on i and m.

The sensitivity of a program's performance to changes in CPU speed is formally defined as the relative performance slowdown with respect to the program running at the maximum CPU speed, i.e., $\frac{T(f)}{T(f_{max})}$. where f_{max} is the maximum CPU speed. Using Equation (1), we can re-write the relative performance slowdown in a form based on a coefficient called β.

$$\frac{T(f)}{T(f_{max})} = \frac{i \cdot \frac{1}{f} + m \cdot l}{i \cdot \frac{1}{f_{max}} + m \cdot l} = \frac{i \cdot \frac{f_{max}}{f} + m \cdot l \cdot f_{max}}{i + m \cdot l \cdot f_{max}}$$

$$= \frac{i}{i + m \cdot l \cdot f_{max}} \cdot \frac{f_{max}}{f} + \left(1 - \frac{i}{i + m \cdot l \cdot f_{max}}\right)$$
$$= \beta \cdot \frac{f_{max}}{f} + (1 - \beta) \tag{3}$$

with

$$\beta = \frac{i}{i + m \cdot l \cdot f_{max}} \tag{4}$$

The coefficient β is, by definition, a value between 0 and 1. The metric represents the fraction of the program workload that scales linearly with the CPU frequency (much like the parallelizable part in Amdahl's Law). 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 $\beta \approx 0$.

Given the value of β, we can compute the desired frequency f^* with respect to a performance criterion δ (a relative performance slowdown compared to the program running entirely at the full speed, e.g., 5%). This desired frequency is in fact the solution of the following optimization problem:

$$\min \left\{ P(f) : \ \frac{T(f)}{T(f_{max})} \leq 1 + \delta \right\}$$
$$= \min \left\{ P(f) : \ \beta \cdot \frac{f_{max}}{f} + (1 - \beta) \leq 1 + \delta \right\}$$
$$= \min \left\{ P(f) : \ f \geq \frac{f_{max}}{1 + \delta/\beta} \right\} \tag{5}$$

If the power function $P(f)$ is an increasing function, which is in practice, then we can describe the desired frequency f^* in a closed form, as follows.

$$f^* = \max \left(f_{min}, \frac{f_{max}}{1 + \delta/\beta} \right) \tag{6}$$

With Equation (6) in place, we propose a power-management algorithm called *β-adaptation*.

The β-adaptation algorithm consists of two phases that are executed in sequence at fixed-length time intervals. The first phase uses Equation (3) and a regression method to compute β up to the current interval. The second phase uses Equation (6) to compute the desired frequency f^* to use in the next interval. These two phases are shown as steps `1` and `2` in Figure 1.

Unfortunately, today's DVS-enabled processors may not be able to generate the desired frequency f^* that is calculated in step `2`. For example, Intel's Pentium-M processor only allows the software to specify the target speed and voltage in multiples of 100 MHz and 16 mV, respectively. For AMD's Opteron processor, the target frequency and voltage can only be specified in multiples of 200 MHz and 25 mV, respectively. In order for our *β-adaptation* algorithm to work on these processors, we emulate the desired frequency using available clock

For every I seconds, do the following:

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

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

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 $\texttt{mips}(f_{j+1})$.

Fig. 1. The *β-adaptation* Algorithm. δ is a relative performance slowdown criterion, I is the length of an interval in seconds, and $\texttt{mips}(f)$ is the average MIPS rate for CPU frequency f. This algorithm assumes that a DVS processor only exports n available frequencies $f_{min} = f_1 < \cdots < f_n = f_{max}$.

frequencies. For example, since the 500-MHz speed is not directly supported by Athlon 64 processor, we emulate the speed by running half of the time at the 400-MHz and half the time at the 600-MHz speed. This emulation process is described as steps `3-6` of Figure 1.

In fact, the frequency emulation of the *β-adaptation* algorithm is a theoretically-based heuristic that is based on a characterization of an energy-optimal DVS scheduling problem originally proposed in [9]. According to that characterization, the emulation of the desired frequency using the two immediately-neighboring settings will result in an energy-optimal DVS schedule; this is what step `3` does. In other words, to emulate the 500-MHz speed, using the 400-MHz and 600-MHz pairs will result in a lower power consumption than using the 200-MHz and 800-MHz pairs. The characterization has its own merit in that it generalizes previous theoretical results. We refer the readers to our workshop paper [8] for more details.

4 Experiments

The *β-adaptation* algorithm can be easily implemented on any commodity DVS-enabled processor. To demonstrate its simplicity and effectiveness, we implemented the algorithm on two commodity processors found in HPC blades and servers, respectively: an AMD Athlon XP-M and an AMD Athlon 64 3200+, as shown in Table 1. The Athlon XP-M processor has a 256-KB level-two cache and 256-MB DDR-266 memory while the Athlon 64 processor has a 1-MB level-two cache and 1-GB DDR-400 memory. In terms of DVS operating points, the Athlon XP-M possesses five frequency-voltage settings with the frequency ranging from 1067-MHz to 1800-MHz and the voltage from 1.15V to 1.45V; the Athlon 64 has four such settings running from 800-MHz to 2000-MHz at 1.0V to 1.55V. For both processors, the time to transition from one setting to another is roughly 100 μs.

Table 1. The tested system configurations

CPU	Athlon XP-M 2200+	Athlon 64 3200+
L1 cache	128KB	128KB
L2 cache	256KB	1024KB
f (MHz)	1067-1800	800-2000
V	1.15-1.45 V	1.10-1.55 V
levels	5	4
memory	256MB DDR266	1024MB DDR400

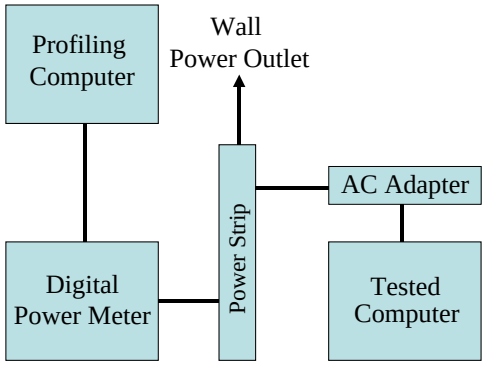

Fig. 2. The measurement set-up.

We evaluated the *β-adaptation* algorithm in the context of the `SPEC` CPU benchmarks and utilized a highly accurate, industrial-strength power meter to collect power-consumption numbers. The benchmarks were compiled with the GNU compilers at the same optimization level, i.e., `-O2`. The power meter is Yokogawa's WT210 digital power meter, which constantly samples the instantaneous wattage at a rate of 50 kHz (i.e., every 20 μs). The `SPEC` CPU benchmarks emphasize the performance of the CPU and memory, but not I/O. We elected to use the `SPEC` benchmarks because they demonstrate a range of performance sensitivity to CPU-frequency changes [9]. All the experimental data was collected by running these `SPEC` benchmarks with the reference data input. Note that this experimental setup included the non-negligible amount of time and energy needed to scale a voltage and frequency, plus the time and energy used to execute the *β-adaptation* algorithm on top of the execution of a `SPEC` benchmark.

As seen in Figure 3, our preliminary evaluation shows that the *β-adaptation* algorithm improves the `SPEC/W` ratio (i.e., performance/watt) of the CFP95, CINT95, and CFP2000 benchmark suites by 26%, 12%, and 17%, respectively, on the Athlon XP-M system and by 30%, 22%, and 35%, respectively, on the Athlon 64 system. The algorithm saves as much as 54% on CPU energy while impacting peak performance by less than 2% when running a SPEC CPU benchmark. Finally, in comparison with a 2.4-GHz Pentium 4 desktop, the *β-adaptation* algorithm improves the `SPEC/W` ratio by another 50% beyond the 50% improvement when the algorithm is run on the Athlon 64 architecture.

We also recently applied our *β-adaptation* algorithm to a four-way Opteron-based SMP machine running Linux 2.6. The `Mop/s/W` ratio for NAS Parallel Benchmarks [1] on class B is 34% better when using the *β-adaptation* algorithm. The power draw of the `SP` benchmark is reduced by 34% with an impact on peak performance of less than 1%.

In summary, the *β-adaptation* algorithm is a theoretically-based lightweight effective heuristic. Our preliminary work demonstrates (via physical measure-

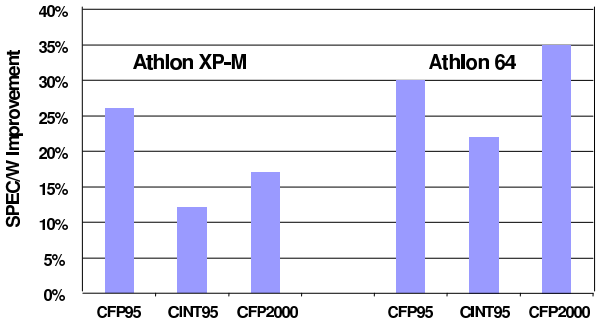

Fig. 3. The effectiveness of the *β-adaptation* algorithm.

ments) that this proactive approach can improve the power efficiency (in terms of the performance-power ratio) of commodity high-end processors by as much as 35% on SPEC benchmarks as well as on NAS Parallel Benchmarks.

5 Concluding Remarks

As both end users and administrators of Linux-based commodity clusters, we have identified two primary roadblocks that prevent such clusters from marching into the HPC mainstream: (1) unreliability and (2) poor ease of use. In this paper, we specifically addressed the former. The latter issue is dealt with by research projects such as Clustermatic (http://www.clustermatic.org/) and a host of others.

We argued that one of the main reasons for unreliability in commodity clusters is hardware failure due to overheating and that of all the components in a typical commodity cluster node, the microprocessor consumes the largest amount of power and dissipates the most heat. As a result, HPC integrators of commodity systems have resorted to more exotic technologies to dissipate the heat generated by a microprocessor (i.e., from heating plates to heat sinks to heat pipes, and now, even to spray or liquid cooling) and have installed thermal sensors and thermal management software as a secondary prevention mechanism.

In contrast, we took a complementary approach in this paper, one that proactively tackled the overheating issue by reducing the heat generation from microprocessors *before* it even becomes a problem. By leveraging the "dynamic voltage and frequency scaling" (DVS) mechanism commonly found in high-end commodity processors, we applied DVS wisely so that the heat generation by processors can be minimized with little effect on system performance. The result was our DVS-based, *β-adaptation* algorithm, which automatically adapts to the application without any prior knowledge of the application itself. The preliminary experimental results demonstrate that our proactive *β-adaptation* algorithm can reduce power consumption by 30% on average with minimal effect on performance when running the SPEC benchmarks and NAS Parallel

Benchmarks. With respect to power efficiency, our technique can increase the performance-power ratio by as much as 35%.

In closing, since productivity is oftentimes quantified as the ratio of the number of jobs a HPC system will run over the total amount of resources that go into buying and running the system, the work proposed in this paper will improve reliability and availability, thus increasing productivity and reducing total cost of ownership.

References

1. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/software/npb/.
2. IBM Blue Gene Project. http://www.research.ibm.com/bluegene.
3. I. Chihaia and T. Gross. Effectiveness of simple memory models for performance prediction. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2004.
4. N. Adiga et al. An overview of the bluegene/l supercomputer. In *the IEEE/ACM SC2002*, November 2002.
5. W. Feng. Making the case for efficient supercomputing. *ACM Queue*, 1(7), October 2003.
6. W. Feng, M. Warren, and E. Weigle. Honey, I shrunk the Beowulf! In *International Conference on Parallel Processing*, August 2002.
7. K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for Linux. In *the Symposium on Operating Systems Design and Implementation*, December 2002.
8. C. Hsu and W. Feng. Effective dynamic voltage scaling through CPU-boundedness detection. In *Workshop on Power-Aware Computer Systems (PACS)*, December 2004.
9. C. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, June 2003.
10. R. Jejurikar and R. Gupta. Dynamic voltage scaling for system-wide energy minimization in real-time embedded systems. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.
11. J. Lorch and A. Smith. PACE: A new approach to dynamic voltage scaling. *IEEE Transactions on Computers*, 53(7), July 2004.
12. J. Markoff and S. Lohr. Intel's huge bet turns iffy. *New York Times*, September 2002.
13. C. Patel. A vision of energy aware computing from chips to data centers. In *The International Symposium on Micro-Mechanical Engineering (JSME)*, December 2003.
14. A. Predtechenski. A method for benchmarks analysis. In *Workshop on Performance Evaluation with Realistic Applications*, January 1999.
15. D. Reed. High-end computing: The challenge of scale. Director's Colloquium, Los Alamos National Laboratory, May 2004.
16. T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *the Conference on Design Automation (DAC)*, June 2001.
17. Supercomputing in Small Space Project. http://sss.lanl.gov.