Efficient 'Cool Down' of Parallel Applications

Osman Sarood, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{sarood1, kale}@illinois.edu

***Abstract*—As we move to exascale machines, both peak power and total energy consumption have become prominent major challenges. There has been a lot of research on saving machine energy consumption for HPC data centers. However, a significant part of energy consumption for HPC data centers can be attributed to cooling the machine room. We have already shown significant reduction in cooling energy consumption by constraining core temperatures in our previous work. In this work, we strive to save machine energy consumption while constraining core temperatures in order to provide a *total* energy solution for HPC data centers that saves both machine and cooling energy consumption. Our approach uses Dynamic Voltage and Frequency Scaling (DVFS) to constrain core temperatures and is particularly designed to reduce the timing penalty associated with DVFS. Using a heuristic that exploits the difference in frequency sensitivity for different parts of an application, we present results that show 17% reduction in machine energy consumption with as little as 0.9% increase in execution time while constraining core temperatures below 60° C.**

I. Introduction

Energy consumption has emerged as a major issue for modern High Performance Computing (HPC) machines. Some of the largest supercomputers draw close to 10 megawatts [1], leading to millions of dollars per annum in energy bills. What is perhaps less well known is the fact that 40% to 50% of the energy consumed by a data center is spent in cooling [2]–[4], to keep the computer room running at a safe temperature. In the past few years, we have seen some low power HPC clusters emerge, such as Green Destiny [5]. Although the energy efficiency for such machines is considerably greater than conventional supercomputers, their processing power is also much inferior to them. A per node comparison of Green Destiny with the Q supercomputer at Los Alamos National Laboratory (LANL) shows that the latter is 15 times faster [5].

Given that the bulk of existing energy optimization research for HPC data centers only considers reducing machine energy consumption, we plan to tackle the bigger problem of reducing the *total* energy consumption i.e. both cooling and machine energy consumptions. A large part of this cooling energy consumption can be attributed to formation of hot spots which force data center operators to over-cool the machine room just to keep machines in the hot spot at an acceptable temperature. System operators can avoid increasing the cooling, provided that core temperatures for all the machines are kept in safe limits because even a small increase in core temperatures e.g. 10-15C° C, can cause a 2X increase in the fault rate [6].

Current day microprocessors contain on-chip temperature sensors which can be accessed by software with minimal overhead. Further, they also provide means to change the frequency and voltage at which the chip runs, known as *Dynamic Voltage and Frequency Scaling* (DVFS). Running processor cores at a lower frequency (and correspondingly lower voltage) reduces their thermal energy dissipation, leading to a cool-down. This suggests a method for keeping processors cool while decreasing the cooling requirement for the machine room. In our earlier work [7], we show that significant amount of cooling energy consumption can be saved by constraining core temperatures using DVFS combined with dynamic load balancing. Although more radical liquid-cooling designs are expected to mitigate some of the hot spot concerns, they are not a panacea. Equipment must be specifically designed to be liquid-cooled, and data centers must be built or retrofit to supply the coolant throughout the machine room. The present lack of commodity liquid-cooled systems and data centers means that techniques to address the challenges of air-cooled computers will continue to be relevant for the foreseeable future. In addition to avoiding hot spots, constraining core temperatures can also reduce cooling energy consumption by simply reducing thermal energy dissipation in the machine room which the cooling unit has to remove. Our earlier work [7] shows that we were able to reduce cooling energy consumption by as much as 63% by constraining core temperatures and lowering the cooling level for the machine room. However, as reducing machine energy consumption was not the aim of our earlier study, we did not end up reducing it significantly. In this work, we try to tackle this *other* part of energy consumption i.e. machine energy consumption. Our scheme allows the application user to specify a maximum temperature threshold and the runtime system ensures that these thresholds are honored while attempting to minimize execution time penalty. We exploit the fact that different parts of the same application could have different sensitivities to frequency due to communication stalls and memory bandwidth requirements, and hence are better off running at different frequency levels. In order to ensure that no processor overheats, a component of the application software periodically checks core temperatures. When it exceeds a pre-set threshold, the software can reduce the frequency and voltage of a part of the application that is least sensitive to frequency. If the temperature is lower than the threshold, the software can correspondingly increase the frequency of the most frequency-sensitive part of the

application currently working below the maximum frequency level due to temperature constraints.

The novelty of our work lies in the fact that we reduce machine energy consumption alongside constraining core temperatures and this leads to reduction in cooling energy consumption. In our work, we use the newly introduced on-chip energy consumption counters supported by Intel's Sandy Bridge processor [8] that have a refresh rate of 1 millisecond. These counters empower the runtime system by allowing it to make more intelligent decisions regarding DVFS in order to constrain core temperatures. Use of these energy counters also allows us to expedite the learning (profiling) process and use our novel heuristic that makes decisions about which part of the application should run at a lower/higher frequency. However, since this microprocessor is very recent, we were unable to find a cluster that had multiple Sandy Bridge nodes and so we resorted to using a single node for all our experiments. As we will show later, the results from our single node experiments suffice to profoundly increase our understanding of application reaction to temperature control. The contributions of this paper can be summarized as follows:

- Decreases the learning (profiling) period to as low as few milliseconds for profiling all parts of the application
- Minimize the timing penalty associated with DVFS to constrain core temperatures and reduce machine energy consumption by using our novel heuristic.
- Using a combination of hardware performance counters and Sandy Bridge energy counters, we present an in depth analysis of how the characteristics of different parts of an application impact CPU core power, core temperatures, and execution time penalty.
- Devise an index that captures how much benefits our scheme can bring for a given application

II. Related Work

Cooling energy optimization and hot spot avoidance have been addressed extensively in the literature of non-HPC data centers [9]–[12], which shows the importance of the topic. As an example, job placement and server shut down have shown savings of up to 33% in cooling costs [9]. Many of these techniques rely on placing jobs that are expected to generate more heat in the cooler areas of the data center. This does not apply to HPC applications where different nodes are running parts of the same application with similar power draw.

Energy optimization work for HPC data centers is broadly divided into two categories: Reducing energy consumption without any performance impact and reduce energy consumption by trading it off with execution time. The former is mostly possible in applications which are load imbalanced and have some slack time available in their Directed Acyclic Graph (DAG) [13]. However, the latter can be applied to load balanced applications as well, where researchers mainly exploit memory bandwidth limitations to reduce machine energy consumption. There are two high level categories to which such techniques belong i.e. techniques based on profiling runs [14] and techniques based on performance counters prediction [15]. The closest work to ours are CPU MISER from Ge et al. [15] and the work from Freeh et al. [14]. CPU MISER builds a model to estimate the load and execution time for each execution phase under different frequency/voltage pairs. It uses performance counters to come up with a frequency/voltage schedule that keeps the execution time delay below a limit while minimizing machine energy consumption. Freeh et al. [14], on the other hand, use profiling in order to obtain the best possible schedule for different application phases. Their technique does not offer any constraint on the execution time delay and strives to achieve maximum energy savings possible for a given application. All the works cited for HPC data centers so far focus on reducing machine energy consumption, ignoring cooling energy consumption. In this paper we address the question of reducing cooling energy consumption by incorporating core temperature constraints. Although researchers have done work at thermal profiling for parallel applications [16], HPC community still lacks research efforts in reducing cooling energy consumption. Our work is different because we use specialized energy counters provided by Sandy Bridge processor to profile core power and timing penalty for each part of the application to obtain an optimal frequency schedule that constrains core temperatures efficiently. More importantly, we present an in depth anaylsis of the relation between core power, core temperature, operating frequency and Although we do not report results showing reduction in cooling energy consumption in this work, according to our previous work [7], this reduction can be as large as 63%.

III. Constraining core temperatures

Modern day systems do not directly respond to high core temperatures. Where they do react, that reaction can cause severe slow down of the application. Simple measures like increasing fan speed have limited effect. Allowing the more extreme response of auto throttling at the core's maximum thermal limits could be disastrous to performance. Right now the only mechanism available to system operators is energy intensive machine room level cooling. According to studies [17], data center operators can save 7% of the total cooling cost by increasing the machine room temperature by 1° C. In order to increase the machine room temperature, data center operators need to be sure that core temperatures would not reach very high values and there will be no hot spot formation.

To see the behavior of core temperatures for different applications, we ran four applications from the NPB parallel benchmark suite [18] on a single node having a quad core Intel core i7-2600 processor. Figure 1 shows the average core temperature across all 4 cores plotted against execution time. Depending on application characteristics, core temperatures settle at different steady state temperatures. This difference in steady state temperatures makes temperature control even more important as some applications can add a significant amount of thermal energy to the machine room due to their higher steady state core temperatures. Core temperatures can be reduced given that core power is reduced. Researchers

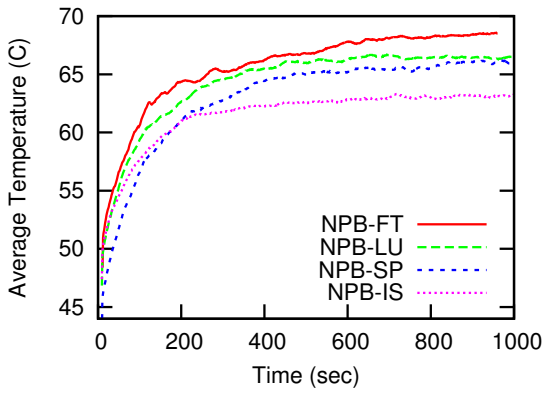

Fig. 1. Temperature profiles without temperature control

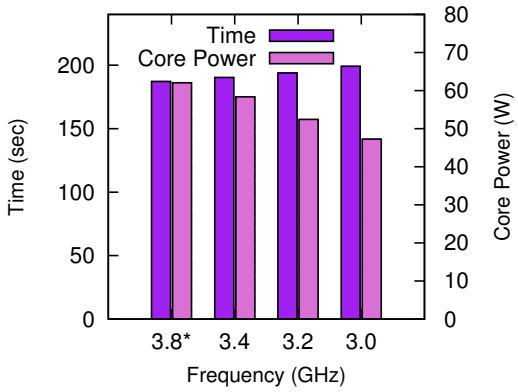

Fig. 2. Execution time and core power for *NPB-FT* for four frequency levels

have widely used the technique of DVFS to reduce core power. This technique allows the runtime system to change frequency/voltage pair in order to reduce power. However, these savings come at the cost of delay in execution time i.e. timing penalty. Figure 2 shows the results of running *NPB-FT* in parallel for four different frequency levels using the same machine used in the earlier experiment. We plot the execution time along with the core power for all 4 cores. Looking at this figure, we can see that the increase in execution time is not significant compared to the reduction in the core power for this application. Hence, reducing frequency would help reduce core power which would consequently reduce core temperatures, with a small timing penalty. Since energy is power integrated over time, whether this saves energy or not depends on the execution time penalty.

To demonstrate the impact of DVFS, we ran a set of experiments using *NPB-FT*. During these experiments, the runtime sampled core temperatures periodically, and when the average core temperature was greater than the maximum threshold, its frequency/voltage pair was lowered by one step. On the other hand, if the average core temperature was lower than the maximum threshold, the frequency/voltage pair was increased by one step. We repeated this experiment for a range of different maximum temperature thresholds and calculated the timing penalty i.e. the percentage delay in execution time, as well as the reduction in machine energy consumption relative to a run where all cores were working at the maximum frequency without any temperature control. As shown in Figure 3, DVFS alone in this setting reduces machine energy consumption but sacrifices execution time considerably. Nevertheless, we were able to constrain core temperatures and save machine energy consumption using DVFS.

IV. Reducing Timing Penalty for DVFS

Having seen the importance of constraining core temperature using DVFS we now investigate the possibility of reducing DVFS-associated timing penalty. It turns out that by dividing the application into smaller execution blocks (*EBs*), we can reduce the timing penalty and machine energy consumption.

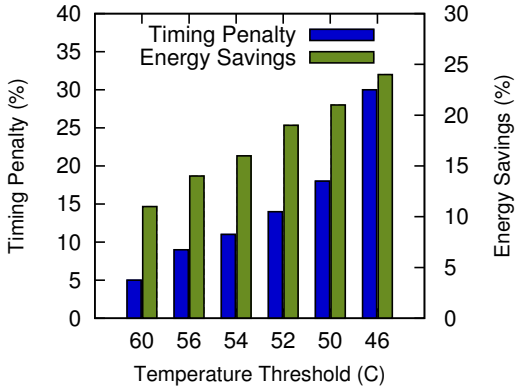

Fig. 3. Timing penalty and savings in energy consumption for *NPB-FT* using different temperature thresholds (temperature sampling at every iteration)

This reduction is possible because different sections of code might have different memory access patterns and hence might not need a very high frequency to run at. For most cases, where there is a lot of memory traffic, the highest levels of frequency consume a lot of energy and consequently dissipate a large amount of heat that increases core temperatures without making any significant difference to execution time.

In order to see the potential of constraining core temperature by executing different parts of an application at different frequencies, we manually divided *NPB-IS* in two parts, EB1 and EB2, which repeat in each iteration of its execution. We then profiled their execution times and core power using all possible frequency levels on the same quad core machine used in earlier experiments. As seen from Figure 4, *EB2* is very insensitive to frequency as compared to *EB1* i.e. the execution time for *EB2* doesn't increase much as we go on decreasing the frequency for it. However, irrespective of being insensitive to frequency, *EB2*'s core power keeps on increasing with an increase in frequency. Hence, if we reduce the frequency of *EB2* from maximum to minimum, it would result in a substantial decrease in the core power (50W to 18W) and hence would cause a reduction in core temperatures without

a significant increase in the execution time. However, the

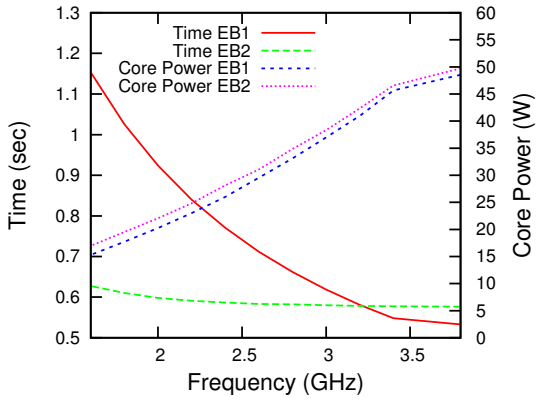

Fig. 4. Execution time and core power for *NPB-IS* for different frequency levels

impact of changing the frequency for individual *EBs* on core power depends on the proportion of execution time each *EB* represents in the total execution time. Figure 4 shows that *EB2* accounts for a higher proportion of total execution time as compared to *EB1* when working at maximum frequency. Hence we should expect a sizable reduction in the core power after shifting *EB2* to the lowest frequency level i.e. 1.6GHz.

V. EB TUNER

In this section we use the insight gained in Section IV to devise a novel technique that constrains core temperatures and saves energy consumption while minimizing timing penalty. This section is divided into two subsections. Section V-A outlines the profiling mechanism which is a pre-requisite for our scheme. In Section V-B we formulate the problem of constraining core temperatures efficiently and describe our scheme, which we refer to as *EBTuner*.

A. Profiling technique

The majority of researchers use standalone power meters having refresh rates in the order of seconds for profiling energy statistics of applications. This implies that if the execution time for an *EB* is less than the refresh rate, the power meter won't be able to profile the *EB* straighforwardly. In order to determine the energy-time tradeoffs under these constraints, researchers usually fix the frequency for all the *EBs* and vary the frequency of one *EB* at a time in order to profile *EBs* for all possible frequency levels. Although this scheme works well in terms of coming up with the trade-offs, it leads to very long profiling periods that require several runs of the entire application [14].

Our goal is to constrain core temperatures that are dependent on power of the cores rather than the total power of the machine. Because of that, our profiling scheme uses core energy consumption information recorded in the Machine Specific Registers (MSRs) on Sandy Bridge processor which is refreshed every 1 millisecond. Using them, we can profile all the *EBs* an application is divided into at the same time for a given frequency level. This helps us profiling the application in order of milliseconds.

In order to profile an application we manually identify portions of code having high memory pressure by looking at hardware performance counters. The goal of profiling is to come up with execution time and core power vectors for each *EB* at the beginning of the application. These vectors, $L = t_i^1, t_i^2, ..., t_i^m$ and $P = p_i^1, p_i^2, ..., p_i^m$, give the execution time per iteration and the core power for EB_i at each of the m frequency levels supported by the CPU.

Since there is already much work done for dividing a program into blocks based on memory pressure [19], we leave out that part from our strategy. We are planning to incorporate our strategy with our earlier work [7] that focuses on reducing cooling energy consumption by using CHARM++ [20] which is based on asynchronous message driven execution. Since CHARM++ already divides computation into smaller sequential chunks i.e. *entry methods*, our technique does not need to divide the application itself. Instead, it would just profile the *entry methods* of CHARM++ and use it to determine optimal frequencies.

B. Temperature Aware EB Tuner

Our scheme is based on selectively running different parts of the applications i.e. *EBs*, at different frequency levels. This idea has already been used by other researchers to reduce machine energy consumption [14], [15]. However, we use it to reduce the DVFS-associated timing penalty for constraining core temperatures. Our scheme currently uses MPI and is limited to a single multi-core node. However, in our future work, we plan to combine it with our multi-node temperature constraining scheme [7] using CHARM++ as it provides efficient task migration infrastructure which is imperative for a multi-node scheme.

Our temperature control scheme is periodically triggered after equally spaced intervals in time, referred to as *steps*. At present, any iterative MPI application can add a call to our utility which then constrains core temperatures effectively. Our control strategy for DVFS is to let the cores work at their maximum frequency as long as their temperature is below a user-specified temperature threshold. If a core's temperature crosses the threshold, it is controlled by lowering the frequency of *one* of the *EBs*. At this time, our scheme needs to identify an *EB* such that a frequency reduction for it would result in the minimum possible timing penalty. The selection of the *best EB* should be such that we minimize application execution time (t_{app}):

$$t_{app} = \max_{p=1}^{N_{procs}} \left(\sum_{i=1}^{N_{EBs}} t_i^{f_i} \right) \quad (1)$$

where N_{EBs} is the number of EBs the application is divided into, $t_i^{f_i}$ is the execution time for EB_i running at frequency f_i, subject to:

$$T \leq T_{max} \quad (2)$$

where T is the average core temperature at each step, and T_{max} is the user specified maximum temperature threshold. Reducing frequency of the *best EB* would cause the core power to reduce which will consequently cause a drop in the core temperature. We adopt a heuristic to select the *best EB*. We define it in a way that considers change in both core power and timing penalty for making a change in frequency of an *EB*. Our heuristic for finding the *EB* with the best *power gradient* when core temperatures go above the threshold is defined as:

$$g_{best} = \max_{i=1}^{N_{EBs}} \left(\frac{p_{avg}^{f_i\downarrow} - p_{avg}}{t_{avg}^{f_i\downarrow} - t_{avg}} \right) \quad (3)$$

where N_{EBs} is the number of *EBs* the application is divided into, p_{avg} and t_{avg} are the average core power and the average execution time per iteration during the last *step*, $p_{avg}^{f_i\downarrow}$ and $t_{avg}^{f_i\downarrow}$ are the *predicted* average core power and average execution time per iteration after decreasing the frequency for EB_i one level lower from what it was during the last step i.e. f_i. Our scheme predicts execution time and core power after using the profiled data. Specifically, it uses the profiled execution time and core power vectors obtained at the beginning of the application (explained in Section V-A) for each *EB* in order to do predictions. In case the cores overheat we select the *EB* having the *maximum* power gradient (g_{best}) from amongst all the *EBs*. This is because we want to *maximize* the reduction in core power (numerator) while trying to *minimize* the timing penalty (denominator). However, in case when average core temperature is below the threshold value, we select the *EB* with the *smallest* power gradient after checking each *EB* at an increased frequency. Hence, instead of decreasing the frequency by one level ($f_i \downarrow$) in Equation 3, we increase it by one level ($f_i \uparrow$).

After conducting experiments with various applications, we determined that sampling temperatures after a period of 1 second (*step* size of 1 sec) is well-suited for constraining core temperatures for reasonable thresholds (Figure 5). After each step, the application calls a method exposed by our utility which passes control to the functionality listed in Algorithm 1. This algorithm along with Table I describes the functionality of our scheme at the start of step k. If the current average core temperature (c_k) is greater than the threshold (T_{max}), we call the method *decreaseFreqEBTuner()* which is responsible for identifying the *best EB* for which the frequency is to be reduced. On the other hand, if current average core temperature is less than the threshold, the method *increaseFreqEBTuner()* is called which in turn identifies the *best EB* for which to increase the frequency (lines 1-5).

The method *decreaseFreqEBTuner()* iterates over all the *EBs* (lines 8-30) and identifies the *best EB* for which the frequency should be reduced by one step. For each EB_i, (line 8), it first predicts the time per iteration for the application by using profiled information (lines 10-16). While calculating the predicted time per iteration i.e. t_{new}, it uses the execution time corresponding to the frequency level $f_i \downarrow$ which is one level lower than EB_i's current frequency. For all remaining *EBs*, its uses execution time corresponding to the same frequency level at which they operated in step $k-1$ i.e. f_i. It next uses the predicted time per iteration (t_{new}) to predict core power assuming that we reduce the frequency for EB_i. In order to do that, it weighs each *EB* according to the proportion of execution time it takes and accumulates the contribution by each *EB* in p_{new} (lines 18-24).

TABLE I
DESCRIPTION FOR VARIABLES USED IN ALGORITHM 1

Variable	Description
N_{EBs}	number of *EBs* the application is divided into
t_{new}	predicted time per iteration for step $k+1$
p_{new}	predicted core power for step $k+1$
f_i	current frequency level for EB_i
t_i^k	time per iteration for EB_i at frequency level k
t_{old}	time per iteration for step $k-1$
p_i^k	core power of EB_i at frequency level k
b_{best}	best EB to change frequency
g_i	power gradient for EB_i for step k
g_{best}	best power gradient of selected EB for step k

We next calculate the *power gradient* (g_i) for EB_i (line 25) by dividing the difference in the current power (p_{cur}) and the predicted power (p_{new}) by the timing penalty associated with reducing the frequency for EB_i one level lower from its current level (f_i). Lines 26-29 are just keeping track of the *best EB* which has the maximum power gradient (g_{best}). We only provide details for *decreaseFreqEBTuner()* method as the method *increaseFreqEBTuner()* is similar to it. Since *increaseFreqEBTuner()* method is a reaction to core temperatures getting cooler than the threshold, we predict core power assuming an increase in frequency by one level. Hence, instead of using $f_i \downarrow$ on lines 12 and 20, we use $f_i \uparrow$. Since we want to minimize timing penalty, we would want to increase the frequency of an *EB* which results in maximum *decrease* in execution time and causes the *smallest* increase in core power. In terms of our heuristic, we want an *EB* that has the *smallest* power gradient instead of one that had the maximum.

VI. PERFORMANCE RESULTS

Obtaining the core power for an application is vital for our scheme as it affects core temperatures. Intel's Sandy Bridge chip, is a relatively new product that deploys on chip counters to supply core energy consumption data to applications through Machine Specific Registers (MSRs). We use a quad-core machine for all our experiments. It has a quad-core Intel core i7-2600 processor with a maximum frequency of 3.4 GHz that can go up to 3.8GHz with Intel's Turboboost. We used a *Watts Up Pro* power meter to measure the machine energy consumption for all our experiments. The operating system on the node is Ubuntu 10.04 with *lm-sensors* and *coretemp* module installed to provide core temperature readings, and the *cpufreq* module installed to enable software-controlled DVFS. We investigate the effectiveness of our scheme by considering Class A datasets of four different parallel applications from the NAS parallel benchmark [18] suite. These applications have a range of power profiles and vary in the intensity with

Algorithm 1 EBTUNER: START OF STEP k

if $c_k > T_{max}$ **then**
  decreaseFreqEBTuner()
else
  increaseFreqEBTuner()
end if

procedure DECREASEFREQEBTUNER()
  $b_{best} = 0$, $g_{best} = 0$
  for $i = 1, N$
    $t_{new} = 0$
    for $s = 1, N$
      if $s = i$
        $t_{new} = t_{new} + t_i^{f_i\downarrow}$
      else
        $t_{new} = t_{new} + t_s^{f_s}$
      end if
    end for
    $p_{new} = 0$
    for $s = 1, N$
      if $s = i$
        $p_{new} = p_{new} + (t_i^{f_i\downarrow} * p_i^{f_i\downarrow})/t_{new}$
      else
        $p_{new} = p_{new} + (t_s^{f_s} * p_s^{f_s})/t_{new}$
      end if
    end for
    $g_i = (p_{cur} - p_{new})/(t_{new} - t_{old})$
    if $g_i > g_{best}$
      $g_{best} = g_i$
      $b_{best} = i$
    end if
  end for
end procedure

which they use the CPU. We divided the applications so that all the *EBs* have an execution time at least on the order of tens of milliseconds to make sure that the DVFS overhead of 100μs [21] becomes negligible (e.g. with 10 ms *EBs*, the overhead is 1%). All results reported in this work are averages of two similar runs with each run taking more than 10 minutes to ensure that each application settles to its steady state. It is important to note that all the experiments were run on real hardware with actual energy consumption measurements (not models), and there are no simulation results in this paper.

A. Constraining core temperature and its impact on frequency and core power

A primary objective of our scheme is to constrain core temperatures below the user defined maximum temperature threshold. Figure 5 shows the average core temperature across all 4 cores plotted against execution time with a maximum temperature threshold of 54° C. Our scheme was effectively able to constrain core temperature below 54° C throughout the 10 minute runs. However, Figure 5 presents the first 150 seconds of these runs in order to analyze some key differences

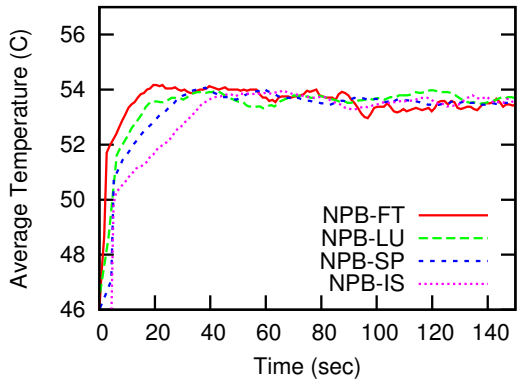

Fig. 5. Temperature profiles with a temperature threshold of 54C using *EBTuner*

TABLE II
STEADY STATE APPLICATION CHARACTERISTICS FOR 54° C THRESHOLD

Description	NPB-FT	NPB-LU	NPB-SP	NPB-IS
MFLOP/s	640	1032	200	0
Frequency	2.24	2.59	2.79	2.88
Timing penalty (%)	6	6	1	1
L1-L2 Traffic (MB/sec)	2416	2114	850	1288
L2-L3 Traffic (MB/sec)	6806	3365	1755	4832
Core Power (W)	35	42	41	40
L3-DRAM Traffic (MB/sec)	372	142	174	668

amongst the applications.

Before analyzing differences in timing penalties amongst applications, we need to analyze temperature gradients shown in Figure 5 as they determine the frequency at which each application runs. *NPB-FT* has the steepest temperature gradient amongst the four applications and is the quickest to reach the temperature threshold of 54° C. On the other hand, *NPB-IS* has the lowest temperature gradient and is the last to reach the temperature threshold of 54° C. The other two applications lie in between these two applications. After looking at the temperature profiles, we now try to relate them with the average frequency for each application plotted against execution time (shown in Figure 6). The average frequency refers to the average of the frequency level used for all *EBs* during each iteration weighted according to the execution time they take.

All applications start at the maximum frequency and as the cores get hotter, DVFS comes into action decreasing their frequencies. We can notice that the average frequencies for all the applications start to decrease in the order in which they reach the threshold temperature (54° C) i.e. *NPB-FT* is the first, followed by *NPB-LU*, *NPB-SP* and *NPB-IS* respectively. Besides that, all applications settle to a different steady state frequency. Table II shows steady-state characteristics for all four applications when running using *EBTuner* with a temperature threshold of 54° C. Both *NPB-FT* and *NPB-LU* end with the same percentage of timing penalty but with average frequencies which are nearly 400 MHz apart. This can be understood if we look at the MFLOPs/sec from Table II. *NPB-*

LU has much higher MFLOPs/sec rate than *NPB-FT* which means that for the same percentage decrease in average frequency, *NPB-LU* should expect a greater timing penalty than *NPB-FT* as the former is more computation bound. Hence, *NPB-LU*, irrespective of suffering a smaller degradation in average frequency, ends up having timing penalty equal to what *NPB-FT* suffers (6%) despite going to an even lower average frequency level. This raises another question: What causes *NPB-FT* to heat much quickly than *NPB-LU* as shown by its higher temperature gradient in Figure 5 despite *NPB-LU*'s much higher MFLOPs/sec rate? This difference is likely caused by the high amount of data transfer going inside the processor (between caches) and to the memory as shown in Table II.

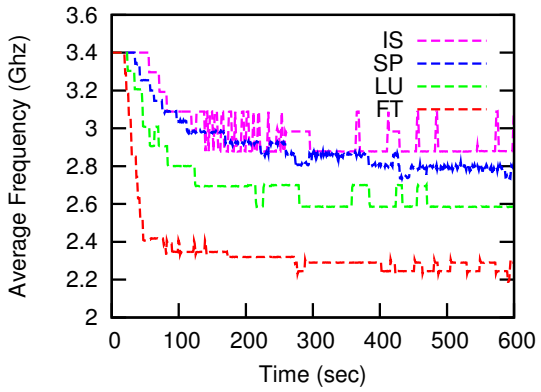

Fig. 6. Average frequency with a temperature threshold of 54° C using *EBTuner*

Since Figure 5 shows all applications settling to the same average core temperature, the laws of thermodynamics dictate that a CPU running at a fixed temperature will transfer a particular amount of heat energy per unit of time to the environment through its heatsink and fan assembly. Thus, each application should end up having the same core power. However the steady-state core power values for all applications from Table II tell a different story. *NPB-LU*, the most compute intensive application considered (Table II), ends up with the highest steady state core power followed by *NPB-SP*, *NPB-IS*, and *NPB-FT* respectively. The most interesting observation is the steady-state core power value (Table II) for *NPB-FT* that is 5W lower than the other three applications. Although core power largely determines core temperatures, we can say it is secondarily dependent on what happens at places nearby i.e. caches and memory controller. *NPB-FT*'s second highest FLOPs/sec rate (only lower than *NPB-LU*) coupled with its high data transfer rate (only lower than *NPB-IS* for main memory access) make it the 'hottest' application.

B. Timing penalty

Now that we have established how core temperatures affect average frequency, let us gain some insights into how the frequency influences the timing penalty. Figure 8 shows timing penalty incurred by each application under DVFS, contrasting

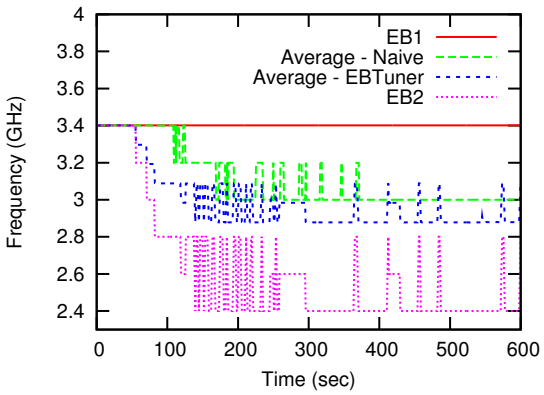

Fig. 7. Frequency comparison of *EBTuner* and *NaiveDVFS* using *NPB-IS* for threshold of 54° C

TABLE III
COMPARISON OF *EBTuner* AND *NaiveDVFS* USING *NPB-IS* (T_{max}=54° C)

Description	MIPs	p_{core} (W)	f (GHz)	Time Penalty(%)
NaiveDVFS	1202	38.51	3.00	5.6
EBTuner	1252	39.49	2.88	0.5

its effect between our scheme, *EBTuner* and *NaiveDVFS* (the strategy mentioned in Section III). Our scheme was able to reduce timing penalty for all temperature thresholds across all four applications. In case of *NPB-IS*, our scheme was able to reduce the timing penalty by more than 50% compared to *NaiveDVFS* for all temperature thresholds. In order to understand the reasons for the improved performance of *EBTuner*, we need to understand the frequency it uses for each *EB*. Figure 7 shows the frequency selected by *EBTuner* for both *EB1* and *EB2* when running with a threshold of 54° C. It also plots the average of both the *EBs* for each iteration i.e. *Average - EBTuner*, and compares it to the frequency selected by the *NaiveDVFS* scheme. In Section IV (Figure 4) we discussed the insensitivity of *EB2* to frequency. Combining that knowledge with our heuristic, we can see that as soon as the temperature for *NPB-IS* hits the threshold value i.e. 54° C (Figure 5), *EBTuner* starts decreasing the frequency of *EB2* owing to its large *power gradient* (due to increase in execution time being very small in Equation 3) . Reducing frequency for *EB2* only, constrains core temperatures without significantly increasing execution time (Table III). However, in case of *NaiveDVFS*, the timing penalty is much larger. Looking at Figure 7 the question arises: Why does *NaiveDVFS* settle at a higher frequency than the average frequency for *EBTuner* and still ends up with a greater timing penalty? Closer analysis reveals that because *EBTuner* keeps *EB1* at maximum frequency and reduce frequency for *EB2* only, it ends up reducing the Million Instructions per second (MIPS) rate only marginally. On the other hand, since *NaiveDVFS* reduces the frequency for the entire application i.e. both *EBs*, it ends up reducing the MIPs rate significantly as *EB1* (the computation intensive) is also run at a lower frequency level. This decrease in MIPs results in a much higher timing penalty (5.6%) compared to *EBTuner*

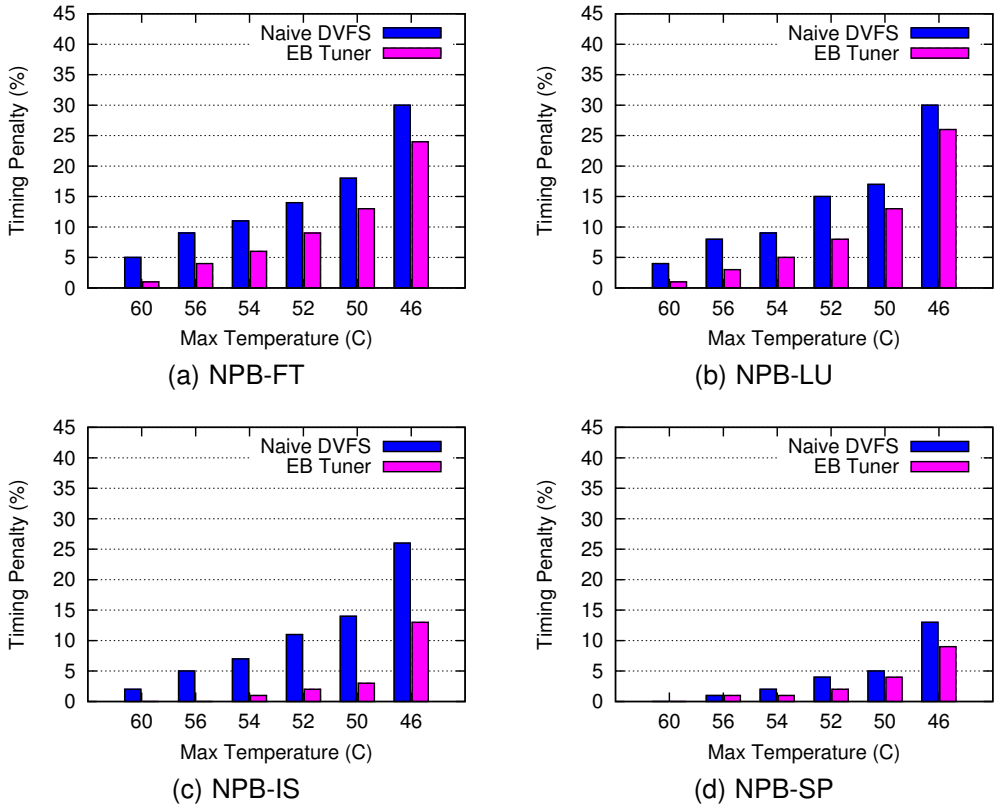

Fig. 8. Timing penalty for different temperature thresholds using Naive DVFS and EB Tuner

as shown in Table III. Irrespective of the different steady state frequencies, Table III shows that for both the cases, the core power is almost the same. We can say that in the case of *NaiveDVFS*, some of the core energy consumption is *wasted* while executing *EB2* at a higher frequency. On the other hand, *EBTuner* removes that inefficiency and consumes the same amount of energy in doing work that increases the MIPS for the application.

To analyze the benefits of our scheme to minimize timing penalty, we define an index that measures the *variance* of sensitivity to frequency amongst different *EBs* of an application. We denote it by σ_{freq} and define it as:

$$\sigma_{freq} = \sqrt{\frac{\sum_{i=1}^{N_{EBs}} (\frac{t_i^{min}}{t_i^{max}} - \frac{T^{min}}{T^{max}})^2}{N_{EBs}}} \quad (4)$$

where N_{EBs} is the number of EBs the application is divided into, t_i^{min} is the average execution time per iteration for EB_i (only) running at the minimum frequency level, t_i^{max} is the average execution time per iteration for EB_i (only) running at the maximum frequency level, and T^{max} is the average execution time per iteration for all *EBs* when running them at the maximum frequency level, and T^{min} is the average execution time per iteration for all *EBs* when running them at the minimum frequency level. Table IV shows these numbers separately along with the 'Average reduction in penalty' for all applications. The 'Average reduction in timing penalty' is the average difference between timing penalties of *NaiveDVFS* and *EBTuner* for all temperature thresholds of each application shown in Figure 8. Table IV suggests a strong correlation between σ_{freq} and 'Average reduction timing penalty'. NPB-IS has the highest value for σ_{freq}, and consequently, gets the highest benefits from our scheme i.e. highest 'Average reduction in timing penalty' value in Table IV. However, despite having very close σ_{freq} to that of *NPB-IS*, the fact that *NPB-FT* has a much smaller 'Average reduction in penalty' value needs further investigation. Compared to *NaiveDVFS*, *EBTuner* reduces timing penalty for *NPB-SP* by 1.6% because of all the *EBs* having very similar frequency sensitivities (σ_{freq}=0.19).

TABLE IV
VARIANCE IN FREQUENCY SENSITIVITY AMONGST *EBs* FOR ALL APPLICATIONS

Description	NPB-FT	NPB-LU	NPB-SP	NPB-IS
t_1^{max}/t_1^{min}	1.01	2.08	1.58	2.16
t_2^{max}/t_2^{min}	2.1	2.19	1.25	1.07
t_3^{max}/t_3^{min}	1.99	1.20	1.14	X
T^{max}/T^{min}	1.49	1.68	1.32	1.6
σ_{freq}	0.53	0.47	0.19	0.55
Average reduction in penalty(%)	5.1	4.9	3.0	7.7

C. Reducing energy consumption and its tradeoff

In our earlier work [7] we have shown that constraining core temperatures can reduce cooling energy consumption by a considerable amount. The focus of this work is to save the *other* major part of energy consumption i.e. machine energy. Figure 9 compares the reduction in machine energy consumption of *EBTuner* and *NaiveDVFS* for all applications using different temperature thresholds. *EBTuner* generally saves more energy consumption for all the applications other than *NPB-SP* because of its smaller timing penalties.

Reduction in energy consumption for load balanced applications generally comes at the cost of execution time. Figure 10 summarizes the essence of our results by plotting the normalized execution time against normalized machine energy consumption for representative applications using different temperature thresholds. Normalization for each application was done with respect to runs where all cores were working at maximum frequency without any temperature control.

These curves give important information: the slope of each curve represents the execution time penalty one must pay in order to save each joule of energy. A movement to the left (reducing the energy consumption) or down (reducing the timing penalty) is desirable. It is clear that for all temperature thresholds across all applications (except for *NPB-SP*), *EBTuner* takes its corresponding point from the Naive DVFS scheme at the same temperature threshold down (saving timing penalty) and to the left (saving energy consumption). For *NPB-IS* and *NPB-SP*, the relatively flat curves show that our scheme does well at saving energy consumption. However, in case of *NPB-SP*, since its σ_{freq} is smaller, *EBTuner* performs almost the same as *NaiveDVFS*. On the other hand, *NPB-FT* and *NPB-LU* (not shown in Figure 10 due to space limitations) have similar, but much steeper curves that imply a high cost for saving energy consumption. However, *EBTuner* brings significant benefits for them compared to *NaiveDVFS*. Figure 9(a) and Figure 8(a) show that *EBTuner* can reduce machine energy by 17% with less than 1% timing penalty while constraining core temperatures below 60° C. On the other hand, *Naive DVFS* can save the same amount of machine energy consumption by paying more than 11% in timing penalty!

VII. Conclusions and Future Work

In this paper, we proposed an approach to constrain core temperature to save both machine and cooling energy consumption with minimum timing penalty. Our scheme uses a combination of DVFS and fine-grain power and performance profiling to achieve this objective. We experimented on four of the NAS parallel benchmarks to demonstrate its substantial benefits in minimizing execution time penalty and reducing machine energy consumption. Furthermore, through detailed analysis, we relate *EB* characteristics to timing penalty for constraining core temperatures, and expressed it mathematically (σ_{freq}) to provide a guide on what to expect from various applications. According to our findings, applications having high σ_{freq} values would maximize the benefits of using *EBTuner* over using the *NaiveDVFS* approach. Our scheme was able to outperform *NaiveDVFS* for all applications in reducing timing penalty. It was also able to reduce machine energy consumption by a greater percentage than *NaiveDVFS* for 3 out of 4 applications. In case of *NPB-FT*, our scheme was able to reduce machine energy consumption by 17% with a timing penalty of less than 1% while constraining core temperatures below 60° C.

In our earlier work [7], we showed that constraining core temperatures can result in significant reduction in cooling energy consumption. In this work, we exploited different frequency sensitivities for different parts of the application (*EBs*) in order to minimize timing penalty and maximize reduction in machine energy consumption. In future we plan to combine both of these techniques for multi-node clusters in order to come up with a load balancer that will place tasks insensitive to frequency on hotter processors in order to minimize execution time penalty and consequently reduce *total* energy consumption.

Acknowledgments

This work was partially supported by the US Department of Energy under grant DOE DE-SC0001845.

References

[1] Top500, "Top500 supercomputer sites," http://www.top500.org.

[2] R. F. Sullivan, "Alternating cold and hot aisles provides more reliable cooling for server farms," White Paper, Uptime Institute, 2000.

[3] C. D. Patel, C. E. Bash, R. Sharma, M. Beitelmal, and R. Friedrich, "Smart cooling of data centers," *ASME Conference Proceedings*, vol. 2003, no. 36908b, pp. 129–137, 2003.

[4] R. Sawyer, "Calculating total power requirements for data centers," *White Paper, American Power Conversion*, 2004.

[5] M. S. Warren, E. H. Weigle, and W.-C. Feng, "High-density computing: A 240-processor beowulf in one cubic meter," 2002.

[6] R. Viswanath, V. Wakharkar, A. Watwe, V. Lebonheur, M. Group, and I. Corp, "Thermal performance challenges from silicon to systems," 2000.

[7] O. Sarood and L. V. Kalé, "A 'cool' load balancer for parallel applications," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[8] I. Corporation, "2nd generation intel core processor family."

[9] C. Bash and G. Forman, "Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center," in *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 29:1–29:6.

[10] L. Wang, G. von Laszewski, J. Dayal, and T. Furlani, "Thermal aware workload scheduling with backfilling for green data centers," in *Proceedings of the 2009 IEEE 28th International Performance Computing and Communications Conference (IPCCC)*, December 2009.

[11] L. Wang, G. von Laszewski, J. Dayal, X. He, A. Younge, and T. Furlani, "Towards thermal aware workload scheduling in a data center," in *International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, December 2009.

[12] Q. Tang, S. Gupta, D. Stanzione, and P. Cayton, "Thermal-aware task scheduling to minimize energy usage of blade server based datacenters," in *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, 2006.

[13] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding Energy Consumption in Large-scale MPI Programs," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2007, pp. 49:1–49:9.

[14] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, pp. 835–848, June 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1263127.1263246

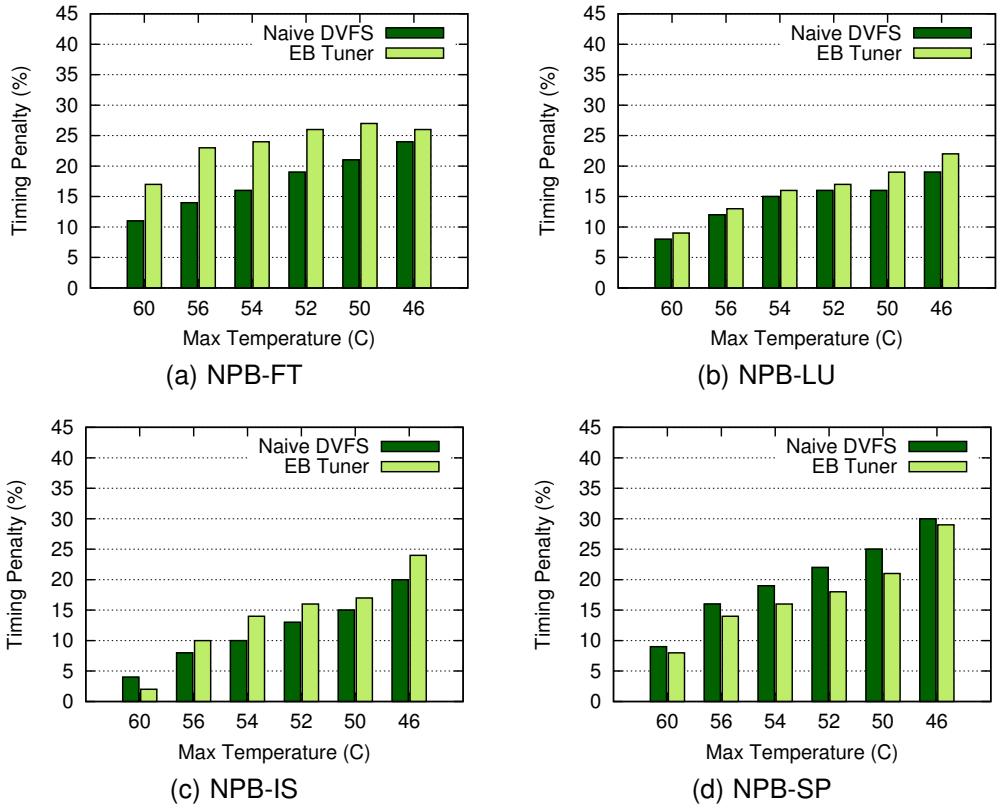

Fig. 9. Machine energy savings for different temperature thresholds using Naive DVFS and EB Tuner

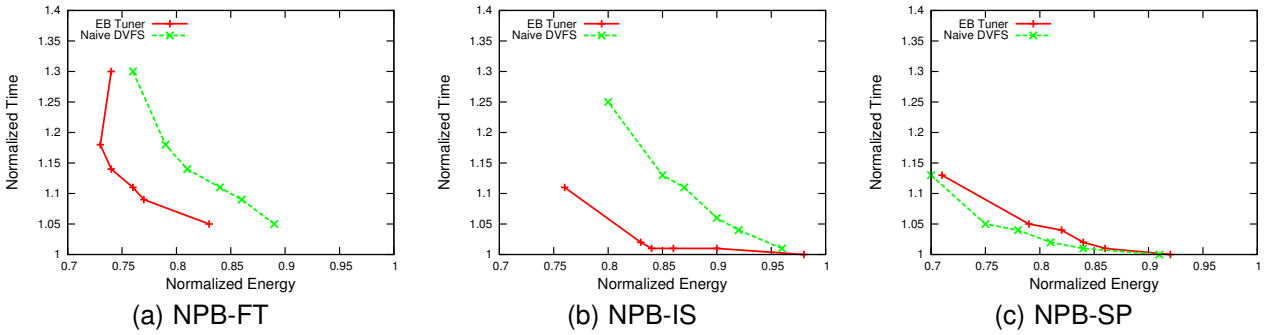

Fig. 10. Normalized timing penalty and machine energy consumption for different temperature thresholds using both *EBTuner* and *NaiveDVFS*

[15] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron, "Cpu miser: A performance-directed, run-time system for power-aware clusters," in *Proceedings of the 2007 International Conference on Parallel Processing*, ser. ICPP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 18–.

[16] K. W. Cameron, H. K. Pyla, and S. Varadarajan, "Tempest: A portable tool to identify hot spots in parallel code," in *Proceedings of the 2007 International Conference on Parallel Processing*, ser. ICPP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–.

[17] D. C. Knowledge, "Google: Raise your data center temperature," http://www.datacenterknowledge.com/archives/2008/10/14/google-raise-your-data-center-temperature/.

[18] D. B. E. B. J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-04-077, 1994.

[19] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Scheduler-based dram energy management," in *IN PROCEEDINGS OF THE 39TH CONFERENCE ON DESIGN AUTOMATION*. ACM Press, 2002, pp. 697–702.

[20] L. Kalé, "A tutorial introduction to Charm," Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Tech. Rep. 92-6, 1992.

[21] O. Khan and S. Kundu, "Predictive thermal management for chip multiprocessors using co-designed virtual machines," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 293–307.