"Cool" Load Balancing for High Performance Computing Data Centers

Osman Sarood, Phil Miller, Ehsan Totoni, and Laxmikant V. Kalé, *Fellow*, *IEEE*

Abstract—As we move to exascale machines, both peak power demand and total energy consumption have become prominent challenges. A significant portion of that power and energy consumption is devoted to cooling, which we strive to minimize in this work. We propose a scheme based on a combination of limiting processor temperatures using dynamic voltage and frequency scaling (DVFS) and frequency-aware load balancing that reduces cooling energy consumption and prevents hot spot formation. Our approach is particularly designed for parallel applications, which are typically tightly coupled, and tries to minimize the timing penalty associated with temperature control. This paper describes results from experiments using five different CHARM++ and MPI applications with a range of power and utilization profiles. They were run on a 32-node (128-core) cluster with a dedicated air conditioning unit. The scheme is assessed based on three metrics: the ability to control processors' temperature and hence avoid hot spots, minimization of timing penalty, and cooling energy savings. Our results show cooling energy savings of up to 63 percent, with a timing penalty of only 2-23 percent.

Index Terms—Green IT, temperature aware, load balancing, cooling energy, DVFS

1 Introduction

Energy consumption has emerged as a significant issue in modern high-performance computing systems. Some of the largest supercomputers draw more than 10 megawatts, leading to millions of dollars per year in energy bills. What is perhaps less well known is the fact that 40 to 50 percent of the energy consumed by a data center is spent in cooling [1], [2], [3], to keep the computer room running at a lower temperature. How can we reduce this cooling energy?

Increasing the thermostat setting on the computer room air-conditioner (CRAC) will reduce the cooling power. But this will increase the ambient temperature in the computer room. The reason the ambient temperature is kept cool is to keep processor cores from overheating. If they run at a high temperature for a long time, the processor cores may be damaged. Additionally, cores consume more energy per unit of work when run at higher temperatures [4]. Further, due to variations in the air flow in the computer room, some chips may not be cooled as effectively as the rest. Semiconductor process variation will also likely contribute to variability in heating, especially in future processor chips. So, to handle such "hot spots", the ambient air temperature is kept at a low temperature to ensure that no individual chip overheats.

Modern 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*, or DVFS. Running processor cores at a lower frequency (and correspondingly lower voltage) reduces the thermal energy they dissipate, leading to a cool-down.

This suggests a method for keeping processors cool while increasing the CRAC set-point (i.e., the thermostat setting). A component of the application software can periodically check the temperature of the chip. When it exceeds a preset threshold, the software can reduce the frequency and voltage of that particular chip. If the temperature is lower than a threshold, the software can correspondingly increase the frequency.

This technique will ensure that no processors overheat. However, in HPC computations, and specifically in tightly coupled science and engineering simulations, this creates a new problem. Generally, computations on one processor are dependent on the data produced by the other processors. As a result, if one processor slows down to half its original speed, the entire computation can slow substantially, in spite of the fact that the remaining processors are running at full speed. Thus, such an approach will reduce the cooling *power*, but increase the execution time of the application. Running the cooling system for a longer time can also *increase* the cooling energy.

We aim to reduce cooling power without substantially increasing execution time, and thus reduce cooling energy. We first describe the temperature sensor and frequency control mechanisms, and quantify their impact on execution time mentioned above (Section 3). Our solution leverages the adaptive runtime system underlying the Charm++ parallel programming system (Section 4). In order to minimize *total* system energy consumption, we study an approach of limiting CPU temperatures via DVFS and mitigating the resultant timing penalties with a load balancing strategy that is conscious of these effects (Section 5). We show the impact of this combined technique on application performance (Section 7) and *total* energy consumption (Section 8).

• *The authors are with the Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N Goodwin Ave., Urbana, IL 61801.*
E-mail: {sarood1, mille121}@illinois.edu.

Manuscript received 5 Nov. 2011; revised 22 May 2012; accepted 28 May 2012; published online 12 June 2012.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2011-11-0870.
Digital Object Identifier no. 10.1109/TC.2012.143.

This paper is a substantially revised version of a conference paper presented at Supercomputing 2011 [5]. The new material includes performance data on MPI benchmarks, in contrast to just the Charm++ applications presented previously, refinements of the earlier techniques, performance data on the new Sandy Bridge processor, including power sensors not available on the older hardware in our cluster, and substantially expanded experimental data.

2 Related Work

Cooling energy optimization and hot spot avoidance have been addressed extensively in the literature of non-HPC data centers [6], [7], [8], [9], which shows the importance of the topic. As an example, job placement and server shut down have shown savings of up to 33 percent in cooling costs [6]. 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 because different nodes are running parts of the same application with similar power consumption. As an example, Rajan and Yu [10] use system throttling for temperature-aware scheduling in the context of operating systems. Given their assumptions, they show that keeping temperature constant is beneficial with their theoretical models. However, their assumption of nonmigratability of tasks is not true in HPC applications, especially with an adaptive runtime system. Le et al. [11] constrain core temperatures by turning the machines on and off and consequently reduce total energy consumption by 18 percent. However, most of these techniques, cannot be applied to HPC applications as they are not practical for tightly coupled applications.

Minimizing energy consumption has also been an important topic for HPC researchers. However, most of the work has focused on machine energy consumption rather than cooling energy. Freeh et al. [12] show machine energy savings of up to 15 percent by exploiting the communication slack present in the computational graph of a parallel application. Lim et al [13] demonstrate a median energy savings of 15 percent by dynamically adjusting the CPU frequency/voltage pair during the communication phases in MPI applications. Springer et al. [14] generate a frequency schedule for a DVFS-enabled cluster that runs the target application. This schedule tries to minimize the execution time while staying within the power constraints. The major difference of our approach to the ones mentioned is that our DVFS decisions are based on saving cooling energy consumption by constraining core temperatures. The *total* energy consumption savings that we report represent both machine and cooling energy consumption.

Huang and Feng describe a kernel-level DVFS governor that tries to determine the power-optimal frequency for the expected workload over a short time interval that reduces machine energy consumption up to 11 percent [15]. Hanson et al. [16] devise a runtime system named PET for performance, power, energy, and thermal management. They consider a more general case of multiple and dynamic constraints. However, they just consider a serial setting without the difficulties of parallel machines and HPC applications. Extending our approach for constraints other than temperature is an interesting future work.

Banerjee et al. [17] try to improve the cooling cost in HPC data centers by an intelligent job placement algorithm yielding up to 15 percent energy savings. However, they do not consider the temperature variations inside a job, so there is no control over the applications. Thus, their approach can be less effective for data centers with a few large-scale jobs rather than many small jobs. They also depend on job preruns to get information about the jobs. In addition, their results are based on simulations and not experiments on a real testbed. Tang et al. [18] reduce 30 percent of cooling energy consumption by scheduling tasks in a data center. However, large-scale parallel jobs' considerations are an issue there too.

Merkel and Bellosa [19] discuss the scheduling of tasks in a multiprocessor to avoid hot cores. However, they do not deal with complications of parallel applications and large-scale data centers. Freeh and Lowenthal [20] exploit the varying sensitivity of different phases in the application to core frequency in order to reduce machine energy consumption for load balanced applications. This work is similar to ours, as it deals with load-balanced applications. They reduce machine energy consumption by a maximum of 16 percent. However, our work is different as we achieve much higher savings in *total* energy consumption primarily by reducing cooling energy consumption.

3 Limiting Temperatures

The design of a machine room or data center must ensure that all equipment stays within its safe operating temperature range while keeping costs down. Commodity servers and switches draw cold air from their environment, pass it over processor heat sinks and other hot components, and then expel it at a higher temperature. To satisfy these systems' specifications and keep them operating reliably, cooling systems in the data center must supply a high enough volume of sufficiently cold air to every piece of equipment.

Traditional data center designs treated the air in the machine room as a single mass, to be kept at an acceptable aggregate temperature. If the air entering some device was too hot, it meant that the CRAC's thermostat should be adjusted to a lower set point. That adjustment would cause the CRAC to run more frequently or intensely, increasing its energy consumption. More modern designs, such as alternating hot/cold aisles [1] or in-aisle coolers, provide greater separation between cold and hot air flows and more localized cooling, easing appropriate supply to computing equipment and increasing efficiency.

However, even with this tighter air management, variations in air flow, system design, manufacturing and assembly, and workload may still leave some devices significantly hotter than others. To illustrate this sensitivity, we run an intensive parallel application on a cluster with a dedicated CRAC unit whose set-point we can manipulate. Details of this setup are described in Section 6. Fig. 1 shows two runs of the application with different CRAC set-point temperatures. For each run, we plot both the average core temperature across the entire cluster, and the maximum deviation of any core from that average.

Unsurprisingly, observed core temperatures correlate with the temperature of the air provided to cool them. With

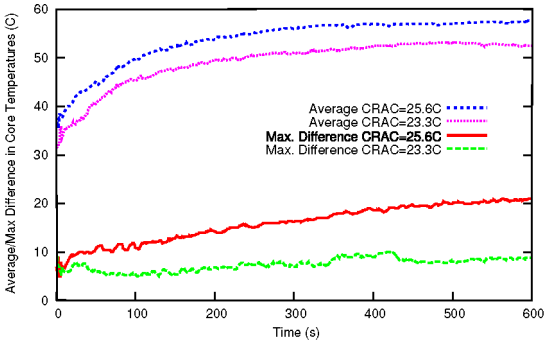


Fig. 1. Average core temperatures and maximum difference of any core from the average for *Wave2D*.

a set-point increase of $2.3°C$, the average temperature across the system increases by $6°C$. More noteworthy is that this small shift creates a substantial hot spot, that worsens progressively over the course of the run. At the higher $25.6°C$ set point, the temperature difference from the average to the maximum rises from 9 to $20°C$. In normal operations, this would be an unacceptable result, and the CRAC set point must be kept low enough to avoid it.

An alternative approach, based on DVFS, shows promise in addressing the issue of overcooling and hot spots. DVFS is already widely used in laptops, desktops, and servers in non-HPC data centers as a means to limit CPU power consumption. However, applying DVFS naïvely to HPC workloads entails an unacceptable performance degradation. Many HPC applications are tightly coupled, such that one or a few slow cores would effectively slow down an entire job. This *timing penalty* implies decreased throughput and increased time-to-solution.

To demonstrate the impact of DVFS, we repeat the earlier experiment with a temperature constraint. We fix a threshold temperature of $44°C$ that we wish to keep all CPUs below. We sample temperatures periodically, and when a CPU's average temperature is over this threshold, its frequency is lowered by one step, i.e., increase P-state by a level. If it is more than a degree below the threshold, its frequency is increased by one step, i.e., decrease P-state by a level. We repeat this experiment over a range of CRAC settings, and compute their performance in time and energy consumption relative to a run with all cores working at their maximum frequency and the CRAC set to $12.2°C$. As shown in Fig. 2, DVFS alone in this setting hurts performance and

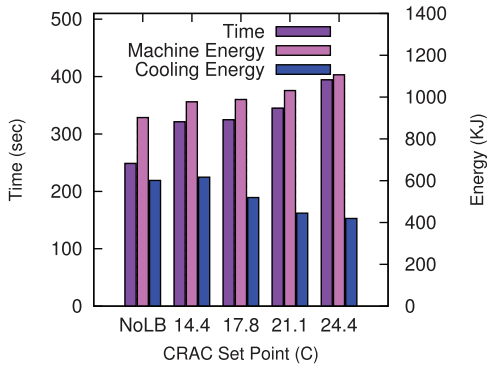


Fig. 2. Execution time and energy consumption for *Wave2D* running at different CRAC set points using DVFS.

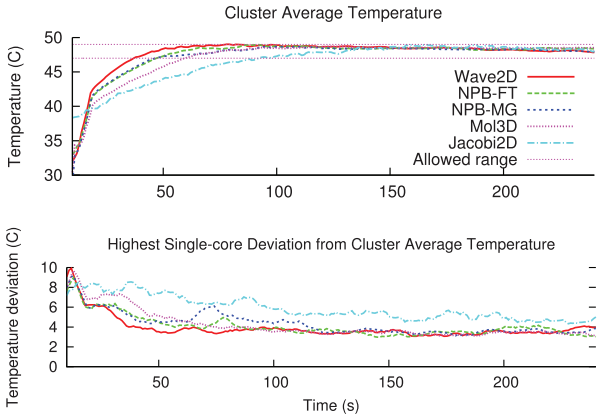


Fig. 3. Our DVFS and load balancing scheme successfully keeps all processors within the target temperature range of 47-$49°C$, with a CRAC set point of $24.4°C$.

provides minimal savings in *total* energy consumption. Most of the savings from cooling energy consumption are offset by an increase in machine energy consumption. Nevertheless, our results in Fig. 3 (described in detail in Section 7) show that DVFS effectively limits both overall temperatures and hot spots.

More radical liquid-cooling designs mitigate some of the hot spot concerns, but 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. Moreover, our techniques for limiting core temperatures can actually reduce the overall thermal load of an HPC system, leading to energy savings even for installations using liquid cooling.

4 CHARM++ AND LOAD BALANCING

CHARM++ is a general-purpose C++-based parallel programming system designed for productive HPC programming [21]. It is supported by an adaptive runtime system that automates resource management. It relies on techniques such as processor virtualization and overdecomposition (having more work units than the number of cores) to improve performance via adaptive overlap of computation and communication and data-driven execution. This means that the developer does not need to program in terms of the physical cores, but instead divides the work into pieces with a suitable grain size to let the system manage them easily.

A key feature of CHARM++ is that the units of work decomposition are *migratable objects*. The adaptive runtime system can assign these objects to any processor and move them around during program execution, for purposes including load balancing, communication optimization, and fault tolerance. To enable effective load balancing, it tracks statistics of each object's execution, including its computation time and communication volume [22].

The runtime system provides a variety of plug-in *load balancing strategies* that can account for different application characteristics. Through a simple API, these strategies take

the execution statistics from the runtime and generate a set of migration instructions, describing which objects to move between which processors. Application developers and users can provide their own strategy implementations as desired. Load balancing strategies can be chosen at compilation or runtime. The majority of these strategies are based on the heuristic "principle of persistence," which states that each object's computation and communication loads tend to persist over time. The principle of persistence holds for a large class of iterative HPC applications. In this study, we have developed a new load balancing strategy that accounts for the performance effects of DVFS-induced heterogeneity. The new strategy is described in detail in Section 5.

At small scales, the cost of the entire load balancing process, from instrumentation through migration, is generally a small portion of the total execution time, and less than the improvement it provides. Where this is not immediately the case, a strategy must be chosen or adapted to match the application's needs [23]. Our approach can be easily adapted to available hierarchical schemes, which have been shown to scale to the largest machines available [24]. By limiting the cost of decision making and scope of migration, we expect these schemes to offer similar energy benefits.

4.1 AMPI

The Message Passing Interface (MPI) is a standardized communication library for distributed-memory parallel programming. MPI has become the dominant paradigm for large-scale parallel computing. Thus, techniques for addressing the energy consumption of large parallel systems must be applicable to MPI applications.

CHARM++ provides an implementation of MPI known as Adaptive MPI (AMPI). AMPI makes the features of the CHARM++ runtime system available to MPI programs. Common MPI implementations implement each unit of parallel execution, or *rank*, as a separate process. Pure MPI applications run one rank per CPU core, while others use fewer ranks and gain additional shared-memory parallelism via threading.

In contrast, AMPI encourages running applications with several ranks per core. AMPI implements these ranks as light-weight user-level threads, many of which can run in each process. The runtime schedules these threads nonpreemptively, and switches them when they make blocking communication calls. Internally, these threads are implemented as migratable objects, enabling the same benefits for MPI programs as for native CHARM++. In particular, AMPI allows us to apply the CHARM++ load balancing strategies without intrusive modifications to application logic.

5 Temperature-Aware Load Balancing Approach

In this section, we introduce a novel approach that reduces energy consumption of the system with minimal timing penalty. It is based on limiting core temperatures using DVFS and task migration. Because our scheme is tightly coupled to task migration, we chose CHARM++ and AMPI as our parallel programming frameworks as they allow easy task (object) migration with low overhead. All the implementations and experiments were done using CHARM++ and AMPI. However, our techniques can be applied to any parallel programming system that provides efficient task migration.

The steps of our temperature-control scheme can be summarized as applying the following process periodically:

1. Check the temperatures of all cores.
2. Apply DVFS to cores that are hotter or colder than desired.
3. Address the load imbalance caused by DVFS using our load balancer, *TempLDB*:
    a. Normalize task and core load statistics to reflect old and new frequencies.
    b. Identify overloaded or underloaded cores.
    c. Move work from overloaded cores to underloaded cores.

The remainder of this section describes this process in detail.

Our temperature control scheme is periodically triggered after equally spaced intervals in time, referred to as *steps*. Other DVFS schemes [15] try to react directly to the demands of the application workload, and thus must sample conditions and make adjustments at intervals on the order of milliseconds. In contrast, our strategy only needs to react to much slower shifts in chip temperature, which occur over intervals of seconds. At present, DVFS is triggered as part of the runtime's load balancing infrastructure at a user-specified period.

Our control strategy for DVFS is to let the cores work at their maximum frequency as long as their temperature is below a threshold parameter. If a core's temperature crosses above the threshold, it is controlled by decreasing the voltage and frequency using DVFS. When the voltage and frequency are reduced, power consumption will drop and hence the core's temperature will fall. Our earlier approach [5] raised the voltage and frequency as soon as temperatures fell below the threshold, causing frequent changes and requiring effort to load balance in every interval. To reduce overhead, our strategy now waits until a chip's temperature is a few degrees below the threshold before increasing its frequency.

The hardware in today's cluster computers does not allow reducing the frequency of each core individually and so we must apply DVFS to the whole chip. This raises the question: what heuristic should we use to trigger DVFS and modulate frequency? In our earlier work [4], we conducted DVFS when *any* of the cores on a chip were considered too hot. However, our more recent results [5] show that basing the decision on *average* temperature of the cores in a chip results in better temperature control.

Another important decision is how much a chip's frequency should be reduced (respectively, raised) when it gets too hot (is safe to warm up). Present hardware only offers discrete frequency and voltage levels built into the hardware, the "P-states." Using this hardware, we observed that reducing the chip's frequency by one level at a time is a reasonable heuristic because it effectively constrains the core temperatures in the desired range (Fig. 3). Lines 1-6 of Algorithm 1 apply DVFS as we have just described. The description of the variables and functions used in the algorithm is given in Table 1.

TABLE 1
Description for Variables Used in Algorithm 1

Variable	Description
$\mathbf{n}$	number of tasks in application
$\mathbf{p}$	number of cores
T_{max}	maximum temperature allowed
k	current load balancing step
C_i	set of cores on same chip as core i
$taskTime_i^k$	execution time of task i during step k (in ms)
$coreTime_i^k$	time spent by core i executing tasks during step k
f_i^k	frequency of core i during step k (in Hz)
m_i^k	core number assigned to task i during step k
$\{task, core\}Ticks_i^k$	num. of clock ticks taken by i^{th} task/core during step k
t_i^k	average temperature of chip i at start of step k (in $^\circ$C)
$overHeap$	heap of overloaded cores
$underSet$	set of underloaded cores

Algorithm 1. Temperature Aware Refinement Load Balancing

1: On every node i at start of step k
2: **if** $t_i^k > T_{max}$ **then**
3: decreaseOneLevel(C_i) {increase P-state}
4: **else if** $t_i^k < T_{max} - 2$ **then**
5: increaseOneLevel(C_i) {decrease P-state}
6: **end if**
7: On Master core
8: **for** $i \in [1, n]$ **do**
9: $taskTicks_i^{k-1} = taskTime_i^{k-1} \times f_{m_i^{k-1}}^{k-1}$
10: $totalTicks \mathrel{+}= taskTicks_i^{k-1}$
11: **end for**
12: **for** $i \in [1, p]$ **do**
13: $coreTicks_i^{k-1} = coreTime_i^{k-1} \times f_i^{k-1}$
14: $freqSum \mathrel{+}= f_i^k$
15: **end for**
16: createOverHeapAndUnderSet()
17: **while** $overHeap$ NOT NULL **do**
18: $donor =$ deleteMaxHeap $(overHeap)$
19: $(bestTask, bestCore) =$ getBestCoreAndTask($donor, underSet$)
20: $m_{bestTask}^k = bestCore$
21: $coreTicks_{donor}^{k-1} \mathrel{-}= taskTicks_{bestTask}^{k-1}$
22: $coreTicks_{bestCore}^{k-1} \mathrel{+}= taskTicks_{bestTask}^{k-1}$
23: updateHeapAndSet()
24: **end while**
25:
26: **procedure isHeavy(i)**
27: return $coreTicks_i^{k-1} > (1 + tolerance) * totalTicks * (f_i^k / freqSum)$
28:
29: **procedure isLight(i)**
30: return $coreTicks_i^{k-1} < totalTicks * f_i^k / freqSum$

When DVFS adjusts frequencies differently across the cores in a cluster, the workloads on those cores change relative to one another. Because this potential for load imbalance occurs all at once, at an easily identified point in the execution, it makes sense to react to it immediately. The system responds by rebalancing the assignment of work to cores according to the strategy described by lines 7-30 of Algorithm 1.

The key principle in how a load balancer must respond to DVFS actuation is that the load statistics must be adjusted to reflect the various different frequencies at which load measurements were recorded and future work will run. At the start of step k, our load balancer retrieves load information for step $k-1$ from CHARM++'s database. This data give the total duration of work executed for each task in the previous interval ($taskTime_i^{k-1}$) and the core that executed it (m_i^{k-1}). Here, i refers to task id and $k-1$ represents last step. We normalize the task workloads by multiplying their execution times by the old frequency values of the core they executed on, and sum them to compute the total load, as seen in lines 8-11. This normalization is an approximation to the performance impact of different frequencies, whose validity can be seen in Fig. 8. However, different applications might have different characteristics (e.g., cache hit rates at various levels, instructions per cycle) that determine the sensitivity of their execution time to core frequency. We plan to incorporate more detailed load estimators in our future work. The scheme also calculates the work assigned to each core and sum of frequencies for all the cores to be used later (lines 12-15).

Once the load normalization is done, we create a *max heap* for overloaded cores ($overHeap$) and a *set* for the underloaded cores ($underSet$) on line 16. The cores are classified as overloaded and underloaded by procedures $isHeavy()$ and $isLight()$ (lines 26-30), based on how their normalized loads from the previous step, $k-1$, compare to the frequency-weighted average load for the coming step k. We use a tolerance in identifying overloaded cores to focus our efforts on the worst instances of overload and minimize migration costs. In our experiments, we set the tolerance to 0.07, empirically chosen for the slight improvement it provided over the lower values used in our previous work.

Using these data structures, the load balancer iteratively moves work away from the most overloaded core ($donor$, line 18) until none are left (line 17). The moved task and recipient are chosen as the heaviest task that the $donor$ could transfer to any underloaded core such that the underloaded core doesn't become overloaded (line 19, implementation not shown). Once the chosen task is reassigned (line 20), the load statistics are updated and the data structures are updated accordingly (lines 21-23).

6 Experimental Setup

To evaluate our approach to reducing energy consumption, we must be able to measure and control core frequencies and temperatures, air temperature, and energy consumed by computer and cooling hardware. It is important to note that all the experiments were run on real hardware, and there are no simulation results in this paper.

We tested our scheme on a cluster of 32 nodes (128 cores). Each node has a single socket with a four-core Intel Xeon X3430 processor chip. Each chip can be set to 10 different frequency levels ("P-states") between 1.2 and 2.4 GHz. It also supports Intel's TurboBoost [25], allowing some cores to overclock up to 2.8 GHz. The operating system on the nodes is CentOS 5.7, with the `lm-sensors` and `coretemp` modules installed to provide core temperature readings,

and the `cpufreq` module installed to enable software-controlled DVFS. The cluster nodes are connected by a 48-port gigabit ethernet switch. We use a Liebert power distribution unit installed on the rack containing the cluster to measure the machine power at 1 second intervals on a per-node basis. We gather these readings for each experiment and integrate them over the execution time to come up with the total machine energy consumption.

The machine and CRAC are hosted in a dedicated machine room by the Department of Computer Science at the University of Illinois at Urbana-Champaign. This CRAC is an air cooler fed by chilled water from a campus plant. It achieves the temperature set point prescribed by the operator by manipulating the flow of the chilled water. The exhaust air coming from the machine room with temperature T_{hot} is compared to the set point and the water flow is adjusted accordingly. This cooling design is similar to the cooling systems of most large data centers. We were able to vary the CRAC set point across a broad range as shown in our results (following sections).

Because the CRAC unit exchanges machine room heat with chilled water supplied by a campus-wide plant, measuring its direct energy consumption (i.e., with an electrical meter) would only include the mechanical components driving air and water flow, and would miss the much larger energy expenditure used to actually cool the water. To more realistically capture the machine room's cooling energy, we use a model [11] based on measurements of how much heat the CRAC actually expels. The instantaneous power consumed by the CRAC to cool the temperature of the exhaust air from T_{hot} down to the cool inlet air temperature T_{ac} can be approximated by

$$P_{ac} = c_{air} * f_{ac} * (T_{hot} - T_{ac}). \quad (1)$$

In this equation, c_{air} is the heat capacity constant and $f_{ac}$is the constant rate of air flow through the cooling system. We use temperature sensors on the CRAC's vents to measure T_{hot} and T_{ac}. During our experiments, we recorded a series of measurements from each of these sensors, and then integrated the calculated power to produce total energy figures.

We believe that by working in a dedicated space, the present work removes a potential source of error from previous data center cooling results. Most data centers have many different jobs running at any given time. Those jobs dissipate heat, interfering with cooling energy measurements and increasing the ambient temperature in which the experimental nodes run. In contrast, our cluster is the only heat source in the space, and the CRAC is the primary sink for that heat.

We investigate the effectiveness of our scheme, using five different applications, of which three are CHARM++ applications and two are written in MPI. These applications have a range of CPU utilizations and power profiles. The first one is *Jacobi2D*, which is a canonical benchmark that iteratively applies a five-point stencil over a 2D grid of points. At the end of each iteration, instead of a reduction to test for convergence, all the processors send a message to processor zero, which causes more idle time. We used this version of Jacobi to have some slack in the computation of each processor and investigate our scheme's behavior in this case. The second application is *Wave2D*, which uses a finite difference scheme over a 2D discretized grid to calculate the pressure resulting from an initial set of perturbations. The third application is a classical molecular dynamics code called *Mol3D*. Our MPI applications are MG and FT from the NAS parallel benchmarks suite [26].

Most of our experiments were run for 300 seconds as it provided ample time for all the applications to settle to their steady state frequencies. All the results we show are averaged over three identically configured runs, with a cool-down period before each. All normalized results are reported with respect to a run where all 128 cores were running at the maximum possible frequency with Intel Turbo Boost in operation and the CRAC set to $12.2°C$. To validate the ability of our scheme to reduce energy consumption for longer execution times, we ran *Wave2D* (the most power hungry of the five applications we consider) for two and a half hours. The longer run was consistent with our findings, with the temperature being constrained well within the specified range and we were able to reduce cooling energy consumption for the entire two and half hour period.

7 Constraining Core Temperatures and Timing Penalty

The approach we have described in Section 5 constrains processor temperatures with DVFS while attempting to minimize the resulting timing penalty. Fig. 3 shows that all of our applications when using DVFS and *TempLDB*, settle to an average temperature that lies in the desired range (the two horizontal lines at 47 and $49°C$ on Fig. 3). As the average temperature increases to its steady-state value, the hottest single core ends up no more than $6°C$ above the average (lower part of Fig. 3) as compared to $20°C$ above average for the run where we are not using temperature control (Fig. 1).

Fig. 4 shows the timing penalty incurred by each application under DVFS, contrasting its effect with and without load balancing. The effects of DVFS on the various applications are quite varied. The worst affected, Wave2D and NAS MG, see penalties of over 50 percent, which load balancing reduces to below 25 percent. Jacobi2D was least affected, with a maximum penalty of 12 percent, brought down to 3 percent by load balancing. In all cases, the timing penalty sharply decreases when load balancing is activated, generally by greater than 50 percent. Before analyzing timing penalty for individual applications let us first see how load balancing reduces the timing penalty compared to naive DVFS.

To illustrate the benefits of load balancing, we use Projections, which is a multipurpose performance visualization tool for CHARM++ applications. Here, we use processor timelines to see the utilization of the processors in different time intervals. For ease of comprehension, we show a representative 16-core subset of the 128-core cluster. The top part of Fig. 5 shows the timelines for execution of *Wave2D* with the naive DVFS scheme. Each timeline (horizontal line) corresponds to the course of execution of one core visualizing its utilization. The green and pink colored pieces show different computation but white ones represent idle time. The boxed area in Fig. 5 shows some of the cores have significant idle time. The top four cores in the boxed area take much longer to execute their computation than the bottom 12 cores and this is why the pink and green parts are

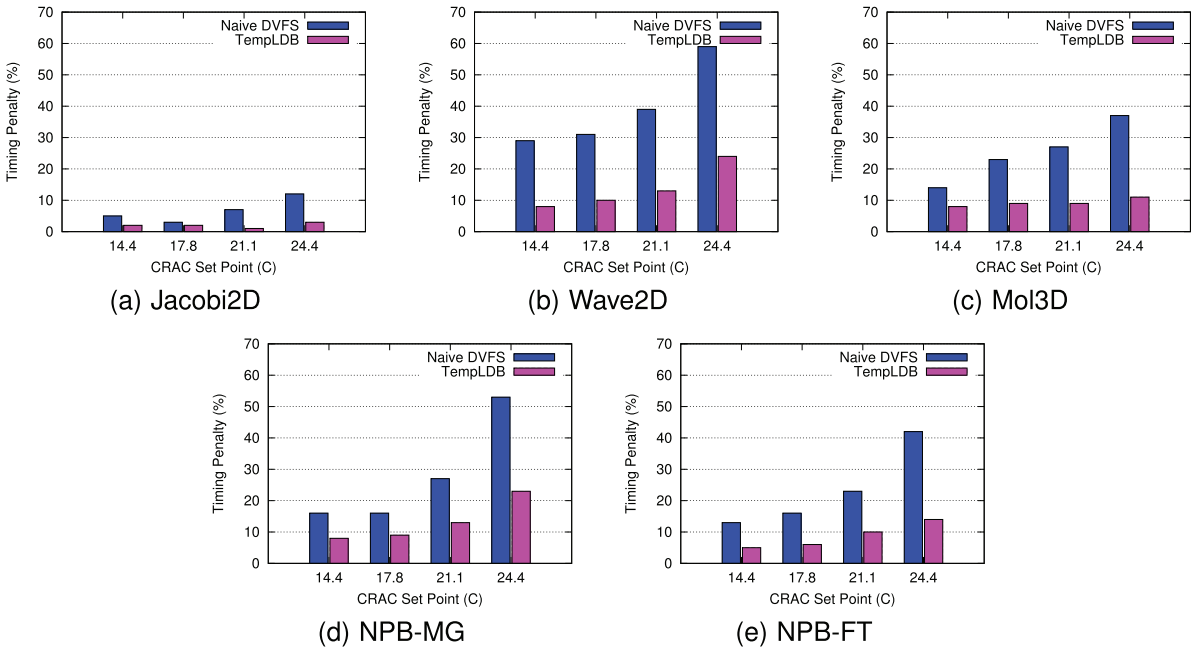


Fig. 4. Execution timing penalty with and without temperature aware load balancing.

longer for the top four cores. However, the other 12 cores execute their computation quickly and stay idle waiting for the rest of cores. This is because DVFS decreased the first four cores' frequencies and so they are slower in their computation. This shows that the timing penalty of naive DVFS is dictated by the slowest cores. The bottom part of Fig. 5 shows the same temperature control but using our *TempLDB*. In this case, there is no significant idle time because the scheme balances the load between slow and fast processors by taking their frequencies into account. Consequently, the latter approach results in much shorter total execution time, as reflected by shorter timelines (and figure width) in the bottom part of Fig. 5.

Now let us try to understand the timing penalty differences among different applications by examining more detailed data. Jacobi2D experiences the lowest impact of DVFS, regardless of load balancing (Fig. 4a). This occurs for several interconnected reasons. From the high level, Fig. 3 shows that it takes the longest of any application to increase temperatures to the upper bound of the acceptable range, where DVFS activates. This slow ramp-up in temperature means that its frequency does not drop until later in the run, and then falls relatively slowly, as seen in Fig. 6 which plots the minimum frequency at which any core was running (Fig. 6a) and the average frequency (Fig. 6b) for all 128 cores. Even when some processors reach their minimum frequency, Fig. 6b shows that its average frequency decreases more slowly than any other application, and does not fall as

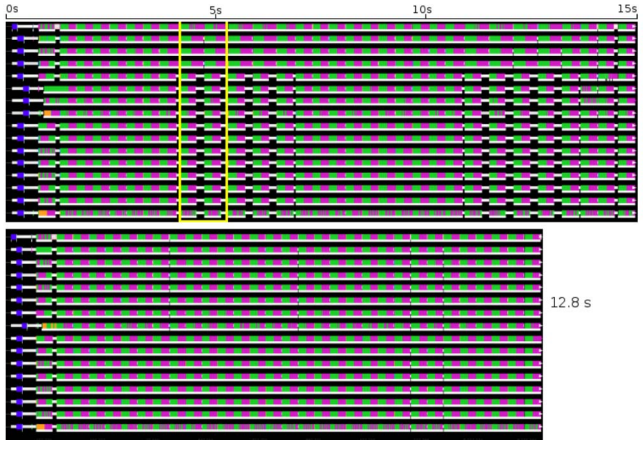


Fig. 5. Execution timelines before and after Temperature Aware Load Balancing for *Wave2D*.

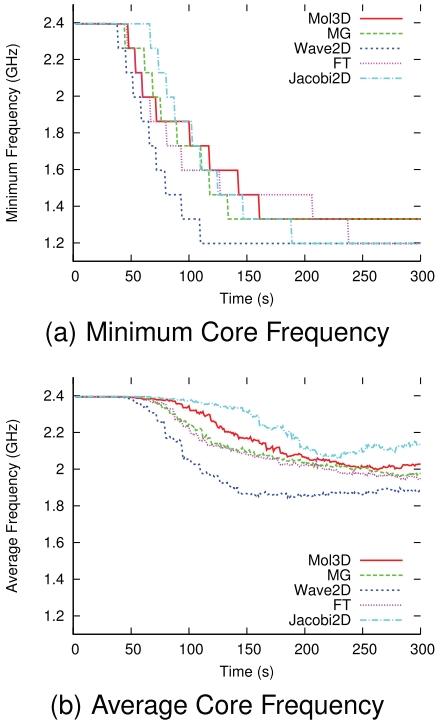
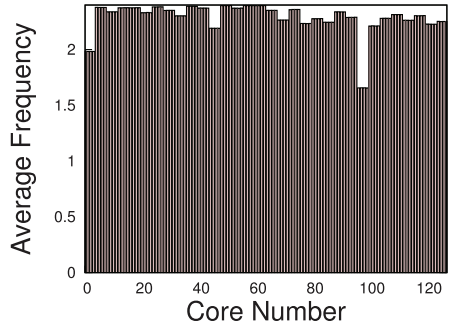
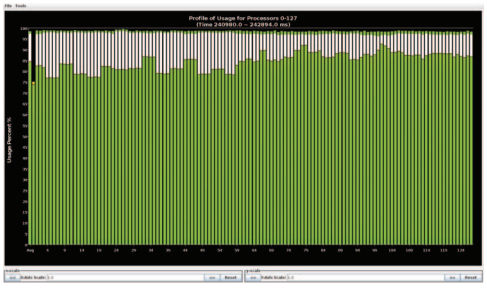


Fig. 6. Minimum and average core frequencies produced by DVFS for different applications at $24.4^{\circ}\mathrm{C}$.



(a) Average frequency of processors for Jacobi2D



(b) Utilization of processors for Jacobi2D

Fig. 7. Overall execution behavior of Jacobi2D using *TempLDB*.

far. The difference in the average frequency and the minimum frequency explains the difference between *TempLDB* and naive DVFS, as the execution time for *TempLDB* is dependent on average frequency whereas the execution time for naive DVFS depends on the minimum frequency at which any core is running.

Another way to understand the relatively small timing penalty of Jacobi2D is to compare its utilization and frequency profiles. Fig. 7a depicts each core's average frequency over the course of the run. Fig. 7b shows the utilization of each core while running Jacobi2D. In both figures, each bar represents the measurement of a single core. The green part of the utilization bars represents computation and the white part represents idle time. As can be seen, utilizations of the right half cores are roughly higher than the left half. Furthermore, the average frequency of the right half processors is roughly lower than the other half. Thus, lower frequency has resulted in higher utilization of those processors without much timing penalty. The reason this variation can occur is that the application naturally has some slack time in each iteration, which the slower processors dip into to keep pace with faster ones.

To examine the differences among applications at another level, Fig. 8 shows the performance impact of running each application with the processor frequencies fixed at a particular value (the marking 2.4+ refers to the top frequency plus Turbo Boost). All applications slow down as CPU frequency decreases, but Jacobi2D feels this effect very lightly compared to the others. This marked difference can be better understood in light of the performance counter-based measurements shown in Table 2. These measurements were taken in equal-length runs of the three Charm++ applications using the PerfSuite toolkit [27]. Jacobi2D has a much lower computational intensity, in terms of FLOP/s, than the other applications. It also retrieves much more data from main memory, explaining its lower sensitivity to

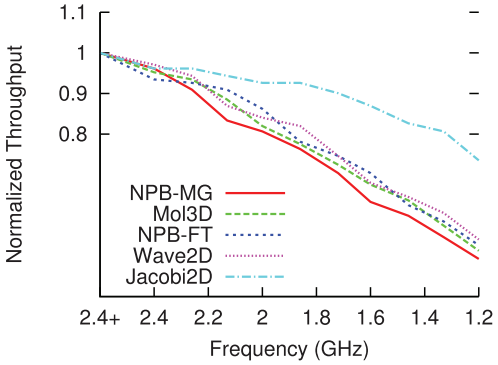


Fig. 8. Frequency sensitivity of the various applications.

frequency shifts. Its lower intensity also means that it consumes less power and dissipates less heat in the CPU cores than the other applications, explaining its slower ramp-up in temperature, slower ramp-down in frequency, and higher steady-state average frequency. In contrast, the higher FLOP counts and cache access rates of Wave2D and Mol3D explain their high frequency sensitivity, rapid core heating, lower steady-state frequency, and hence the large impact DVFS has on their performance.

8 Energy Savings

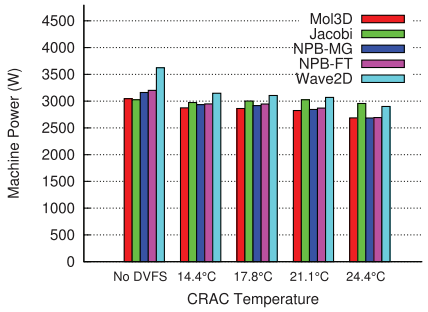
In this section, we evaluate the ability of our scheme to reduce *total* energy consumption. Our current load balancing scheme with the allowed temperature range strategy resulted in less than 1 percent time overhead for applying DVFS and load balancing (including the cost of object migration). Due to that change, we now get savings in both cooling energy consumption as well as machine energy consumption, although savings in cooling energy consumption constitute the main part of the reduction in total energy consumption. In order to understand the contribution for both cooling energy consumption and machine energy consumption, we will look at them separately.

8.1 Cooling Energy Consumption

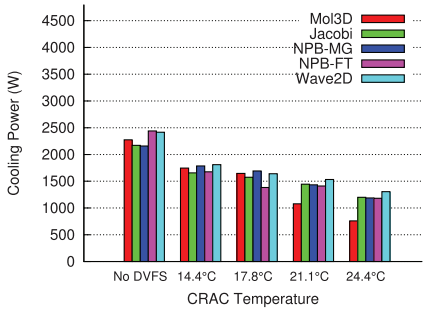
The essence of our work is to reduce cooling energy consumption by constraining core temperatures and avoiding hot spots. As outlined in (1), the cooling power consumed by the air conditioning unit is proportional to the difference between the hot air and cold air temperatures going in and out of the CRAC, respectively. As mentioned in [1], [2], [3], cooling cost can be as high as 50 percent of the total energy budget of the data center. However, in our calculation, we take it to be 40 percent of the total energy consumption of a baseline run with the CRAC at its lowest set point, which is equivalent to 66.6 percent of the measured machine energy during that run. Hence, we use

TABLE 2
Performance Counters for Charm++ Applications on One Core

Counter Type	Jacobi2D	Mol3D	Wave2D
MFLOP/s	373	666	832
Traffic L1-L2 (MB/s)	762	1017	601
Cache misses to DRAM (millions)	663	75	402



(a) Machine power consumption



(b) Cooling power consumption

Fig. 9. Machine and cooling power consumption for no-DVFS runs at a $12.2^{\circ}\mathrm{C}$ setpoint and various *TempLDB* runs.

the following formula to estimate the cooling power by feeding in actual experimental results for hot and cold air temperatures:

$$P_{cool}^{LB} = \frac{2 * \left(T_{hot}^{LB} - T_{ac}^{LB}\right) * P_{machine}^{base}}{3 * \left(T_{hot}^{base} - T_{ac}^{base}\right)}, \quad (2)$$

where T_{hot}^{LB} represents the temperature of hot air leaving the machine room (entering the CRAC) and T_{ac}^{LB} represents the temperature of the cold air entering the machine room. T_{hot}^{base} and T_{ac}^{base} represent the hot and cold air temperatures with the largest difference while running *Wave2D* at the coolest CRAC set point (i.e., $12.2^{\circ}\mathrm{C}$), and $P_{machine}^{base}$ is the power consumption of the machine for the same experiment.

Fig. 9 shows the machine power consumption and the cooling power consumption for each application using *TempLDB*. In Fig. 9b we note that the cooling power consumption falls as we increase the CRAC set point for all the applications. A higher CRAC set point means the cores heat up more rapidly, leading DVFS to set lower frequencies. Thus, machine power consumption falls as a result of the CPUs drawing less power (Fig. 9a). The machine's decreased power draw and subsequent heat dissipation means that less energy is added to the machine room air. The lower heat flux to the ambient air means that the CRAC requires less power to expel that heat and maintain the set-point temperature, as seen in Fig. 9b.

Wave2D consumes the highest cooling power for three out of the four CRAC set points we used, which is consistent with its high machine power consumption. Fig. 10 shows the savings in cooling energy in comparison to the baseline run where all the cores are working at the maximum frequency without any temperature control. These figures include the extra time the cooling needs to run corresponding to the timing penalty introduced because of applying DVFS. Due to the large reduction in cooling power (Fig. 9b) our scheme was able to save as much as 63 percent of the cooling energy in the case of *Mol3D* running at a CRAC set point of $24.4^{\circ}\mathrm{C}$. We can see that the savings in cooling energy consumption are better with our technique than naive DVFS for most of the applications and the corresponding set points. This is mainly due to the higher timing penalty for naive DVFS runs, which causes the CRAC to work for much longer than the corresponding *TempLDB* run.

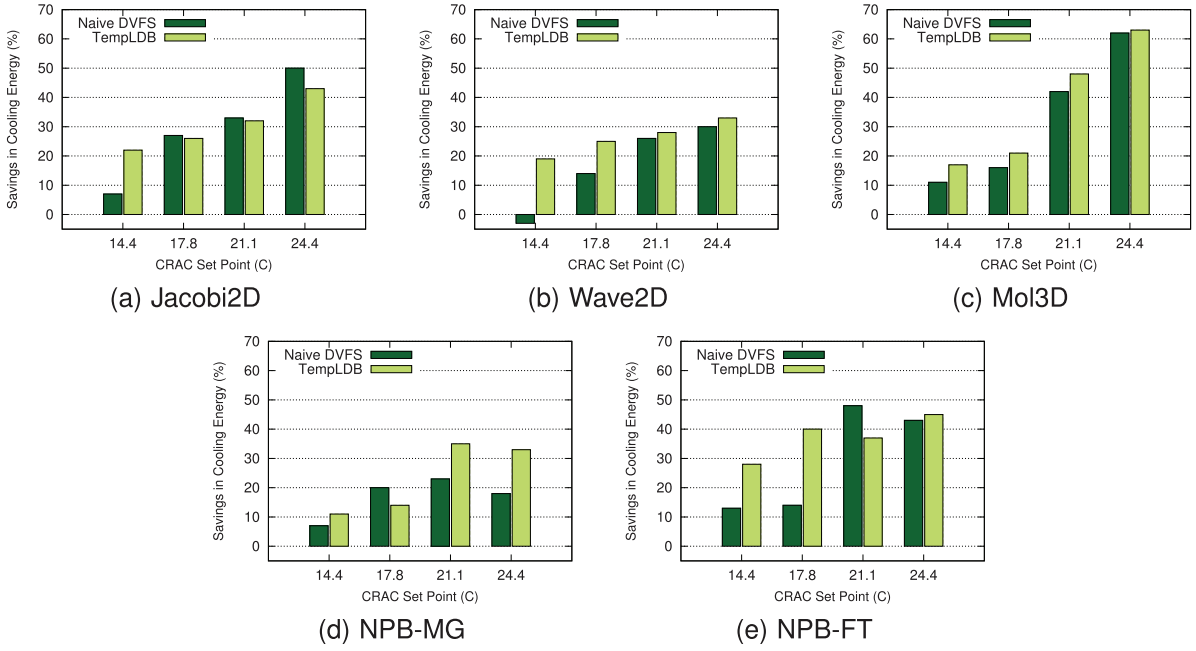


Fig. 10. Savings in cooling energy consumption with and without Temperature Aware Load Balancing (higher is better).

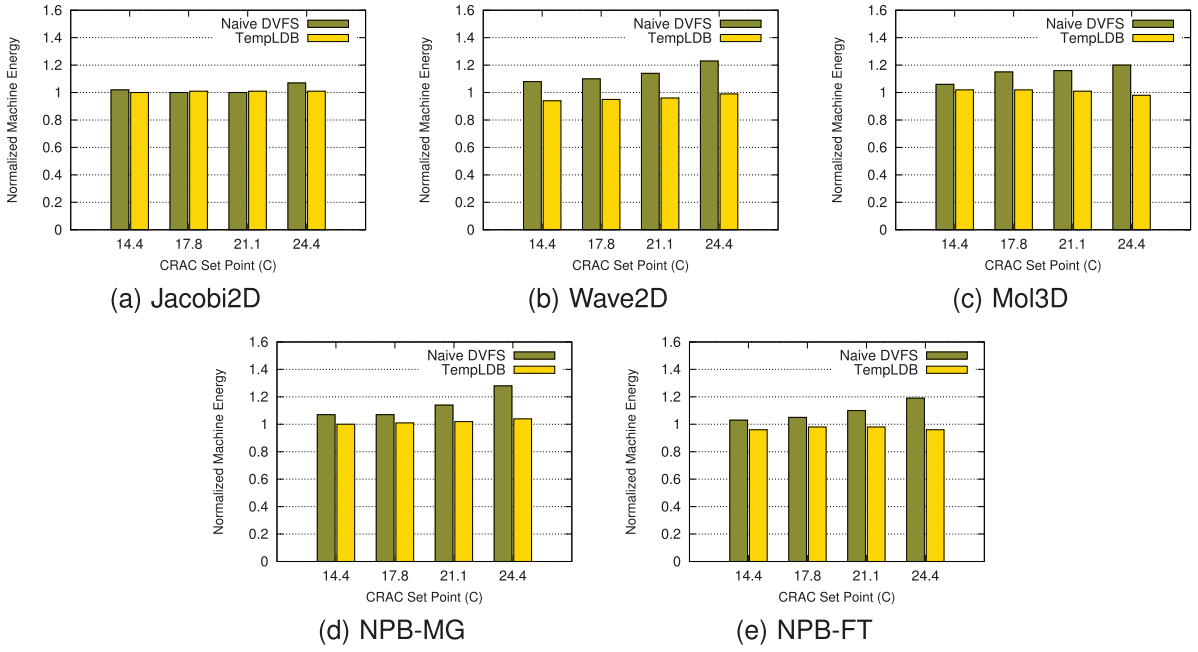


Fig. 11. Change in machine energy consumption with and without Temperature Aware Load Balancing (values below one represent savings).

8.2 Machine Energy Consumption

Although *TempLDB* does not optimize for reduced machine energy consumption, we still end up showing savings for some applications. Fig. 11 shows the change in machine energy consumption. A number less than one represents a reduction in machine energy consumption, whereas a number greater than one points to an increase.

It is interesting to see that *NPB-FT* and *Wave2D* end up saving machine energy consumption when using *TempLDB*. For *Wave2D*, we end up saving 6 percent of machine energy consumption when the CRAC is set to 14.4°C whereas the maximum machine energy savings of *NPB-FT*, 4 percent, occurs when the CRAC is set to 14.4 or 24.4°C. To find the reasons for these savings in machine energy consumption, we performed a set of experiments where we ran the applications with the 128 cores of our cluster fixed at each of the available frequencies. Fig. 12 plots the normalized machine energy for each application against the frequency at which it was run. Power consumption models dictate that CPU power consumption can be regarded as being proportional to the cube of frequency which would imply that we should expect the power to fall as a cubic of frequency whereas the execution time increases only linearly in the worst case. This would imply that we should always reduce energy consumption by moving to a lower frequency. This proposition does not hold because of the high base power drawn by everything other than the CPU and memory subsystem, which is 40 W per node for our cluster. We can say that while moving to each successive lower frequency we reach a point where the savings in the CPU energy consumption are offset by an increase in base energy consumption due to the timing penalty incurred, leading to the U-shaped energy curves. When our scheme lowers frequency as a result of core temperature crossing the maximum temperature value, we move into the more desirable range of machine energy consumption, i.e., closer to the minimum of the U-shape energy curves.

To see that this is the case, Fig. 13 shows the cumulative time spent by all 128 cores at different frequency levels for *Wave2D* using *TempLDB* at a CRAC set point of 24.4°C. We can see that most of the time is spent at frequency levels between 1.73-2.0 GHz, which corresponds to the lowest point for normalized energy for *Wave2D* in Fig. 12.

In order to study the relationship between machine power consumption and average frequency we plotted the power consumption for each application over the course of a run using *TempLDB* in Fig. 14. It was surprising for us to notice that despite starting at the same level of machine power

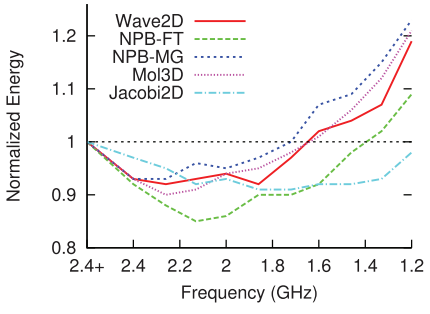


Fig. 12. Normalized machine energy consumption for different frequencies using 128 cores.

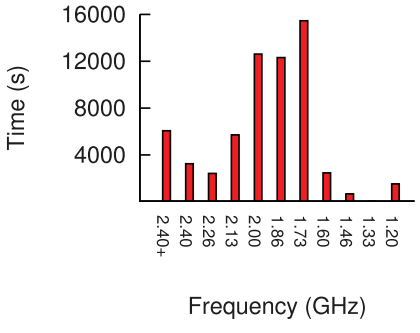


Fig. 13. The time *Wave2D* spent in different frequency levels.

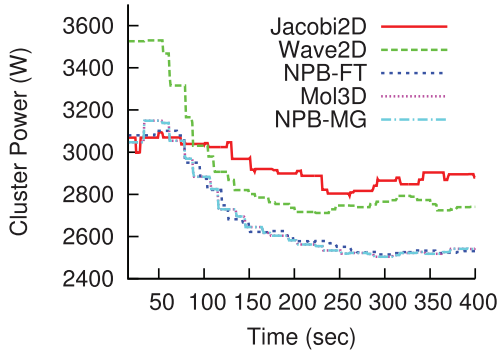


Fig. 14. Total power draw for the cluster using *TempLDB* at CRAC set point of $24.4°C$.

consumption as *NPB-FT* and *NPB-MG*, *Jacobi2D* ended up having a much higher average frequency (Fig. 6b). The other interesting observation we can make from this graph is the wide variation in steady state power consumption among the applications.

Since all the applications are 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 heat sink and fan assembly. Thus, each application should end up having the same CPU power consumption. This would mean that the difference in power draw among the applications in Fig. 14 is caused by something other than CPU power consumption. Table 2 shows *Jacobi2D* and *Wave2D* have many more cache misses than *Mol3D* and thus end up with a higher power consumption in the memory controller and DRAM, which do not contribute to increased core temperatures but do increase the total power draw for the machine.

In order to verify our hypothesis, we ran two of our applications, *Jacobi2D* and *Wave2D*, on a single node containing a four-core Intel Core i7-2600K, with a temperature threshold of $50°C$. Using our load balancing infrastructure and the newly added hardware energy counters in Intel's recent Sandy Bridge technology present in this chip, we can measure the chip's power consumption directly from machine specific registers. Both applications begin execution with the CPU at its maximum frequency, which our system decreases as temperatures rise. As expected, both applications settled near a common steady state of power consumption for the CPU package (cores and memory combined). When Sandy Bridge-EP processors become available, with energy counters for the memory subsystem, it should be possible to observe the separate contributions of the cores and memory hierarchy.

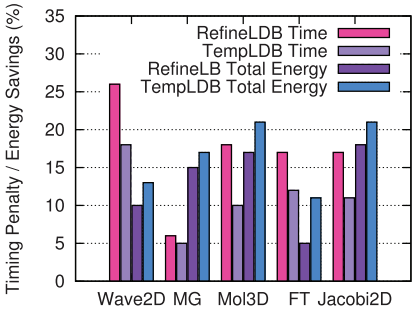


Fig. 16. Timing penalty and energy consumption improvement of *TempLDB* over *RefineLDB*.

Before highlighting the key findings of our study, we compare our load balancer, i.e., *TempLDB*, with a generic CHARM++ load balancer, i.e., *RefineLDB*. *RefineLDB*'s load balancing strategy relies on execution time data for each task without taking into account the frequency at which each core is working at. Similar to *TempLDB*, *RefineLDB* also migrates extra tasks from the overloaded cores to the underloaded cores. We implemented our temperature control scheme using DVFS into this load balancer but kept the load balancing part the same. Because *RefineLDB* relies only on task execution time data to predict future load without taking into account the transitions in core frequencies, it ends up taking longer and consumes more energy to restore load balance. The improvement that *TempLDB* makes can be seen from Fig. 16 which shows a comparison between both load balancers for all the applications with the CRAC set point at $22.2°C$.

9 Tradeoff in Execution Time and Energy Consumption

The essence of our results can be seen in Fig. 15, which summarizes the tradeoffs between execution time and *total* energy consumption for a representative trio of our applications. Each application has two curves, one for each of the *Naive DVFS* and *TempLDB* runs. 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 CRAC set points

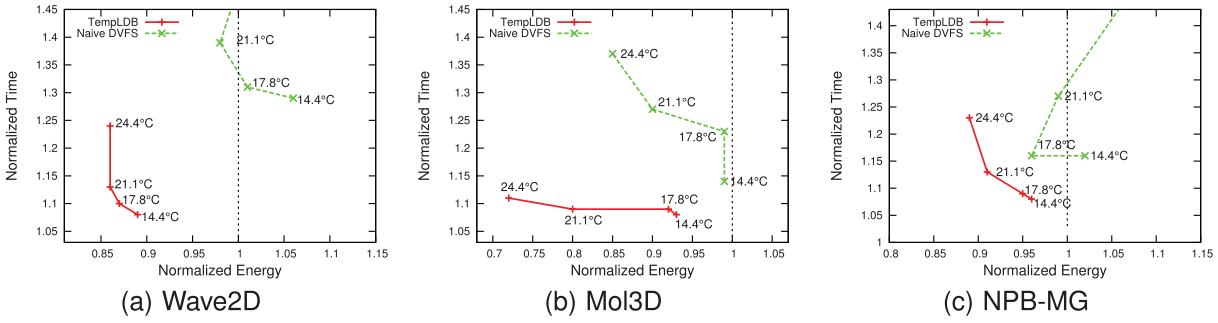


Fig. 15. Normalized time against normalized total energy for a representative subset of applications.

across all applications, *TempLDB* takes its corresponding point from the *Naive DVFS* scheme at the same CRAC set-point down (saving timing penalty) and to the left (saving energy consumption).

From Fig. 15a, we can see that *Wave2D* is only conducive to saving energy with the CRAC set below 21.1°C, as the curve becomes vertical with higher set points. However, we should note that the temperature range of 47-49°C was much lower than the average temperature *Wave2D* reached with the CRAC set at the coolest set point of 12.2°C without any temperature control. Thus, a higher CRAC set point imposes too much timing penalty to provide any total energy savings beyond the 21.1°C set point. It is worth noting that even at 14.4°C we are able to reduce its total energy consumption by 12 percent.

For *Mol3D*, the nearly flat curve shows that our scheme does well at saving energy, since we do not have to pay a large execution time penalty in order to reduce energy consumption. The same effect holds true for *Jacobi2D*. *NPB-MG*'s sloped curve places it in between these two extremes. It and *NPB-FT* truly present a tradeoff, and users can optimize according to their preferences.

10 Conclusion and Future Work

In this paper, we proposed an approach to saving cooling energy by constraining core temperature while minimizing the associated timing penalty using task migration. Our scheme uses DVFS and *TempLDB* to meet these requirements. We experimented (using dedicated hardware) on five different CHARM++ and MPI applications to demonstrate its substantial benefits in energy savings. While saving energy, our scheme was also able to neutralize hot spots in our testbed. Furthermore, through detailed analysis, we characterized the relationship between application characteristics and the timing penalty we observed when constraining core temperature. According to our findings, applications with a high FLOP/s rate, e.g., *Wave2D*, can reduce *total* energy consumption by 12 percent by paying a timing penalty of 7 percent. On the other hand, *Jacobi2D* had a lower FLOP/s rate and some slack available, enabling our scheme to reduce *total* energy consumption by 18 percent with as little as 3 percent timing penalty. In case of *Mol3D*,which falls in between *Wave2D* and *Jacobi2D* in terms of FLOP/s, our scheme was able to reduce total energy consumption by 28 percent by paying a timing penalty of 11 percent.

Load balancing based on overdecomposition is the novel technique we bring to the broader problem of power, energy, and thermal optimization. We plan to combine existing work such as incorporating the computation DAG of an application in our scheme so that we can place critical tasks on "cooler" cores in order to further reduce timing penalty. The ultimate objective is to combine our technique with schemes that minimize machine energy consumption so that we can provide a *total* energy solution for HPC data centers, i.e., reduce both machine and cooling energy consumption. To achieve this objective, we plan to exploit the varying sensitivities of different parts of code to core frequency [20] and combine it with our *TempLDB*. We also plan to apply similar principles and techniques to limit power consumption of the data center to allow operation within a fixed *power* budget.

References

[1] R.F. Sullivan, "Alternating Cold and Hot Aisles Provides More Reliable Cooling for Server Farms," white paper, Uptime Inst., 2000.

[2] C.D. Patel, C.E. Bash, R. Sharma, M. Beitelmal, and R. Friedrich, "Smart Cooling of Data Centers," *Proc. ASME Conf.,* vol. 2003, no. 36908b, pp. 129-137, 2003.

[3] R. Sawyer, "Calculating Total Power Requirements for Data Centers," white paper, Am. Power Conversion, 2004.

[4] O. Sarood, A. Gupta, and L.V. Kale, "Temperature Aware Load Balancing for Parallel Applications: Preliminary Work," *Proc. Seventh Workshop High-Performance, Power-Aware Computing (HPPAC '11),* 2011.

[5] O. Sarood and L.V. Kalé, "A 'Cool' Load Balancer for Parallel Applications," *Proc. ACM/IEEE Conf. Supercomputing,* Nov. 2011.

[6] C. Bash and G. Forman, "Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center," *Proc. USENIX Ann. Technical Conf.,* pp. 29:1-29:6, 2007.

[7] L. Wang, G. von Laszewski, J. Dayal, and T. Furlani, "Thermal Aware Workload Scheduling with Backfilling for Green Data Centers," *Proc. IEEE 28th Int'l Performance Computing and Comm. Conf. (IPCCC),* Dec. 2009.

[8] L. Wang, G. von Laszewski, J. Dayal, X. He, A. Younge, and T. Furlani, "Towards Thermal Aware Workload Scheduling in a Data Center," *Proc. Int'l Symp. Pervasive Systems, Algorithms, and Networks (ISPAN),* Dec. 2009.

[9] Q. Tang, S. Gupta, D. Stanzione, and P. Cayton, "Thermal-Aware Task Scheduling to Minimize Energy Usage of Blade Server Based Datacenters," *Proc. IEEE Second Int'l Symp. Dependable, Autonomic and Secure Computing,* 2006.

[10] D. Rajan and P. Yu, "Temperature-Aware Scheduling: When Is System-Throttling Good Enough?" *Proc. Ninth Int'l Conf. Web-Age Information Management (WAIM '08),* pp. 397-404, July 2008.

[11] H. Le, S. Li, N. Pham, J. Heo, and T. Abdelzaher, "Joint Optimization of Computing and Cooling Energy: Analytic Model and a Machine Room Case Study," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS),* June 2012.

[12] 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," *Proc. ACM/IEEE Conf. Supercomputing,* pp. 49:1-49:9, 2007.

[13] M.Y. Lim, V.W. Freeh, and D.K. Lowenthal, "Adaptive, Transparent CPU Scaling Algorithms Leveraging Inter-node MPI Communication Regions," *Parallel Computing,* vol. 37, nos. 10/11, pp. 667-683, 2011.

[14] R. Springer, D.K. Lowenthal, B. Rountree, and V.W. Freeh, "Minimizing Execution Time in MPI Programs on An Energy-Constrained, Power-Scalable Cluster," *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '06),* pp. 230-238, 2006.

[15] S. Huang and W. Feng, "Energy-Efficient Cluster Computing via Accurate Workload Characterization," *Proc. IEEE/ACM Ninth Int'l Symp. Cluster Computing and the Grid (CCGRID '09),* pp. 68-75, 2009.

[16] H. Hanson, S. Keckler, R.K, S. Ghiasi, F. Rawson, and J. Rubio, "Power, Performance, and Thermal Management for High-Performance Systems," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* Mar. 2007.

[17] A. Banerjee, T. Mukherjee, G. Varsamopoulos, and S. Gupta, "Cooling-Aware and Thermal-Aware Workload Placement for Green HPC Data Centers," *Proc. Int'l Green Computing Conf.,* pp. 245-256, Aug. 2010.

[18] Q. Tang, S. Gupta, and G. Varsamopoulos, "Energy-Efficient Thermal-Aware Task Scheduling for Homogeneous High-Performance Computing Data Centers: A Cyber-Physical Approach," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 11, pp. 1458-1472, Nov. 2008.

[19] A. Merkel and F. Bellosa, "Balancing Power Consumption in Multiprocessor Systems," *Proc. First ACM SIGOPS/EuroSys European Conf. Computer Systems (EuroSys '06),* pp. 403-414, 2006.

[20] V.W. Freeh and D.K. Lowenthal, "Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster," *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '05),* pp. 164-173, http://doi.acm.org/10.1145/1065944.1065967, 2005.

[21] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *Proc. Conf. Object Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, A. Paepcke ed., pp. 91-108, Sept. 1993.
[22] R.K. Brunner and L.V. Kalé, "Handling Application-Induced Load Imbalance Using Parallel Objects," *Proc. Int'l Workshop Parallel and Distributed Computing for Symbolic and Irregular Applications*, pp. 167-181, 2000.
[23] P. Jetley, F. Gioachin, C. Mendes, L.V. Kale, and T.R. Quinn, "Massively Parallel Cosmological Simulations with ChaNGa," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.*, 2008.
[24] G. Zheng, A. Bhatele, E. Meneses, and L.V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *Int'l J. High Performance Computing Applications*, vol. 25, pp. 371-385, Mar. 2011.
[25] "Intel Turbo Boost Technology," http://www.intel.com/technology/turboboost/, 2012.
[26] 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," Technical Report RNR-04-077, NASA Ames Research Center, 1994.
[27] R. Kufrin, "Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux," *Proc. Linux Cluster Conf.*, 2005.

Osman Sarood received the BS degree in computer science from Lahore University of Management Sciences in 2004. He is currently working toward the PhD degree in computer science at the University of Illinois at Urbana-Champaign, supported by a fellowship from the Fulbright Program. His research areas include parallel programming and energy efficiency for HPC data centers.

Phil Miller received the BS degree in computer science from Harvey Mudd College in 2008. He is currently working toward the PhD degree in computer science at the University of Illinois at Urbana-Champaign. He is supported by US National Science Foundation (NSF) grant OCI-0725070.

Ehsan Totoni received the BS degree in computer engineering from Sharif University of Technology, the MS degree from University of Illinois at Urbana-Champaign, and is working toward the PhD degree. His research areas include parallel programming, simulation and performance analysis, scientific computing and energy efficiency. He is supported by US Department of Energy (DOE) grant DE-SC006706.

Laxmikant V. Kalé received the BTech degree in electronics engineering from Banaras Hindu University, Varanasi, India, in 1977, and the ME degree in computer science from Indian Institute of Science in Bengaluru, India, in 1979. He received the PhD degree in computer science from State University of New York, Stony Brook, in 1985. He joined the faculty of the University of Illinois at Urbana-Champaign as an assistant professor in 1985, where he is currently employed as a full professor. His research spans parallel computing, including parallel programming abstractions, scalability, automatic load balancing, communication optimizations, and fault tolerance. He has collaboratively developed several scalable CSE applications. A paper he coauthored on scaling the molecular dynamics program NAMD was one of the winners of the Gordon Bell award in 2002. He is a fellow of the IEEE, a member of the IEEE Computer Society and the Association for Computing Machinery.