A ‘Cool’ Load Balancer for Parallel Applications

Osman Sarood
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
sarood1@illinois.edu

Laxmikant V. Kale
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kale@illinois.edu

ABSTRACT

Meeting power requirements of huge exascale machines of the future would be one major challenge. Our focus in this paper is to minimize cooling power and we propose a technique, that uses a combination of DVFS and temperature aware load balancing to constrain core temperatures as well as save cooling energy. Our scheme is specifically designed to suit parallel applications which are typically tightly coupled. The temperature control comes at the cost of execution time and we try to minimize the timing penalty.

We experiment with three applications (with different power utilization profiles), run on a 128-core (32-node) cluster with a dedicated air conditioning unit. We calibrate the efficacy of our scheme based on three metrics: ability to control average core temperatures thereby avoiding hot spot occurence, timing penalty minimization, and cooling energy savings. Our results show cooling energy savings of up to 57% with timing penalty mostly in the range of 2 to 20%.

1. INTRODUCTION

Cooling energy is a substantial part of the total energy spent by an High Performance Computing (HPC) computer room or a data center. According to some reports, this can be as high as 50% [19], [3], [17] of the total energy budget. It is deemed essential to keep the computer room adequately cold in order to prevent processor cores from overheating beyond their safe thresholds. For one thing, continuous operation at higher temperatures can permanently damage processor chips. Also, processor cores operating at higher temperatures consume more power while running identical computations at the same speeds due to the positive feedback loop between temperature and power [8].

Cooling is therefore needed to dissipate the energy consumed by a processor chip, and thus to prevent overheating of the core. This consumed energy has a static and dynamic component. The dynamic component increases as the cube of the frequency at which the core is run. Therefore, an alternative way of preventing overheating is to reduce the frequency. Modern processors and operating systems support such frequency control (e.g. DVFS). With this, it becomes possible to run a computer in a room with high ambient temperature, by simply reducing frequencies whenever temperature goes above a threshold.

However, this method of temperature control is problematic for HPC applications, which tend to be tightly coupled. If only one of the cores is slowed down by 50%, the entire application will slow down by 50% due to dependencies between computations on different processors. This is further exacerbated when the dependencies are global, such as when global reductions are used with a high-frequency. Since individual processors may overheat at different rates, and at different points in time, and since physical aspects of the room and the air flow may create regions which tend to be hotter, the situation where only a small subset of processors are operating at a reduced frequency will be quite common. For HPC applications, this method therefore is not suitable as it is.

The question we address in this paper is whether we can substantially reduce cooling energy without a significant timing penalty. Our approach involves a temperature-aware dynamic load balancing strategy. In some preliminary work presented at a workshop [16], we have shown the feasibility of the basic idea in the context of a single eight-core node. The contributions of this paper include development of a scalable load-balancing strategy demonstrated on 128 core machine, in a controlled machine room, and with explicit power measurements. Via experimental data, we show that cooling energy can be reduced to the extent of up to to 57%, with the timing penalty only in the range of 2 to 20% in most cases.

We begin in Section 2 by introducing the frequency control method, and documenting the timing penalty it imposes on HPC applications. In Section 3 we describe our temperature-aware load balancer. It leverages object-based overdecomposition and the load-balancing framework in the Charm++ runtime system. Section 4 outlines the experimental setup for our work. We then describe (Section 5) performance data to show that, with our strategy, the temperatures are retained within the requisite limits, while the timing penalties are small. Some interesting issues that arise in understanding how different applications react to temperature control are analyzed in Section 6. Section 7 undertakes a detailed analysis of the impact of our strategies on machine energy and cooling energy. Section 8 summarizes related work and sets our work in its context, which is followed by a summary in Section 9.

2. CONSTRAINING CORE TEMPERATURES

Unrestrained, core temperatures can soar very high. The most common way to deal with this in today's HPC centers is through the use of additional cooling arrangements. But as we have already mentioned, cooling itself accounts for around 50% [19, 3, 17] of the total energy consumption of a data center and this can rise even higher with the formation of hot spots. To motivate the technique of this paper, we start with a study of the interactions of core temperatures in parallel applications with the cooling settings of their surroundings. We run *Wave2D*, a finite differencing application, for ten minutes on 128 cores in our testbed. We provide more details of the testbed in Section 4. The cooling knob in this experiment was controlled by setting the cooling room air conditioning (CRAC) to different temperature settings. Figure 1 shows the average core temperatures and the maximum difference of any core from the average temperature corresponding to two different CRAC set points. As expected, cooling settings have a pronounced effect on the core temperatures. For example, the average core temperatures corresponding to CRAC set point 23.3 °C are almost 6 °C less than those for CRAC set point 25.6 °C. Figure 1 also shows the maximum difference between the average temperature and any core's temperature and as can be seen, the difference worsens as we decrease external cooling (an increase of 11 °C). The result: Hotspots!

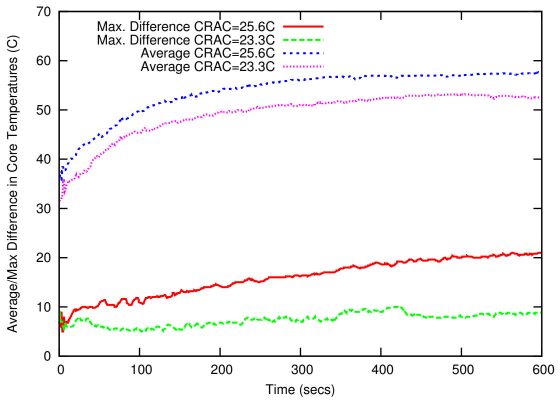


Figure 1: Average core temperatures along with max. difference of any core from the average for *Wave2D*

The issue of core overheating is not new. DVFS is a widely accepted solution to cope with it. DVFS is a technique which is used to adjust the frequency and input voltage of a microprocessor. It is mainly used to conserve the dynamic power consumed by a processor. A shortcoming of DVFS, is that it comes with an execution time and machine energy penalty. To establish the severity of these penalties, we performed an experiment with 128 cores, running *Wave2D* for a fixed number of iterations. We used DVFS to keep core temperatures under 44 °C by periodically checking core temperatures and reducing the frequency by one level whenever a core got hot. The experiment was repeated for five different CRAC set points. The results, in Figure 2 show the normalized execution time and machine energy. Normalization is done with respect to the run where all cores run at full frequency without DVFS. The high timing penalty (seen from Figure 2) coupled with an increase in machine energy makes

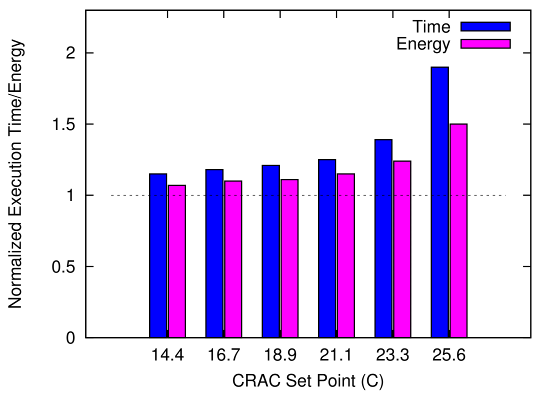


Figure 2: Normalized time and machine energy using DVFS for *Wave2D*

it infeasible for HPC community to use such a technique. Now that we have established that DVFS on its own can not efficiently control core temperatures without incurring unacceptably high timing penalties, we now propose our approach to ameliorate the deficiencies in using DVFS without load balancing.

3. TEMPERATURE AWARE LOAD BALANCING

In this section, we propose a novel technique based on task migration that can efficiently control core temperatures and simultaneously minimizes the timing penalty. In addition, it also ends up saving total energy. Although our technique should work well with any parallel programming language which allows object migration, we chose Charm++ for our tests and implementation because it allows simple and straightforward task migration. We introduce Charm++ followed by a description of our temperature aware load balancing technique.

3.1 Charm++

Charm++ is a parallel programming runtime system that works on the principle of processor virtualization. It provides a methodology where the programmer divides the program into small computations (objects or tasks) which are distributed amongst the P available processors by the runtime system [5]. Each of these small problems is a migratable C++ object that can reside on any processor. The runtime keeps track of the execution time for all these tasks and logs them in a database which is used by a load balancer. The aim of load balancing is to ensure equal distribution of computation and communication load amongst the processors. Charm++ uses the load balancing database to keep track of how much work each task is doing. Based on this information, the load balancer in the runtime system, determines if there is a load imbalance and if so, it migrates object from an overloaded processor to an underloaded one [24]. The load balancing decision is based on the heuristic of *principle of persistance*, according to which computation and communication loads tend to persist with time for a certain class of iterative applications. Charm++ load balancers have proved to be very successful with iterative applications such as NAMD [13].

3.2 Refinement based temperature aware load balancing

We now describe our refinement based temperature aware load balancing scheme which does a combination of DVFS and intelligent load balancing of tasks according to frequencies in order to minimize execution time penalty. The general idea is to let each core work at the maximum possible frequency as long as it is within the maximum temperature threshold. Currently, we do DVFS on a per-chip instead of a per-core basis as the hardware did not allow us to do otherwise. When we change the frequency of all the cores on the chip, the core input voltage also drops resulting in power savings. This raises a question: What condition should trigger a change in frequency? In our earlier work [16], we did DVFS when *any* core on a chip crossed the temperature threshold. But our recent results show that basing DVFS decision on average temperature of the chip provides better temperature control. Another important decision is to determine how much should the frequency be lowered in case a chip exceeds the maximum threshold. Modern day processors come with a set of frequencies (frequency levels) at which they can operate. Our testbed had 10 different frequency levels from 1.2GHz to 2.4GHz (each step differs by 0.13GHz). In our scheme, we change the frequency by only one level at each decision time.

The pseudocode for our scheme is given in Algorithm 1 with the descriptions of variables and functions given in Table 1. The application specifies a maximum temperature threshold and a time interval at which the runtime periodically checks the temperature and determines whether any node has crossed that threshold. The variable k in Algorithm 1 refers to the interval number the application is currently in. Our algorithm starts with each node computing the average temperature for all cores present on it i.e. t_i^k. Once the average temperature has been computed, each node matches it against the maximum temperature threshold (T_{max}). If the average temperature is greater than T_{max}, all cores on that chip shift one frequency level down. However, if the average temperature is less than T_{max}, we increase the frequency level of all the cores on that chip (lines 2-6). Once the frequencies have been changed, we need to take into account the speed differential with which each core can execute instructions. We start by gathering the load information from the load balancing database for each core and task. In Charm++, this load information is maintained in milliseconds. Hence, in order to neutralize the frequency difference amongst the loads of each task and core, we convert the load times into clock ticks by multiplying load for each task and core with the frequency at which it was running (lines 8-15). It is important to note that without doing this conversion, it would be incorrect to compare the loads and hence load balancing would result in inefficient schedules. Even with this conversion, the calculations would not be completely accurate, but will give much better estimates. We also compute the total number of ticks required for all the tasks (line 10) for calculating the weighted averages according to new core frequencies. Once the ticks are calculated, we create a *max heap* i.e. *overHeap*, for overloaded and a *set* for underloaded cores i.e. *underSet* (line 16). The categorization of over and underloaded cores is done by the *isHeavy* and *isLight* procedures on lines (25-28). A core i is overloaded if its currently assigned ticks are greater than what it should be assigned i.e. a weighted average of

Table 1: Description for variables used in Algorithm 1

Variable	Description
n	number of tasks in application
p	number of cores
T_{max}	maximum temperature allowed
C_i	set of cores on same chip as core i
e_i^k	execution time of task i during step k (in ms)
l_i^k	time spent by core i executing tasks during step k (in ms)
f_i^k	frequency of core i during step k (in Hz)
m_i^k	core number assigned to task i during step k
$ticks_i^k$	num. of ticks taken by ith task/core during step k
t_i^k	average temperature of node i at start of step k (in $^\circ$C)
S^k	$\{e_1^k, e_2^k, e_3^k, \ldots, e_n^k\}$
$overHeap$	heap of overloaded cores
$underSet$	set of underloaded cores
P^k	$\{l_1^k, l_2^k, l_3^k, \ldots, l_p^k\}$

totalTicks according to the cores new frequency (line 26). Notice the *1+tolerance* factor in the expression at line 26. We have to use this in order to do refinement only for cores that are overloaded by some considerable margin. We set it to 0.03 for all our experiments. This means that a core is considered to be overloaded if its currently assigned ticks are greater than its average weighted ticks by a factor of 1.03. Similar check is in place for *isLight* procedure (lines 27) but we do not include tolerance as it does not matter.

Once the max heap for the overloaded cores and a set for underloaded cores are ready, we start with the load balancing. We pop the max element (tasks with maximum number of ticks) out of *overHeap* (referred as *donor*). Next, we call the procedure *getBestCoreAndTask* which selects the best task to donate to the best underloaded core. The *bestTask* is the largest task currently assigned to *donor* such that it does not overload a core from the *underSet*. And the *bestCore* is the one which will remain underloaded after being assigned the *bestTask*. After determining the *bestTask* and *bestCore*, we do the migration by recording the task mapping and (line 20) updating the *donor* and *bestCore* with number of ticks in *bestTask*. We then call *updateHeapAndSet* (line 23) which rechecks the *donor* for being overloaded. If it is, we reenter it to *overHeap*. It also checks donor for being underloaded so that it is added to the *underSet* in case it has ended up with too little load. This ends the job of migrating one task from overloaded core to an underloaded core. We repeat this procedure until *overHeap* is empty. It is important to notice that the value of tolerance can affect the overhead of our load balancing. If that value is too large, it might ignore load imbalance whereas if it is too small, it can result in a lot of overhead for object migration. We have noticed that any value from 0.05 to 0.01 performs equally good.

4. EXPERIMENTAL SETUP

The primary objective of this work is to constrain core temperature and save energy spent on cooling. Our scheme ensures that all the cores fall below a user-defined maximum threshold. We want to emphasize that all results reported in this work are actual measurements and not simulations. We have used a 160 core (40 node, single socket) testbed equipped with a dedicated CRAC. Each node is a single socket machine with Intel Xeon X3430 chip. It is a quad core chip supporting 10 different frequency levels ranging from 1.2GHz to 2.4GHz. We use 128 cores out of the 160 cores available for all the runs that we report. All the nodes

Algorithm 1 Temperature Aware Refinement Load Balancing

```
1: At node $i$ at start of step $k$
2: if $t_i^k > T_{max}$ then
3:     decreaseOneLevel($C_i$) //reduce by 0.13GHz
4: else
5:     increaseOneLevel($C_i$) //increase by 0.13GHz
6: end if
7: At Master core
8: for i $\in S^{k-1}$ do
9:     $ticks_i^{k-1} = e_i^{k-1} \times f_{m_i^{k-1}}^{k-1}$
10:    $totalTicks = totalTicks + ticks_i^{k-1}$
11: end for
12: for i $\in P^{k-1}$ do
13:    $ticks_i^{k-1} = l_i^{k-1} \times f_i^{k-1}$
14:    $freqSum = freqSum + f_i^k$
15: end for
16: createOverHeapAndUnderSet()
17: while $overHeap$ NOT NULL do
18:    donor = $overHeap$->deleteMaxHeap
19:    (bestTask,bestCore) =
       getbestCoreAndTask(donor,underSet)
20:    $m_{bestTask}^k = bestCore$
21:    $ticks_{donor}^{k-1} = ticks_{donor}^{k-1} - bestSize$
22:    $ticks_{bestCore}^{k-1} = ticks_{bestCore}^{k-1} + bestSize$
23:    undateHeapAndSet()
24: end while
25: procedure isHeavy(i)
26: return ($ticks_i^{k-1}$ > (1 + $tolerance$) * ($totalTicks$ * $f_i^k$)
    / $freqSum$)
27: procedure isLight(i)
28: return ($ticks_i^{k-1}$ < $totalTicks$ * $f_i^k/freqSum$)
```

run ubuntu 10.4 and we use `cpufreq` module in order to do DVFS. The nodes are interconnected using a 48-port gigabit ethernet switch. We use the Liebert Power unit installed with the rack to get power readings for the machines.

The CRAC in our testbed is an air cooler that uses centrally chilled water for cooling the air. It manipulates the flow of chilled water to achieve the temperature set point prescribed by the operator. The exhaust air (T_{hot}) i.e. the hot air coming in from the machine room, is compared against the set point and the flow of the chilled water is adjusted accordingly to cover the difference in the temperatures. This model of cooling is favorable considering that the temperature control is responsive to the thermal load (as it tries to bring the exhaust air to temperature set point) instead of room inlet temperature [9]. The machines and the CRAC are located in the Computer Science department of University of Illinois Urbana Champaign. We were fortunate enough to not only be able to use DVFS on all the available cores but to also change the CRAC set points.

There isn't a straightforward way of measuring the exact power draw of the CRAC as it uses the chilled water to cool the air which in turn is cooled centrally for the whole building. This made it impossible for us to use a power meter. But that isn't unusual as most data centers use similar cooling designs. Instead of using a power meter, we installed temperature sensors at the outlet and inlet of the CRAC. These sensors measure the air temperature coming from and going out to the machine room.

The heat dissipated into the air is affected by core temperatures and the CRAC has to cool this air for maintaining a constant room temperature. The power consumed by CRAC (P_{ac}) to bring the temperature of exhaust air (T_{hot}) down to the cool inlet air (T_{ac}) is [9]:

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

where c_{air} is the hear capacity constant, f_{ac} is the constant flow rate of the cooling system. Although we are not using a power meter, our results are very accurate because there is no interference from other heat sources as is the case with larger data centers where jobs from other users running on nearby nodes might dissipate a lot of heat which would distort cooling energy estimation for your experiments.

To the best of our knowledge, this is the largest testbed on which any HPC researcher has reported results with DVFS. Also, we are not aware of any other work on constraining core temperatures and showing its benefit in cooling energy savings. In contrast to most of the earlier work that emphasized on savings from machine power consumption using DVFS. Most importantly, our work is unique in using load balancing to mitigate effects of transient speed variations in HPC world.

We demonstrate the effectiveness of our scheme by using three applications having different utilization and power profiles. The first is a canonical benchmark, *Jacobi2D*, that uses 5 point stencil to average values in a 2D grid using 2D decomposition. The second application, *Wave2D*, uses a finite differencing scheme to calculate pressure information over a discretized 2D grid. The third application, *Mol3D*, is from molecular dynamics and is a real world application to simulate large biomolecular systems. For *Jacobi2D* and *Wave2D*, we choose a problem size of 22,000x22,000 and 30,000x30,000 grids respectively. For *Mol3D* however, we ran a system containing 92,224 atoms. We did an initial run of these applications without DVFS with CRAC working at 13.9 °C and noted the maximum average core temperature reached for all 128 cores. We then used our temperature aware load balancer to keep the core temperatures at 44 °C which was the maximum average temperature reached in the case of *Jacobi2D* (this was the lowest peak average temperature amongst all three applications). While keeping the threshold fixed at 44 °C, we decreased the cooling by increasing the CRAC set point. In order to gauge the effectiveness of our scheme, we compared it with the scheme in which DVFS is used to constrain core temperatures, without using any load balancing (we refer to it as *w/o TempLDB* throughout the paper).

5. TEMPERATURE CONTOL AND TIMING PENALTY

Temperature control is important for cooling energy considerations since it determines the heat dissipated into the air which the CRAC is responsible for removing. In addition to that, core temperatures and power consumption of a machine are related with a positive feedback loop, so that an increase in any of them causes an increase in the other [8]. Our earlier work [16] shows evidence of this where we ran *Jacobi2D* on a single node with 8 cores and measured the machine power consumption along with core temperatures. The results showed that increase in core temperature can cause an increase of up to 9% in machine power consumption and this figure can be huge for large data centers. For our testbed in this work, Figure 3 shows the average temperature for all 128 cores over a period of 10 minutes using our temperature aware load balancing. The CRAC was set to 21.1 °C for these experiments. The horizontal line is drawn as a reference to show the maximum temperature threshold (44 °C) used by our load balancer. As we can see, irrespective of how large the temperature gradient is, our scheme is able to restrain core temperature to within 1 °C. For example, core temperatures for *Mol3D* and *Wave2D* reach the threshold i.e. 44 °C much sooner than *Jacobi2D*. But all three applications stay very close to 44 °C after reaching the threshold.

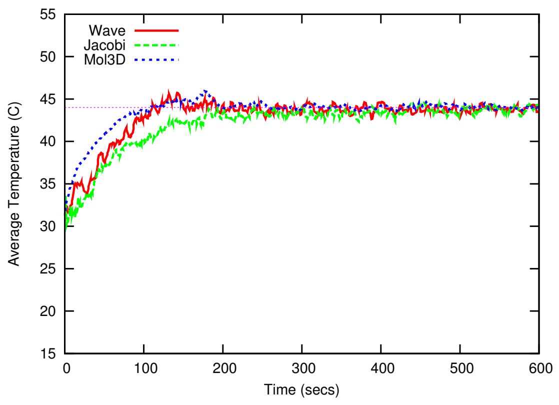


Figure 3: Average core temperature with CRAC set point at 21.1 °C

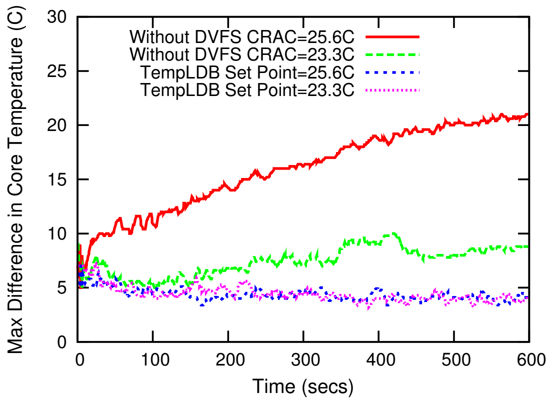


Figure 4: Max difference in core temperatures for Wave2D

Temperature variation *across* nodes is another very important factor. Spatial temperature variation is known to cause hot spots which can drastically increase the cooling costs of a data center. To get some insight into hot spot formation, we performed some experiments on our testbed with different CRAC set points. Each experiment was run for 10 minutes. Figure 4 shows the maximum difference any core has from the average core temperature for *Wave2D* when run with different CRAC set points. The *Without DVFS* run refers to all cores working at full frequency and no temperature control at core-level. It was observed that for the case of *Without DVFS* run, the maximum difference is due to one specific node getting hot and continuing to be so throughout the execution i.e. a hot spot. On the other hand, with our scheme, no single core is allowed to get a lot hotter than the maximum threshold. Currently, for all our experiments, we do temperature measurement and DVFS after every 6-8 seconds. More frequent DVFS would result in more execution time penalty since there is some overhead of doing task migration to balance the loads. We will return to these overheads later in this section.

The above experimental results showed the efficacy of our scheme in terms of limiting core temperatures. However, as shown in Section 2, this comes at the cost of execution time. We now use savings in execution time penalty as a metric to establish the superiority of our temperature aware load balancer in comparison to using DVFS without any load balancing. For this, we study the normalized execution times, t_{norm}, with and without our temperature aware load balancer, for all three applications under consideration. We define t_{norm} as follows:

$$t_{norm} = t_{LB}/t_{base} \tag{2}$$

where t_{LB} represents the execution time for temperature aware load balanced run and t_{base} is execution time without DVFS so that all cores work at maximum frequency. The value for t_{norm} in case of *w/o TempLDB* run is calculated in a similar manner except that we use t_{NoLB} instead of t_{LB}. We experiment with different CRAC set points. All the experiments were performed by actually changing the CRAC set point and allowing the room temperature to stabilize before any experimentation and measurements were done. To minimize errors, we averaged the execution times over three similar runs. Each run takes longer than 10 minutes to allow fair comparison between applications. The results of this experiment are summarized in Figure 5. The results show that our scheme consistently performs better than *w/o TempLDB* scheme as manifested by the smaller timing penalties for all CRAC set points. As we go on reducing the cooling (i.e. increasing the CRAC set point), we can see degradation in the execution times i.e. an increase in timing penalty. This is not unexpected and is a direct consequence of the fact that the cores heat up in lesser time and scale down to lower frequency thus taking longer to complete the same run.

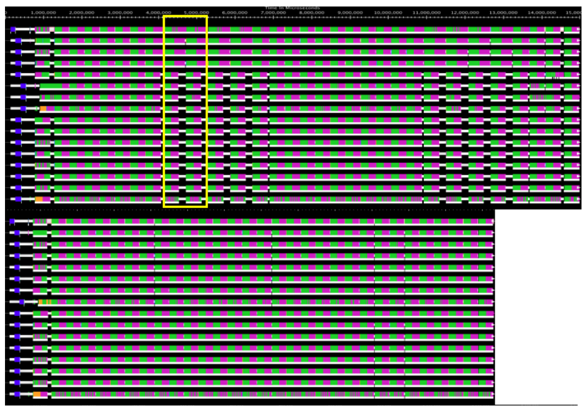
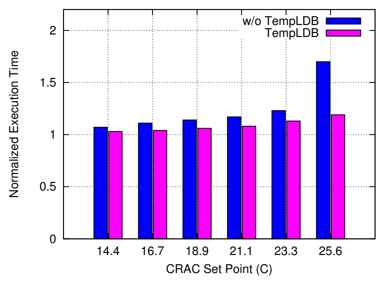
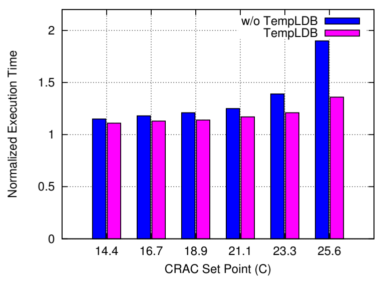


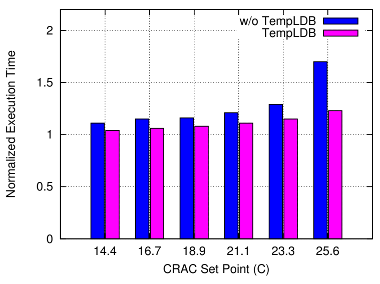
Figure 6: Projections timeline with and without Temperature Aware Load Balancing for *Wave2D*



(a) Jacobi2D



(b) Wave2D



(c) Mol3D

Figure 5: Normalized execution time with and without Temperature Aware Load Balancing

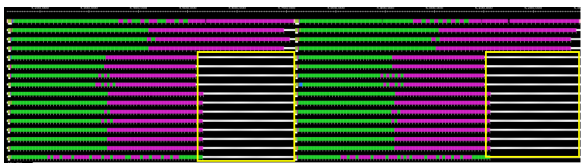


Figure 7: Zoomed Projections for 2 iterations

It is interesting to observe from Figure 5 that the difference in our scheme and the *w/o TempLDB* scheme is small to start with but grows as we increase the CRAC set point. This is because when the machine room is cooler, the cores take longer to heat up in the first place. As a result, even the cores falling in the hot spot area do not become so hot that they go to a very small frequency (we decrease frequency in steps of 0.13GHz). But as we keep on decreasing the cooling, the hot spots become more and more visible, so much so that when the CRAC set point is 25.6 °C, Node 10 (hot spot in our testbed) runs at the minimum possible frequency almost throughout the experiment. Our scheme does not suffer from this problem since it intelligently assigns loads by taking core frequencies into account. But without our load balancer, the execution time increases greatly (refer to Figure 5 for CRAC set point 25.6 °C). This happens because in the absence of load balancing, execution time is determined by the slowest core i.e. core with the minimum frequency.

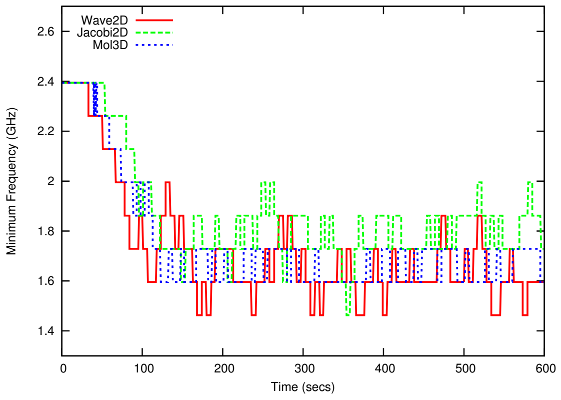


Figure 8: Minimum frequency for all three applications

For a more detailed look at our scheme's sequence of actions, we use Projections [6], a performance analysis tool from the Charm++ infrastructure. Projections provides a visual demonstration of multiple performance data including processor timelines showing their utilization. We carried out an experiment on 16-cores instead of 128 and use projections to highlight the salient features of our scheme. We worked with a smaller number of cores since it would have been difficult to visually understand a 128-core timeline. Figure 6 shows the timelines and corresponding utilization for all 16 cores throughout the execution of *Wave2D*. Both runs in the figure had DVFS enabled. The upper run i.e. the top 16 lines, is the one where *Wave2D* is executed without temperature aware load balancing whereas the lower part i.e. the bottom 16 lines, repeated the same execution with our temperature aware load balancing. The length of the timeline indicates the total time taken by an experiment. The green and pink colors show the computations, whereas the white lines represents idle time. Notice that the execution time with temperature aware load balancing is much less than that without it. To see how processors spend their time, we zoomed into the boxed part of Figure 6 and reproduced it in Figure 7. It represents 2 iterations of *Wave2D*. This zoomed part belongs to the run without temperature aware load balancing. We can see that because of DVFS, the first four cores work at a lower frequency than the remaining 12 cores. They, therefore, take longer to complete their tasks as compared to the remaining 12 cores (longer pink and green portions on the first 4 cores). The remaining 12 cores finish their work quickly and then keep on waiting for the first 4 cores to complete their tasks (depicted by white spaces towards the end of each iteration). These results clearly suggest that the timing penalty is dictated by the slowest cores. We also substantiate this by providing Figure 8 which shows the minimum frequency of any core during a *w/o TempLDB* run (CRAC set point at 23.3 °C). We can see from Figure 5 that *Wave2D* and *Mol3d* have higher penalties as compared to *Jacobi2D*. This is because the minimum frequency reached in these applications is lower than that reached in *Jacobi2D*.

We now discuss the overhead associated with our temperature aware load balancing. As outlined in Algorithm 1, our scheme has to measure core temperatures, do DVFS, decide new assignments and then exchange tasks according to the new schedule. The major overhead in our scheme comes from the last item i.e. exchange of tasks. In comparison, temperature measurements, DVFS, and load balancing decisions take negligible time. To calibrate the communica-

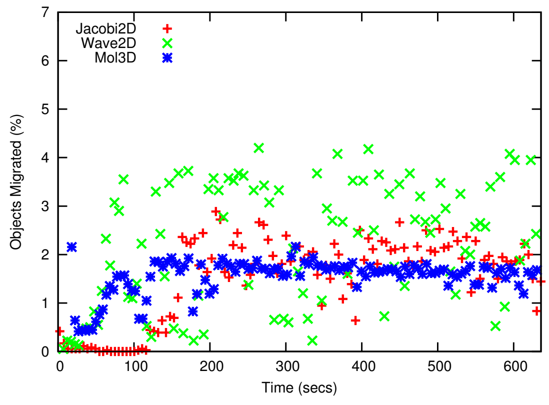


Figure 9: Percent objects migrated during temperature aware load balancer run

tion load we incur on the system, we run an experiment with each of the three applications for ten minutes and count the number of tasks migrated at each step when we check core temperatures. Figure 9 shows these percentages for all three applications. As we can see, the numbers are very small to make any significant difference. The small overhead of our scheme is also highlighted by its superiority over the *w/o TempLDB* scheme which does temperature control through DVFS but no load balancing (and so, no object migration). One important observation to be made from this figure is the larger number of migrations in *Wave2D* as compared to the other two applications. This is because it has a higher CPU utilization. *Wave2D* also consumes/dissipates more power than the other two applications and hence has more transitions in its frequency. We explain and verify these application-specific differences in power consumption in the next Section.

6. UNDERSTANDING APPLICATION REACTION TO TEMPERATURE CONTROL

One of the reasons we chose to work with three different applications was to be able to understand how application-specific characteristics react to temperature control. In this section, we highlight some of our findings and try to provide a comprehensive and logical explanation for them.

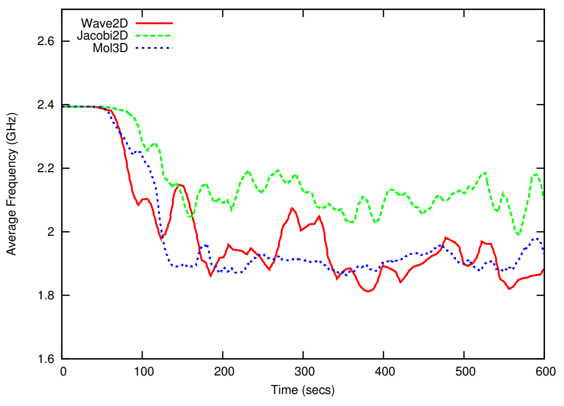


Figure 10: Average frequency for all three applications with CRAC at 23.3 °C

We start by referring back to Figure 5 which shows that *Wave2D* suffers the highest timing penalty followed by *Mol3D* and *Jacobi2D*. Our intuition was that this difference could be explained by the frequencies at which each application is running along with their CPU utilizations (see Table 2). Figure 10 shows the average frequency across all 128 cores during the execution time for each application. We were surprised with Figure 10 because it showed that both *Wave2D* and *Mol3D* run at almost the same average frequency throughout the execution time and yet *Wave2D* ends up having a much higher penalty than *Mol3D*. Upon investigation, we found that *Mol3D* is less sensitive to frequency than *Wave2D*. To further gauge the sensitivity of our applications to frequency, we ran a set of experiments in which each application was run at all available frequency levels. Figure 11 shows the results where execution times are normalized with respect to a base run where all 128 cores run at maximum frequency i.e. 2.4GHz. We can see from Figure 11 that *Wave2D* has the steepest curve indicating its sensitivity to frequency. On the other hand, *Mol3D* is the least sensitive to frequency as shown by its small slope. This gave us one explanation for the higher timing penalties for *Wave2D* as compared to the other two. However, if we use this line of reasoning only, then *Jacobi2D* is more sensitive to frequency (as shown by Figure 11) and has a higher utilization (Table 2) and should therefore have a higher timing penalty than *Mol3D*. But Figure 5 suggests otherwise. Moreover, the average power consumption of *Jacobi2D* is also higher than *Mol3D* (see Table 2) which should imply cores getting hotter sooner while running *Jacobi* than with *Mol3d* and shifting to lower frequency level. On the contrary, Figure 10 shows *Jacobi* running with a much higher frequency than *Mol3D*. These counter intuitive results could only be explained in terms of CPU power consumption which is higher in case of *Mol3D* than for *Jacobi2D*. To summarize, these results suggest that although the total power consumption of the entire machine is smaller for *Mol3D*, the proportion consumed by CPU is higher as compared to the same for *Jacobi2D*.

For some mathematical backing to our claims, we look at the following expression for core temperatures [9]:

$$T_{cpu} = \alpha T_{ac} + \beta P_i + \gamma \tag{3}$$

Here T_{cpu} is the core temperature, T_{ac} is temperature of the air coming from the cooling unit, P_i is power consumed by the chip, α, β and γ are constants which depend on heat capacity and air flow since our CRAC maintains a constant airflow. This expression shows that core temperatures are dependent on power consumption of the chip rather than the whole machine, and therefore it is possible that the cores get hotter for *Mol3D* earlier than with *Jacobi2D* due to higher CPU power consumption.

So far, we have provided some logical and mathematical explanations for our counter-intuitive results. But we wanted to explore them thoroughly and find more cogent evidence to our claims. As a final step towards this verification, we ran all three applications on 128 cores using the performance capabilities of Perfsuite [7] and collected information about different performance counters summarized in Table 2. We can see that *Mol3D* faces fewer cache misses and has 10 times more traffic between L1 and L2 cache (counter type 'Data Traffic L1-L2') resulting in higher MFLOP/s than *Jacobi2D*. The difference between the total power consumption of *Jacobi2D* and *Mol3D* can now be ex-

Table 2: Performance counters for one core

Counter Type	Jacobi2D	Mol3D	Wave2D
Execution Time (secs)	474	473	469
MFLOP/s	240	252	292
Traffic L1-L2 (MB/s)	995	10,500	3,044
Traffic L2-DRAM (MB/s)	539	97	577
Cache misses to DRAM (billions)	4	0.72	4.22
CPU Utilization (%)	87	83	93
Power (W)	2472	2353	2558
Memory Footprint(% of memory)	8.1	2.4	8.0

plained in terms of more DRAM access for *Jacobi2D*. We sum our analysis up by remarking that MFLOP/s seems to be the most viable deciding factor in determining the timing penalty that an application would have to bear when cooling in the machine room is lowered. Figure 12 substantiates our claim. It shows that *Wave2D*, which has the highest MFLOP/s (Table 2) suffers the most penalty followed by *Mol3D* and *Jacobi2D*.

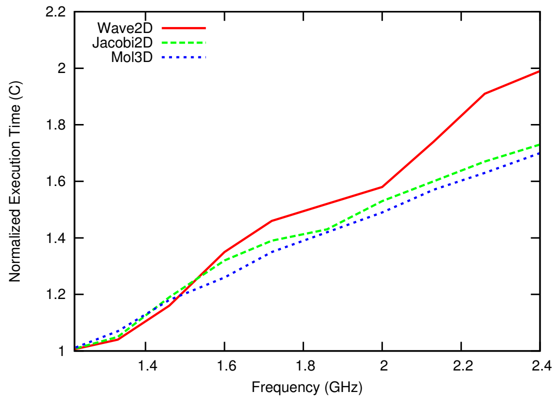


Figure 11: Normalized execution time for different frequency levels

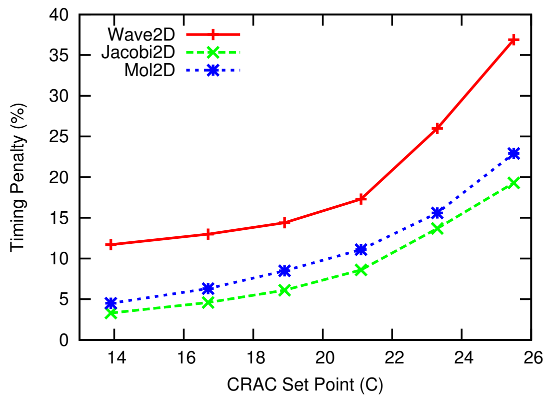


Figure 12: Timing penalty for different CRAC set points

7. ENERGY SAVINGS

This section is dedicated to a performance analysis for our temperature aware load balancing in terms of energy consumption. We first look at machine energy and cooling energy separately and then combine them to look at the total energy.

7.1 Machine Energy Consumption

Figure 13 shows the normalized machine energy consumption (e_{norm}), calculated as:

$$e_{norm} = e_{LB}/e_{base} \tag{4}$$

where e_{LB} represents the energy consumed for temperature aware load balanced run and e_{base} is execution time without DVFS with all cores working at maximum frequency. e_{norm}, for *w/o TempLDB* run is calculated in a similar way with e_{LB} replaced by e_{NoLB}. Static power of CPU, along with the power consumed by power supply, memory, hard disk and the motherboard mainly form the idle power of a machine. A node of our testbed has an idle power of 40W which represents 40% of the total power when the machine is working at full frequency assuming 100% CPU utilization. It is this high idle/base power which inflates the total machine consumption in case of 'w/o TempLDB' runs as shown in Figure 13. This is because for every extra second of penalty in execution time, we will pay an extra 40J per node in addition to the dynamic energy consumed by the CPU. Considering this, our scheme does well to keep the normalized machine energy consumption close to 1 as shown in Figure 13.

We can better understand the reason why the *w/o TempLDB* run is consuming much more power than our scheme if we refer back to Figure 7. We can see that although the lower 12 cores are idle after they are done with their tasks (white portion enclosed in the rectangle), they still consume idle power thereby increasing the total energy consumed.

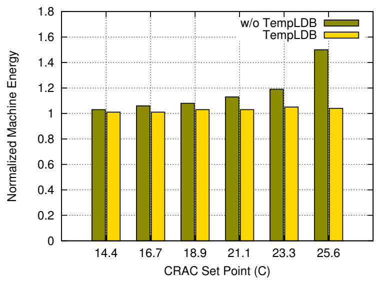
7.2 Cooling Energy Consumption

While there exists some literature discussing techniques for saving cooling energy, those solutions are not applicable to HPC where applications are tightly coupled. Our aim in this work, is to come up with a framework for analyzing cooling energy consumption specifically from the perspective of HPC systems. Based on such a framework, we can design mechanisms to save cooling energy that are particularly suited to HPC applications. We now refer to Equation 1 to infer that T_{hot} and T_{ac}, are enough to compare energy consumption for CRAC as the rest are constants.

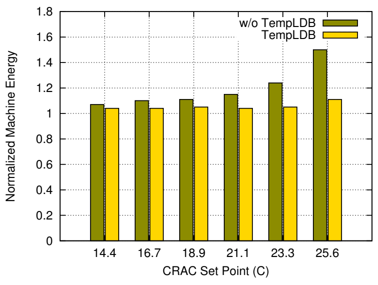
So we come up with the following expression for normalized cooling energy (c_{norm}):

$$c_{norm} = \frac{T_{hot}^{LB} - T_{ac}^{LB}}{T_{hot}^{base} - T_{ac}^{base}} * t_{norm}^{LB} \tag{5}$$

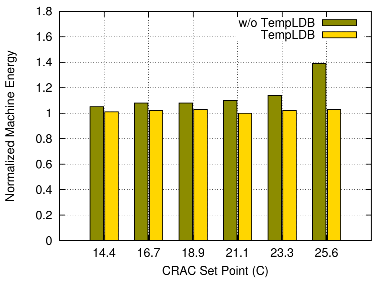
where T_{hot}^{LB} represents temperature of hot air leaving the machine room (entering the CRAC) and T_{ac}^{LB} represents temperature of the cold air entering the machine room respectively when using temperature aware load balancer. Similarly, when running all the cores at maximum frequency without any DVFS, T_{hot}^{base} is the temperature of hot air leaving the machine room and T_{ac}^{base} is the temperature of the cold air entering the machine room. t_{norm} is the normalized time for the temperature aware load balanced run. Notice



(a) Jacobi2D



(b) Wave2D



(c) Mol3D

Figure 13: Normalized machine energy consumption with and without Temperature Aware Load Balancing

that we include the timing penalty in our cooling energy model so that we incorporate the additional time for which cooling must be done.

Figure 14 shows the normalized cooling energy for both with and without temperature aware load balancer. We can see from the figure that both schemes end up saving some cooling energy but temperature aware load balancing outperforms *w/o TempLDB* scheme by a significant margin. Our temperature readings showed that the difference between T_{hot} and T_{ac} was very close in both cases i.e. our scheme and the *w/o TempLDB* scheme, and the savings in our scheme was a result of savings from t_{norm}.

7.3 Total Energy Consumption

Although most data centers report cooling to account for 50% [19, 3, 17] of total energy, we decided to take a conservative figure of 40% [11] for it in our calculations of total energy. Figure 15 shows the percentage of total energy we save and the corresponding timing penalty we end up paying for it. Although it seems that *Wave2D* does not give us much room to decrease its timing penalty and energy, we would like to mention that our maximum threshold of 44 °C was very conservative for it. On the other hand, results from *Mol3D* and *Jacobi2D* are very encouraging in the sense that if a user is willing to sacrifice some execution time, he can save a considerable amount of energy keeping core temperatures in check. It should also be noticed that our current constraints are very strict considering that we do not allow any core to go above the threshold. If we were to allow for a range of temperatures instead of one strict threshold, we can improve the timing penalty even more. For example, we did an experiment with *Mol3D*, where the allowed core temperature range was set to 44 °C – 49 °C and CRAC set point was 23.3 °C. With these settings, we saved 18% energy after paying only 4% timing penalty.

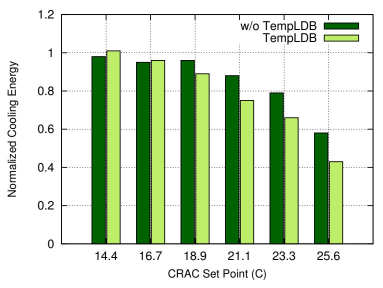
To quantify energy savings achievable with our technique, we plot normalized time against normalized energy (Figure 16). The figure shows data points for both our scheme and *w/o TempLDB* scheme. We can see that for each CRAC set point, our scheme moves the corresponding *w/o TempLDB* point towards the left (reducing energy) and down (reducing timing penalty). The slope of these curves would give us the number of seconds the execution time increases for each joule saved in energy. As we see *Jacobi2D* has a higher potential for energy saving as compared to *Mol3D* because of the lower MFLOP/s.

8. RELATED WORK

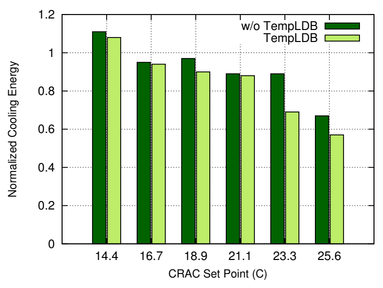
Most researchers from HPC have focused on minimizing machine energy consumption as opposed to cooling energy [15, 1, 21]. Given a target program, a DVFS enabled cluster, and constraints on power consumption, they [18] come up with a frequency schedule that minimizes execution time while staying within the power constraints. Our work differs in that we base our DVFS decisions on core temperatures for saving cooling energy whereas they devise frequency schedules according to task schedule irrespective of core temperatures. Their scheme works with load balanced applications only whereas ours has no such constraints. In fact one of the major features of our scheme is that it strives to achieve a good load balance. A runtime system named, PET (Performance, power, energy and temperature management), by Hanson et al [4], tries to maximize performance while respecting power, energy and temperature constraints. Our goal is similar to them but we achieve it in a multicore environment which adds an additional dimension of load balancing.

The work of Banarjee et al [1] comes closest to ours in the sense that they also try to minimize cooling costs in an HPC data center. But their focus is on controlling the CRAC set points rather than the core temperatures. In addition, they need to know the job start and end times beforehand to come up with the correct schedule whereas our technique does not rely on any pre-runs. Merkel et al [10] also explore the idea of task migration from hot to cold cores. However, they do not do it for parallel applications and therefore do not have to deal with complications in task migration decisions because of synchronization primitives. In another work, Tang et al.[21] have proposed a way to decrease cooling and avoid hot spots by minimizing the peak inlet temperature from the machine room through intelligent task assignment. But their work is based on a small-scale data center simulation while ours is comprised of experimental results on a reasonably large testbed.

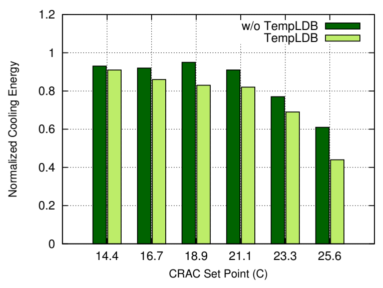
Work related to cooling energy optimization and hot-spot avoidance has been done extensively in non HPC data centers [2, 12, 22, 23, 20]. But most of this work relies on placing jobs such that jobs expected to generate more heat are placed on nodes located at relatively cooler areas in the machine room and vice versa. Rajan et al [14] discuss the effectiveness of system throttling for temperature aware scheduling. They claim system throttling rules to be the best one can achieve under certain assumptions. But one of their assumptions, non-migrateability of tasks, is clearly not true



(a) Jacobi2D

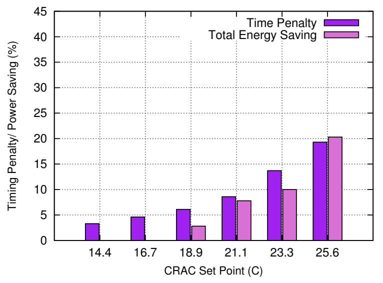


(b) Wave2D

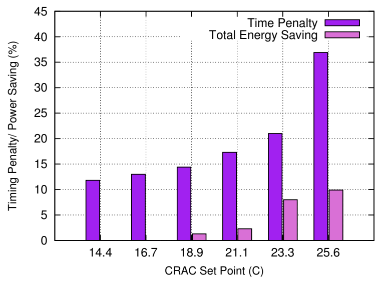


(c) Mol3D

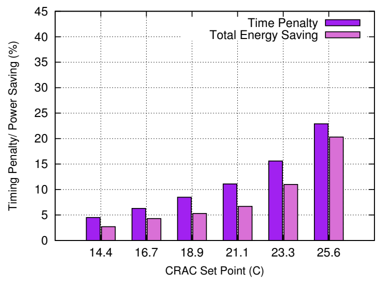
Figure 14: Normalized cooling energy consumption with and without Temperature Aware Load Balancing



(a) Jacobi2D

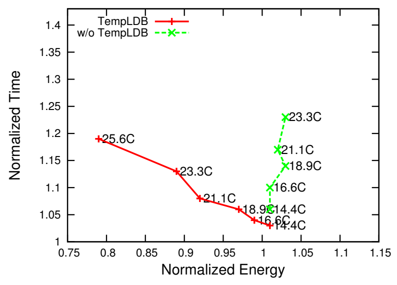


(b) Wave2D

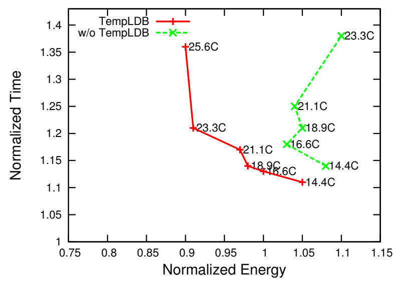


(c) Mol3D

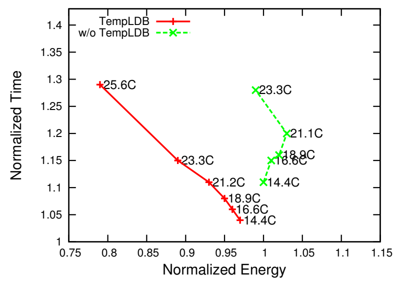
Figure 15: Timing penalty and power savings in percentage for temperature aware load balancing



(a) Jacobi2D



(b) Wave2D



(c) Mol3D

Figure 16: Normalized time as a function of normalized energy

for HPC applications we target. Another recent approach is used by Le at al [9] where they switch machines on and off in order to minimize total energy to meet the core temperature constraints. However, they do not consider parallel applications.

9. CONCLUSION

We experimentally showed the possibility of saving cooling and total energy consumed by our small data center for tightly coupled parallel applications. Our technique not only saved cooling energy but also minimized the timing penalty associated with it. Our approach was conservative in a manner that we set hard limits on absolute values of core temperature. However, our technique can readily be applied to constrain core temperatures within a specified temperature range which can result in much less timing penalty. We carried out a detailed analysis to reveal the relationship between application characteristics and the timing penalty that can be expected if it were to constrain core temperatures. Our technique was successfully able to identify and neutralize a hot spot from our testbed.

We plan to extend our work by incorporating critical path analysis of parallel applications in order to make sure that we always try to keep all tasks on critical path on the fastest cores. This would further reduce our timing penalty and possibly reduce machine energy consumption. We also plan to extend our work in such a way that instead of using DVFS to constrain core temperatures, we apply it to meet a certain maximum power threshold that a data center wishes not to exceed.

Acknowledgments

We are thankful to Prof. Tarek Abdelzaher for letting us use the testbed for experimentation.

10. REFERENCES

[1] A. Banerjee, T. Mukherjee, G. Varsamopoulos, and S. Gupta. Cooling-aware and thermal-aware workload placement for green hpc data centers. In *Green Computing Conference, 2010 International*, pages 245 –256, 2010.

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

[3] R. S. C. D. Patel, C. E. Bash. Smart cooling of datacenters. In *IPACK'03: The PacificRim/ASME International Electronics Packaging Technical Conference and Exhibition.*

[4] H. Hanson, S. Keckler, R. K, S. Ghiasi, F. Rawson, and J. Rubio. Power, performance, and thermal management for high-performance systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1 –8, march 2007.

[5] L. Kalé. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.

[6] L. V. Kalé and A. Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, Newport Beach, CA, April 1993.

[7] R. Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux. In *In Proc. of the Linux Cluster Conference, Chapel*, 2005.

[8] E. Kursun, C. yong Cher, A. Buyuktosunoglu, and P. Bose. Investigating the effects of task scheduling on thermal behavior. In *In Third Workshop on Temperature-Aware Computer Systems (TAC'S 06*, 2006.

[9] 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. In *The Second International Green Computing Conference (in submission)*, 2011.

[10] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06. ACM.

[11] L. Minas and B. Ellison. *Energy Efficiency For Information Technolog: How to Reduce Power Consumption in Servers and Data Centers.* Intel Press, 2009.

[12] L. Parolini, B. Sinopoli, and B. H. Krogh. Reducing data center energy consumption via coordinated cooling and load management. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 14–14, Berkeley, CA, USA, 2008. USENIX Association.

[13] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[14] D. Rajan and P. Yu. Temperature-aware scheduling: When is system-throttling good enough? In *Web-Age Information Management, 2008. WAIM '08. The Ninth International Conference on*, pages 397 –404, july 2008.

[15] 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*, pages 49:1–49:9, 2007.

[16] O. Sarood, A. Gupta, and L. V. Kale. Temperature aware load balancing for parallel applications: Preliminary work. In *The Seventh Workshop on High-Performance, Power-Aware Computing (HPPAC'11)*, Anchorage, Alaska, USA, 5 2011.

[17] R. Sawyer. Calculating total power requirments for data centers. American Power Conversion, 2004.

[18] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 230–238, New York, NY, USA, 2006. ACM.

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

[20] Q. Tang, S. Gupta, D. Stanzione, and P. Cayton. Thermal-aware task scheduling to minimize energy usage of blade server based datacenters. In *Dependable, Autonomic and Secure Computing, 2nd IEEE International Symposium on*, pages 195 –202, 2006.

[21] Q. Tang, S. Gupta, and G. Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *Parallel and Distributed Systems, IEEE Transactions on*, 19(11):1458 –1472, 2008.

[22] L. Wang, G. von Laszewski, J. Dayal, and T. Furlani. Thermal aware workload scheduling with backfilling for green data centers. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, pages 289 –296, 2009.

[23] L. Wang, G. von Laszewski, J. Dayal, X. He, A. Younge, and T. Furlani. Towards thermal aware workload scheduling in a data center. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 116 –122, 2009.

[24] G. Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.