Variation Among Processors Under Turbo Boost in HPC Systems

Bilge Acun, Phil Miller, Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL, 61801
{acun2, mille121, kale}@illinois.edu

Abstract

The design and manufacture of present-day CPUs causes inherent variation in supercomputer architectures such as variation in power and temperature of the chips. The variation also manifests itself as frequency differences among processors under Turbo Boost dynamic overclocking. This variation can lead to unpredictable and suboptimal performance in tightly coupled HPC applications. In this study, we use compute-intensive kernels and applications to analyze the variation among processors in four top supercomputers: Edison, Cab, Stampede, and Blue Waters. We observe that there is an execution time difference of up to 16% among processors on the Turbo Boost-enabled supercomputers: Edison, Cab, Stampede. There is less than 1% variation on Blue Waters, which does not have a dynamic overclocking feature. We analyze measurements from temperature and power instrumentation and find that intrinsic differences in the chips' power efficiency is the culprit behind the frequency variation. Moreover, we analyze potential solutions such as disabling Turbo Boost, leaving idle cores and replacing slow chips to mitigate the variation. We also propose a speed-aware dynamic task redistribution (load balancing) algorithm to reduce the negative effects of performance variation. Our speed-aware load balancing algorithm improves the performance up to 18% compared to no load balancing performance and 6% better than the non-speed aware counterpart.

1. Introduction

Heterogeneity in supercomputer architectures is often predicted as a characteristic of future exascale machines with non-uniform processors, for example, machines with GPGPUs, FPGAs, or Intel Xeon Phi co-processors. However, even today's architectures with nominally uniform processors are not homogeneous, i.e. there can be performance, power, and thermal variation among them. This variation can be caused by the CMOS manufacturing process of the transistors in a chip, physical layout of each node, differences in node assembly, and data center hot spots.

These variations can manifest themselves as frequency difference among processors under dynamic overclocking. Dynamic overclocking allows the processors to automatically run above their base operating frequency since power, heat, and manufacturing cost prevent all processors from constantly running at their maximum validated frequency. The processor can improve performance by opportunistically adjusting its voltage and frequency within its thermal and power constraints. Intel's Turbo Boost Technology is an example of this feature. Overclocking rates are dependent on each processor's power consumption, current draw, thermal limits, number of active cores, and the type of the workload [3].

High performance computing (HPC) applications are often more tightly coupled than server or personal computer workloads. However, HPC systems are mostly built with commercial off-the-shelf processors (with exceptions for special-purpose SoC processors as in the IBM Blue Gene series and moderately custom products for some Intel customers [5]). Therefore, HPC systems with recent Intel processors come with the same Turbo Boost Technology as systems deployed in other settings, even though it may be less optimized for HPC workloads. Performance heterogeneity among components and performance variation over time can hinder the performance of HPC applications running on supercomputers. Even one slow core in the critical path can slow down the whole application. Therefore heterogeneity in performance is an important concern for HPC users.

In future generation architectures, dynamic features of the processors are expected to increase, and cause their variability to increase as well. Thus, we expect variation to become a pressing challenge in future HPC platforms. Our goal in this paper is to measure and characterize the sources of variation, and propose solutions to mitigate their effects. The main contributions of this paper are:

- Measurement and analysis of performance variation of up to 16% between processors in top supercomputing platforms (§ 2)
- Measurement and analysis of frequency, power, and temperature of processors on Edison (§ 4)
- A demonstration of the performance degradation of HPC applications caused by variation (§ 4)
- Identify specific measurement and control features that future architectures should provide to enable responses to in-homogeneity at the level of applications, runtime

systems, and job schedulers (§ 5)
- Analysis of potential solutions to mitigate effects of inhomogeneity: disabling Turbo Boost, replacing slow chips, idling cores, and dynamic task redistribution (§ 6)
- A speed-aware dynamic task redistribution technique which improves performance up to 18% (§ 6.4)

To the best of our knowledge, there is no other work which measures and analyzes performance, frequency, temperature, and power variation among nominally equal processors under Turbo Boost at large scale (See related work in Section 7).

2. Motivation

Homogeneous synchronous applications running on multiple cores or processors are limited by the slowest rank. Hence, even one slow core can degrade the performance of the whole application. If one core is slower than others by $x\%$, then the whole application would run $x\%$ slower if the slow core is on the critical path. For applications with non-homogeneous workloads, this effect is not as straightforward to measure. In the worst case scenario, the heaviest loaded rank would be on the slowest core and that could make the application up to $x\%$ slower.

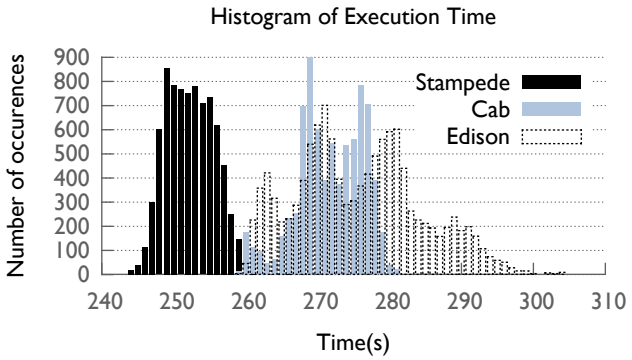

Figure 1: The distribution of benchmark times on 512 nodes of each machine looping MKL-DGEMM a fixed number of times on each core.

The impact of the core-to-core performance difference is also based on what fraction of the cores are fast and what fraction of them are slow. If there are only a few fast cores and most of the cores are slower, then the situation is not unfavorable. However the opposite of this condition, i.e most of the cores are fast but some of them are slow, is unfavorable. Figure 1, shows the histogram of the core performance running a benchmark that calls Intel MKL-DGEMM sequentially on the Edison, Cab and Stampede supercomputers. Overall core-to-core performance difference is around 16%, 8% and 15% respectively.

To further understand this time difference, we look into the relation between total time and the average frequency of

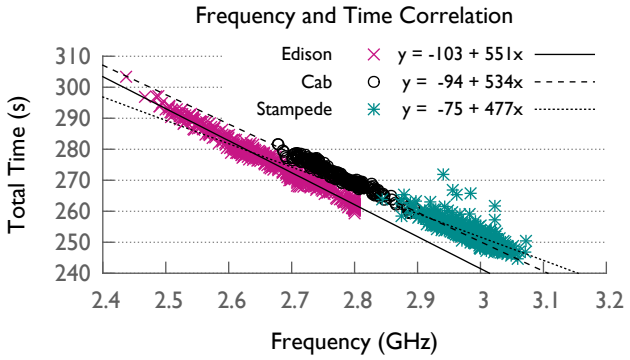

Figure 2: Average frequency shows a negative correlation with total execution time on both Edison, Cab and Stampede when running MKL-DGEMM.

the cores from the whole execution. The frequency difference among the chips is a result of the dynamic overclocking feature of the processors. Figure 2 shows the correlation for Edison, Cab, and Stampede supercomputers on 512 compute nodes. There is an inverse linear correlation between the time and the frequency of the processors with fit lines shown in the figure. The values of the R^2 correlation coefficient are 0.977, 0.965, 0.303 respectively, where 1 indicates a perfect linear trend. Edison and Cab show almost perfect inverse correlation, while Stampede has a lower R^2 value because of the interference or noise: the execution time of some of the cores were longer even though they were not running at a slower frequency. We note a few other features of these measurements. Edison processors (Ivy Bridge) span a wider range of frequencies, nearly 400 MHz, than the 300 MHz spread among processors in Cab and Stampede (Sandy Bridge). Many of Edison's processors reach the maximum possible frequency, while none of those in Cab or Stampede do the same. These observations may indicate broader generational trends among Intel processors.

We also look for variation in a platform which does not have a dynamic overclocking feature: Blue Waters, which has AMD Bulldozer processors. Blue Waters cores do not show any significant performance difference among them. Overall performance variation is less than 1% among 512 compute nodes. Therefore we do not further analyze Blue Waters.

To summarize our motivation, we show that there is a substantial frequency and consequent execution time difference among the cores under dynamic overclocking running the same workload. In HPC applications, this variation is particularly bad for performance because slower processors will hold back execution through the load imbalance and critical path delays they introduce. Moreover, this effect worsens with scale, because a larger set of processors increases the probability of encountering more extreme slow outliers.

Table 1: Platform hardware and software details

Platforms	Edison	Cab	Stampede	Blue Waters
Processor	Intel Xeon(R) E5-2695 v2 (Ivy Bridge)	Intel Xeon(R) E5-2670 (Sandy Bridge)	Intel Xeon(R) E5-2680 (Sandy Bridge)	AMD 6276 Interlagos
Clock Speed	Nominal 2.4 GHz	Nominal 2.6 GHz	Nominal 2.7 GHz	Nominal 2.3 GHz
Turbo Speed	# Cores: 1 / 2 / 3 / 4 / 5-12 GHz: 3.2 / 3.1 / 3.0 / 2.9 / 2.8	# Cores: 1-2 / 3-4 / 5-6 / 7-8 GHz: 3.3 / 3.2 / 3.1 / 3.0	# Cores: 1-2 / 3-5 / 6 / 7-8 GHz: 3.5 / 3.4 / 3.2 / 3.1	No Boost Feature
TDP	115 W	115 W	130 W	-
Cores per node	$12 \times 2 = 24$	$8 \times 2 = 16$	$8 \times 2 = 16$	$16 \times 2 = 32$
Cache size (L3)	30MB(shared)	20MB(shared)	20MB(shared)	16MB(shared)

3. Experimental Setup and Background

3.1. Platforms

We have used 4 different top supercomputing platforms in our experiments. We list the detailed specifications of the platforms in Table 1.

Edison is a Cray XC30 supercomputer at NERSC [2]. Each compute node has 2 Intel Ivy Bridge processors with Intel's Turbo Boost version 2.0 feature enabled. The actual CPU frequency can peak up to 3.2 GHz if there are 4 or fewer cores active within a chip. When all cores are active, the cores can peak up to 2.8 GHz [4]. The platform gives users the ability to change the nominal frequency and the Linux kernel's power governors. It has 14 fixed frequency states ranging from 1.2GHz to 2.4GHz and users can specify the frequency at job launch with the `aprun --pstate` command. The `--p-governor` flag sets the power governor.

Edison has 5576 compute nodes. We have used up to 1024 randomly allocated nodes in our experiments. Thus, we believe our results are representative of the whole machine.

Cab is a supercomputer at LLNL [1]. Each computer node has dual socket Intel Sandy Bridge processors with Intel Turbo Boost version 2.0 enabled. The actual CPU frequency can peak up to 3.3 GHz if there are 1 or 2 cores active within a chip. When all cores are active, the cores can peak up to 3.0 GHz.

Stampede is TACC's supercomputer [7]. Each compute node of Stampede has Intel Sandy Bridge processors and one Xeon Phi coprocessor. We do not use the coprocessor in our experiments. The processors have Intel Turbo Boost version 2.0 enabled. The actual CPU frequency can peak up to 3.5 GHz if there are 1 or 2 cores active within a chip. When all cores are active, the cores can peak up to 3.1 GHz.

Blue Waters is a Cray XE/XK system at NCSA. For our experiments we use the XE nodes which have two AMD processors with 16 Bulldozer cores [22]. The processors in this system do not have a dynamic overclocking feature like Intel's Turbo Boost.

3.2. Applications

3.2.1. Matrix Multiplication

Dense matrix multiplication is a relatively compute-bound operation that simultaneously stresses a broad subset of a system's hardware. We run the sequential matrix multiply kernels in a loop on each core with data which fit in the last level (L3) cache i.e. we use three 296x296 double-precision matrices, which requires around 2MB data per core and 24MB per chip where the L3 cache is 30MB on Ivy Bridge cores. Using data which fits in cache eliminates the effect of memory and cache related performance variation in our timings.

MKL-DGEMM: This kernel comes from Intel's Math Kernel Library (Intel MKL) version 13.0.3 on Edison and 13.0.2 on Stampede. Specifically, we call the `cblas_dgemm` function. We use this as a representative of a maximally hardware-intensive benchmark.

NAIVE-DGEMM: This kernel is a simplistic hand-written 3-loop sequential, double-precision dense matrix multiply. We use this as a representative of application code with typical compiler optimization settings, but that has not been hand-optimized or auto-tuned for maximum performance on a given system architecture.

Data alignment, padding, compiler flags We use 2MB alignment using `mkl_malloc()` or `posix_memalign()` (respectively) with 0, 64, or 128 bytes of padding for the data buffers to avoid cache aliasing. Preliminary experiments showed that neglecting this effect created substantial performance perturbations. We do not explore that issue both because it has been addressed by substantial previous research and because more realistic applications are much less likely to encounter it as consistently as our micro-benchmark. We use Intel's *icc* compiler (version 15.0.1 on Edison, and 13.0.2 on Stampede) with *-O3* and *-march=native* flags.

3.2.2. LEANMD

This is a mini-app version of NAMD, a production-level molecular dynamics application for high performance biomolecular simulation systems [24]. It does bonded, short-range, and long-range force calculations between atoms. In

our experiments, we use the benchmark size of around 1.8 million atoms. The benchmark is written in CHARM++ parallel programming framework.

3.2.3. JACOBI2D

This is a 5-point stencil application on a 2D grid. The application uses the CHARM++ parallel programming framework for parallelization. The grid is divided into multiple small blocks, each is represented as an object. We use multiple different grid sizes and block sizes in our experiments. For each iteration, the application executes in 3 stages, i.e. local computation, neighbor communication and barrier-based synchronization.

3.3. Measurement Methodology

We sample the time, hardware counters, temperature and power every 10 matrix multiply iterations for NAIVE-DGEMM and 100 iterations for MKL-DGEMM. This makes the sampling time roughly 20 milliseconds. For other benchmarks we do 1 second periodic measurements through an external module. The temperature and power measurements are specific to Edison.

Frequency Measurements: We use PAPI [10] to read the hardware counters. Specifically, we measure the total clock cycles, reference clock cycles, and cache misses. Total cycles (`PAPI_TOT_CYC`) "measure the number of cycles required to do a fixed amount of work" and reference clock cycles (`PAPI_REF_CYC`) "measure the number of cycles at a constant reference clock rate, independent of the actual clock rate of the core" [6]. We use the total and reference cycles to calculate the cycle ratio (`PAPI_TOT_CYC / PAPI_REF_CYC`). The cycle ratio gives us the effective clock rate of the processor. If the ratio is greater than one, it means the processor is running above the nominal speed and below means slower than the nominal speed. When running a workload under Turbo Boost this ratio is typically greater than one. On the other hand, if the processor is idle this ratio will typically be less than one [6]. In summary, we can obtain the clock frequency of the processor using the following formula:

$$Freq_{\text{effective}} = Freq_{\text{nominal}} \times \frac{TotalCycles}{ReferenceCycles}$$

Temperature Measurements on Edison: Edison users have read access to the temperature data of the cores through the `/sys/devices/platform/coretemp` interface.

Power Measurements on Edison: Edison allows read access to node level power meters for all users through the file: `/sys/cray/pm_counters/power` or using PAPI 'native' counters. We use the first option to get each compute node's power consumption. The power measurements are available as the whole compute node's power (CPUs, RAM, and all other components) in watts. These meters read out with an apparent 4 W resolution. Cab, Stampede and Blue Waters do not provide an application-accessible interface to access power consumption without a specific privilage.

We note that the CPUs in Edison, Cab and Stampede have model-specific registers (MSRs) that report CPU-level power and energy measurements. However, these are only accessible to OS kernel code or processes running with administrative privileges. We discuss this limitation further in Section 5.

3.4. Eliminating OS Interference

Operating systems and other extraneous processes can induce significant noise into the application [23]. On Edison and Blue Waters, we eliminate the effect of OS interference by binding all the OS processes to one core using the process launcher option `aprun -r 1`. From our observations, these systems use the last core in each node to satisfy this option. We then report measurements focusing on core 0 in each chip to avoid the effect of those OS processes. Cab and Stampede do not provide such an option.

3.5. Turbo Boost and Frequency

On the Intel Ivy Bridge EP processors found in Edison, there are two ways in which the observed operating frequency of a chip can differ from the nominal 2.4 GHz for which they are rated. P-states are software controlled, and Turbo Boost is hardware controlled within the limits of software-provided parameters and hardware constraints.

The software-driven P-state variable can be used to set the desired baseline frequency, ranging from 1.2 GHz to 2.4 GHz in increments of 100 MHz [4]. Various experiments have used this control in HPC runtime system, job scheduler, and resource manager software to optimize for energy usage, temperature, reliability, and performance under resource constraints [14, 18, 27]. On Edison, users can only set a uniform P-state across all nodes for the duration of a job using the `aprun --pstate` flag when launching compute processes. For the results examined in this paper, we use only the default maximum setting of this variable except in Section 6.1.

Intel Turbo Boost provides dynamic hardware-driven operation at frequencies above the baseline frequency requested by the P-state setting. All active cores within a chip operate at the same frequency. Software settings can bound the allowed range of frequency values that Turbo Boost can select, but none of the platforms currently allows users to control this. The hardware constraints are based on limits of operating temperature (max of 76° C), keeping power consumption below TDP, and current draw (value not documented in available sources) [4]. At 1 ms intervals, the hardware controller examines sensor values for these parameters and adjusts accordingly: if any limit is exceeded, then the CPU slows down 100 MHz; if no limit is exceeded and the frequency is below the maximum, then the CPU speeds up 100 MHz.

Table 2: Distribution of observed steady-state frequencies of 1K Chips on Edison

Application	Idle cores	Frequency (GHz) 2.4−2.5	2.5	2.5−2.6	2.6	2.6−2.7	2.7	2.7−2.8	2.8
MKL-DGEMM	0	5	31	116	125	254	154	211	128
	1	0	0	0	20	42	116	256	590
NAIVE-DGEMM	0	0	0	0	0	2	49	23	950
	1	0	0	0	0	0	2	0	1022
LEANMD	0	0	0	0	0	0	0	186	838
	1	0	0	0	0	0	0	8	1012
JACOBI2D	0	0	0	0	0	0	200	100	720
	1	0	0	0	0	0	50	50	924

Table 3: Frequency distribution of MKL-DGEMM on Cab

Frequency (GHz) 2.6−2.7	2.7	2.7−2.8	2.8	2.8−2.9	2.9
16	56	548	184	210	10

Table 4: Frequency distribution of MKL-DGEMM on Stampede

Frequency (GHz) 2.8−2.9	2.9	2.9−3.0	3.0	3.0−3.1
13	19	555	183	254

4. Measurement and Analysis of Variation

In this section, we measure and analyze the performance, frequency, thermal, and power variation among the chips. We use Edison in rest of the paper since it shows the highest variation not attributable to OS interference, and it gives users access to the most power and temperature measurements. Even this access is much less than ideal, as we discuss in Section 5.

We note that there is small intra-chip variation (i.e. variation between the cores within one chip) that is not caused by frequency; however, this variation is not significant. Therefore we only focus on the inter-chip variation that arises even though they are all of the same product model.

4.1. Inter-chip Frequency Variation

The chips takes a warm-up period from the job launch to settle down on a set frequency or a duty-cycle determined frequency average. Figure 4 illustrates this warm-up period with temperature, frequency and power measurements of a randomly selected compute node. The node has two sockets that behave differently. The temperature of Chip 1 is a few degrees higher over the run and it has a stable 2.8 GHz frequency. On the other hand, chip 2 starts at 2.8 GHz and the frequency drops to 2.5 GHz after around 18 seconds. Until the drop point, node power slowly increases from 320W to 330W and once Chip 2 hits the threshold, its frequency drops, causing its power level to drop. Duration of the warm-up period can vary depending on the application's compute intensity. For MKL-DGEMM the warm-up is around 20 seconds whereas for NAIVE-DGEMM it's around 1 minute. We exclude the warm-up period in our following reported measurements.

Table 2 shows the distribution of the steady-state frequencies of the chips on Edison. For example, during the run of MKL-DGEMM, 67 of the 512 chips run at the maximum possible frequency of 2.8 GHz. Since these chips are efficient and stable, we call these *fast chips*. These make up 13% of the whole tested allocation. There are other chips which are stable but run at a lower frequency, i.e 75 of the 512 chips run at a stable 2.7 GHz with MKL-DGEMM. These are stable but *slow chips*. Moreover, some of the chips have an average frequency that is not one of the set values (i.e. 2.8, 2.7, 2.6 or 2.5 GHz). This means that the chip could not settle down on a stable frequency and it is oscillating between two frequencies, i.e. 100 of the 512 chips have an average between 2.7 GHz and 2.8 GHz with MKL-DGEMM. We term these *variable chips*. Tables 3 and 4 show the corresponding data for MKL-DGEMM on Cab and Stampede, respectively.

To understand how these types of chips behave over time, we have selected 1 core from 3 chips which behave differently and show how the iteration time and the frequency changes over the iterations in Figure 3. The selected slow core has the highest iteration time compared to the other 2 selected cores, and its frequency of 2.7 GHz does not change over time. The fast core has the lowest iteration time and a frequency of 2.8 GHz which changes minimally over time. On the other hand, the variable core's iteration time the frequency make a wave pattern. By comparing the left and right figures we can observe that when the iteration time increases(or decreases) in the variable core, the frequency decreases(or increases). We analyzed the time and frequency correlation earlier in Figure 2, and the same correlation applies here as well.

Table 2 also shows the effect when one core is left idle. We try leaving one core idle from socket 2 in each compute node, to eliminate the potential for OS interference by binding the OS processes to the idle core. Leaving the core idle not only eliminates the interference but also reduces the number and severity of slow and variable chips as well. Since the

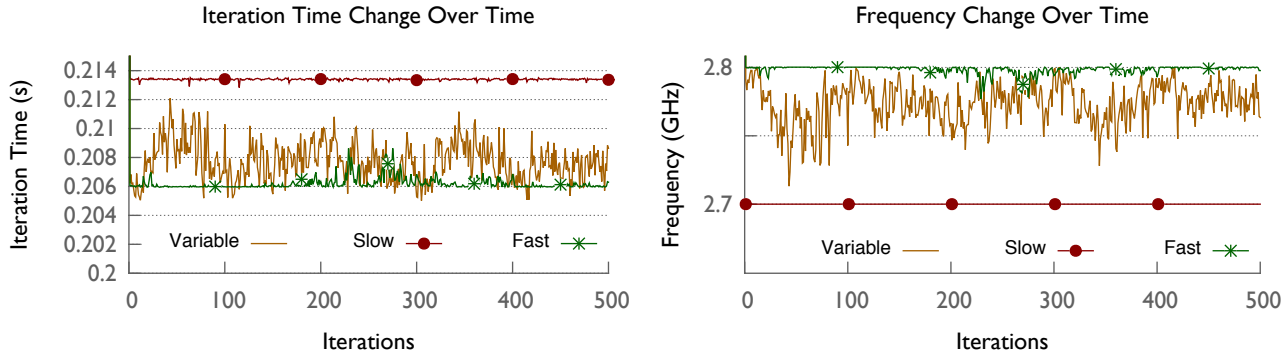

Figure 3: Iteration time (left plot) and frequency (right plot) over iterations are shown from cores selected from 3 chips showing distinct behavior: slow, variable, and fast.

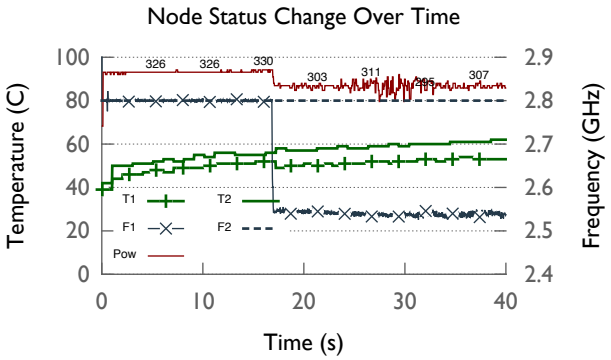

Figure 4: Plot shows the power (Pow (W)) of a randomly selected node, temperature (T1, T2) and frequency (F1, F2) of the two chips on the node.

chips run faster and more stably when one core is left idle, we discuss this arrangement as a potential means to avoid slow processors in Section 6.3.

MKL-DGEMM is a highly-optimized kernel which puts a lot of pressure on the CPU whereas NAIVE-DGEMM is not as intense. Consequently, while the chips fall as far down as 2.5 GHz with MKL-DGEMM, with NAIVE-DGEMM they fall down to 2.7 GHz. In a run on 1024 compute nodes of Edison, we see that 92% of the chips are fast and only about 7% of processors are unable to sustain a steady 2.8 GHz over the few minutes of our NAIVE-DGEMM benchmark run. Others either persistently vary between 2.7 and 2.8 GHz during the run, or stabilize after a variable length of time at 2.7 GHz. These off-nominal chips are exactly those on which the benchmark as a whole took longer to run. LEANMD and JACOBI2D applications shows a similar behavior to NAIVE-DGEMM. The more applications are optimized for performance, the more they are likely to encounter a chip running at a slower frequency. For example, an application using AVX instructions and data tiling for memory performance would have a high CPU intensity, whereas more time waiting for communication, synchronization or high memory access latency would give CPU more idle time which results in lower temperature and power values. We observed by experiment that such applications may show little frequency variation or none at all.

In a CPU-intensive parallel application, processors that are even slightly slower or less efficient than their cohort can potentially create a vicious cycle for themselves. Faster processors will experience idle time due to load imbalance and critical path delays. During that time, they will cool down and bank energy, stabilizing their temperature, power consumption, and frequency. The slower processors will run closer to a 100% duty cycle, pushing their temperature and power consumption up and their steady-state frequency down. As they get hotter, draw more power, and slow down, they become worse off relative to the faster chips. Thus, the effect amplifies and feeds back on itself.

4.2. Temperature and/or Power as Cause of Frequency Variation

There are several possible reasons for the frequency variation that we have observed. Turbo Boost adjusts the clock frequency based on the processor's power, current, temperature, active core count, and the frequency of the active cores. Active core count is irrelevant here, because Edison's processors can boost to a maximum of 2.8 GHz with 5–12 cores running.

We first try to understand if the frequency variation is caused by slow or variable processors reaching their temperature limit. In Figure 5 shows what frequency level processors are running at for each temperature bin for NAIVE-DGEMM and MKL-DGEMM benchmarks. We periodically collect frequency and temperature data from the whole execution including the warm-up time. Then, we bin the data points in terms of temperature and calculate which percentage of them run at what frequency level. We can see that chips running MKL-DGEMM span a wide range of frequencies and temperatures, with no apparent correlation. At every temperature level, there are processors from each frequency. The fastest chips reach temperatures as high as the slower chips. For NAIVE-DGEMM, as the temperature goes higher, the percentage of chips running at high frequency drops. However, very few chips are anywhere near the documented thresh-

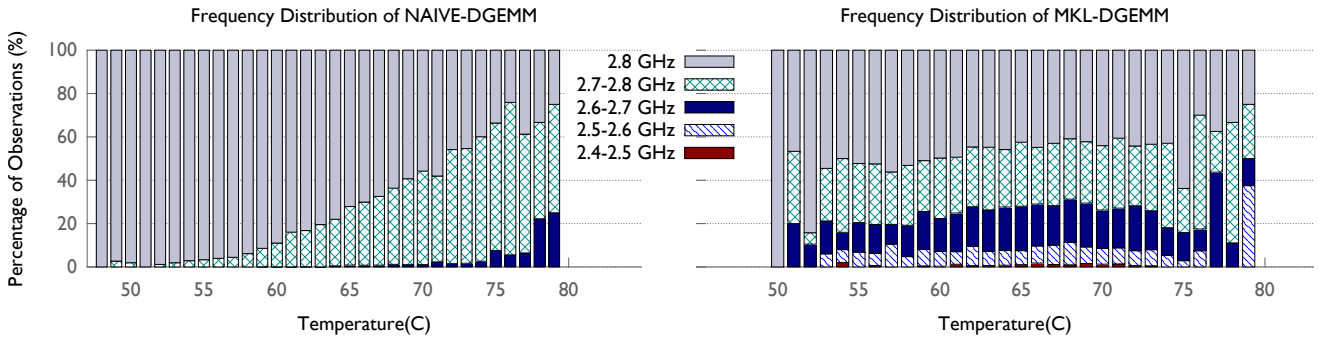

Figure 5: Plots show what percentage of the chips run at what frequency in each temperature level for NAIVE-DGEMM(left plot) and MKL-DGEMM(right plot) benchmarks. The data points are collected from the whole execution of the benchmarks and classified according to their temperature bins. Temperature of the chips running is not directly correlated with their frequency under the heavier load of MKL-DGEMM.

Power (W)	Total = 1024	F & F = 901	F & S = 63	F & V = 36	S & S = 13	S & V = 9	V & V = 2
260	4	4					
265	8	8					
270	60	60					
275	108	108					
280	162	162					
285	141	139	2				
290	123	119	2	2			
295	107	104	1	2			
300	89	75	7	6	1		
305	88	60	18	7	2	1	
310	77	41	19	7	5	5	
315	51	18	14	10	5	3	1
320	6	3	0	2	0	0	1

F = Fast Chip = 2.8 GHz
S = Slow Chip = 2.7 GHz
V = Variable Chip = 2.7 <f< 2.8GHz

Table 5: The distribution of whole-compute-node power consumption while running NAIVE-DGEMM, for nodes containing the various possible combinations of chips. Histograms are given to illustrate the strong regularity present in the distribution of power consumption values in the fast/fast case. This regularity suggests a relatively simple underlying stochastic process. As a compute node includes slower and variable chips, its distribution of measured power shifts upward, suggesting those chips are at their limiting power while the fast chips are not.

old temperature of 76° C. The data for LEANMD and JACOBI2D looks very similar to NAIVE-DGEMM. Thus, we conclude that the chips running slower than nominal have not slowed down due to reaching their thermal limit.

Another reason for the frequency variation could be power draw of the cores. The package control unit (PCU) in the processors with Turbo Boost Technology 2.0 has an intelligent exponential weight-moving average (EWMA) algorithm [25] to adjust the frequency of the cores. According to this algorithm, energy consumption of the processor is tracked within fixed time periods. Within these periods if the CPU is consuming less power than a threshold power limit, then it accumulates energy credit which can be used in the following period to boost the CPU frequency. That power threshold limit is thermal design power (TDP) which is the power level in the steady state where the frequency of the CPU could safely be higher than the nominal frequency [25]. We can observe this time interval in the EWMA algorithm from the variable core in Figure 3. The frequency changes in a rapid wave pattern over iterations. The core accumulates energy credits when the frequency is low and uses those credits to increase the frequency back again.

Table 5 shows the distribution of node power consumption from NAIVE-DGEMM. We have grouped the nodes by the pair of categories assigned to the processors they contain. Since we only have power measurements of the whole compute node on Edison, this pairing is necessary to identify the power consumption of different chip types. There are 3 distinct frequency levels with NAIVE-DGEMM: 2.8 GHz, 2.7 GHz, and between 2.7-2.8 GHz. We simply name them fast, slow, and variable chips. Since there are two chips in a node on Edison, a node can have one of 6 different combination of chips. These are: fast & fast, fast & slow, fast & variable, slow & slow, slow & variable, variable & variable.

We can see that the variable processors systematically consume more power than slow processors, which in turn con-

sume more power than fast processors. This occurs because a variable core is running at an average frequency right at the edge of what its power consumption will allow. As temperature rises even slightly, the power consumption increases to a point where the PCU will not allow the chip to ever step up to its higher frequency, and so it stabilizes at a lower frequency and hence slightly lower but still near-threshold level of power consumption. TDP of the processors is 115 Watts. However, since the power data is node-level power which includes not just CPU power but also power of RAM and other components in the node, the measured power is higher than $115 \times 2 = 230$ W.

Power measurements of LEANMD and JACOBI2D shows a very similar distribution to NAIVE-DGEMM. MKL-DGEMM also shows a similar distribution, however with a much narrower power range of 18 Watts: [302-320], instead of 60 Watts: [260-320] in NAIVE-DGEMM. The node categorization is more complicated then the categorization in Table 5 since there are 8 different frequency levels and it makes 56 different node types.

Although it is hard to make a concrete conclusion without CPU-level power data, our measurements show that that processors' frequency is likely throttled down due to the power limit. Increase in temperature increases the power consumption however, the lack of correlation between the temperature and frequency suggests the frequency variation is not directly due to temperature-driven throttling.

5. Architecture and System Needs

As supercomputing platforms are becoming more heterogeneous with thousands of processors and as processors are becoming more dynamic, forecasts predict more variation in the future. Therefore, it is important that applications, or runtime systems underneath the applications, be aware of this heterogeneity and to do necessary optimization to mitigate the effect of performance variation.

Many supercomputing platforms do not give users access to power or temperature measurements, or rights to control the frequency of the processors or apply power-capping algorithms. Access to these measurements and controls would give researchers the opportunity to understand the behavior of the hardware and hence improve application performance.

Edison supercomputer provides read access to node-level power, and core-level temperature measurements. However, node-level power measurements are not fine-grained enough, to make detailed studies and CPU-level power measurements are necessary. Edison also provides control of frequency and power in job-allocation-level i.e. all nodes participating the job launch has same frequency/power settings. However, this is also not fine-grained enough. Given the variation we observe among chips, every chip can have a different optimal power, frequency setting. Therefore, chip level power and frequency control is necessary. Cab and Stampede do not provide access to either power or temperature measurements and do not provide any frequency/power control mechanisms.

Table 6: Desired access of measurement and controls. ✓: There is support and access. ✗: There is no support. ◐: Hardware supports, but the software does not allow.

Need \ Platform	Edison	Cab	Stampede
Frequency Data	✓	✓	✓
Temperature Data	✓	◐	◐
Node Level Power Data	✓	◐	◐
Chip Level Power Data	◐	◐	◐
Core Level Power Data	✗	✗	✗
Per-chip Power Capping	◐	◐	◐
Per-chip Frequency Scaling	◐	◐	◐
Per-core Frequency Scaling	✗	✗	✗

Most current power and energy related studies are usually done in small clusters or using only a few processors. Access to power related controls and measurements would also enable researchers to extend their studies to much larger platforms, to the benefit of the whole HPC community.

6. Potential Solutions to Mitigate Performance Variation

In this section, we analyze potential solutions to mitigate the variation problem. The solutions do not all require power related measurement/control rights. These solutions are: disabling Turbo Boost, replacing slow chips, leaving some cores idle, and dynamic task redistribution.

6.1. Disable Turbo Boost

Turbo Boost enables the cores of Edison to speed up to 2.8 GHz when all cores are active. In this section, we show the performance of the applications when the frequency is fixed to run at 2.4 GHz using the `aprun --pstate` option (note that 2.4 GHz is the maximum possible non-Turbo frequency to set the processors at). Setting the frequency at 2.4 GHz removes the frequency variation among the chips and makes every chip run at 2.4 GHz.

Table 7: Percentage slowdown of applications when the frequency is fixed at maximum frequency of 2.4GHz

Application	% Slowdown
MKL-DGEMM	9.1
NAIVE-DGEMM	18.1
LEANMD	16.8
JACOBI2D	4.2

Our measurements show that with Turbo Boost enabled, even the slowest and the most variable chips are consistently running beyond the nominal clock speed of 2.4 GHz and the applications definitely have a performance gain. Table 7

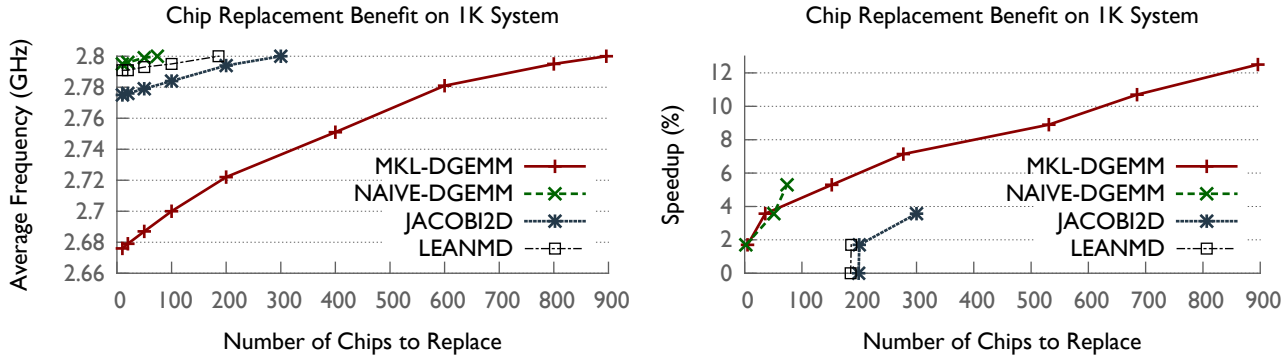

Figure 6: How many chips we should replace to get performance benefit?

shows the slowdown of the applications when the frequency is fixed at 2.4 GHz compared to the default case where Turbo Boost is on. All of the applications show significant performance degradation. NAIVE-DGEMM gets 18.1% performance degradation whereas MKL-DGEMM gets 9.1%. The applications that are running at higher frequency levels with Turbo-Boost on (i.e. NAIVE-DGEMM), shows more slowdown when Turbo-Boost is disabled (i.e. compared to MKL-DGEMM). LEANMD gets 16.8% because of its compute intensity, whereas application has a comparatively larger memory access latency and therefore the slowdown is only 4.3%. Memory bound applications are less affected by disabling Turbo-Boost.

As all applications lose performance with Turbo Boost off, disabling Turbo Boost is not an ideal solution in terms of performance even though it removes the frequency variation. In fact, newer generation processors are becoming more and more dynamic. We have shown an example of this in Section 2 where newer generation Ivy Bridge processors have a wider frequency boost range than older generation Sandy Bridge processors. Processors can take advantage of power and thermal headroom to improve performance by opportunistically adjusting their voltage and frequency based on embedded constraints. Instead of disabling these dynamic features, software and applications should be able to work well with these architectural features.

6.2. Replacing Slow Chips

In this section we analyze replacing the chips that are running at a lower frequency as a solution to mitigate the performance variation. We seek answer to the question: How many chips should we replace to get x% performance benefit? Figure 6 shows the answer for each application.

The left plot shows how average frequency of all chips changes with replacing the slow chips (i.e. not running at 2.8 GHz) with fast ones (i.e. running at 2.8 GHz) starting from the slowest. The right plot shows the percentage speedup. The speedup here is calculated by the improvement in the minimum frequency of all chips. We use minimum number here since the slowest chip will be the bottleneck in a synchronized application without dynamic load balancing.

The number of chips to replace to get a given level of performance benefit varies from application to application. MKL-DGEMM requires many more chips to be replaced compared to the other applications. Getting all of the chips running at 2.8GHz requires a significant number of the chips to be replaced and therefore is not feasible. However, replacing 50 chips, which is around 5% of the chips, would give an instant 5% speedup for MKL-DGEMM and NAIVE-DGEMM applications. There is a trade-off between the replacement cost, and the performance benefit which varies from application to application.

6.3. Leaving cores idle

The mitigation of sluggishness and variability when leaving a core idle (cf. Table 2) suggests that this could be done intentionally as a means to regain threatened performance. A chip with one or more cores idle would systematically have more head room in power consumption, heat output, and cache capacity. Experiments with a core idle in one chip per node show measured cache misses on each core that were much lower than on a fully occupied chip, which would also imply less power consumed by the memory controller (and in DRAM, though that does not presently impact CPU frequency).

This trade-off would not be generally worthwhile on Edison for CPU-bound applications optimizing for time to completion. In Figure 7, we calculate the aggregate throughput if one core from each of the chips running below the highest frequency are selectively left idle, starting from the slowest. The average frequency increases almost to the maximum frequency with selective idling. Still, the aggregate throughput, is higher for all applications when there is no idling.

We also experimented with leaving different core ID's within chips idle to see if the selection matters, but we did not observe much difference. So rather than one specific core slowing down its chip, whole chips seem to be uniformly more or less efficient.

On other systems, with different core densities, clock speeds, and prevalence of slow and variable chips, this cal-

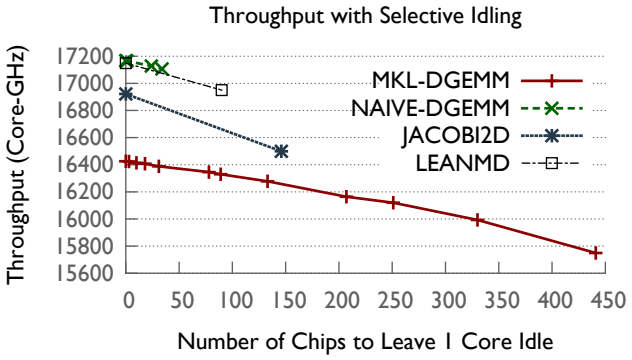

Figure 7: Throughput of all cores when 1 core is left idle from chips starting from the slowest chip.

culation could turn out differently. Given how close this trade-off is in this setting, it's worth considering situations in which the trade-off may differ. An application and problem for which the working set fits better in cache or the memory bus is less contended with one less working core might get higher performance by idling one or more cores. To consider those effects, we also calculate the throughput with real execution time instead of frequency-based throughput. In this case, MKL-DGEMM shows a small increase in the throughput, up to 0.05%, when 1 core from each of the 20 chips that get the greatest benefit is left idle. When more chips are left idle, the throughput starts falling down again. Thus, the benefit is negligible.

On a system that charged for energy consumption instead of or in addition to time, leaving cores idle can reduce total cost - especially if one can select the least efficient cores [20]. A system provisioned with more nodes than the facility can fully power and cool could be used more fully by leaving cores idle to stay under the effective power cap [27]. Reducing the power drawn by each chip would also reduce their operating temperature, potentially lowering fault rates and thus improving overall time to completion [28].

6.4. Dynamic Work Redistribution

Dynamic introspective load balancing and work stealing can fully address the observed variation at a cost of overhead and implementation complexity. The load balancing algorithms should take the CPU frequency and performance of the cores into account when distributing the work among the cores. The load balancing can be done by the application itself or by a run-time system. Moving only a small portion of the workload at run-time with an intelligent load balancing strategy can be sufficient to compensate for the performance variation; therefore, the load balancing overhead could be negligible or lower than the benefit obtained from re-balancing the application.

Algorithm 1 presents a refinement based load balancing algorithm, REFINELB, available in the CHARM++ framework [33]. In CHARM++ , the work is represented as C++ objects which can migrate from processor to processor. This algorithm moves away objects from the most overloaded processors (defined as $heavyProcs$ in algorithm 1) to the least overloaded ones ($lightProcs$) until the overloaded processors' load reaches the average load. A processor is considered overloaded if its load is more than a threshold ratio above the average load of the whole set of processors. This threshold value is typically set to 1.002. The main goal of the algorithm is to balance the load with a minimum number of objects to be migrated.

Algorithm 1 Refinement Load Balancing Algorithm

Idea: Move heaviest object to lightest processor until the processor's load reachs the average load
Input: V_o (set of objects), V_p (set of processors)
Result: Map: $V_o \rightarrow V_p$ (An object mapping)

```
1:  Heap heavyProcs = getHeavyProcs(V_p)
2:  Set lightProcs = getLightProcs(V_p)
3:  while heavyProcs.size() do
4:    donor = heavyProcs.deleteMax()
5:    while (lightProc = ligthProcs.next()) do
6:      obj, lightProc = getBestProcAndObj(donor, V_o)
7:      if (obj.load+lightProc.load<avgLoad) break
8:    end while
9:    deAssign(obj, donor)
10:   assign(obj, lightProc)
11: end while
```

Algorithm 2 Speed-Aware Refinement Algorithm

Idea: When moving objects between processors, take processor speed into account.
Input: S_p (Speed of processor p)
Replace line 7 in Algorithm 1 with the following:

```
7: if (obj.load × S_donor ÷ S_lightProc + lightProc.load < avgLoad) break
```

The processor load is calculated by past execution time information of each object on the processor and the background load collected by the CHARM++ framework. However, when moving one object from a heavy to light processor, the algorithm does not take into account speed of the processors, instead assuming processor speeds are equal. An object's load can change as a result of migration i.e can take less time when it's moved from a slow processor to a fast processor. Therefore the object's estimated load needs to be scaled with the speed of the processors. Algorithm 7 shows the necessary change in to make in the RefineLB algorithm to make it speed-aware. This scaling is done using the frequency of the processors; a more advanced method can use instructions

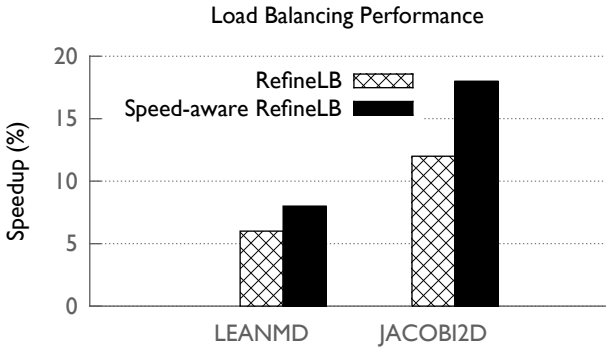

Figure 8: Speedup of RefineLB and Speed-aware RefineLB compared to no load balancing case.

per cycle or a more detailed performance model, such as a frequency-dependent Roofline [31], to get a better estimation.

In Figure 8, we show the performance of RefineLB and our speed-aware RefineLB compared to no load balancing with JACOBI2D and LEANMD applications running on 6 nodes. Note that these applications have no inherent load imbalance, so each core initially has an equal workload. Load balancing with RefineLB improves the performance by 6% in LEANMD and 12% in JACOBI2D compared to no load balancing. Moreover, speed-aware RefineLB outperforms RefineLB by 2% in LEANMD and 6% in JACOBI2D. A better load estimation with speed-awareness results in more object migration from slow chips to the fast ones compared to non-speed aware version. Load balancing has overheads of measurement, decision making and the actual migration. Since the number of migrated objects are a small portion of the total number of objects, the overhead is not too much and will be compensated after a few iterations.

To conclude, dynamic load balancing is the most feasible solution giving the best performance and it does not require any change in the machine infrastructure. We have shown how a dynamic application runtime can cope with the dynamic, unpredictable behavior of the chips.

7. Related Work

There are several published evaluations of earlier generations of Intel's Turbo Boost technology. Charles et al. show that Turbo Boost increases the performance of the applications, but it can increase the power consumption more than the performance benefit it gives [11]. Especially with memory intensive applications, the performance benefit coming from CPU frequency boost may not be significant. This means that performance per watt may not be better under Turbo Boost for all workloads. Balakrishnan et al. also show that for some benchmarks, performance per watt under Turbo Boost is worse [9]. On the other hand, Kumar et al. show [19] performance per watt is higher in most situations when compared with symmetric multi-core processors. Regardless of the conclusion of the performance per watt metric under Turbo Boost, none of these studies examine the variability caused by Turbo Boost on HPC platforms with thousands of processors working in concert.

Rountree et al. show that there is variation and hence performance degradation in applications under power capping [16, 26]. However, they do not study variation in the absence of power capping below TDP or under Turbo Boost.

Application performance on the Edison supercomputer under different CPU frequency settings has been studied before [8]. Austin et al. show energy optimal CPU frequency points for various applications. However they do not analyze CPU frequency variation in their study and only focus on fixed frequencies below nominal speed.

Variation within a multicore processor has been demonstrated by Dighe et al [13]. Langer and Totoni propose a variation aware scheduling algorithm [20, 29] with an integer linear programming approach to find the best task to core match in a simulated environment with variation. Hammouda et al. propose noise tolerant stencil algorithms to tolerate the performance variations caused by various sources including dynamic power management, cache performance and OS jitter [15].

There have been various other studies showing the thermal variation among supercomputer architectures [17, 32]. Moreover, there are various studies to mitigate the temperature variation or hot spots among cores or processors. Menon et al. demonstrate a thermal aware load balancer technique using task migration to remove the hot-spots in HPC data centers [21]. Wang et al. propose thermal aware workload scheduling technique for green data centers [30]. Choi et al. propose a thermal aware task scheduling technique to reduce the temperature of the cores within a processor [12].

To the best of our knowledge, there is no other paper which comprehensively measures and analyzes performance, frequency, temperature, and power variation among nominally equal processors under Turbo Boost at large scale.

8. Conclusion and Future Work

In this paper, we have analyzed the performance variation caused by dynamic overclocking on top supercomputing platforms. We have shown the performance degradation caused by frequency variation on math kernels and HPC applications. As we move towards exascale machines, we expect this variation to increase further. Modern processors are becoming more dynamic in order to take advantage of headroom in the operating temperature and power consumption, and can adjust their voltage and frequency based on their thermal and energy constraints. Turning off these dynamic features is not the ideal solution to mitigate the variation. We should look for ways to mitigate variation from the application software. We show a speed-aware dynamic task migration strategy to tackle this problem and show up to 18% performance improvement.

References

[1] Cab supercomputer at LLNL. https://computing.llnl.gov/tutorials/bgq/ https://computing.llnl.gov/?set=resources&page=OCF_resources#cab.

[2] Edison Supercomputer at NERSC. https://www.nersc.gov/users/computational-systems/edison/.

[3] Intel Turbo Boost Technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[4] Intel Xeon Processor E5 v2 Product Family, Specification Update. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v2-spec-update.pdf.

[5] Lenovo showcases high-performance computing innovations at supercomputing 2014. http://news.lenovo.com/article_display.cfm?article_id=1865.

[6] PAPI 5.4.1.0, Cycle Ratio. https://icl.cs.utk.edu/papi/docs/da/dab/cycle__ratio_8c_source.html.

[7] Stampede supercomputer at TACC. https://www.tacc.utexas.edu/stampede/.

[8] Brian Austin and Nicholas J Wright. Measurement and interpretation of microbenchmark and application energy use on the Cray XC30. In *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 51–59, Piscataway, NJ, USA, 2014. IEEE.

[9] Ganesh Balakrishnan. Understanding Intel Xeon 5500 Turbo Boost Technology. *How to Use Turbo Boost Technology to Your Advantage, IBM*, 2009.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[11] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel® Core i7 Turbo Boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 188–197. IEEE, 2009.

[12] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 213–218. ACM, 2007.

[13] Saurabh Dighe, Sriram R Vangal, Paolo Aseron, Shasi Kumar, Tiju Jacob, Keith A Bowman, Jason Howard, James Tschanz, Vasantha Erraguntla, Nitin Borkar, et al. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *Solid-State Circuits, IEEE Journal of*, 46(1):184–193, Jan 2011.

[14] Rong Ge, Xizhou Feng, and Kirk W Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 34–, Washington, DC, USA, 2005. IEEE.

[15] Adam Hammouda, Andrew R Siegel, and Stephen F Siegel. Noise-tolerant explicit stencil computations for nonuniform process execution rates. *ACM Trans. Parallel Comput.*, 2(1):7:1–7:33, April 2015.

[16] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 78. ACM, 2015.

[17] Laxmikant Kale, Akhil Langer, and Osman Sarood. Power-aware and Temperature Restrain Modeling for Maximizing Performance and Reliability. In *DoE Workshop on Modeling and Simulation of Exascale Systems and Applications (MODSIM)*, Seattle, Washington, August 2014.

[18] Nandini Kappiah, Vincent W Freeh, and David K Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 33–, Washington, DC, USA, 2005. IEEE.

[19] Rakesh Kumar, Keith Farkas, Norman P Jouppi, Parthasarathy Ranganathan, Dean M Tullsen, et al. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th*

Annual IEEE/ACM International Symposium on, pages 81–92. IEEE, 2003.

[20] Akhil Langer, Ehsan Totoni, Udatta S. Palekar, and Laxmikant V. Kalé. Energy-efficient computing for HPC workloads on heterogeneous manycore chips. In *Proceedings of Programming Models and Applications on Multicores and Manycores*. ACM, 2015.

[21] Harshitha Menon, Bilge Acun, Simon Garcia De Gonzalo, Osman Sarood, and Laxmikant Kalé. Thermal aware automated load balancing for HPC applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[22] National Center for Supercomputing Applications. Blue Waters project. http://www.ncsa.illinois.edu/BlueWaters/.

[23] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003.

[24] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[25] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, (2):20–27, 2012.

[26] Barry Rountree, Dong H Ahn, Bronis R de Supinski, David K Lowenthal, and Martin Schulz. Beyond DVFS: A First Look at Performance Under a Hardware-enforced Power Bound. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012.

[27] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kale. Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, New York, NY, USA, 2014. ACM.

[28] Osman Sarood, Esteban Meneses, and L. V. Kale. A 'cool' way of improving the reliability of HPC machines. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, USA, November 2013.

[29] Ehsan Totoni. *Power and Energy Management of Modern Architectures in Adaptive HPC Runtime Systems*. PhD thesis, Dept. of Computer Science, University of Illinois, 2014.

[30] Lizhe Wang, Gregor von Laszewski, Jai Dayal, and Thomas R 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. IEEE, 2009.

[31] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[32] Kaicheng Zhang, Seda Ogrenci-Memik, Gokhan Memik, Kazutomo Yoshii, Rajesh Sankaran, and Pete Beckman. Minimizing thermal variation across system components. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1139–1148. IEEE, 2015.

[33] Gengbin 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.