© 2017 by Bilge Acun. All rights reserved.

MITIGATING VARIABILITY IN HPC SYSTEMS AND APPLICATIONS FOR PERFORMANCE AND POWER EFFICIENCY

BY

BILGE ACUN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Tarek Abdelzaher
Professor Josep Torrellas
Dr. Pete Beckman, Argonne National Laboratory

Abstract

Power consumption and process variability are two important, interconnected, challenges of future generation large-scale High Performance Computing (HPC) data centers. For example, current production petaflop supercomputers consume more than 10 megawatts of machine and cooling power that costs millions of dollars every year [1]. As HPC moves towards exascale computing, these costs will increase and power consumption is expected to become a major concern. Not solely dynamic behavior of HPC applications but also dynamic behavior of HPC systems makes it challenging to optimize the performance and power efficiency of large scale applications. Dynamic behavior of applications include irregular or imbalanced applications. Dynamic behavior of HPC systems include thermal, power, and frequency variations among processors. Smart and adaptive runtime systems have great potential to handle these challenges transparently from the application.

In this dissertation, I first analyze frequency, temperature, and power variations in large-scale HPC systems using thousands of cores and different applications. After I identify the cause of each of these variations, I propose solutions to mitigate these variations to improve performance and power efficiency. When analyzing frequency variation, I attribute manufacturing related intrinsic differences in the chips' power efficiency as the culprit behind frequency variation under dynamic overclocking. I propose speed-aware dynamic load balancing strategies to mitigate the performance overhead due to frequency variation. When analyzing temperature variation, I focus on inefficiencies in fan-based air cooling systems. I propose proactive and decoupled fan control mechanisms that reduce temperature varia-

tions and reduce cooling power consumption by predicting core temperatures using a learning based model. When analyzing power variations, I identify manufacturing related sources of power variation that are static and dynamic. I propose different variation aware node assembly methods to mitigate the power variation. Finally, I propose a fine-grained runtime based technique to mitigate application level variations that are caused by the characteristics of the application itself (for example, applications with different kernel types or phases) in order to reduce the energy consumption.

To my family.

Acknowledgments

Many people have been a part of this journey:

Thank you to Prof. Laxmikant Kale, my advisor for five years, for his invaluable guidance and support.

Thank you to my commitee members Prof. Josep Torrellas, Prof. Tarek Abdelzaher and Dr. Pete Beckman for their crucial feedback on my work.

Thank you to all of the Parallel Programming Laboratory (PPL) members, past and present. In particular, I would like to thank Phil Miller for sparking the initial idea for this work which turned into my thesis. Also, among my many appreciated PPL colleagues, I would like to thank by name my first collaborators who are co-authors on publications preceding this work. They include: Nikhil Jain, Harshitha Menon, Abhishek Gupta, Osman Sarood, Ehsan Totoni, Akhil Langer, Abhinav Bhatele, Yanhua Sun, Lukasz Wesolowski, Xiang Ni, Esteban Meneses, Michael Robson, Ronak Buch and Eric Mikida.

Thank you to Yoonho Park and Eun Kyung Lee for their mentorship connected with my internship at the IBM T.J. Watson Research Center.

Thank you to Prof. Brian Lily for his encouragement and support which indirectly but significantly contributed to the thesis.

Of course, I thank all of my friends, from around twenty different amazing countries, all of whom I met here and their friendships have all made my life richer.

Finally, I would like to thank my family for their love and support. Thank you for believing in me.

Grants

This work was supported by multiple generous funding sources, and it made use of resources from several supercomputing centers. I would like to thank all of the following sources that contributed towards:

Fellowship sources: The Saburo Muroga Endowed Fellowship by the Department of Computer Science at University of Illinois at Urbana-Champaign (UIUC) provided support for academic year 2012-2013.

Assistantship sources: This work was supported by the National Institutes of Health (grant number PHS-5-P41-RR05969) and by Cisco Systems Inc. with their gift award (CG 587589).

Supercomputing centers and allocations: This work would not be possible without the Cori and Edison supercomputers at National Energy Research Scientific Computing Center (NERSC), Stampede at Texas Advanced Computing Center (TACC), Cab at Lawrence Livermore National Laboratory (LLNL), Blue Waters at UIUC, National Center for Supercomputing Applications (NCSA), and Minsky at IBM T.J. Watson Research Center.

This research used resources of NERSC, which is supported by the Office of Science of the U.S. Department of Energy (DOE) (contract number DE-AC02-05CH11231). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation (grant number OCI-1053575). This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the State of Illinois. Blue Waters

is a joint effort of UIUC and NCSA. This research also used computer time on Livermore Computing's high performance computing resources, provided under the M&IC Program. Finally, I am thankful to Prof. Tarek Abdelzaher for letting me use the testbed at UIUC for experimentation (grant number NSF CNS 09-58314).

Table of Contents

List of Figures x

List of Tables xiv

List of Algorithms xv

List of Abbreviations and Acronyms xvii

CHAPTER 1 Overview 1
- 1.1 Dissertation Organization 4

CHAPTER 2 A Dynamic Runtime Interacting with Data Center's Resource Manager 6
- 2.1 An Adaptive Runtime System for HPC 9
- 2.2 Throughput Maximization Under A Power Budget 13
- 2.3 Improving Reliability Through Temperature Restraint and Load Balancing . 16
- 2.4 Dynamic Configuration of System Components 18
- 2.5 Architecture and System Needs 20
- 2.6 Summary 21

CHAPTER 3 Analyzing Large Scale Processor Variability 22
- 3.1 Experimental Setup 23
- 3.2 Measurement and Analysis of Variation in Large Scale Systems 30
- 3.3 Temperature and/or Power as Cause of Frequency Variation 39
- 3.4 Related Work 44
- 3.5 Summary 45

CHAPTER 4 Mitigating Frequency Variation 47
- 4.1 Disable Turbo Boost 48
- 4.2 Replacing Slow Chips 49
- 4.3 Leaving cores idle 51
- 4.4 Dynamic Work Redistribution 52
- 4.5 Summary 62

CHAPTER 5 Mitigating Temperature Variation 63
5.1 Motivation and Observations 66
5.2 Neural Network-Based Temperature Prediction Model 70
5.3 Model-Guided Proactive Fan Control 75
5.4 Mitigating Intra-Chip Temperature Variation 85
5.5 Related Work 87
5.6 Summary 89

CHAPTER 6 Mitigating Across Component Power Variation 90
6.1 Understanding the Sources of the Power Variation 91
6.2 Power Variation Analysis of Node Components 93
6.3 Variation Aware Node Assembly 97
6.4 Summary 103

CHAPTER 7 Mitigating Application-Level Variability 104
7.1 Motivation 106
7.2 Runtime Guided Frequency Regulation 111
7.3 Experimental Results 120
7.4 Related Work 123
7.5 Summary 124

CHAPTER 8 Conclusion 125
8.1 Future Directions 126

REFERENCES 128

List of Figures

1.1 Figures are taken from "United States Data Center Energy Usage Report" published by the Lawrence Berkeley National Laboratory in 2016 [3]. 2

2.1 Figure shows overall system design with two major components interacting with each other: Resource Manager and the Runtime System. 7
2.2 Illustration of overdecomposition with migratable objects and message driven execution in runtime system. 10
2.3 Various components of adaptive runtime system and their interaction with the resource manager. 12
2.4 Power-aware speedups of four applications running on 20 nodes. The applications vary from being CPU intensive to memory intensive. 14
2.5 Comparison of average completion time of jobs with SLURM and PARM, in Rigid(R) and Malleable(M) variants. SetL has jobs that have low sensitivity to CPU power and SetH has jobs that have high sensitivity. 15
2.6 Reduction in execution time and change in MTBF for different temperature thresholds 17

3.1 The distribution of benchmark times on 512 nodes of each machine looping MKL-DGEMM a fixed number of times on each core. 30
3.2 Average frequency shows a negative correlation with total execution time on both Edison, Cab and Stampede when running MKL-DGEMM. 31
3.3 KNL execution time variability of 256 nodes 17408 cores. Overall variation is 53.4%. Frequency variation is almost 200 MHz. 32
3.4 Plots show the execution time and frequency of four randomly selected nodes sorted by the core and node IDs. 34
3.5 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. 35
3.6 Iteration time (top plot) and frequency (bottom plot) over iterations are shown from cores selected from 3 chips showing distinct behavior: slow, variable, and fast. 36
3.7 Different chip types naturally form clusters and can be identified looking at their percentage variability in the iteration time. 37

3.8 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. The chips' operating temperatures are not directly correlated with their frequency under the heavier load of MKL-DGEMM. 40
3.9 Processors that reach TDP drops their frequency and stay at 134 W level, just under 1 W than TDP. 512 Haswell processors in Cori platform are shown. 43
3.10 The processors that hit the power cap, the ones circled, have a wide range of temperatures similar to the ones that do not hit. Temperature does not directly correlate with power. 44

4.1 How many chips we should replace to get performance benefit? 50
4.2 Throughput of all cores when 1 core is left idle from chips starting from the slowest chip. 52
4.3 Speedup of RefineLB and Speed-aware RefineLB compared to no load balancing case. 54
4.4 Homogeneous Processors under Turbo-Boost using Speed-aware RefineLB . . 55
4.5 RefineLB and Speed-aware RefineLB performance compared to without load balancing case 56
4.6 Comparing RefineLB and Speed-aware RefineLB 59
4.7 Speed-aware RefineLB performance on Jacobi-2D with different grid sizes. . 60
4.8 Speed-aware RefineLB performance on Jacobi-2D with varying block size. Load balancer is triggered at iteration 800. 61

5.1 Neural network-based thermal prediction approach. 65
5.2 POWER8 server node architecture illustration. 66
5.3 There is an 8°C steady-state temperature variation among the cores within a node running a balanced benchmark, LeanMD. 67
5.4 Fan power grows cubic polynomially with respect to fan speed. 68
5.5 Fans show different behavior for different applications. 68
5.6 Plots show fan power (top) and core temperature (bottom) behavior from 10% to 100% CPU utilization levels. Purple and green lines in the bottom plots represent cores in Chip-1 and Chip-2 respectively. 69
5.7 (a) Mean absolute error using different back-propagation algorithms; (b) Training time using different back-propagation algorithms; (c) Mean absolute error per core using Levenberg-Marquardt algorithm; (d) Mean absolute error using different number of neurons; (e) Training time using different number of neurons; and (f) Model validation while increasing CPU frequency. Plots are shown with 95% confidence interval. 74
5.8 Power and temperature comparison of the three fan control mechanisms with *LeanMD* benchmark starting at 100s. (Preemptive cooling has been started 50s ahead of time for clarification of the plots and the idea. It can achieve a similar effect if triggered within few seconds of the application start as well.) 81

5.9 The difference between the maximum and the stable fan gives the power reduction that preemptive cooling can achieve with LeanMD. 82
5.10 Precooling should start within a few seconds of application start at the latest. 82
5.11 Steady-state temperature distribution of 1,800 cores running DGEMM kernel in two different architectures: Cori (left) with Intel Xeon Haswell processors and Minsky (right) with IBM POWER8 processors. 83
5.12 Steady-state temperature distribution of different applications on Minsky: kNeigbor, LeanMD, Stencil3D (in order). Despite their different characteristics, applications show almost the same temperature distribution when the cores are used in balance. 83
5.13 The top plot shows steady-state temperature distribution of the DGEMM benchmark with decoupled fans on Minsky. (Notice the range change in the y-axis.) Independent fan control cannot remove temperature variations fully overall, because of the intra-chip variations. 84
5.14 The distribution of intra-chip temperature variation among cores. 84
5.15 Load balancing reduces within chip temperature variations from 5°C to 2°C. (Inter-chip variation is mitigated by decoupled fans.) 86

6.1 Node architecture of Oak Ridge National Laboratory (ORNL) Summit-Dev Supercomputer with IBM Power8 CPUs, NVIDIA Tesla P100 GPUs, DDR4 memory and Mellanox EDR Infiniband network adapter. 91
6.2 Power and frequency characterization of a manufacturing yield. Note that the distribution ratios in the figure do not represent actual numbers. 92
6.3 Static (idle) power distribution of node components (notice the x axis range change for the total node distribution in bottom right). Note that some of the difference in the idle power shown in this plot is caused by diffent instructions per second (IPS) number levels in the CPUs despite being idle. However, we still observe wide power differences among processors having similar IPS values as well. 95
6.4 Idle versus active power of chips running DGEMM benchmark. 95
6.5 Power distribution of chips running different benchmarks. 96
6.6 Temperature and power correlation of the chips. 96
6.7 Illustration of node assembly types 97
6.8 Illustration of node assembly Type-1: Categorized assembly 98
6.9 Distribution of the active power of node components: CPU, GPU, Memory (in order) fit to Gaussian distribution. 101
6.10 Power reduction with Type-1 node assembly compared to random assembly at different data center loads with a size of 5,000 nodes. The nodes that are not active are assumed to be turned-off. 101
6.11 Illustration of node assembly Type-2: Application characteristics aware assembly 102
6.12 Illustration of node assembly Type-3: Balanced power node assembly 102

7.1 Core level Dynamic Voltage and Frequency Scaling (DVFS) on Haswell architecture shows proportional/linear decrease in power when core frequencies are dropped one by one. On the other hand, since Sandy Bridge do not have per core voltage regulators, all core frequencies needs to be dropped together to for a a reduction in power and temperature. 107
7.2 Core level frequency scaling on IBM Power8 chip. 108
7.3 Timeline of two processors running the OpenATOM benchmarks. Each color represents a Charm++ entry method. Notice how different entry methods are executed on the two processes at the same time range. 111
7.4 Code snippet from Charm++ stencil application. 113
7.5 Runtime control flow. 114
7.6 Optimal frequency of DGEMM kernel remains more or less stable despite the number of active cores running the kernel, wheres optimal frequency of the MEMOPS kernel drops significantly as more cores are activated. . . . 116
7.7 Optimal frequency of the MEMOPS kernel depends more on the number of active cores than the data size. 116
7.8 Timeline of two kernels executing one another where the runtime applies optimal frequency. 118
7.9 Timeline of the synthetic benchmark having two kernels (represented by light blue and dark blue colors) randomly overlapping. Each line represents a process, in this case four processes are running in parallel. 118
7.10 Plots shows how much energy can be reduced if a kernel that has an energy optimal frequency of 2.3 GHz is transitioned from different frequency levels. Different lines represent different kernel durations. 120
7.11 **Per-core:** Uses energy optimal frequency for each kernel in core level. **Per-chip-K1:** Uses per-chip DVFS with optimal frequency of kernel-1 which is 2.2 GHz, **Per-chip-K2:** Uses per-chip DVFS with optimal frequency of kernel-2 which is 3.3 GHz, **Per-chip-K1&2:** Uses per-chip DVFS with optimal frequencies of kernel 1 and 2 running together which is 2.7 GHz. 122

List of Tables

2.1 Desired access to hardware-level measurement and controls. ✓: There is support and access. ✗: There is no support. ◯: Hardware has support, but the system's software lacks support or forbids access. 20

3.1 Platform hardware and software details 29
3.2 Distribution of observed steady-state frequencies of 1K Chips on Edison . . . 38
3.3 Frequency distribution of MKL-DGEMM on Cab 38
3.4 Frequency distribution of MKL-DGEMM on Stampede 38
3.5 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 or close to their limiting power while the fast chips are not. 41

4.1 Percentage slowdown of applications when the frequency is fixed at maximum frequency of 2.4GHz 48
4.2 Platform hardware details 56
4.3 Object Migration with Different Load Balancers 57

5.1 Peak fan power and energy consumption of different fan control mechanisms and benchmarks. 77
5.2 Decoupling the left two and the right two fans reduces chip-to-chip temperature variations, power, and energy consumption. 79
5.3 Fan Power Reduction in Large Scale 79

7.1 Platform hardware and software details 106

List of Algorithms

1 Refinement Load Balancing Algorithm 53

2 Speed-Aware Refinement Algorithm 53

List of Abbreviations and Acronyms

CMOS Complementary Metal-Oxide-Semiconductor. 22, 64, 91

DOE Department of Energy. vi, 1, 6, 63

DVFS Dynamic Voltage and Frequency Scaling. xiii, 3, 85, 88, 104–111, 117, 120, 122–124

FIVR Fully Integrated Voltage Regulator. 106

HPC High Performance Computing. ii, viii, 1, 3, 4, 6–9, 13, 16, 18, 19, 21–23, 26, 32, 44, 45, 47, 85, 87, 88, 90, 105, 125, 127

ILP Integer Linear Program. 14

LBM Load Balancing Module. 11

LLNL Lawrence Livermore National Laboratory. vi, 3, 24

LM Local Manager. 11, 19

NCSA National Center for Supercomputing Applications. vi, vii, 24

NERSC National Energy Research Scientific Computing Center. vi, 23, 24, 42

ORNL Oak Ridge National Laboratory. xii, 3, 63, 90, 91

PARM Power Aware Resource Manager. 13, 14

PRM Power Resiliency Module. 11, 12, 15, 17, 18

RAPL Running Average Power Limit. 13, 41, 104

RM Resource Manager. 19

RTS Runtime System. 11, 47, 70

TACC Texas Advanced Computing Center. vi, 24

TDP Thermal Design Power. 13, 29, 41, 42, 91

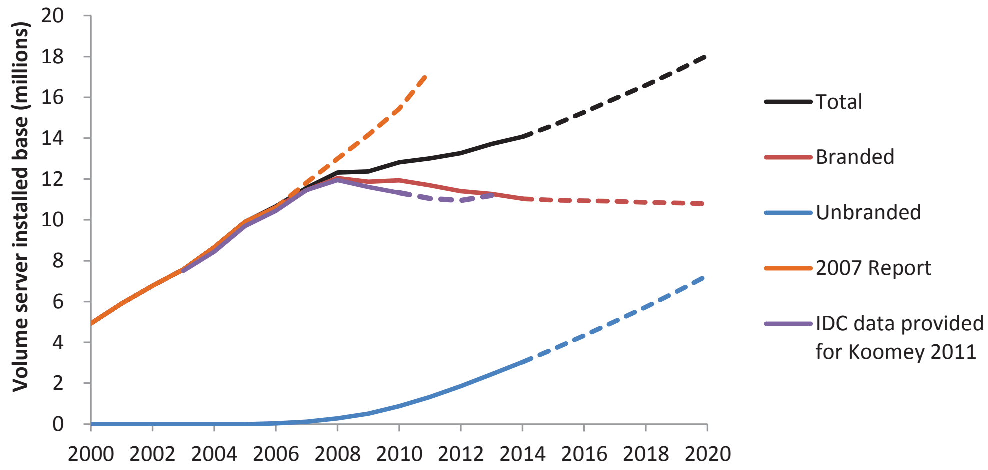
UIUC University of Illinois at Urbana-Champaign. vi, vii, 1, 56

CHAPTER 1

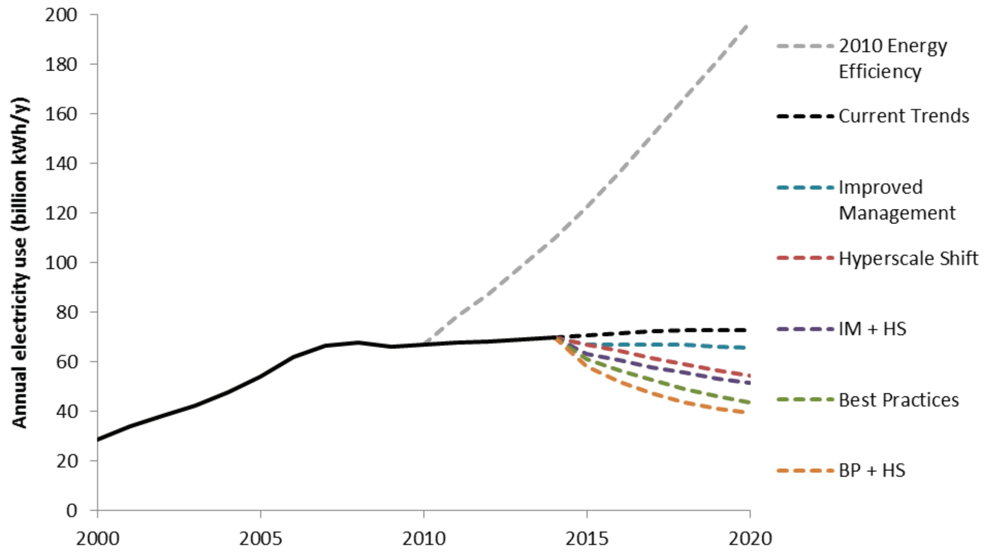
Overview

Just as no two people are alike, the same can be said for processors. Just as an ideal relationship should accommodate for differences, an ideal system should accommodate the same at scale.

As the size of the data centers and HPC centers keeps growing, power consumption grows as well. Current petascale systems, such as Blue Waters supercomputer at UIUC, consume tens of megawatts of power leading to millions of dollars in energy bills, significant power strains on local, state, or regional energy grid systems, and the environmental impact on natural resources that provide the power for the supercomputer. As the scale has been growing rapidly over the last few decades, the extrapolations of the trends has shown that each data-center may need their own nuclear power plants for operational energy needs in near future. This has made scientists and engineers take action to improve power and energy efficiency of data centers. For example, the DOE set a 20 MW power goal for an exascale system to be built [2]. Fortunately, the efforts to improve the design and operation of the data centers have been paying off. A 2016 report by Lawrence Berkeley National Laboratory shows that the total energy consumption of United States data centers has been flat-lined despite the increased volume of installed servers [3]. As shown in Figure 1.1, the predictions show that the energy usage may remain constant until 2020, as the in-efficient practices are replaced with more efficient technologies. These efficient practices include not only hardware related advancements, such as optimizing the idle power consumption



(a) Total Volume Server Installed Base Estimates



(b) Projected Data Center Total Electricity Use

Figure 1.1: Figures are taken from "United States Data Center Energy Usage Report" published by the Lawrence Berkeley National Laboratory in 2016 [3].

of the processors, but also improved operations of the data center, such as increasing the average utilization of the servers. However, for beyond 2020, the report cautions that further technological advancements are necessary to ensure that the energy demand does not grow at the rate proportional to the growth of the data centers as once it used to be in early 2000s.

It is important to note that the majority (as much as 80%) of the data center electricity is spent by servers and infrastructure divided equally among them [3]. Servers and infrastructure costs, which includes cooling, are the focus of this dissertation. The rest of the electricity is consumed by network and storage, which is about less than 20% [3].

It is also important to note the motivations behind reducing the energy versus power consumption which may not necessarily be the same. Some supercomputing facilities, such as LLNL, only pay energy charge per kWh, despite they have a contracted maximum power capacity [4]. Therefore, reducing the energy consumption directly reduces the costs. On the other hand, some other facilities, such as ORNL that hosts the largest supercomputer in the United States – Titan, is charged based on its maximum power usage [4]. Moreover, power line infrastructure of the data center can have a maximum draw limit. Therefore, reducing the maximum power usage or increasing the throughput under a strict power budget can be a motivation to reduce the costs. Hence, power and energy efficient system design have both their own merits.

This dissertation proposes novel software and hardware techniques to improve power and energy efficiency of data centers specifically by mitigating various kinds of variability in processors. A unified model where the runtime dynamically interacts with the data center's resource manager is also introduced for this purpose. DVFS and power capping are common approaches to reduce the energy consumption or power consumption. Naive usage of these methods can lead to performance degradation which is unfavorable by HPC users. Not only the dynamic behavior of HPC applications (such as irregular or imbalanced applications) but also the dynamic behavior of HPC systems (such as thermal, power variations among processors) can cause high overhead when DVFS, and power capping methods are applied naively. Smart and dynamic runtime systems have great potential handling these challenges and improving performance and power efficiency transparent from the application under user

or administrator supplied constraints.

To understand the system variability, first, this dissertation gives a detailed analysis and identification the sources of temperature, power, and frequency variation in large-scale HPC systems. Then, novel ways to reduce the different types of variation are proposed to improve performance, power, and energy efficiency with minimal or no performance overhead.

1.1 Dissertation Organization

The dissertation is organized as follows:

Chapter 2 gives a high level overview of a data center system design where the resource manager interacts with the runtime of the application dynamically to do performance, power and energy optimizations. This chapter contains materials from my co-authored article "Power, Reliability, and Performance: One System to Rule them All" published in the IEEE Computer journal [5].

Chapter 3 gives a detailed analysis of frequency, power and temperature at large-scale using top supercomputers. It also shows how manufacturing related variations cause performance variations if the processors are running under dynamic-overclocking. This chapter revises and adds upon my paper "Variation Among Processors Under Turbo Boost in HPC Systems" published in International Conference on Supercomputing [6].

Chapter 4 provides evaluations of different solutions to mitigate the frequency variation. Frequency variation can degrade the performance of tightly coupled HPC applications. Solutions to mitigate the performance degradation include: disabling Turbo Boost, replacing slow chips, idling cores, and dynamic task redistribution. This chapter uses data from my articles "Variation Among Processors Under Turbo Boost in HPC Systems" published in International Conference on Supercomputing [6] and "Mitigating Processor Variation through Dynamic Load Balancing" published in IEEE International Parallel and Distributed Processing Symposium Workshops [7].

Chapter 5 focuses on temperature variation. First, it provides a temperature prediction model using neural networks. Then, it gives solutions to mitigate temperature variation with the help of the prediction model. The solutions include: a proactive fan control mechanism and a model-guided temperature balancing algorithm. This chapter includes work from my paper "Support for Power Efficient Proactive Cooling Mechanisms" published in IEEE International Conference on High Performance Computing, Data and Analytics [8] as well as my paper "Neural Network-Based Task Scheduling with Preemptive Fan Control" published in International Workshop on Energy Efficient Supercomputing [9].

Chapter 6 proposes techniques address power variation. Two techniques are analyzed to mitigate the power variation. These are a power-variation aware node assembly method and a variation-aware job scheduler that accompanies it. The ideas proposed in this chapter are patent pending [10].

Chapter 7 addresses application related variations during the execution. Unstructured or irregular application behavior contributes to the variability in the system. In this section, we propose and evaluate a runtime based function-level optimization approach to do fine-grained optimizations.

Finally, **Chapter 8** summarizes the main contributions and provides a discussion of future research.

CHAPTER 2

A Dynamic Runtime Interacting with Data Center's Resource Manager

DOE set a 20 MW power budget for an exascale supercomputer to be built in the next few years [2]. It is anticipated that such a system will face major challenges with reliability, power management, and thermal variations. We believe smart runtime systems have a great potential in overcoming the barriers toward exascale computing.

This chapter gives the high-level design of a unified adaptive runtime system where different modules such as job scheduler, resource manager and the job runtime system interact with each other to optimize for performance and power consumption under user or administrator supplied constrains in an environment with system failures.

Traditionally, the emphasis of HPC data centers and applications has been on performance. However, it is anticipated that future generation supercomputing systems will face major challenges in reliability, power management, thermal variations. Disruptive solutions are required to optimize performance in the presence of these challenges. For each job, a smart parallel runtime system that interacts with the whole machine's resource manager is key to overcome the challenges of next generation supercomputing data centers. In the past, it has been demonstrated that a smart and adaptive runtime system can:

- improve efficiency in a power-constrained environment [11],
- increase performance with load balancing algorithms [12, 13],

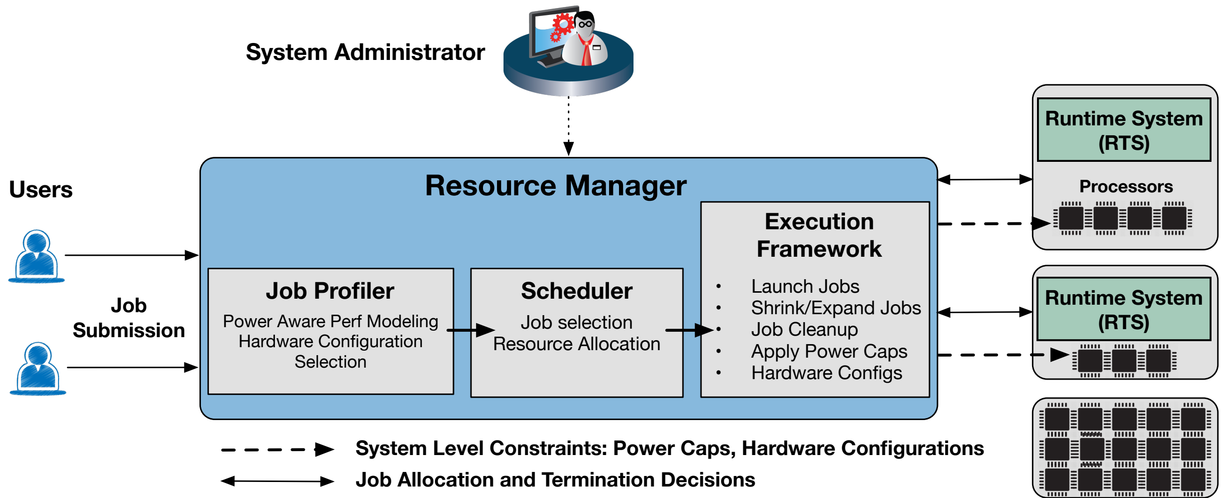


Figure 2.1: Figure shows overall system design with two major components interacting with each other: Resource Manager and the Runtime System.

- control the reliability of supercomputers with substantial thermal variations [14],
- configure hardware components to save power [15, 16].

Although these research directions were developed in isolation, they indicate that smart runtime systems have a great potential to overcome barriers towards exascale computing. What the HPC community lacks is an integrated solution that combines past research into a single system that optimizes across multiple dimensions. We propose a comprehensive design in which the data center resource manager dynamically interacts with the individual runtime systems of jobs to optimize performance and power consumption in an environment with system failures under constraints supplied by users or administrators.

At the heart of the proposed solution for power-efficiency, reliability and performance of a HPC data center lies an adaptive and dynamic parallel runtime system. An adaptive runtime system can migrate tasks and data from one processor to any other processor allocated to the job. This ability can solve many challenges that upcoming supercomputers face - application load imbalance across processors, high fault rates, power and energy constraints, and thermal variations. These challenges are often contradictory in terms of their requirements. For example, applying power and temperature constraints can compromise performance and lead to load imbalance across processors. We strive to achieve a healthy balance where we

try to maximize performance in presence of the known as well as contingent constraints and events.

In Figure 2.1, we show a unified diagram of several important components of a data center, their functions and interactions with each other in order to address the challenges of power, reliability and performance. Currently, a data center user is primarily concerned the performance of his or her job. In the future, however, power consumption of their jobs will become a major concern. On the other hand, data center administrators have different and more complex concerns - while they want to guarantee good performance to individual jobs, they need to ensure that the total power consumption of the data center does not exceed its allocated budget and that the job throughput of the data center remains high despite node failures and thermal variations. We achieve the objectives of both the user and the system administrators by allowing dynamic interaction between the system resource manager or scheduler and the job runtime system. While the job scheduler ensures that at any time the system resources are optimally allocated to the jobs based on their power and performance characteristics, the job runtime system implements the decision of the job scheduler by being malleable to shrink or expand itself to the nodes assigned by the scheduler and by doing dynamic load balancing whenever beneficial.

Furthermore, the runtime system can turn on/off or reconfigure various hardware components without impacting application performance, if adequate hardware control is provided by vendors. Our evaluations demonstrate that these runtime capabilities result in greater power efficiency for common HPC applications.

For the rest of this chapter, we first give background information on an adaptive runtime system that serves perfectly for our design needs in Section 2.1. Then, we summarize and demonstrate the past work that fits in the context of dynamic runtime that interacts with data center's resource manager in Sections 2.2, 2.3, 2.4. Finally, in Section 2.5, we discuss the architecture and system needs that are required to enable our design.

2.1 An Adaptive Runtime System for HPC

An adaptive runtime is an essential component of a system optimized for power efficiency, reliability and performance. Adaptive runtime systems enable dynamic collection of performance data, dynamic task migration (load balancing), temperature restraint and power capping with optimal performance.

Charm++ is a C++ based parallel programming framework supported by an adaptive runtime system, which enhances user productivity and allows programs to run portably from small multicore computers (e.g., laptops and phones) to the largest supercomputers [17]. It enables users to easily expose and express much of the parallelism in their algorithms while automating many of the requirements for high performance and scalability. Charm++ has been in production use for over fifteen years and it has thousands of users across a wide variety of computing disciplines with multiple large scale applications including: NAMD for molecular dynamics, ChaNGa for cosmology and OpenAtom for quantum chemistry simulations, and many others [17].

Charm++ has three main attributes: over-decomposition, asynchronous message-driven execution, and migratability. Over-decomposition entails dividing the computation in an application into small work and data units so that there are many more such units than the number of processors. Message-driven execution involves scheduling work units based on when a message is received for them. Migratability refers to the ability to move data and work units between processors. These attributes enable the Charm++ adaptive runtime system to provide many useful features including dynamic load balancing, fault tolerance, and job malleability.

An application written in Charm++ contains a collection of parallel objects that are distributed among processors and communicate via messages. These objects form the basic unit of computation and data that can be assigned and re-assigned to processors. The programmer over-decomposes the problem into many more objects than the number of processors. The Charm++ runtime system handles the assignment of these objects to processors, and it may dynamically migrate objects to balance the load, handle faults, or to shrink-expand the number of processors the application is running on.

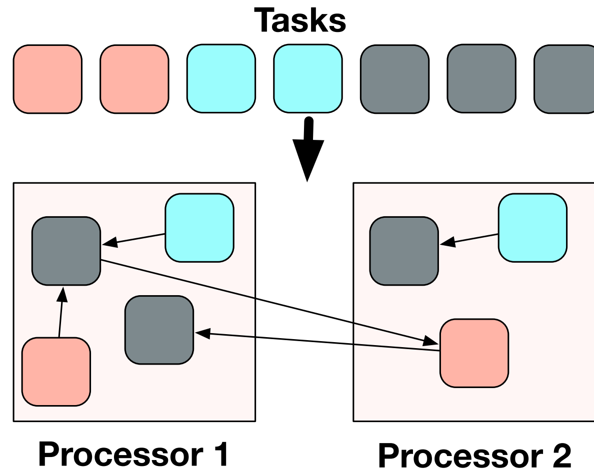


Figure 2.2: Illustration of overdecomposition with migratable objects and message driven execution in runtime system.

Charm++ collects information about the application and the system in a distributed database including load of processors, load of each object, communication pattern, and core temperatures. When the number of processors are large, then the centralized data collection becomes a performance bottleneck. Therefore, data collection and decision making are done in a hierarchical fashion. This information is used by different modules of the adaptive runtime to make decisions such as improving load balance, handling faults, and enforcing power constraints.

Load balancing: Charm++ uses a measurement-based mechanism for load balancing. It relies on a heuristic known as *the principle of persistence*, which states that for over-decomposed iterative applications, the computation load and the communication pattern of tasks or objects tend to persist over time. It uses the load statistics of the application code collected by the runtime system. This has the advantage that it provides an automatic, application independent way of obtaining the load statistics without any input from the user. Using the load statistics, Charm++ executes a chosen load balancing strategy to determine a mapping of objects to processors and then carries out migrations based on this mapping. Charm++ consists of a suite of load balancers including several centralized, distributed and hierarchical strategies. The runtime system can also automate the decision of when to call the load balancer [18]. It can use the instrumented load information to predict the future load

and make load balancing decisions. It automatically triggers load balancing when imbalance is detected and when the benefit of load balancing is more than its overhead.

Fault tolerance: The Charm++ runtime system implements both *proactive* and *reactive* strategies for reliability [19]. In the proactive techniques, the runtime system evacuates all objects from a node that a monitoring system predicts is going to crash soon. Since failure prediction is not completely accurate, the reactive techniques recover the information lost after a failure brings down one node of the system. Those latter strategies are mostly based on checkpoint and restart. Therefore, the global application's state is routinely stored and recovery implies retrieving a prior global state.

Shrink-expand: The migratability of Charm++ objects enables a unique ability called *job malleability*; during runtime, a job can shrink (decrease) or expand (increase) the number of nodes it is running on. This feature does not require any additional code from the application developer [20]. Shrink or expand operations can be triggered by an external command or it could be an internal decision made by the runtime. During a shrink operation, the runtime system reduces the number of processors that the application is running on. First, it moves the objects away from the processors that are not going to be used anymore. The unused processors can then be returned back to the resource manager. For an expand operation, the runtime launches new processes on the additional processors that the resource manager has allocated and distributes objects from current processors to the newly allocated processors.

Figure 2.3 shows the internal components and functioning of the Charm++ Runtime System (RTS). There are three important components of the RTS - Local Manager (LM), Load Balancing Module (LBM), and Power Resiliency Module (PRM). Each processor has an LM that is responsible for managing the objects residing on that processor and for interacting with other components of the RTS. The LM of each processor periodically sends its total compute load and compute load of each of its objects to the LBM, and the CPU temperature is sent to the PRM. LBM makes the load balancing decisions using MetaBalancer and it also redistributes load in response to shrink-expand commands from the resource manager. Object migration decisions are communicated to the respective LM by the LBM. PRM, on the other hand, is responsible for ensuring that the CPU temperatures remain below the

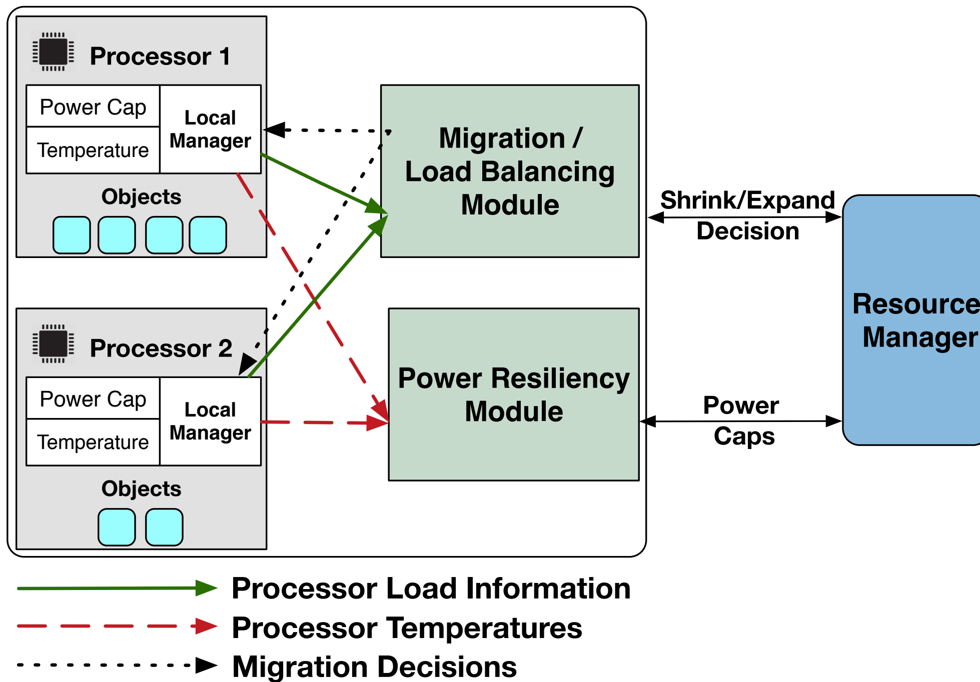


Figure 2.3: Various components of adaptive runtime system and their interaction with the resource manager.

job specific temperature threshold. The PRM controls CPU temperature by adjusting the power cap of the CPU. When a processor's temperature is above the threshold, then its power cap is lowered. And when the temperature is well below the desired threshold, then the corresponding power cap is increased while ensuring that the total power of the job remains below the power budget allocated to the job (the determination of this budget is described in the next section). Jobs may not have administrator rights to constrain the power consumption of their CPUs. Therefore, the new power caps are communicated to the resource manager which applies them to each CPU.

2.2 Throughput Maximization Under A Power Budget

Recent advances in processor hardware design allow users to control the amount of power consumed by the processor using software with Running Average Power Limit (RAPL) driver [21]. Processors can be power capped to run below their Thermal Design Power (TDP) value, where TDP is the maximum amount of power a processor can consume. The maximum number of nodes in a data center with a power budget is determined by the TDP of the nodes. Power capping makes it possible to control the power consumption of nodes and thus have additional nodes while remaining within the power budget of the data center. This is called an *overprovisioned* system [22]. Earlier research shows that an increase in the power allocated to a processor does not yield a proportional increase in job's performance [11]. Different jobs react differently to an increase in power allocated to the CPU. The idiosyncrasies in jobs performance based on allocated CPU power, points to the possibility of running different applications at different power levels. Overprovisioned systems can significantly improve performance of applications that are not sensitive to CPU power by capping CPU powers to values below their TDP and adding more nodes to get benefits from strong scaling. The Power Aware Resource Manager (PARM) [11] leverages this capability by optimally distributing the available resources to the jobs - the total power budget of the data center and the compute nodes.

The response of an application to CPU power can be captured by its power-aware speedup. The *power-aware speedup* is the ratio of the execution time of a job running on a CPU capped at a certain power level compared to the execution time of the same job when running on the lowest allowed power level allowed by the CPU [11]. A higher value for power-aware speedup implies that the application is sensitive to changes in the amount of power allocated to the CPU.

Figure 2.4 shows *power-aware speedups* of four HPC applications having different characteristics under different CPU power caps [11]. LeanMD, which is a molecular dynamics application has the highest power-aware speedup since it is the most CPU intensive one. Whereas Jacobi2D, which is a stencil application, has the lowest since it is memory intensive. PARM makes scheduling decisions by selecting jobs and their resource configurations

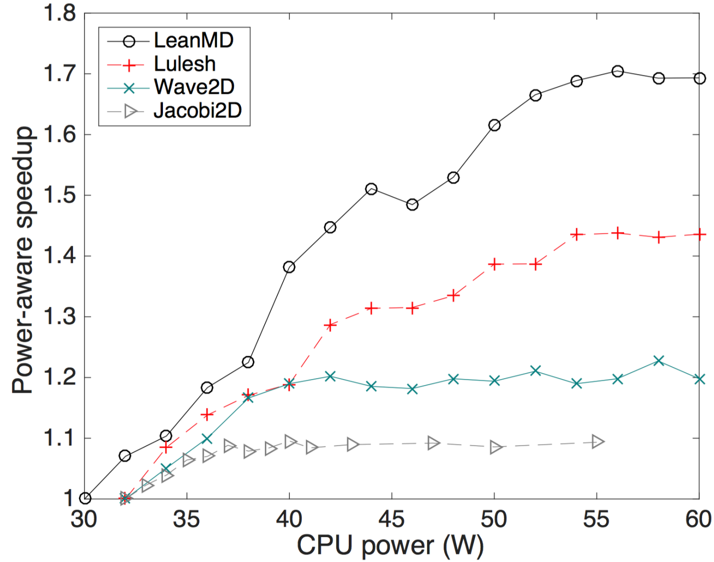


Figure 2.4: Power-aware speedups of four applications running on 20 nodes. The applications vary from being CPU intensive to memory intensive.

(e.g., power budget and compute nodes) such that the total power-aware speedup of running jobs is maximized.

PARM is an essential part of the overall system we propose in this work. It dynamically interacts with the adaptive runtime system of jobs, the system hardware, the user and the system administrator to perform several critical tasks (Figure 2.1). There are three important components of PARM:

- Job Profiler: Before a job is added to the scheduler queue, it is profiled to develop a power-aware strong scaling model that is used to calculate the power aware speedups. This profiling mechanism has negligible overhead as it is sufficient to run the application for a few iterations to get the necessary data points.

- Scheduler: PARM implements its resource allocation optimization strategy as an Integer Linear Program (ILP) with the objective of maximizing power-aware speedup of running jobs under power constraints. Whenever a new job arrives or a running job terminates, PARM's scheduler is triggered, and re-optimizes scheduling and resource allocation decisions. PARM's ILP is fast enough to run frequently with negligible overhead.

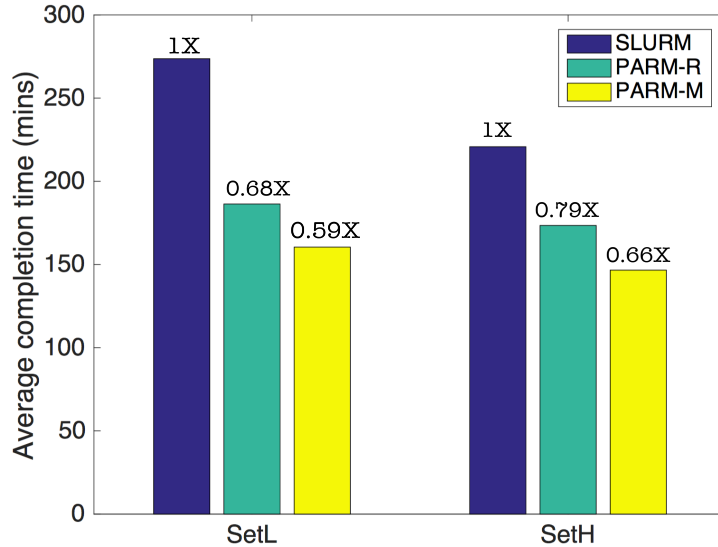


Figure 2.5: Comparison of average completion time of jobs with SLURM and PARM, in Rigid(R) and Malleable(M) variants. SetL has jobs that have low sensitivity to CPU power and SetH has jobs that have high sensitivity.

- Execution Framework: This component implements the scheduler decisions by launching jobs, sending shrink or expand decisions to the runtime system of the jobs, and by applying power caps on compute nodes. Job runtime systems interact with the execution framework to convey job termination, completion of shrink or expand operation and any changes to CPU power caps as determined by PRM module of the runtime system.

Figure 2.5 shows the benefits of using PARM as compared to power-unaware SLURM which is an open source resource manager used in many supercomputers. Two versions of PARM are compared - PARM-Rigid and PARM-Malleable. In PARM-Rigid, node allocation decision to any job is rigid, that is it cannot be changed once the job starts running. PARM-Malleable, on the other hand, has an additional degree of freedom that allows it to change the nodes allocated to a running job which is made possible by the shrink and expand feature of Charm++. The number at the top of each bar in Figure 2.5 represents average completion time as a percentage of the average completion time using SLURM scheduler. PARM-Malleable was able to reduce average completion time of jobs by up to 41%.

2.3 Improving Reliability Through Temperature Restraint and Load Balancing

Checkpoint and restart is the most popular mechanism to provide fault tolerance in HPC. The total execution time T of an application, on an unreliable system, is given by the equation:

$$T = T_{solve} + T_{checkpoint} + T_{recover} + T_{restart}$$

where T_{solve} represents the total effort required to solve the problem; $T_{checkpoint}$ accumulates all the time spent on saving the checkpoints of the system; $T_{recover}$ stands for the total work that is lost and must be recovered as a result of failures in the system; $T_{restart}$ is usually constant and represents the amount of time required to resume execution after a crash. A system using checkpoint and restart has to choose an appropriate checkpoint period (denoted by τ). There is a delicate balance in the value of τ. A long value of τ (low checkpoint frequency) decreases $T_{checkpoint}$, but may increase $T_{recover}$. Conversely, a short value of τ (high checkpoint frequency) means a reduced $T_{recover}$, but may enlarge $T_{checkpoint}$. The optimum value of τ strongly depends on the mean-time-between-failures (MTBF) of the system.

The MTBF of an electronic component is directly affected by its temperature. That relation is usually exponential and there is some experimental evidence that a $10°C$ increase on a processor's temperature decreases its MTBF in half [14]. Therefore, the reliability of a system can be controlled by restraining the temperature of its components. The cooler the system runs, the more reliable it is, but the slower it runs. That is because temperature constraints are realized by restraining the power of the CPU. The runtime allows each core to work at the maximum possible power as long as it is within the maximum temperature threshold. If any of the cores goes above the maximum temperature threshold, then their power is further reduced causing its temperature to fall. However, this can cause a performance degradation for tightly coupled applications due to thermal variations. The LBM will automatically detect any load imbalance and will make the load balancing decisions [12, 13].

The runtime system must strike a balance in the temperature at which each component should be restrained. Moreover, that balance depends on the application. Different codes

generate different thermal profiles on the system at different stages of the application. Some codes are more computationally intensive and tend to heat-up the processors more quickly. Appropriate application-based temperature thresholds are stored as part of the Job Profiler in Figure 2.1. In the end, the runtime system aims at reducing the total execution time of an application, considering the MTBF of the system and subject to the power limitations [14].

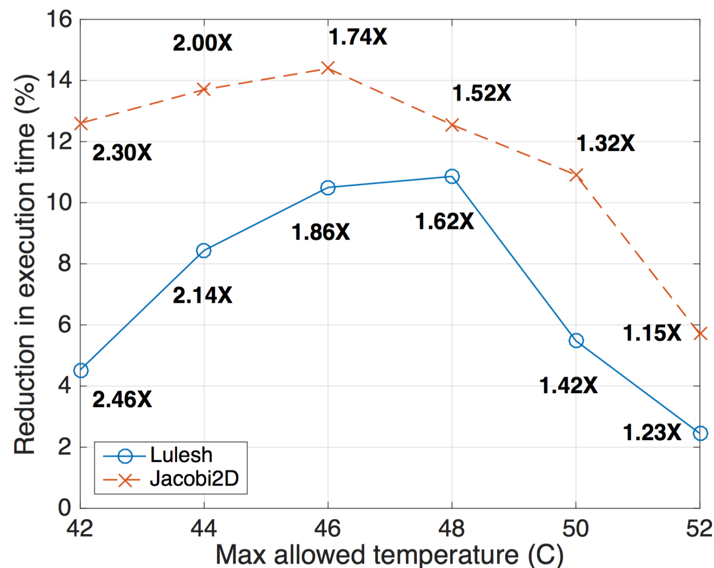


Figure 2.6: Reduction in execution time and change in MTBF for different temperature thresholds

Figure 2.6 shows percentage reduction in execution time after constraining core temperatures to different thresholds for two different applications. The reduction in execution time shown in Figure 2.6 is calculated compared to the baseline case where processor temperature is not constrained. Figure 2.6 also shows the ratio of MTBF for the machine using our scheme relative to the baseline case where core temperatures are not constrained. For example, by restraining core temperatures to 42°C in case of Jacobi2D, the MTBF for the machine increased 2.3 times while the execution time reduced by 12% compared to the baseline case where core temperatures are not constrained. The inverted U-shape of both of the curves strongly suggests a trade-off between reliability (MTBF) and the slowdown induced by the temperature restraint.

The resource manager sends the PRM of the runtime system (Figure 2.3) the upper bounds of the temperatures that honor the power envelope of the system. Those temperature values

are used as input to an internal *resilience* component in PRM and they will be changed according to the algorithms that optimize performance and consider the MTBF of the system and the characteristics of the application running. The output will be propagated to further components in PRM that will later consolidate the final power limits and they will be communicated back to the resource manager. A dynamic runtime system is fundamental in controlling the reliability of the system and honoring the power envelope at the same time. Since thermal variations are dynamic, a reactive runtime system efficiently responds to those changes and provides a healthy balance between performance and reliability in the system.

2.4 Dynamic Configuration of System Components

The runtime system can take advantage of the currently available hardware "knobs" for controlling power such as frequency scaling and power capping. However, greater power savings are possible if there is more runtime control over hardware components. We demonstrate, via cycle accurate simulations, that the runtime system can turn-off or reconfigure many components without significant performance penalty based on the properties of the running HPC application.

HPC systems should ideally be *energy proportional*; the hardware components should consume power and energy only when their functionality is being used. However, network links are always "on", independent of their utilization. In addition, processor caches consume large amounts of power, even when they are not improving the performance of the running application. We propose a runtime system approach that can save this wasted energy by dynamically re-configuring the hardware based on the application needs.

Caches consume up to 40% of a processor's power [15]. A large fraction of cache power consumption can be saved by turning-off some cache banks in cases where the application performance would not be degraded. Many common HPC applications cannot take advantage of the caches effectively. For example, molecular dynamics applications typically have small working data sets and do not need the large last level caches (LLC). On the other hand, grid-based physical simulation applications typically have very large data sets that do not fit in caches and the data reuse in cache is minimal. However, the hardware is not

able to predict the application behavior. Therefore, we propose a runtime system approach where the runtime uses profiling data to reconfigure the cache to save power without significant performance loss. Using a set of representative HPC applications, our previous study demonstrates that on average, 67% of cache energy can be saved with only 2.4% performance penalty [15].

A similar approach applies to HPC networks as well. Networks consume up to 30% of system power even when there is no communication since the links are always on. Our previous study demonstrates that typical HPC applications do not use a large fraction of the links in most of their execution time [16, 23]. The reason is that HPC topologies such as Dragonfly are designed to handle the most challenging communication patterns such as all-to-alls in FFT modules. However, typical applications have sparse communication patterns such as nearest neighbor that cannot exploit the massive number of links in HPC networks like Dragonfly. We propose using the runtime system to turn the links on and off adaptively. This hardware configuration case is harder to handle since the usage of network links can be impacted by features such as adaptive routing. Therefore, the runtime system should handle different hardware designs based on their exact specification. Our results demonstrate that up to 80% of the power consumption of network links can be saved using our adaptive runtime strategy [16].

In our unified design, the best hardware configuration is determined by the Job Profiler in Resource Manager (RM) before running each job. Using this information, LM is responsible for applying the configuration. Power models are built using the best configuration as well. As a hypothetical example, RM might know that turning-off half of LLC does not affect performance, but it reduces the maximum power from 70W to 60W. Hence, 60W is used by RM for making scheduling decisions instead of 70W. Tighter incorporation of power-performance trade-offs of hardware configuration in scheduling decisions of the resource manager is the subject of future work.

Table 2.1: Desired access to hardware-level measurement and controls.
✓: There is support and access. ✗: There is no support.
◯: Hardware has support, but the system's software lacks support or forbids access.

Need \ Platform	Cori	Edison	Cab	Stampede
Measurements				
Frequency Data	✓	✓	✓	✓
Temperature Data	✓	✓	◯	◯
Node Level Power Data	✓	✓	◯	◯
Chip Level Power Data	✓	◯	◯	◯
Core Level Power Data	✗	✗	✗	✗
Controls				
Application-level Frequency Scaling	✓	✓	◯	◯
Per-chip Frequency Scaling	◯	◯	◯	◯
Per-chip Power Capping	◯	◯	◯	◯
Per-core Frequency Scaling	◯	✗	✗	✗

2.5 Architecture and System Needs

As supercomputing platforms are becoming more heterogeneous with thousands of processors, forecasts predict challenges in power, energy consumption and process variation in the future. Therefore, it is important that applications, or runtime systems underneath the applications, be aware of the characteristics of the undelying architecture and do necessary optimization to reduce the power, energy consumption and mitigate the effect of performance variations.

Many supercomputing platforms do not give users access to power or temperature measurements or rights to control the frequency of the processors or to 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 power and energy consumption, application performance.

Table 2.1 summarizes the support for the desired user access to hardware-level measurement and controls in four top supercomputing platforms that are used in this dissertation. Cori plaform provides the greatest support among the four platforms. It provides all of the measurement data and as well as application-level frequency control. Such access avail-

able to all end-users without restriction is rare among supercomputing platforms. Edison 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 – CPU-level power measurements are necessary. Edison also provides control of frequency and power the the level of job allocations, i.e., all nodes participating in a job have the same frequency or power settings for the duration of the job. This also is too coarse-grain. Given the variation we observe among chips, every chip can have a different optimal power and frequency setting. Therefore, dynamic 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 or power control mechanisms.

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

2.6 Summary

Important challenges, such as power, reliability, and thermal variations, loom in the future of supercomputing. Addressing these concerns is imperative to harness the next generation of high performance machines. We propose a unified system design with a smart runtime which interacts with the system resource manager.

The combined system offers several important features. First, it honors the power constraints by wisely scheduling jobs and re-allocating their resources when utilization changes. Second, it controls the reliability by a temperature-aware module that cools down the system to an application-based optimal level. Third, it can re-configure the hardware via the runtime without sacrificing performance.

CHAPTER 3

Analyzing Large Scale Processor Variability

Heterogeneity in supercomputer architectures is often predicted as a characteristic of future exascale machines with non-uniform processors. For example, this could include 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 Complementary Metal-Oxide-Semiconductor (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 costs prevent 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 [24].

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 [25]).

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 work is to measure and to characterize the sources of variation, and to propose solutions to mitigate their effects.

The main contributions of this work include:

- Measurement and analysis of performance variation of up to 16% between processors in top five supercomputing platforms: Cori, Edison, Cab, Stampede, Blue Waters on 1K chips
- Measurement and analysis of frequency, power, and temperature of processors on Cori, Edison

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 3.4).

3.1 Experimental Setup

3.1.1 Platforms

We have used five different top supercomputing platforms in our experiments. We list the detailed specifications of the platforms in Table 3.1.

Cori is a Cray XC40 supercomputer at NERSC [26]. It contains two types of nodes: with Intel Haswell and with Intel Knights Landing processors. Both processor types have Intel's Turbo Boost version 2.0 feature enabled. For Haswell processors, when all cores are

active, cores can peak up to 2.9 GHz from the nominal frequency of 2.3 GHz. Whereas Knights Landing processors can peak up to only 1.5 GHz from their nominal frequency of 1.4 GHz. Users can specify the frequency of their jobs at launch time with `srun --cpu-freq` command and specify power cap using `srun --power` command. The specified configuration is applied to all nodes within the job allocation. If no configuration is specified, processors operate under Turbo Boost.

Edison is a Cray XC30 supercomputer at NERSC [27]. 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 [28]. 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 1.2 GHz to 2.4 GHz 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 [29]. Each computer node has dual 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 [30]. Each compute node of Stampede has two 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 [31]. The processors in this system do not have a dynamic overclocking feature like Intel's Turbo Boost.

3.1.2 Applications

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 multiplication kernels in a loop on each core with data which fit in the last level cache (L3), 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.

LEANMD

This is a mini-app version of NAMD, a production-level molecular dynamics application for high performance biomolecular simulation systems [32]. 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. This benchmark is written in the Charm++ parallel programming framework.

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 represented as an object. For each iteration, the application executes in 3 stages, i.e., local computation, neighbor communication and barrier-based synchronization.

3.1.3 Measurement Methodology

We sample the time, hardware counters, temperature, and power for every 10 iterations of matrix multiplication for NAIVE-DGEMM and 100 iterations for MKL-DGEMM. This gives a sampling period of roughly 20 milliseconds. For other benchmarks, we do 1 second periodic measurements through an external module. Our frequency measurement is based on hardware counters and can be done on most of the HPC platforms, the only requirement is the installation of PAPI. The temperature and power measurements are specific to Cori and Edison.

Frequency Measurements: We use PAPI [33] 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" [34]. 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, then 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, then this ratio is typically greater than one. On the other hand, if the processor is idle, then this ratio will typically

be less than one [34]. 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 Cori and Edison: Cori and Edison users have read access to the temperature data of the cores through the `/sys/devices/platform/coretemp` interface.

Power Measurements on Cori and Edison: Cori and 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 privilege.

We note that the CPUs in Cori, 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 2.5.

Cori allows read access to `/sys/class/powercap/intel-rapl` system files which provides chip and memory level power data.

3.1.4 Eliminating OS Interference

Operating systems and other extraneous processes can induce significant noise into the application [35]. 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.

Note that, after Edison switched from `aprun` to `srun` interface in early 2016, this support has been discontinued. Unfortunately Cori, Cab and Stampede do not provide such an option.

3.1.5 Variation Metric

We calculate the effective variation of a set of numbers S by the following formula:

$$Variation_S(\%)=[(max_S - min_S) \div mean_S]\times 100$$

Since one slow processor can slow-down the whole application, we use the difference between max_S and min_S in the variation formula rather than a summary statistic such as standard variance or standard deviation.

We have used this variation metric in different contexts in this dissertation. The first example is the variation among the cores, since there can be performance difference among the cores running the same benchmark. In this case, S is the set of times that all the cores take to execute the same computational kernel. Another example is when analyzing the iteration time range within one core. S is then the set of iteration times within that core. If the iteration time changes over time in a core (i.e., if the variation in S is high), then we describe that core as a variable core.

Table 3.1: Platform hardware and software details

Platforms	**Cori - Node Type 1**	**Cori - Node Type 2**
Processor	Intel Xeon(R) E5-2698 v3 (Haswell)	Intel Xeon Phi(R) 7250 (Knights Landing)
Clock Speed	Nominal 2.3 GHz	Nominal 1.4 GHz
Turbo Speed	#Cores: 1-2/3/4/5/6/7/8/9-16 GHz: 3.6/3.5/3.4/3.3/3.2/3.1/3.0/2.9	#Cores: 1/2-68 GHz: 1.6/1.5
TDP	135 W	215 W
Cores per node	$16 \times 2 = 32$	68
Cache size (L3)	40 MB (shared)	No L3 cache
Linux Kernel	3.12.60	3.12.60

Platforms	**Edison**	**Blue Waters**
Processor	Intel Xeon(R) E5-2695 v2 (Ivy Bridge)	AMD Opteron 6276 (Interlagos)
Clock Speed	Nominal 2.4 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	No Boost Feature
TDP	115 W	115 W
Cores per node	$12 \times 2 = 24$	$16 \times 2 = 32$
Cache size (L3)	30MB (shared)	16MB (shared)
Linux Kernel	3.0.101	3.0.101

Platforms	**Cab**	**Stampede**
Processor	Intel Xeon(R) E5-2670 (Sandy Bridge)	Intel Xeon(R) E5-2680 (Sandy Bridge)
Clock Speed	Nominal 2.6 GHz	Nominal 2.7 GHz
Turbo Speed	#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
TDP	115 W	130 W
Cores per node	$8 \times 2 = 16$	$8 \times 2 = 16$
Cache size (L3)	20MB (shared)	20MB (shared)
Linux Kernel	2.6.32	2.6.32

3.2 Measurement and Analysis of Variation in Large Scale Systems

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.

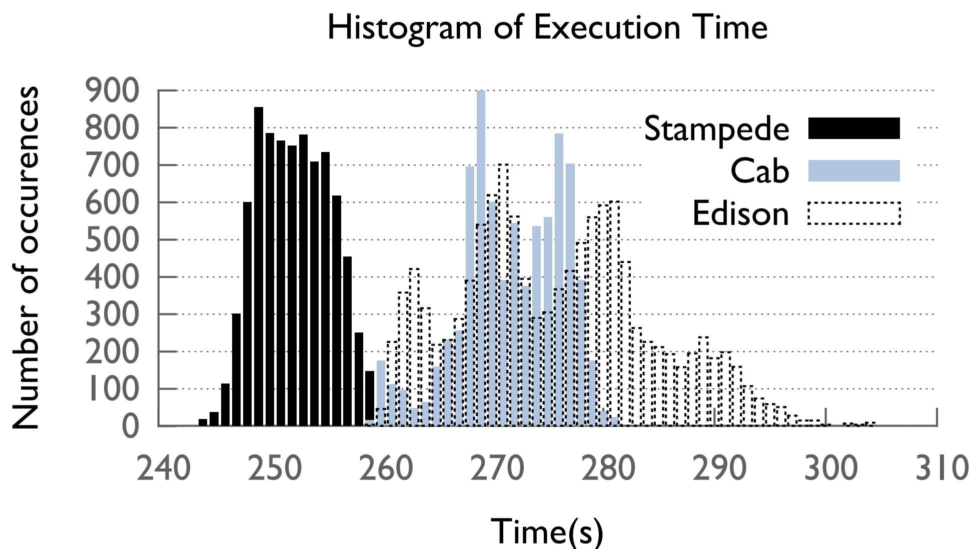


Figure 3.1: The distribution of benchmark times on 512 nodes of each machine looping MKL-DGEMM a fixed number of times on each core.

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 3.1, shows the histogram of the core performance running a benchmark that calls Intel MKL-DGEMM sequentially on the Edison, Cab, and Stampede supercomputers. The overall core-to-core performance difference is around 16%, 8% and 15% respectively.

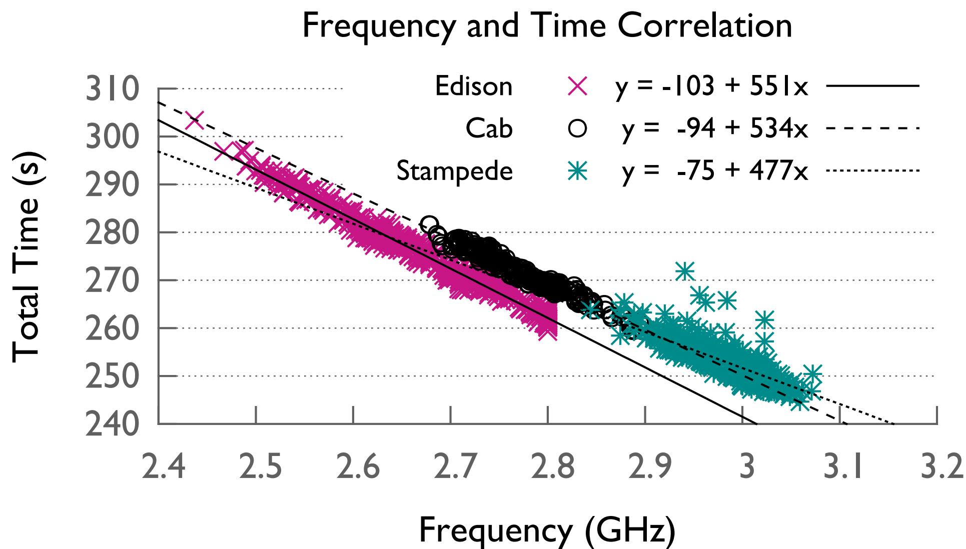


Figure 3.2: Average frequency shows a negative correlation with total execution time on both Edison, Cab and Stampede when running MKL-DGEMM.

To further understand this time difference, we looked into the relationship between the total time and the average frequency of the cores from the whole execution. The frequency difference among the chips was a result of the dynamic overclocking feature of the processors. Figure 3.2 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.

Analysis of Intel Xeon Phi Architecture

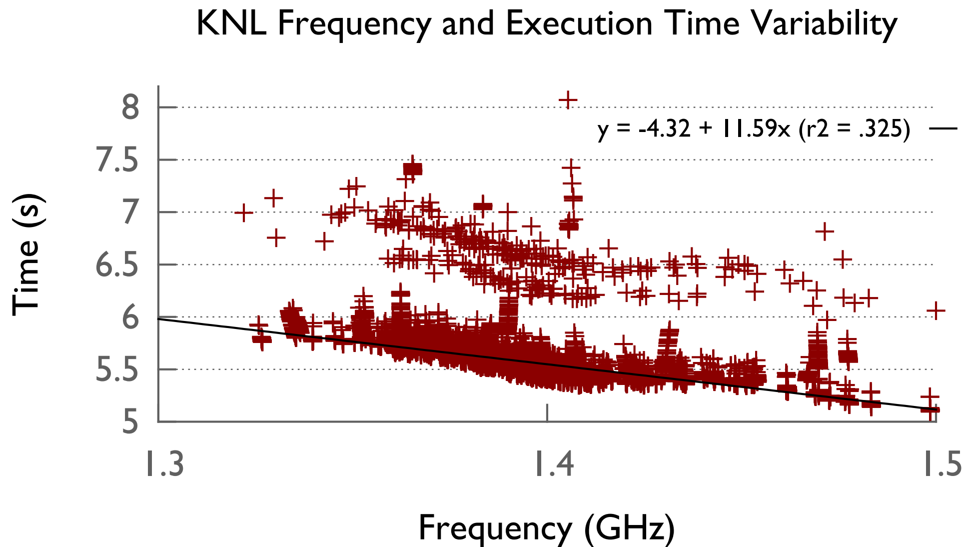


Figure 3.3: KNL execution time variability of 256 nodes 17408 cores. Overall variation is 53.4%. Frequency variation is almost 200 MHz.

Intel's new generation Xeon Phi architecture shows quite different characteristics compared to processors in the Intel Xeon family, therefore it deserves its own subsection. Here, we analyze Knights Landing processors (KNL), as hardware details shown in Table 7.1. Knights Landing processors have lots of small cores (68) within a chip where two cores form a *tile*. There is no L3 cache, but there is a slightly larger L2 cache that is shared between two cores

in a tile. The total L2 size is 34 MB which makes 1 MB per tile. KNLs have a narrow Turbo Boost range with a low nominal frequency. When all cores are active, it can only boost 100 MHz, from 1.4 GHz up to 1.5 GHz. Moreover, if vector instructions are used, the effective frequency can be even lower than the nominal frequency - which was never the case for Xeon architectures.

We use the our MKL-DGEMM benchmark to do variabilty analysis on KNLs. This benchmark is highly optimized and uses vector instructions. An important parameter to tune using this benchmark is the matrix size. In Xeon architectures, we use a matrix size that fits in L3 cache so there is no memory related variability that masks the manufacturing related variability. Since there is no L3 cache in Xeon Phi's, we try using a matrix size that fits in L2 cache instead.

Figure 3.3 shows the execution time variation and frequency correlation of 256 KNL nodes in Cori. Overall variation is 53.4%, which is quite high. However, as seen from the plot, not all performance variation is related to frequency variation. Therefore, the R-squared value is showing weak correlation between the execution time and frequency with a value of 0.325. Possible causes of this variation are OS noise and shared L2 cache contention. In fact, when we do the experiment using only one core per tile, the performance variation almost completely disappears. Using one core per tile removes the cache contention, leaves free cores for OS processes, and processors do not hit TDP anymore. This makes it difficult to study the manufacturing related frequency variation in isolation.

Figure 3.4 focuses on four randomly selected nodes to show the behavior of the cores within the node. There is no intra-chip frequency variation, so this implies Turbo Boost changes the frequency of the cores alltogether. However, despite the fact that there is no frequency variation, there is intra-chip perfromance variation. Especially, the first and the last core (with IDs 0,67) in every node shows low performance. These cores created the parallel line above the fit line in Figure 3.3. The cause of this is likely to be OS processes on those cores.

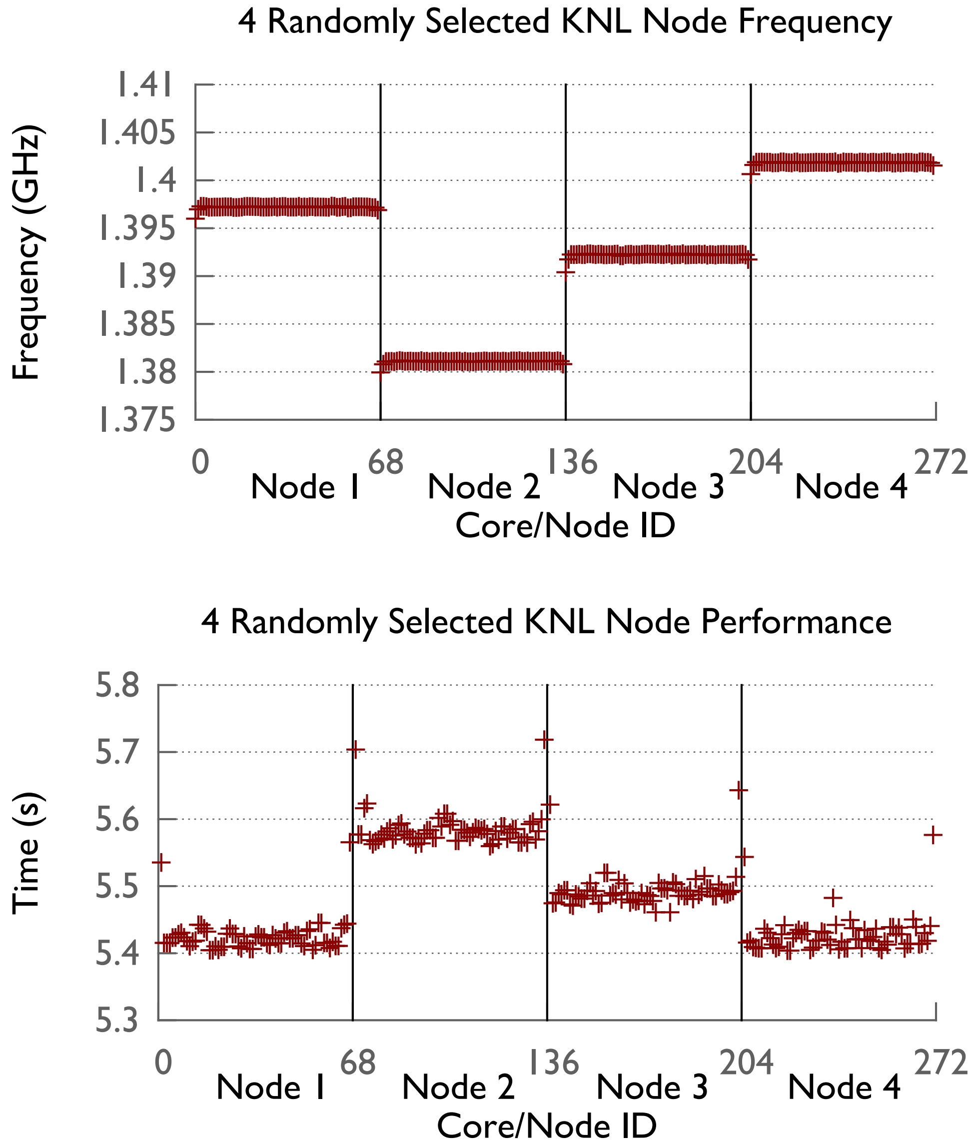


Figure 3.4: Plots show the execution time and frequency of four randomly selected nodes sorted by the core and node IDs.

3.2.1 Inter-chip Frequency Variation

We first 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 (on architectures shown other than Xeon Phi). Therefore we only focus on the inter-chip variation that arises even though they are all of the same product model.

There is a warm-up period from job launch to the time at which the chips settle at a

particular frequency or a duty-cycle-determined frequency average. Figure 5.2 illustrates this warm-up period with temperature, frequency, and power measurements of a selected compute node. The node has two chips 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. The 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 is around 1 minute. We exclude the warm-up period in our following reported measurements.

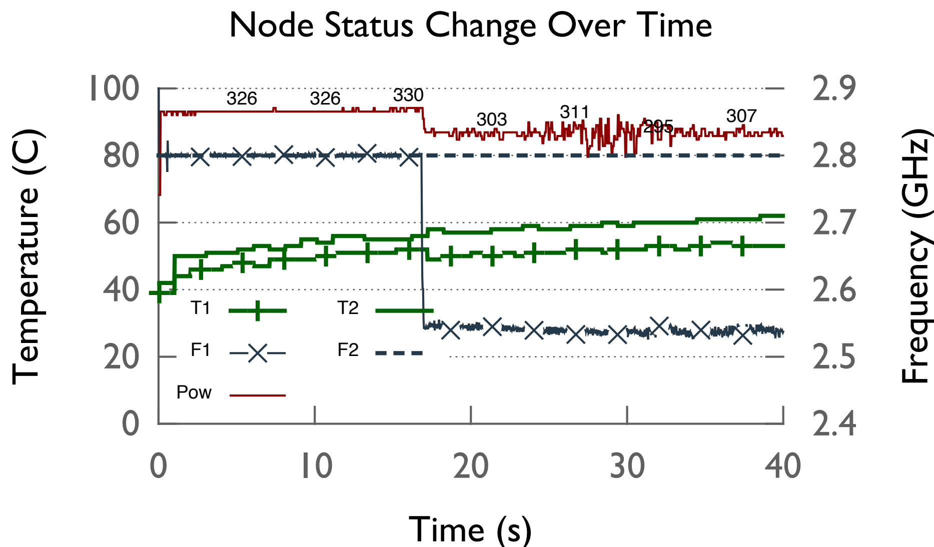


Figure 3.5: Plot shows the power (Pow (W)) of a randomly selected node, temperature (T1, T2) and frequency (F1, F2) of the two chips on the node.

Table 3.2 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

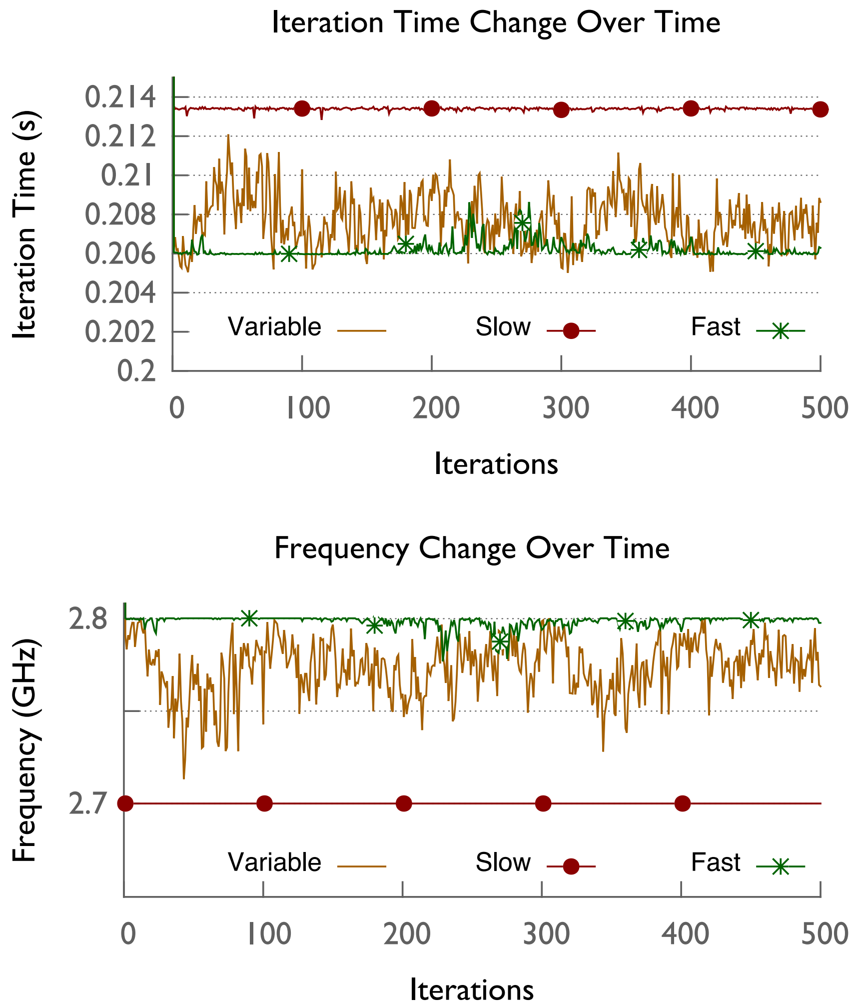


Figure 3.6: Iteration time (top plot) and frequency (bottom plot) over iterations are shown from cores selected from 3 chips showing distinct behavior: slow, variable, and fast.

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 3.3 and 3.4 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 3.6. 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

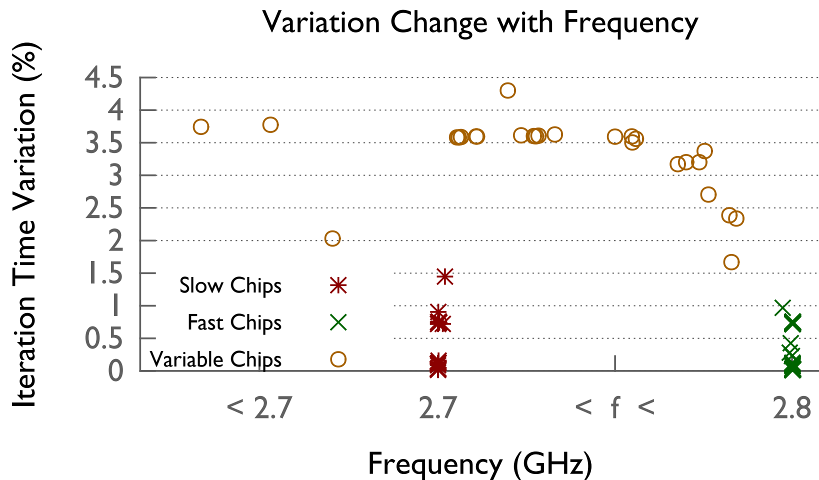


Figure 3.7: Different chip types naturally form clusters and can be identified looking at their percentage variability in the iteration time.

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 3.2, and the same correlation applies here as well. Figure 3.7 shows how variable chips can be easily identified by looking at the percentage variability in their interation time. Up to 1.5% variability can happen even in stable chips, but beyond that number means the chips cannot sustain a steady frequency under the workload and is a variable chip.

Table 3.2 also shows the effect when one core is left idle. We try leaving one core idle from the chip in 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 interference, but also reduces the number and severity of slow and variable chips as well. Since the 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 Chapter 6.

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

Table 3.2: 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 3.3: 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 3.4: 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

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, then 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

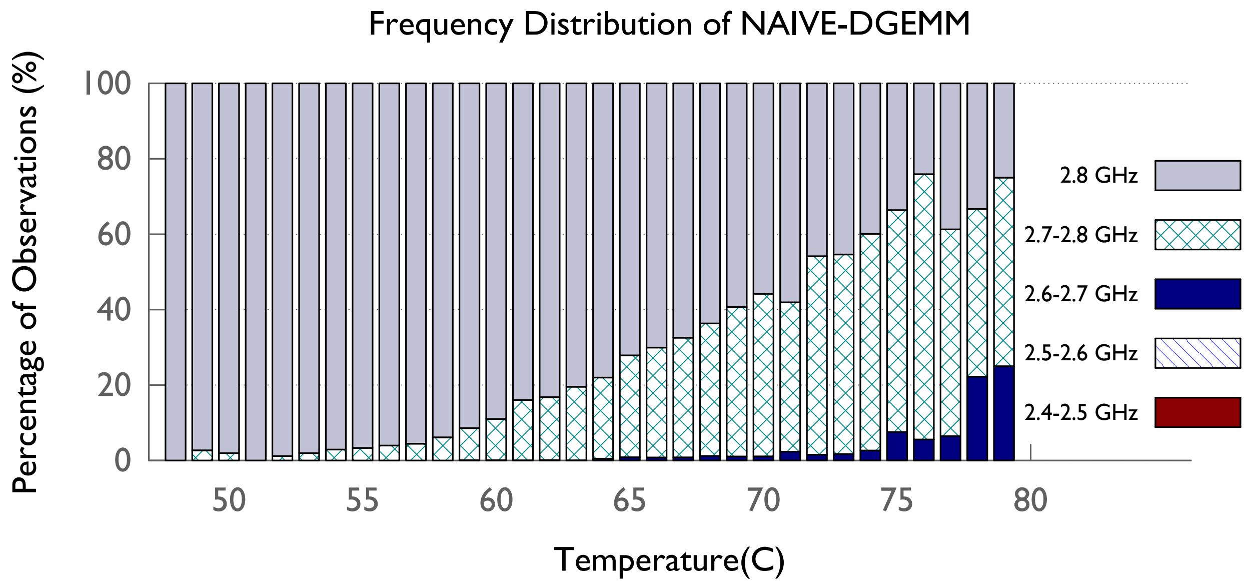
time waiting for communication, synchronization or high memory access latency would give the CPU more idle time, which results in lower temperature and power values. We observed by experiment that such applications may show little frequency variation or none at all.

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.

3.3 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. Figure 3.8 shows what frequency level processors are running at for each temperature bin for the 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 show operation at each 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 threshold temperature of 76°C. The data for LEANMD and JACOBI2D looks very similar to NAIVE-DGEMM. Thus, we



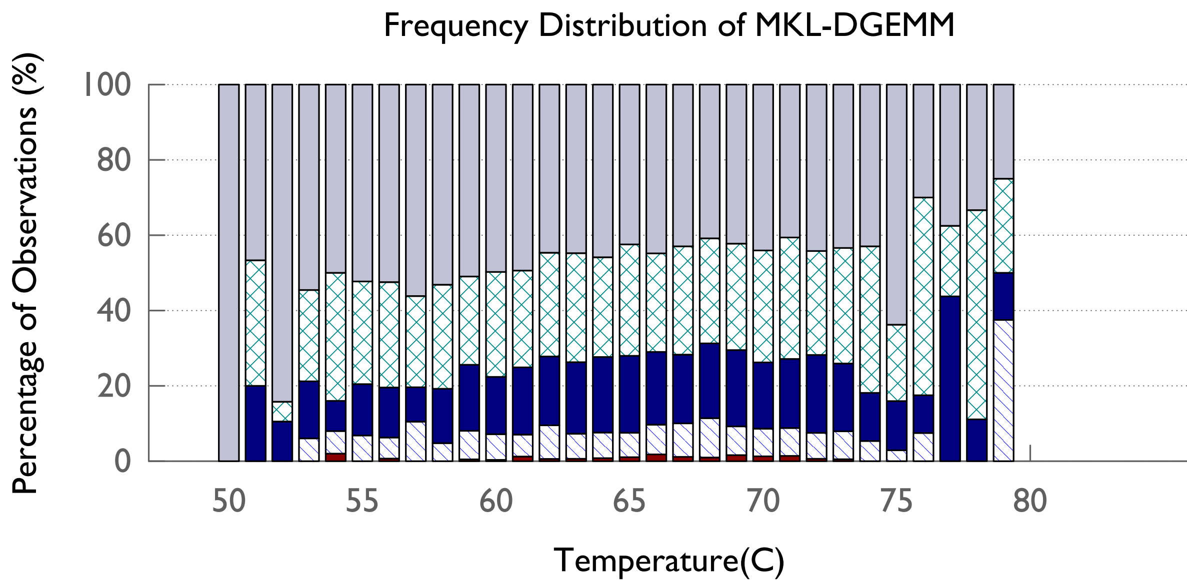


Figure 3.8: Plots show what percentage of the chips run at what frequency in each temperature level for NAIVE-DGEMM (left plot) and MKL-DGEMM (right plot) benchmarks. The data points are collected from the whole execution of the benchmarks and classified according to their temperature bins. The chips' operating temperatures are not directly correlated with their frequency under the heavier load of MKL-DGEMM.

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 the power draw of the cores. The

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

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

Table 3.5: 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 or close to their limiting power while the fast chips are not.

package control unit (PCU) in processors with Turbo Boost Technology 2.0 has an intelligent exponential weight-moving average (EWMA) algorithm [36] 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. By default, that power threshold limit is the processor's TDP, which is the power level in the steady state where the frequency of the CPU could safely be higher than the nominal frequency [36]. Intel's RAPL [21] feature lets software set a lower threshold. We can observe this time interval in the EWMA algorithm from the variable core in Figure 3.6. 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 3.5 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 six different combination of chips. These are: fast & fast, fast & slow, fast & variable, slow & slow, slow & variable, and variable & variable.

We can see that the variable processors systematically consume more power than slow processors, which in turn consume 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. The processors have a TDP of 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 show 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 3.5 since there are 8 different frequency levels and thus 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.

A few months after these measurements are made on Edison, NERSC's new generation platform, Cori, have become available with Intel Haswell and Knights Landing processors supporting new power measurement features. Unlike Edison, Cori supports chip power, as well as memory power measurements that are available to all users of the supercomputer. Therefore, we repeated our Edison experiments on Cori, especially to find an answer to the

question: is power the reason of the processors to get throttled down?

Figure 3.9 shows the power and frequency correlation of power and frequency using 512 Haswell chips. The chips that have lower frequency than the maximum frequency of 2.5 GHz consume 134 W which is just 1 W under the TDP of the chip, 135 W. Figure 3.9 shows the temperature and power correlation of the same chips and the processors that hit the power cap show a wide range of temperatures from 56 to 79°C. There is no obvious correlation between temperature and power. This allows us to conclude that temperature is not the sole reason for frequency throttling, hitting the power cap can also be a factor. Note that the performance variation among the Haswell chips on Cori with the MKL-DGEMM benchmark is 14.76% which is not much different than the earlier generation processors.

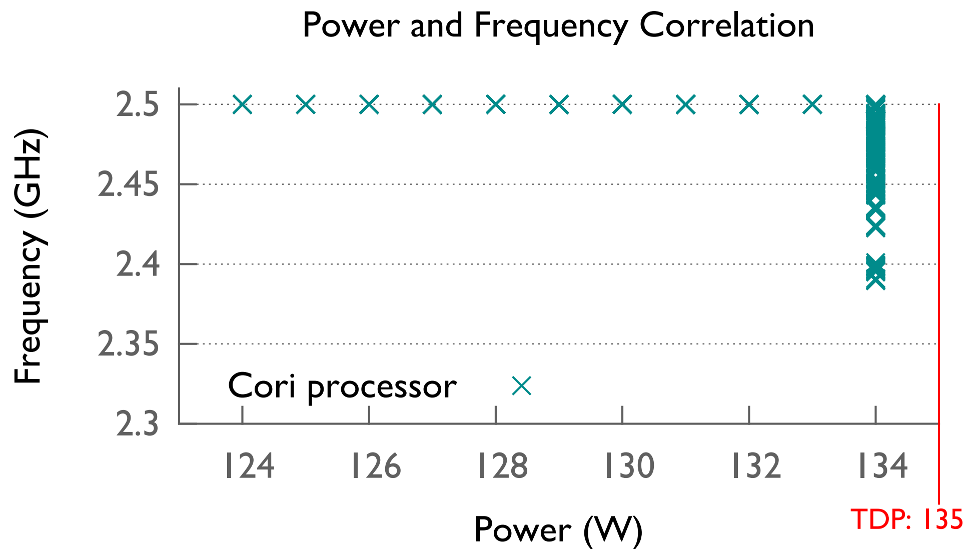


Figure 3.9: Processors that reach TDP drops their frequency and stay at 134 W level, just under 1 W than TDP. 512 Haswell processors in Cori platform are shown.

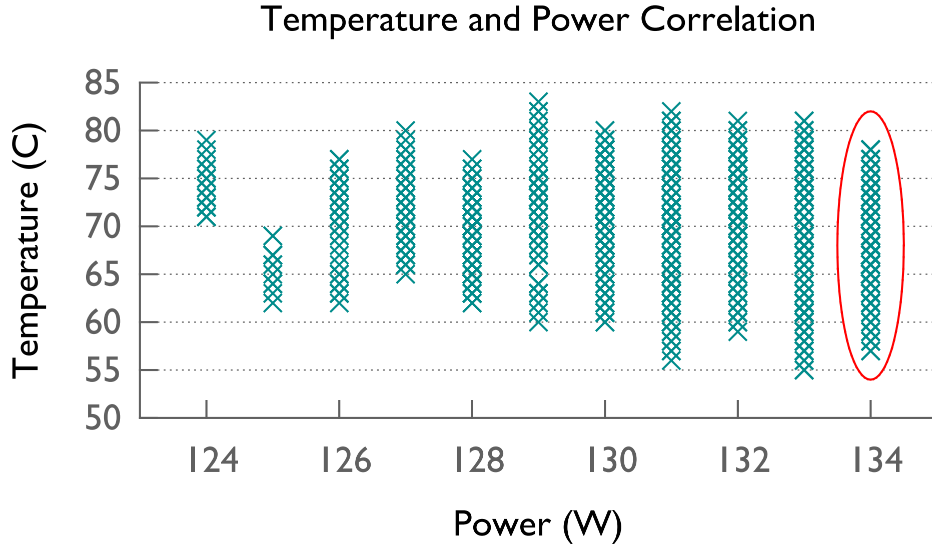


Figure 3.10: The processors that hit the power cap, the ones circled, have a wide range of temperatures similar to the ones that do not hit. Temperature does not directly correlate with power.

3.4 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 [37]. 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 [38]. On the other hand, Kumar et al. show [39] 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 [40, 41]. 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 [42]. 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 [43]. Various different variation-aware thread mapping and scheduling algorithms have been proposed for multicore processors to minimize power or maximize performance with and without a power bugdet [44]. Langer and Totoni propose a variation aware scheduling algorithm [45, 46] 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 [47].

There have been various other studies showing the thermal variation among supercomputer architectures [48, 49]. 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 [13]. Wang et al. propose thermal aware workload scheduling technique for green data centers [50]. Choi et al. propose a thermal aware task scheduling technique to reduce the temperature of the cores within a processor [51].

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

3.5 Summary

In this chapter, 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. Processors have dynamic overclocking features in order to take advantage of headroom in the operating temperature and power consumption, and to adjust their voltage and frequency based on their thermal

and energy constraints. If the current trend continues in the future, then the observed variation may increase further at larger scales. Turning off these dynamic features may not be the ideal solution to mitigate the variation because it can sacrifice available performance. We should look for ways to mitigate variation from the application software, as we do in the next chapter.

CHAPTER 4

Mitigating Frequency Variation

For decades, the goal of HPC has primarily been to obtain better performance. Although this trend had started to change when the scale and the power consumption of data centers kept increasing significantly. In this chapter, we analyze potential solutions to mitigate the frequency variation problem from a performance perspective. Without taking power or energy efficiency into the picture, we exclusively focus on how to increase performance of applications where the proccessors have different frequency levels that might dynamically change. The solutions we propose do not all require power related measurement or control rights. These solutions are: disabling Turbo Boost, replacing slow chips, leaving some cores idle, and dynamic task redistribution (i.e., speed aware load balancing) with Charm++ RTS.

The main contributions of this work include:

- A demonstration of the performance degradation of HPC applications caused by variation
- Analysis of potential solutions to mitigate effects of in-homogeneity: disabling Turbo Boost, replacing slow chips, idling cores, and dynamic task redistribution
- A speed-aware dynamic task redistribution technique which improves performance up to 18%

Runtime-based load balancing solutions to mitigate OS related performance variabilities have been proposed before, for example, with the Legion runtime [52]. However, to the best of our knowledge, a runtime based speed-aware load balancing strategy have not been

evaluated before.

4.1 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 4.1: 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 4.1 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 computational intensity, whereas the 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

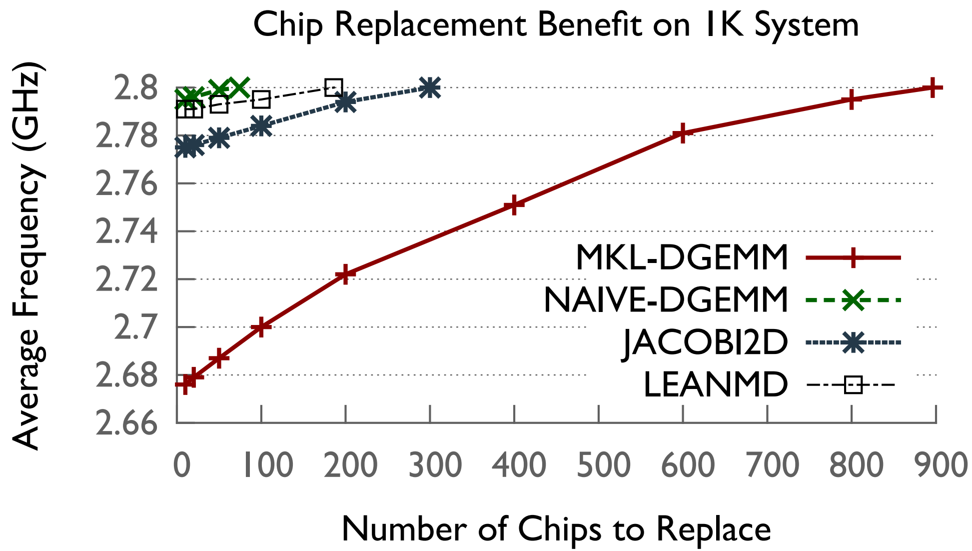
fact, newer generation processors are becoming more dynamic. We have shown an example of this in the previous section where newer generation Ivy Bridge processors have a wider frequency boost range than older generation Sandy Bridge processors. Processors can take advantage of power and thermal headroom 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.

4.2 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 4.1 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 the 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.



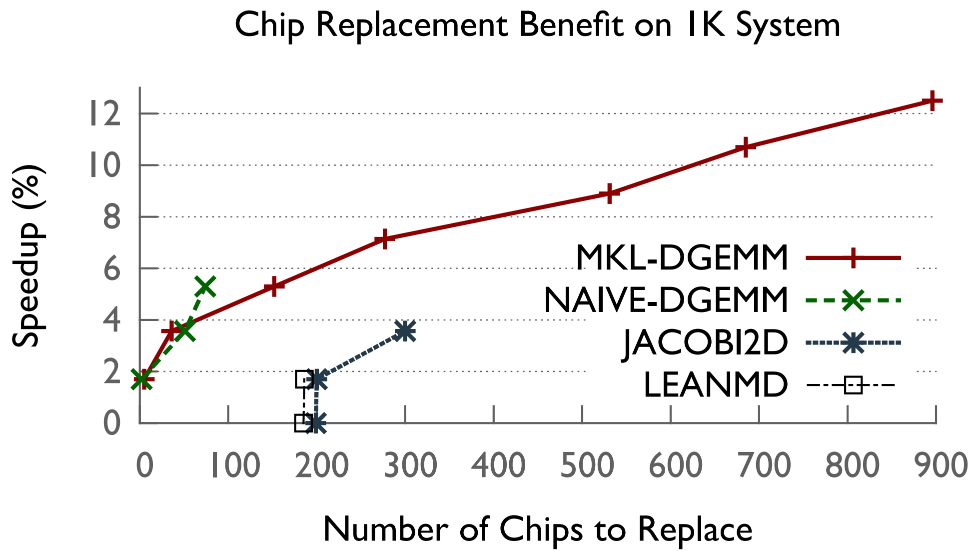


Figure 4.1: How many chips we should replace to get performance benefit?

4.3 Leaving cores idle

The mitigation of sluggishness and variability when leaving a core idle (cf. Table 3.2) 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. This 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 4.2, 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, entire chips seem to be uniformly more or less efficient.

On other systems, with different core densities, clock speeds, and the prevalence of slow and variable chips, this calculation 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 [46]. A

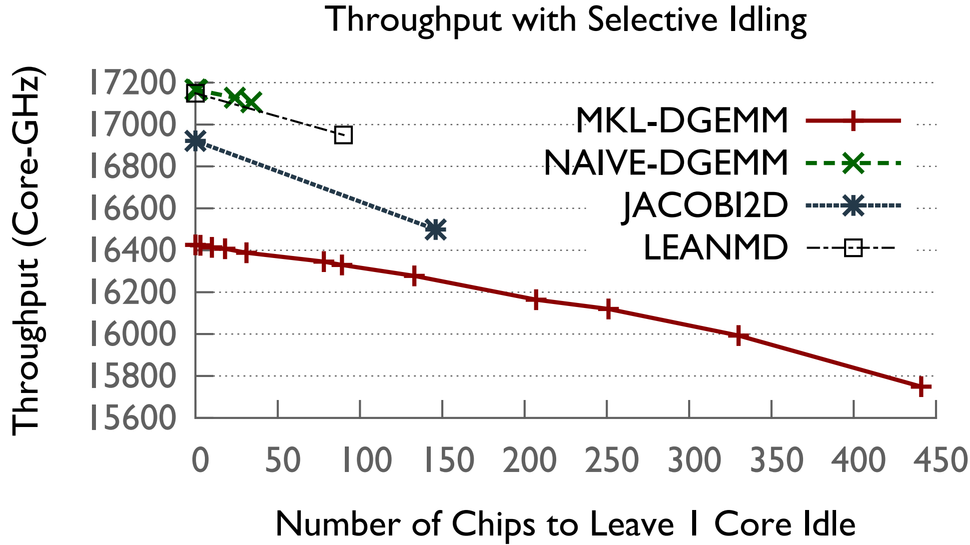


Figure 4.2: Throughput of all cores when 1 core is left idle from chips starting from the slowest chip.

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 [11]. Reducing the power drawn by each chip would also reduce their operating temperature, potentially lowering fault rates and thus improving overall time to completion [14].

4.4 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 [53]. 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 ($obj.load \times S_{donor} \div S_{lightProc} + lightProc.load < avgLoad$) 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, 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

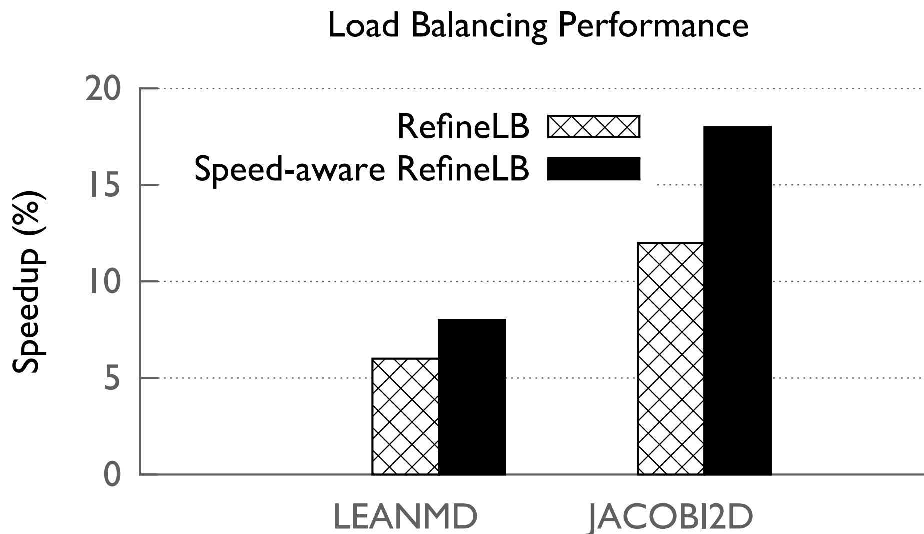


Figure 4.3: Speedup of RefineLB and Speed-aware RefineLB compared to no load balancing case.

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 per cycle or a more detailed performance model, such as a frequency-dependent Roofline [54], to get a better estimation.

In Figure 4.3, 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.

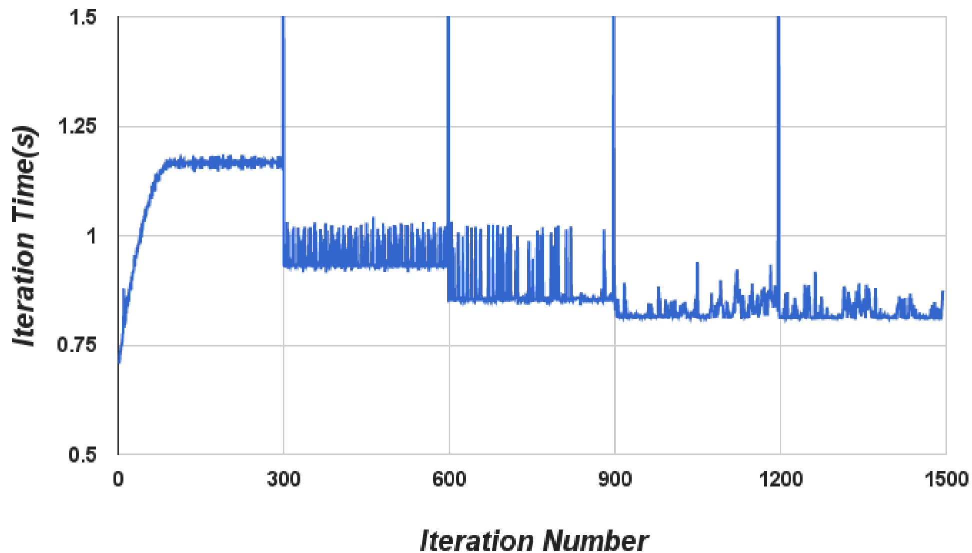


Figure 4.4: Homogeneous Processors under Turbo-Boost using Speed-aware RefineLB

In Figure 4.4, we show our speed-aware load balancer can play an effective role where the frequency variation between processors increases over time as the processor continues running. Here, we run Jacobi-2D with 24000 grid size and 400 block size on 2 same-model processor nodes where one of the nodes keeps running at 2.4 GHz, while the other node slowly throttles down from 2.4 GHz to 1.8 GHz due to its high temperature. During every iteration, the fast node has finishes its computation earlier and it has more time to cool-down until the barrier synchronization after each iteration. Whereas the slow node is overloaded and it has no time to cool-down. The load balancer is triggered every 300 steps. Since the slow processor is throttled-down more and more over time, one load balancing is not enough to balance the load. Only after the first three migrations, performance stabilizes. Compared to execution time before the first load balancer, we get 30% improvement after last migration.

The overhead of the load balancing does not exceed 2 seconds in all experiments shown. It is a small overhead which is compensated after a few iterations. For larger node-counts, more scalable load balancing algorithms can be made speed-aware in a similar way we made *RefineLB* speed-aware.

4.4.1 Speed-Aware Load Balancer in a Heterogeneous System

In this section, we evaluate our speed-aware load balancing technique under a system where the processor speeds are different inherently because of their models. We use a heterogeneous cluster configuration where the processor types are different (i.e., half of them have 2.0 GHz base clock speed and the other half has 2.4 GHz). We use an 80-node local cluster at UIUC. The hardware details of the processors can be found in Table 7.1.

Table 4.2: Platform hardware details

Processor	Intel Xeon E5-2620	Intel Xeon X3430
Clock Speed	2.0 GHz	2.4 GHz
Max Turbo Speed	2.5 GHz	2.8 GHz
Cores	6	4
Cache size(L3)	15MB	8MB

In Figure 4.5, we evaluate *RefineLB* and *Speed-aware RefineLB* performance against no load balancing performance. We use Jacobi-2D application with grid size as 20000 and block size as 200.

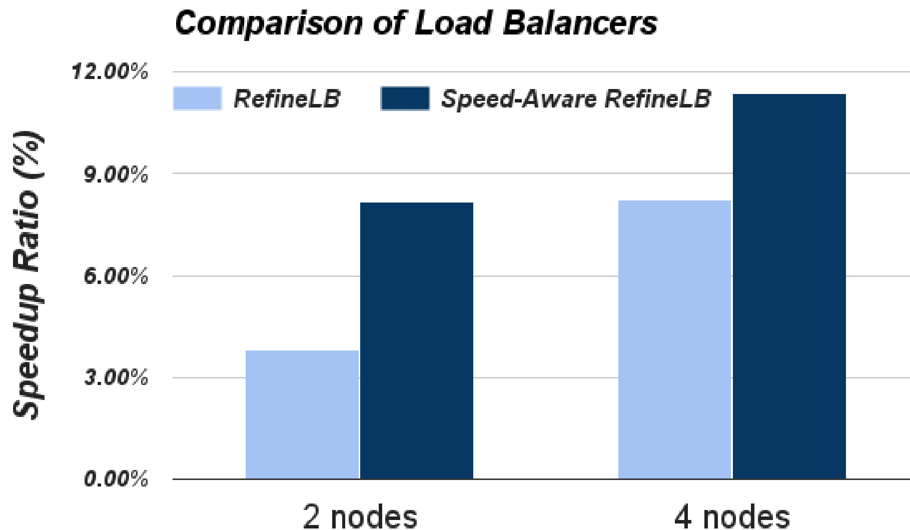


Figure 4.5: RefineLB and Speed-aware RefineLB performance compared to without load balancing case

Table 4.3: Object Migration with Different Load Balancers

Load Balancer	**# Migrations** (out of 10000)
RefineLB	95
Speed-aware RefineLB	204

RefineLB improves the performance by 4% and 11% on 2 and 4 nodes (processors) respectively. Speed-aware version performs around 5% and 3% better compared to *RefineLB* . By taking processor clock speed into consideration and combining it with history workload information, *Speed-aware RefineLB* has a more precise estimation of resulted workload of target processor after migration. Therefore, it can move objects more aggressively than non-speed-aware version. *Speed-aware RefineLB* migrates 204 whereas *RefineLB* migrates 95 objects out of 10000 objects from the fast processors to the slow processors.

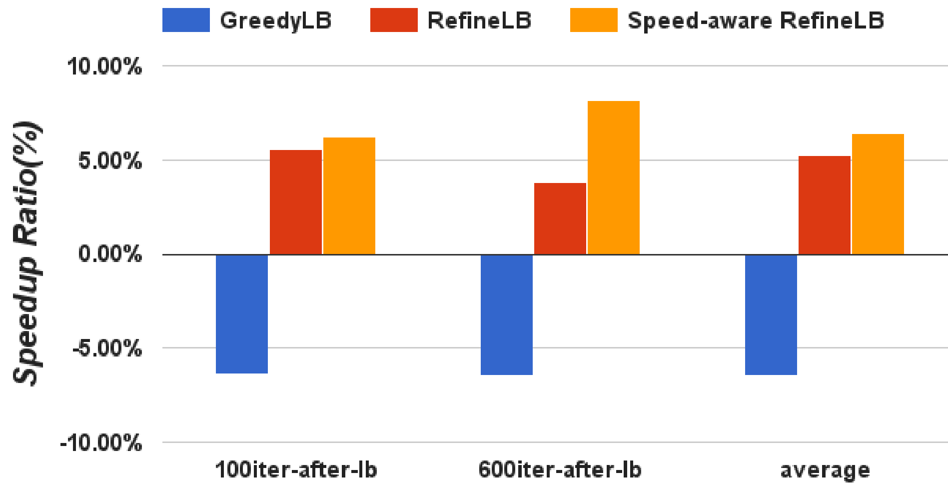
Problem Size Effect

The different problem size of Jacobi-2D application varies the intensity of computation, communication and memory accesses. In the two sets of experiments below, we show that depending on the grid and block sizes, the benefit of load balancing changes.

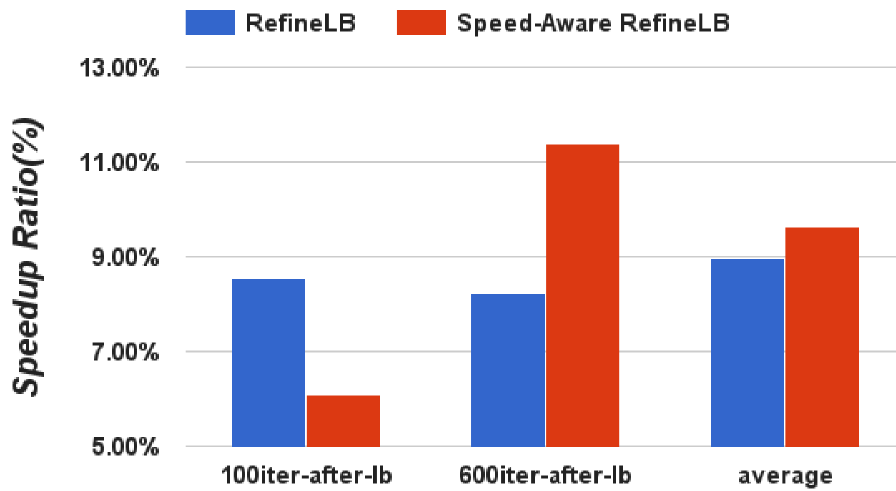
Different Grid Sizes: The grid size is the total workload size of the array before divided into smaller blocks. We fix block size at 200 and vary grid size from 16000 to 24000 which is around 1.9 GB to 4.2 GB of data in total. Figure 4.7a shows effect of grid size with our speed-aware load balancer. We show the processor frequency behavior over iterations in Figure 4.7b and 4.7c. When the grid size is 16000, it takes 1.47% longer to finish execution, the load balancer has negative effect on performance and both processors run close to their peak speed. Smaller grid size does not require much computation between two synchronizations (iterations), and processors turn into communication stage before heat-up. The reason of the slowdown could be the higher communication overhead when the objects are moved from slow to fast processor. When grid size is set at 20,000 and 24,000, we have 8.51% and 3.03% performance gain respectively. The frequency of fast node does not change over time, while slow node's frequency drops after about 180 iterations from 2.4 GHz to 2.0 GHz. Larger

grid size introduces more workload for both processors and leads to frequency throttling. On one hand, the fast node only needs to spend shorter time to finish the work and then it gains some time to cool-down before synchronization. On the other hand, the slow node has no time to only cool-down. When the grid size increases to 24000 load balancer works less effectively because of the increased memory access latency.

Different Block Sizes: The block size determines the workload of each migratable object. We show speedup ratios in Figure 4.8a with grid size fixed at 20000 and block size varying from 100 to 800. Processor frequency behavior with block size 100 and 800 are shown in Figure 4.8b and Figure 4.8c respectively. The highest performance improvement is 28% when block size is set as 100. Performance gains decreases to 8.2% and 2.5% when block size is increased to 200 and 400. This can be explained as the following. Firstly, the system creates more objects with smaller block size, resulting in heavier background load. Secondly, the smaller block size is more flexible to migrate. In our load balancer algorithm, we conservatively compute workload of the target processor after object migration. Larger objects tend to make target processor overloaded and less likely to be moved. Moreover, when the block size is 800, we see 12.7% performance improvement and we see objects moved from the fast node to the slow node. This inverse behavior could be caused by the different cache size of the processors. Overall, load balancing still has a significant performance benefit.

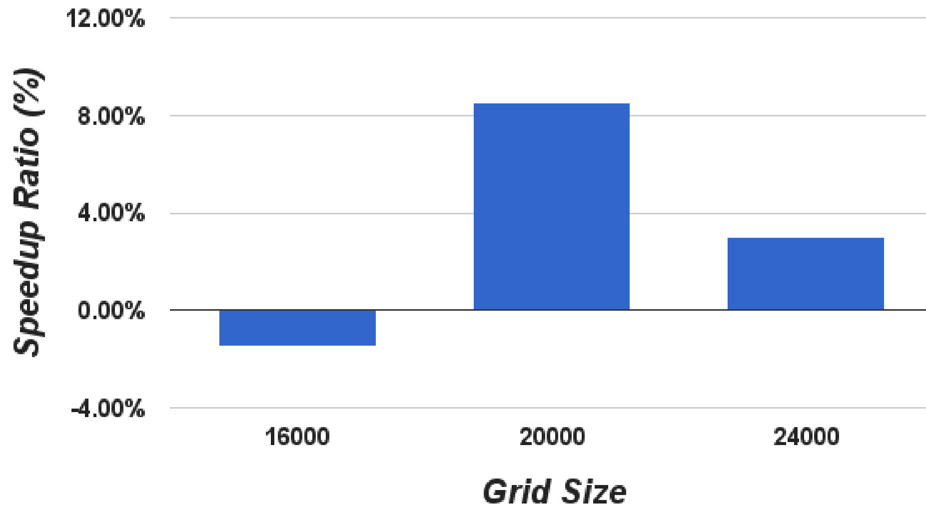


(a) 12 cores

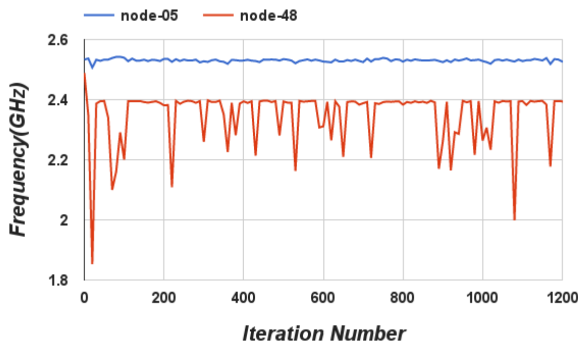


(b) 24 cores

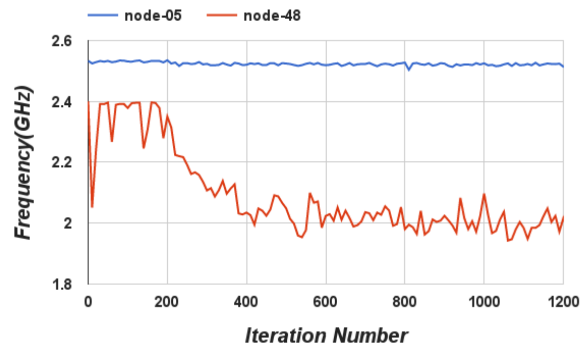
Figure 4.6: Comparing RefineLB and Speed-aware RefineLB



(a) Speedup with different grid sizes

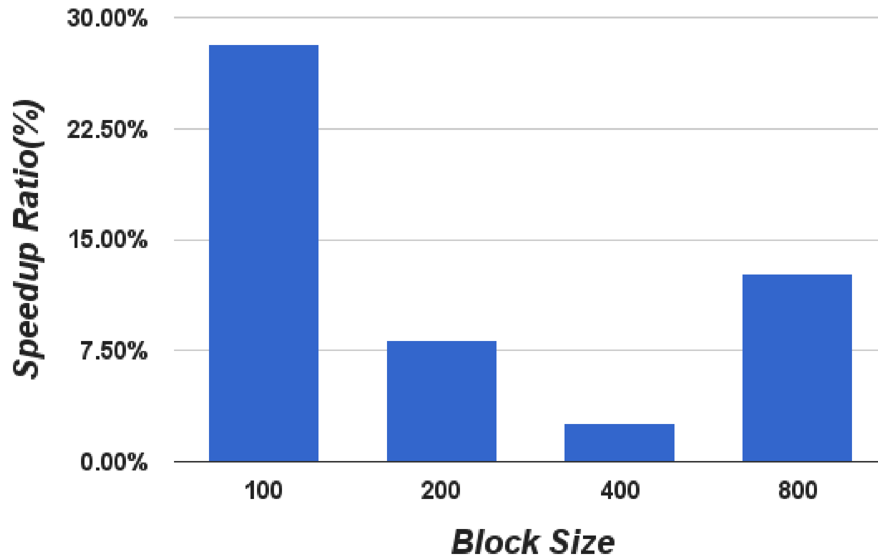


(b) Frequency when grid size=16000

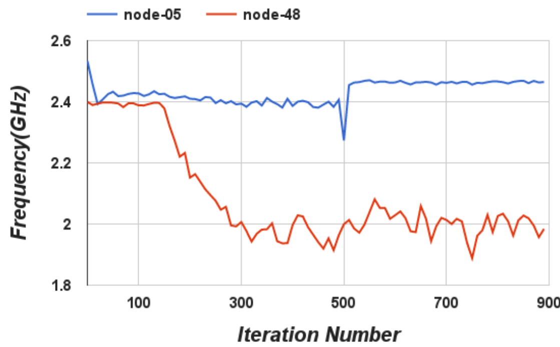


(c) Frequency when grid size=20000

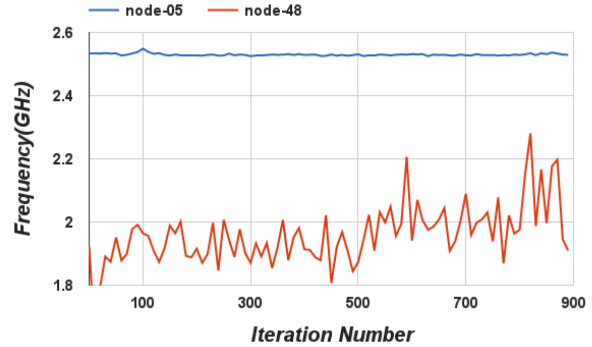
Figure 4.7: Speed-aware RefineLB performance on Jacobi-2D with different grid sizes.



(a) Speedup with different block sizes



(b) Frequency when block size=100



(c) Frequency when block size=800

Figure 4.8: Speed-aware RefineLB performance on Jacobi-2D with varying block size. Load balancer is triggered at iteration 800.

4.5 Summary

In this chapter, we evaluate four different techniques to mitigate the frequency variation: disabling Turbo Boost, replacing slow chips, leaving cores idle, and dynamic load balancing. We conclude that speed-aware dynamic load balancing is the most feasible solution giving the best performance and it does not require any change in the machine infrastructure. Speed-aware load balancing can be both used in platforms where processors have the same or different nominal clock frequencies (i.e., frequency variation is caused either by dynamic overclocking or due to different processor types). We show how a dynamic runtime can cope with the dynamic, unpredictable behavior of the chips.

For a future direction, speed-aware dynamic load balancing can be improved by taking into account memory and compute load of Charm++ objects separately. In the current infrastructure, the load of the objects are represented by their execution time and our speed-aware load balacing strategy scales the execution time with the speed of the processors. This gives a good enough estimation for most scenarios, especially for compute intensive applications. However, for memory intensive applications, where the load is dominated by the memory load times, the estimation will not be correct since frequency only effects the CPU load whereas the memory load time remains the same even if the CPU speeds are different. For this reason Charm++ infrastructure can be changed to track the memory and computational load of the processors independently.

CHAPTER 5

Mitigating Temperature Variation

Increasing scale of data centers and the density of server nodes pose significant challenges in producing power and energy efficient cooling infrastructures. Current fan based air cooling systems have significant inefficiencies in their operation causing power peaks in fan power consumption and temperature variations among cores. In this chapter, we identify the cause these problems and propose proactive cooling mechanisms to mitigate the power peaks and temperature variations. An accurate temperature prediction model lies behind the basis of our solutions. We use a neural network-based modeling approach for predicting core temperatures of different workloads, under different core frequencies, fan speed levels, and ambient temperatures. The model provides guidance for our proactive cooling mechanisms. We propose a preemptive fan control mechanism that can remove the power peaks in fan power consumption and reduce the maximum cooling power by 53% on average as well as energy consumption up to 22%. Moreover, through our decoupled fan control method and thermal-aware load balancing algorithm, we show that temperature variations in large scale platforms can be reduced from 25°C to 2°C, making cooling systems more efficient with negligible performance overhead.

Given the 20 MW power limit for an exascale system set by the DOE, reducing the power consumption of high performance data centers has become an important challenge [2]. Moreover, some supercomputing facilities, such as ORNL that hosts the largest supercomputer in the United States – Titan, is charged based on its maximum power usage [4]. This has

increased the importance of making optimizations for power-constrained systems, improving the performance under a strict power budget and similar power focused research [11, 22, 55] compared to prior focus on energy optimizations.

Up to half of the data center power consumption can be cooling costs [56]. Addressing data center cooling costs is becoming more challenging as supercomputers start using high-density, fat nodes with high core counts and accelerators such as GPGPUs. Moreover, the pressure of reducing the power costs and carbon footprint without sacrificing from application performance has driven these systems to operate under higher ambient and liquid inlet temperatures [57–59].

Thermal-aware workload management strategies can improve cooling efficiency, increase the life-time of the chips, and reduce the risk of over-heating-related system failures [14, 60, 61]. However, implementing these strategies requires an accurate temperature model of the system. Accurately predicting core temperatures is difficult due to multiple factors including sophisticated workloads, complex thermodynamics within the data center, physical layout of each core/node, and CMOS manufacturing differences. Some previous model-based approaches to temperature prediction assume that all instances of a given hardware component behave similarly [62, 63]. This assumption significantly limits the prediction accuracy, because tuning model parameters for individual hardware instances quickly becomes impractical.

We propose a learning-based temperature prediction modeling approach that can capture complex parameters that cause on-chip temperature variations using a neural network (NN). Figure 5.1 shows our approach which consists of four steps: (1) monitor and collect core temperatures data under different utilization rates, core frequencies, fan speed, ambient temperatures (2) pre-process data, (3) train the neural network model, and (4) deploy the neural network to provide core temperature predictions. We construct the neural network model for each chip independently. This provides more accurate temperature predictions than existing model-based approaches. Furthermore, we study different neural-network back-propagation algorithms and neural network structures such as the number of layers and the number of neurons to gain an understanding of the memory and computation requirements of the neural network deployment.

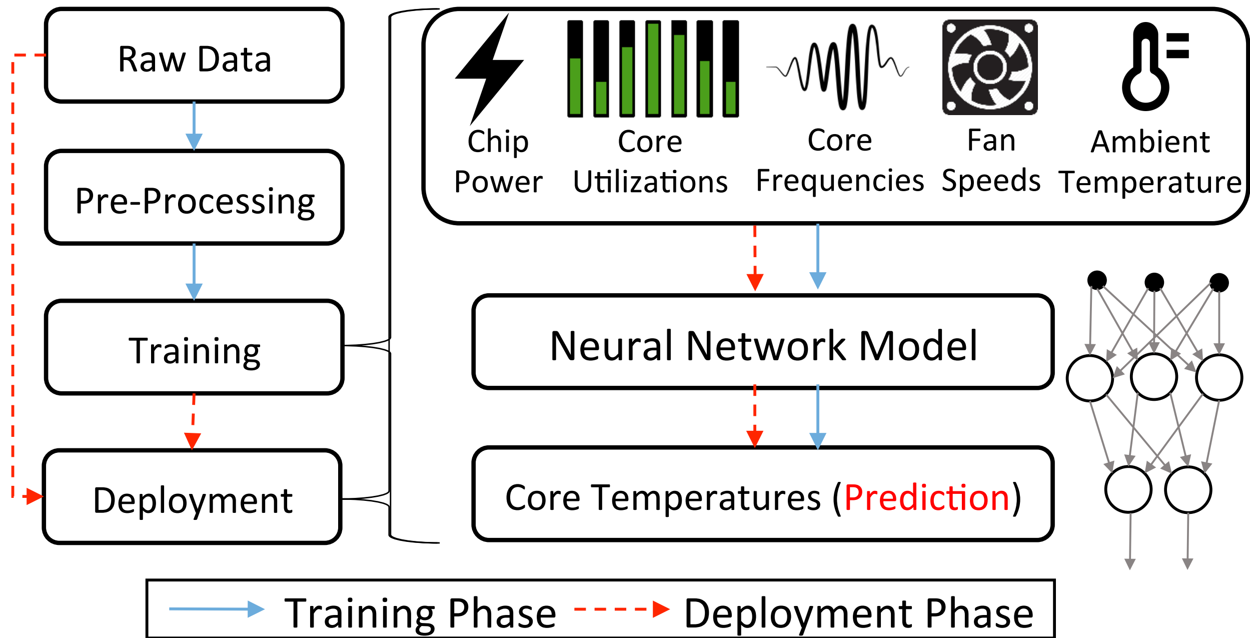


Figure 5.1: Neural network-based thermal prediction approach.

Having an accurate temperature prediction model enables us to implement different proactive cooling mechanisms that can reduce power and energy. First, we identify inefficiencies in existing air cooling control systems. These deficiencies include reactive fan behavior and temperature variations. We analyze temperature variations within a chip and across chips using more than 1,000 cores in two different architectures. Finally, we propose model-guided proactive mechanisms that mitigate these problems.

The main contributions of this chapter are the following:

- We propose a simple yet powerful neural network-based temperature prediction model that can predict steady state core temperatures accurately under different core utilization levels, frequency levels, and fan speeds.

- We propose a proactive fan control mechanism, *precooling*, that removes the power oscillations and can reduce the maximum fan power by 45.6% on average as well as energy consumption by 9.4% by predicting core temperatures.

- We analyze inter-chip and intra-chip temperature variations in two different architectures using 1,000 cores with 25°C variation. We show that decoupling the fans reduces the

variation down to 10°C, the power by additional 7.7%, and the energy by 13%. We show that the remaining 10°C is intra-chip variation and can be reduced to 2°C via thermal-aware load balancing.

5.1 Motivation and Observations

In this section, we share some observations about the cooling fans in the server nodes that set the basis of our motivation in this chapter. Cooling fans are used to reduce the temperature of hardware components in the servers by drawing cool air from outside into the server node and moving heat away from the heat sinks. The particular server nodes we study in this chapter are IBM POWER8 nodes with four individual fans at the edge of the case as illustrated in Figure 5.2. These four fans are responsible for cooling the two chips, memory buffers, and GPUs in the server node. If the temperature of any of these hardware components hits their threshold (73°C, 79°C, and 74°C, respectively), then the fans increase their speed rapidly in a *reactive* way. If all of the components are under their threshold, then fan speed is calculated based on the ambient temperature. When analyzing the fans, we observe three critical behaviors: (1) synchronous control, (2) cubic polynomial exponential increase in power, and (3) oscillations in speed.

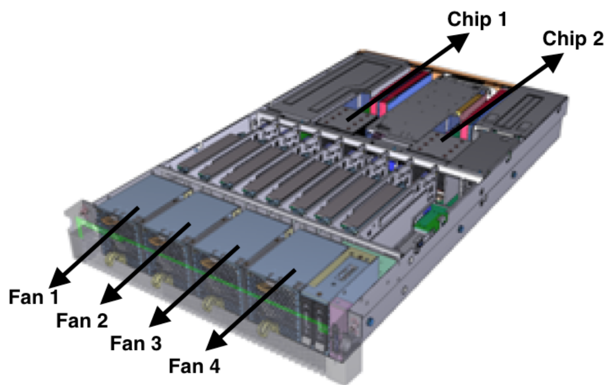


Figure 5.2: POWER8 server node architecture illustration.

Synchronous Control: The four individual fans in the node operate synchronously (i.e., they have the same speed in terms of Revolutions Per Minute (RPMs)). However, because

of the physical layout Fans 1 and 2 have more cooling effect on Chip 1 and Fans 3 and 4 have more effect on Chip 2. Figure 5.3 shows the temperature variations among the cores within a compute node. There is an 8°C difference between state temperatures of the cores with maximum being 72°C while running a balanced benchmark. The fans keep the maximum temperature of the cores right under the threshold of 73°C, however Chip 1 is over-cooled 8°C with maximum temperature of 64°C. Synchronous control of the fans can result in even larger with an imbalanced workload. For example, when the workload is running on Chip 2 and Chip 1 is left idle, then the temperature difference among the cores increases up to 25°C. When trying to cool down the active chip, the idle chip is over-cooled.

In summary, "hot" cores can cause the cooling system to activate unnecessarily and result in over-cooling surrounding hardware. "Hot" chips can have the same effect, and we analyze the temperature variations at larger scale in Section 5.3. Low core temperatures have the potential benefit of reducing fault rates [14]. In this chapter, we consider the cores under a certain threshold (i.e., fan kick-off threshold) to be equal in terms of reliability and focus our evaluation on power and energy consumption.

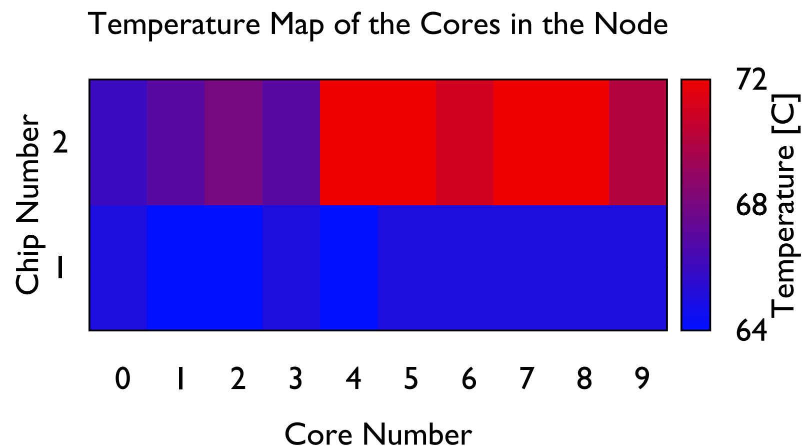


Figure 5.3: There is an 8°C steady-state temperature variation among the cores within a node running a balanced benchmark, LeanMD.

Cubic Polynomial Growth in Power: Power consumption of the fans grows cubic polynomially with respect to fan speed [64]. Figure 5.4 shows the total power of the four fans at different fan speed levels. We use simple curve fitting to calculate the model function

parameters shown in the plot. Total power can peak at over 200 W at 10,000 RPM. Even the bursts with short duration can cause significant power peaks. We show examples of such scenarios next.

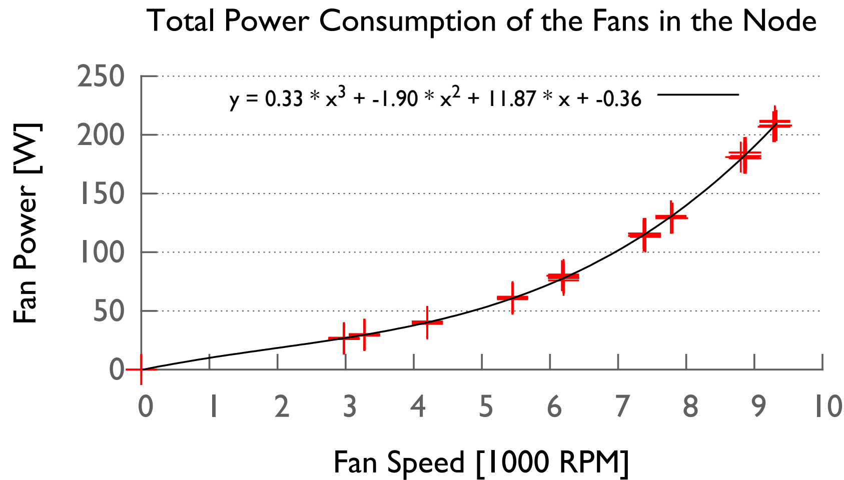


Figure 5.4: Fan power grows cubic polynomially with respect to fan speed.

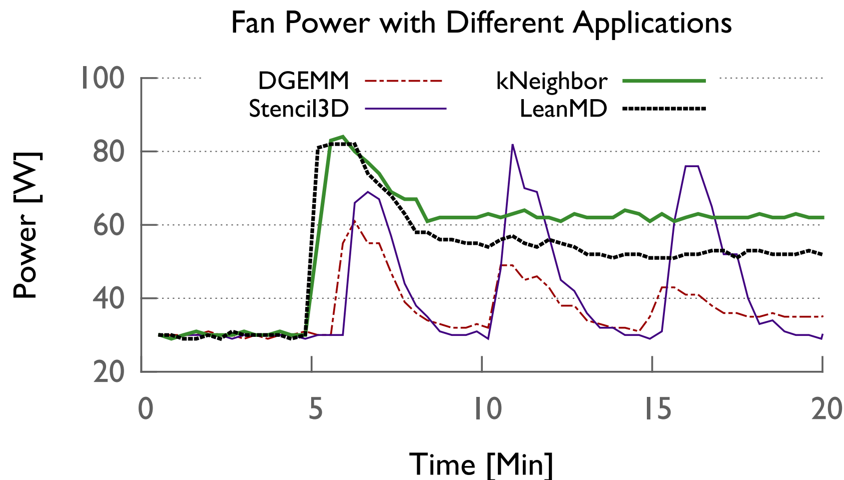


Figure 5.5: Fans show different behavior for different applications.

Oscillations in Fan Speed: The *reactive* behavior of the fans result in oscillations in fan speed and power. Figure 5.6 shows this behavior over time under different CPU utilization levels. We use a synthetic workload generator which starts at around 300 seconds. The lower utilization levels shows lower peaks in fan power, whereas higher levels cause bigger jumps.

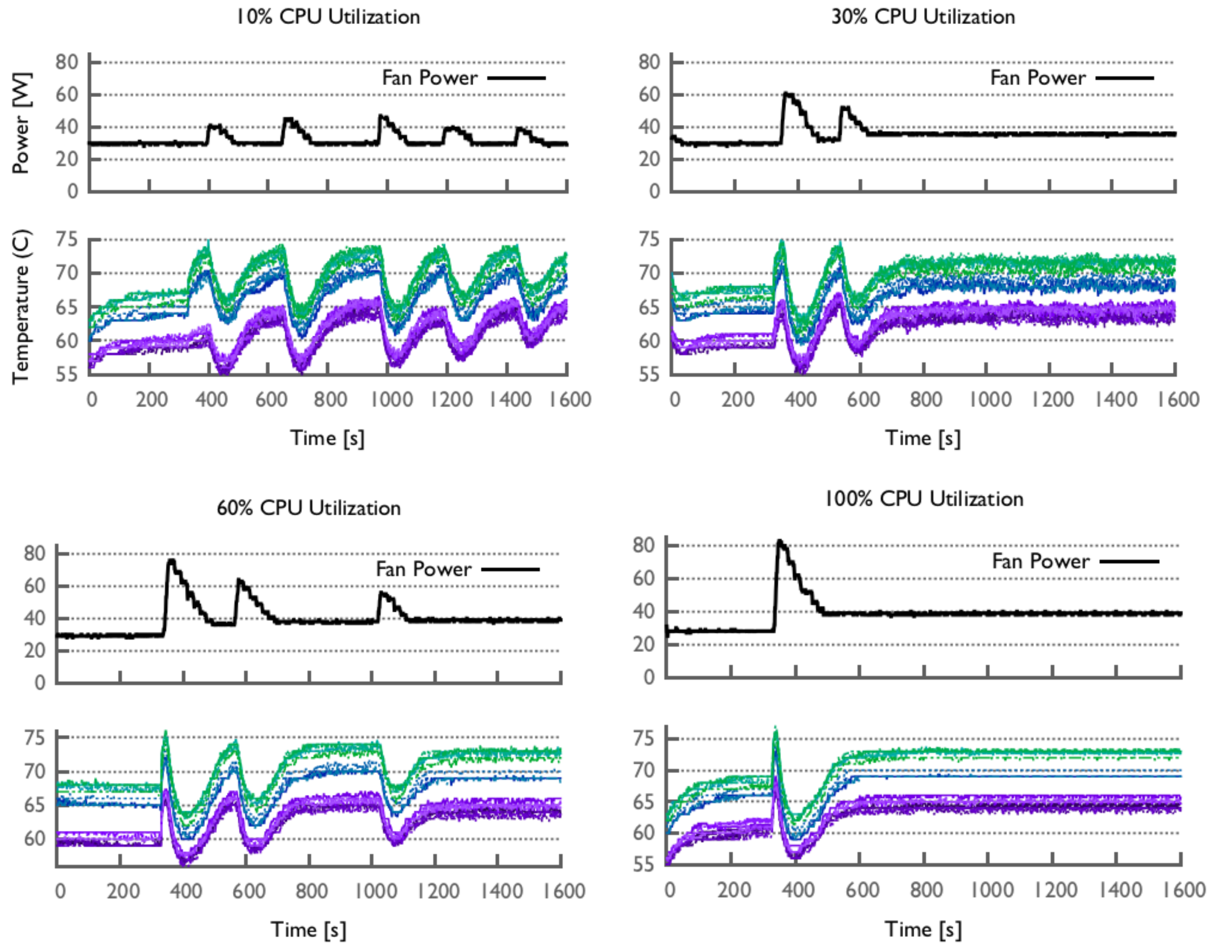


Figure 5.6: Plots show fan power (top) and core temperature (bottom) behavior from 10% to 100% CPU utilization levels. Purple and green lines in the bottom plots represent cores in Chip-1 and Chip-2 respectively.

In the 100% CPU utilization case, the total fan power peaks to 81W from idle power of 29W, and settles at 40W which is half of the maximum fan power. This behavior can differ based on the application, as it can be seen in Figure 5.5. DGEMM has the least peaks since its running on 20 threads, whereas the other benchmarks are using all 160 SMT threads on the node. kNeighbor and LeanMD benchmarks are computationally intensive and fans are able to stabilize after a single peak. On the other hand, Stencil3D is a memory intensive benchmark and fans do not seem to be able stabilize even after 15 minutes of the application running.

This oscillatory behavior has multiple downsides. First, the maximum power consumption

is an important cost metric for large data centers and since the fans settle much lower than their maximum power, the budgeted power will not be fully used. Second, the oscillations can cause fan wear and reduce hardware lifetime. Finally, making power or temperature predictions and creating power or temperature profiles for applications becomes more difficult.

5.2 Neural Network-Based Temperature Prediction Model

In this section, we describe our experimental setup and provide the details of our temperature prediction model and its validation. Our model is a proof-of-concept prototype that can do fine-grained temperature predictions for our proactive cooling mechanisms, which is described later in Section 5.3.

5.2.1 Experimental Setup

We use an IBM cluster with POWER8 processors running at 3.5 GHz nominal clock speed in our experiments. Each node in the cluster has two sockets. Each socket has 10 physical cores. Each core has 8 hardware threads. The On-Chip Controller (OCC) provides temperature and power data. The Baseboard Management Controller (BMC) provides fan speed data and fan speed control. The MATLAB Neural Network Toolbox [65] is used for the modeling. Then, the neural network parameters such as weights, biases, minimum and maximum output values are extracted, and transferred into parameters stored in the Charm++ RTS for load balancing. The ambient temperature in the cluster room measured to be between 19°C-23°C.

Benchmarks: We use a set of benchmarks that exhibit different characteristics for training and performing our experiments. These consist of compute intensive applications *DGEMM and LeanMD*, a memory intensive application *Stencil3D*, a communication intensive application *kNeighbor*, and *lookbusy* synthetic workload generator. *Stencil3D* is a 7-point stencil application based on a 3D grid using the Jacobi kernel. *DGEMM* is a naive 3-loop double precision matrix multiply code. *LeanMD* is the mini-app version of the molecular dynamics application NAMD [66]. It simulates the behavior of atoms based on the Lennard-

Jones potential in a 3D space consisting of atoms which are divided into cells with short and long-range force calculations. *kNeighbor* is a micro-benchmark with a near-neighbor communication pattern where each object exchanges fixed-sized messages with a fixed set of fourteen neighbors in every iteration. *lookbusy* generates random instructions for Linux systems and is useful for analyzing different scenarios easily as custom custom CPU utilization levels can be specified for each core [67].

5.2.2 Model Description and Validation

The computational neural network is inspired by biological neural networks to predict or approximate functions that can depend on a large number of inputs. There are two phases in using a neural network: training and deployment. During the training phase, neurons in each layer are adjusted iteratively using training data. Then the trained neurons are used to predict the new output in the deployment phase. Figure 5.1 shows our neural network design. First, input data is pre-processed because training data selection and specific input variables have a large impact on the overall accuracy of the temperature prediction. We used stabilized temperatures for our model as we would like to predict the stable state where the heat extraction is same as heat generation. Second, pre-processed input data is fed into the input layer of the neural network. The input layer consists of core utilization rate, power, fan speeds, and ambient temperature. The output layer consists of core temperatures. After the training phase, the neural network is deployed to provide core temperature predictions. The information flow of the training phase and the deployment phase is shown in Figure 5.1 as the solid blue line and dotted red line, respectively.

In a neural network, each connection between neighboring layers has a weight to scale data and a bias that allows shifting of the activation function. Data (with 9 data points) from the input layer are inserted as inputs to the next consecutive layers (hidden layers). Then the hidden layers (each with 20 nodes) sum the data fed to them, scale (weight) the data, and process it until the data reaches the last layer (output layer with 10 data points). 9 input data points includes power, core utilization and frequencies, fan speeds, and ambient temperature. The 10 output data points are core temperature as there are 10 temperature

sensors per socket. The weights and biases of one neural network are updated as follows in the training phase:

$$w_{ij}(k+1) = w_{ij}(k) - \eta \frac{\delta e_k}{\delta w_{ij}} \tag{5.1}$$

η is the learning rate parameter, which determines the rate of learning. w_{ij} represents the scalar value of weight on the connection from layer i to j. e_k represents the error of NN at the kth iteration. $\delta e_k / \delta w_{ij}$ determines the weighted search direction for this iterative method.

The weights and biases of the network are updated only after the entire training set has been applied to the network. 90% of the randomly selected data is used as a training set and 10% is used as a validation set. An entire data set was collected for different utilization rates (idle-100%), CPU frequencies (2 - 3.5 GHz), and fan speeds (3300 - 5800 RPM). The gradients calculated for each training set are added together to determine the change in weights, and biases. The weights and biases are updated in the direction of the negative gradient of the performance function. We tried different back-propagation algorithms in our neural network. *Levenberg-Marquardt* updates weight and bias values according to the Levenberg-Marquardt optimization [68]. It is often the fastest back-propagation algorithm, but it requires more memory than other algorithms. *Scaled conjugate gradient* updates weight and bias values according to the scaled conjugate gradient method [69]. *Resilient* updates weight and bias values according to the resilient back-propagation algorithm [70].

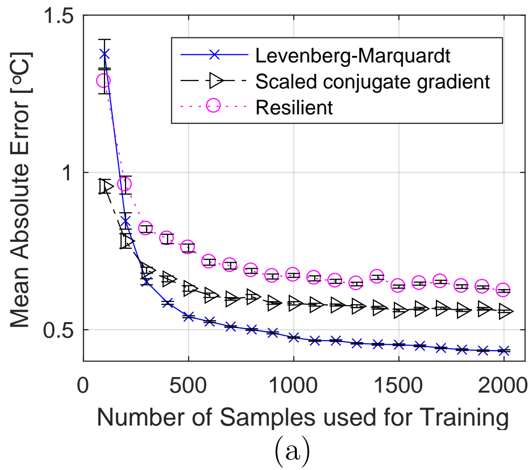
We use Newton's Law of Cooling when predicting temperature in the transient state as follows:

$$T(t) = T_p + (T_0 - T_p)e^{-kt}. \tag{5.2}$$

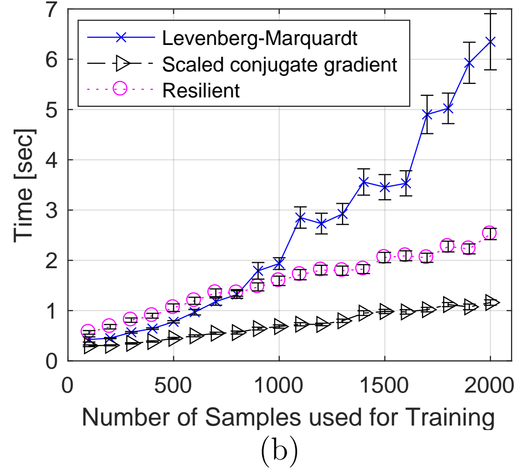
$T(t)$ represents the predicted temperature of a CPU in time t in transient state, where T_p and T_0 represent predicted next stable state temperature and current stable state temperature of the core, respectively. We assume positive constant k does not change for a specific type of CPU and fan speed, and the ambient temperature does not change either during the heating/cooling phase.

Figure 5.7 (a) and (d) shows the Mean Absolute Error (MAE) of using different back propagation algorithms and using different number of neurons, respectively. MAE is a good indicator of predictor performance and tends to decrease with the number of training samples. Our neural network becomes more knowledgeable about the relationship between temperature changes and resource allocation as it processes more training samples. The Levenberg-Marquardt algorithm performs best in terms of accuracy (showing minimum mean error) but has more computational overhead (showing higher execution time for training) with a larger amount of neurons. Also, Figure 5.7 (c) shows MAE is consistent with all the cores.

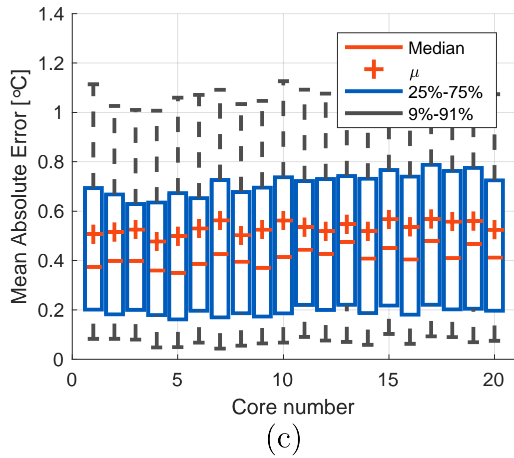
The time elapsed for training is shown in the Figure 5.7 (b) and (e). Scaled conjugate gradient performs best in terms of computational overhead showing less time for training than the other algorithms. Because using 20 neurons for the hidden layer does not improve model accuracy, we used 15 neurons for the hidden layer. We used the Levenberg-Marquardt back-propagation algorithm because our highest priority is model accuracy. However, scaled conjugate gradient, or resilient algorithms can be used if learning overhead is a constraint. We verified the accuracy of neural network approach by repeating the experiment and comparing the predicted core temperature over time with the actual core temperature as shown in Figure 5.7 (f). Our neural network will be able to make accurate predictions despite slight hardware differences, variable heat, and air circulation patterns (thermodynamic phenomena) of different nodes and, furthermore, regions inside a data center.



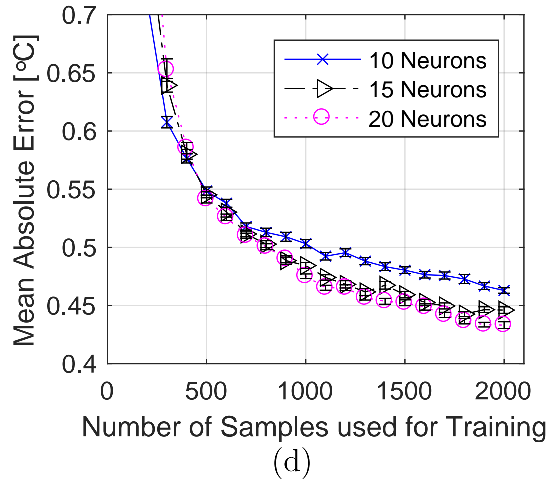
(a)



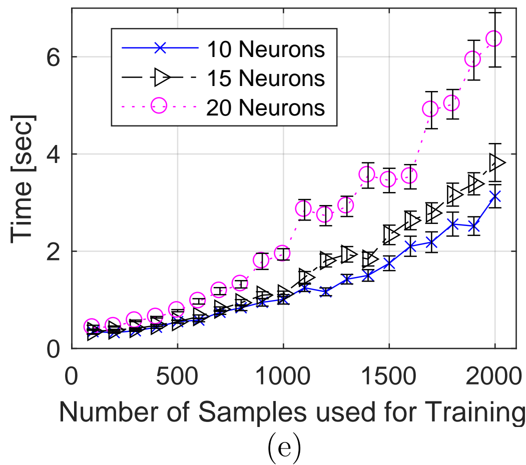
(b)



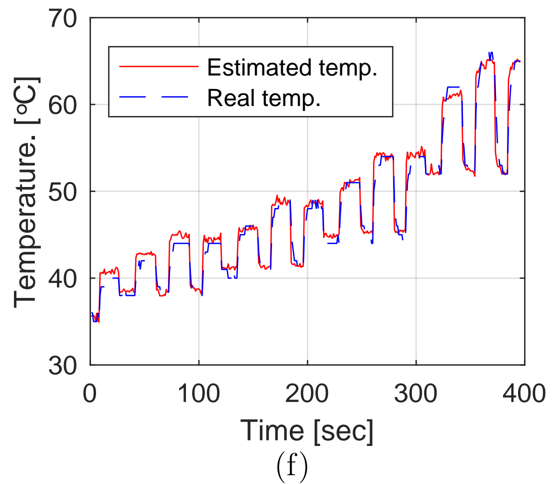
(c)



(d)



(e)



(f)

Figure 5.7: (a) Mean absolute error using different back-propagation algorithms; (b) Training time using different back-propagation algorithms; (c) Mean absolute error per core using Levenberg-Marquardt algorithm; (d) Mean absolute error using different number of neurons; (e) Training time using different number of neurons; and (f) Model validation while increasing CPU frequency. Plots are shown with 95% confidence interval.

5.3 Model-Guided Proactive Fan Control

The temperature prediction model can be used to implement proactive cooling mechanisms that mitigate the problems described earlier in Section 5.1. These problems include temperature variations and reactive and oscillatory behavior in fan speed. In this section, we propose a preemptive fan control mechanism and compare it with two other mechanisms including the existing reactive control mechanism. Below are the fan control mechanisms we compare:

1) **Reactive Fan Control (Default Mode):** This is the existing method used in the server nodes. As we described in the Section (5.1), the fan speed is determined from core and ambient temperatures. If any core exceeds the temperature threshold of 73°C, then the fans increase their speed to cool down the chips. The floor fan speed, i.e., when all components are under their threshold, is determined by the ambient temperature. This approach consumes minimal power if the components are under the threshold. However, once the cores hit the threshold, then the fans react rapidly. This results in peaks in power consumption and in certain scenarios such as compute-intensive workloads where the fan power can triple. The fans can also be triggered by other components such as GPUs or memory hitting their respective threshold. In our experiments, CPU core temperatures are the sole reason for fans to trigger.

2) **Constant Fan Control:** As the name implies, this technique uses a constant fan speed regardless of the processor utilization or the benchmark running. This constant speed is sufficient to cool down the most compute-intensive benchmark, therefore it guarantees the components will operate under the temperature threshold. This constant fan speed can be determined by running a synthetic benchmark which fully utilizes the CPU or the most compute-intensive benchmark that is projected to run, and measuring the fan speed that can keep core temperatures under the threshold. Because this is the worst case, any other benchmark will be guaranteed to operate under the temperature limit. There are advantages and disadvantages with this technique. The advantages are the removal of the oscillatory behavior and a potential reduction in the maximum fan power. The disadvantage is the

increased overall power and energy consumption, because the cores will be over-cooled when they are idle or running a non-intensive benchmark.

3) **Preemptive Fan Control:** This is our model-guided fan control mechanism that preemptively cools the processor before the cores hit the threshold. We also refer to it as *precooling*. The idea is conceptually similar to *prefetching* where the data is brought from memory before it is used. In *precooling*, the processor is cooled before the application starts or the fan speed speed is adjusted for different phases of the application, i.e., before a computationally intensive phase. The *precooling* speed is determined from the temperature prediction model using some hints about the application profile as we described in Section 5.2.2. The model determines the steady-state temperatures under different fan speeds. Then, the desired speed can be set via the job scheduler of the cluster or the runtime system of the application. The job scheduler can set the *precooling* speed before the application starts. As we showed earlier, in the most cases fans are able to find a stable level after one or a few peaks. A job scheduler approach would eliminate that initial peak for most scenarios. At the end of the application, the job scheduler sets the speed back to the optimal idle speed. On the other hand, runtime systems can do more fine-grained fan optimization for applications that have different phases. The runtime system can detect the start of a different phase and determine a new optimal fan speed. The phase could be either transitioning from non-CPU-intensive to high-intensive or vice versa.

In Figure 5.8, we compare the fan power and the core temperatures under these three cooling control mechanisms using the LeanMD benchmark. At the start, when the cores are idle, the reactive and preemptive control mechanisms have the same speed and temperatures, whereas cores under constant fan control are much cooler since fan runs at a higher speed even when the processor is idle. Before the application start at 100s, preemptive control starts the cooling process and the fan speed is set to the stable level of the benchmark using the temperature prediction model. In the reactive mechanism, the fans start after the maximum core temperature hit the threshold, which is around 15 seconds after the application start. Only after 300 seconds (5 minutes) from the application start, the fans gradually lower their speed and stabilize at 4900 RPM and 51W. Preemptive and reactive

mechanisms converge to the same temperature and power level at around 400s. On the other hand, constant fan control shows 5°C lower maximum core temperature. Since we determine the constant speed for the worst case benchmark in our benchmark set (which is kNeighbor), it causes over-cooling for the LeanMD benchmark.

Table 5.1: Peak fan power and energy consumption of different fan control mechanisms and benchmarks.

	Reactive	**Preemptive**	**Constant**
DGEMM	61W, 19.4kJ	38W, 17.7kJ	63W, 31.3kJ
Stencil3D	81W, 21.0kJ	42W, 18.7kJ	63W, 31.3kJ
kNeighbor	84W, 29.7kJ	63W, 27.7kJ	63W, 31.3kJ
LeanMD	83W, 26.8kJ	51W, 23.7kJ	63W, 31.3kJ

Table 5.1 shows the maximum power and total energy consumption of the three mechanisms in a 500-second window where the application starts at 100s as shown in Figure 5.8. Overall, the preemptive mechanism reduces the peak fan power by 48% with DGEMM, 50% with Stencil3D, 25% with kNeighbor, and 39% with LeandMD compared to the reactive mechanism. Constant fan control is able to reduce the peak fan power at most by 25%, but it increases the energy consumption by 61% which is quite significant and undesirable. In short, preemptive cooling mechanism provides the minimum power and energy in all cases. We have also observed reduction in chip's maximum power consumption due to the reduction in the peak temperature of the cores in preemptive and constant fan control. However, because the reduction is not significant (just a few watts), we are not including it in our results.

After showing the benefits on one node, next we look at the fan behavior on 90 nodes and how much preemptive cooling can help at large scale. Figure 5.9 shows the maximum and stable fan power of 90 nodes running the LeanMD benchmark. The difference between maximum and stable power is the savings that preemptive cooling can provide for each node. The reason that each node has a different fan power despite running the same benchmark is because of the temperature variations across the nodes. We will provide a temperature analysis and address this issue later in this section.

When to start *precooling?*

A critical factor in *precooling* is to determine when precooling should start. Is it before the application starts, when the application starts, or even after the application starts? How many seconds before or after the application starts? When the CPU is transitioning from a low-utilization state to high-utilization state, it is important to have sufficient time to cool the cores enough. This is not as critical when the CPU is transitioning from a high-utilization state to low-utilization state, since the fans are going to be set to a lower level and delay in setting the new level will not harm the cores but only cause minimal energy loss when compared with the optimal case. Figure 5.10 shows that precooling needs to start within a few seconds after the application start at the latest. The window can be extended by a few seconds more if the application is not as intense and the temperatures are not increasing rapidly.

So far, we have shown how preemptive fan control can remove fan oscillations and power peaks. Next, we are going to analyze and mitigate the temperature variation problem by decoupling the fans.

Decoupling the Fans

The four fans in the server node change their speed synchronously, however each of the individual fans has a different cooling effect on the two chips because of the physical layout, which can be seen in earlier Figure 5.2. The fans are more effective in cooling closer chips. If the fans are decoupled (i.e., controlled independently), then temperature variations can be reduced and the fans could consume less power and energy. To evaluate the potential benefits of decoupling the fans, we first analyze temperature variations in large scale. Figure 5.11 shows temperature distribution of 1,800 cores in two different platforms: Cori and Minsky. Cori has Intel Haswell generation processors with 36 cores per node, and Minsky has IBM POWER8 processors with 20 cores per node. Our goal is not to compare the cooling efficiency of one platform to another. We are simply showing the significance of the temperature variation problem and that it is not unique to a specific platform. The rest of our experiments are done solely on Minsky. The maximum core-to-core temperature difference is 22°C for Cori and 25°C for Minsky. Figure 5.12 shows the temperature distribution of the cores on Minsky using three other benchmarks. While it may seem a surprise that all of the

Table 5.2: Decoupling the left two and the right two fans reduces chip-to-chip temperature variations, power, and energy consumption.

LeanMD	**Synchronous**	**Decoupled**
Fan Speeds (RPM)	4900, 4900	3900, 4900
Max Temp Chip 1, 2(C)	63, 71	70, 71
Temp Variation(C)	8	5
Power (W)	51	44
Energy (kJ)	23.7	20.6

benchmarks show a similar temperature distribution, there are two main reasons this. First, all of the benchmarks use the cores in a balanced manner. Second, all of the benchmarks exceed the fan kick-off threshold, therefore the fans stabilize the core temperatures optimally just below the threshold. So, it is expected to see a similar temperature distribution.

The 25°C variation on Minsky can be partially prevented by decoupling the fans. We have observed that the left two fans affect Chip 1 and the right two fans affect Chip 2 most significantly. The effect on the diagonal chips are minimal. Therefore, we decouple the control of the left two fans and right two fans from each other and evaluate potential power reductions. Figure 5.13 shows the new temperature distribution of the cores after fans are decoupled. The maximum temperature variation is reduced from 25°C to 12°C with the lowest temperature of 64°C. The remaining variation is intra-chip variation that fan decoupling cannot resolve. Figure 5.14 shows the distribution of intra-chip temperature variation. While the majority of the chips have less than 6°C temperature difference among their cores, the range can go up to 10°C. We analyze ways to mitigate the intra-chip temperature variation, and the potential benefits in the next section (5.4).

Table 5.3: Fan Power Reduction in Large Scale

Optimization \ **Benchmarks**	DGEMM	Stencil3D	kNeighbor	LeanMD	Average
Reactive Fan Control	5868 W	13433 W	6769 W	6770 W	8210 W
Preemptive Fan Control	3893 W	8526 W	4381 W	4224 W	5256 W
Preemptive and Decoupled Fan Control	3179 W	7972 W	3765 W	3569 W	4621 W
Total Power Reduction (%)	45.8	59.3	55.6	52.7	53.3

Table 5.2 shows various node parameters when the fans are decoupled. First, the left two

fans are now running at 1,000 RPM less, and this results in 6W reduction in the stable fan power. Power reduction also results in less energy consumption. The maximum core temperature in Chip 1 is increased, but it is still kept under the threshold. In-node temperature variation reduces from 8 to 5°C. The remaining 5°C temperature variation is intra-chip variation that decoupling cannot mitigate.

To summarize, Table 5.3 shows the cumulative power consumption of the 90 nodes under reactive control, preemptive control, and decoupled control. On average, preemptive and decoupled fan control methods can reduce the maximum power by 53.3% (45.6%, 7.7% respectively) and energy by 22.4% (9.4%, 13% respectively).

Safety of Model-Based Cooling Control

Machine learning models can be error prone, therefore using a learning mechanism for a safety-critical feature such as temperature control requires fail-safe mechanisms. There can be different reasons for errors in learning algorithms including noise, estimation bias, and variance (commonly known as over-fitting and under-fitting). If the model predicts the temperature lower than it is going to be and the cores exceeds a certain level, then the fall-back mechanisms need to take control quickly. To handle such scenarios, we can set a maximum threshold level lower than hardware controller's level to have a safety margin in case there are 2-3°C errors in the prediction. The system can still continue running while continuing to train the model to make better predictions in the future. If the error is more than that, then hardware controllers can take over the control in their usual way and override the model-guided mechanism.

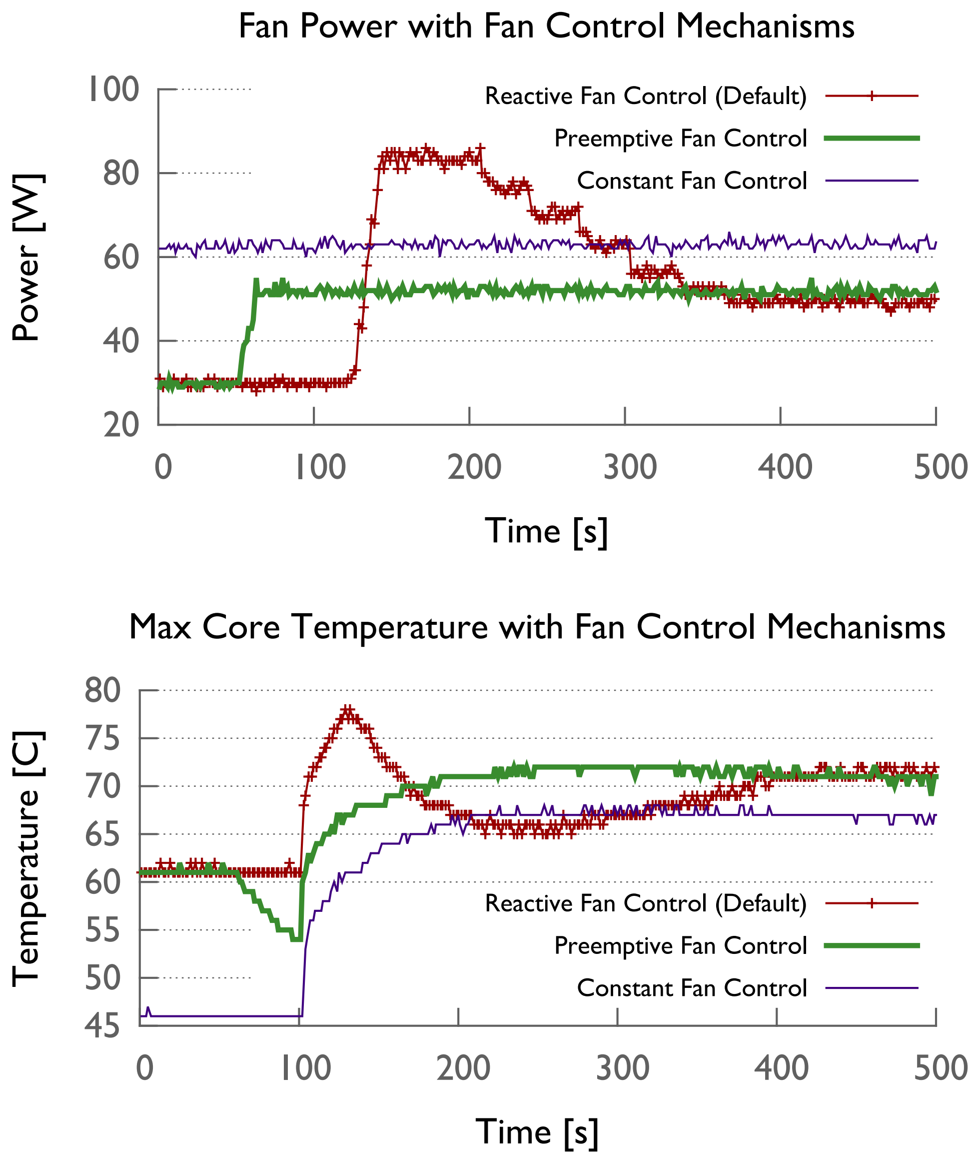


Figure 5.8: Power and temperature comparison of the three fan control mechanisms with *LeanMD* benchmark starting at 100s. (Preemptive cooling has been started 50s ahead of time for clarification of the plots and the idea. It can achieve a similar effect if triggered within few seconds of the application start as well.)

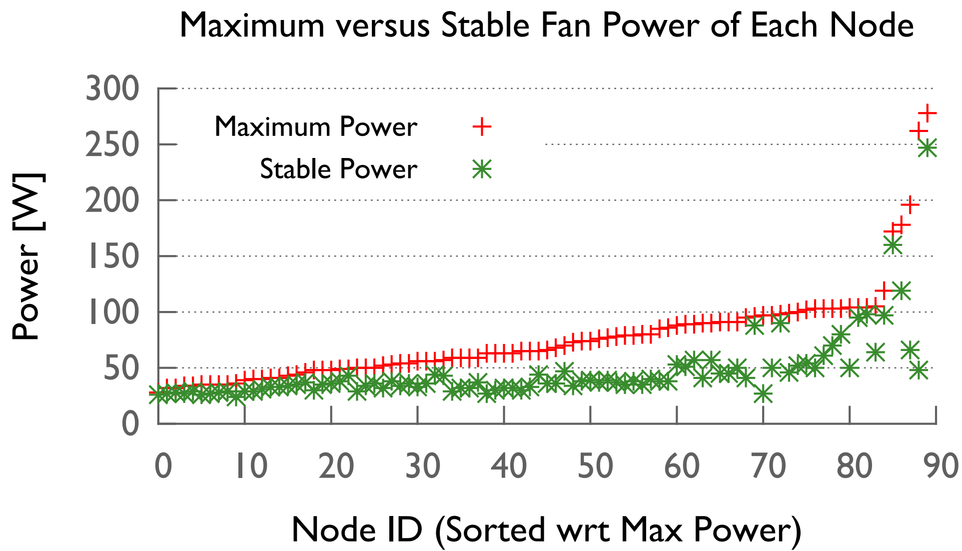


Figure 5.9: The difference between the maximum and the stable fan gives the power reduction that preemptive cooling can achieve with LeanMD.

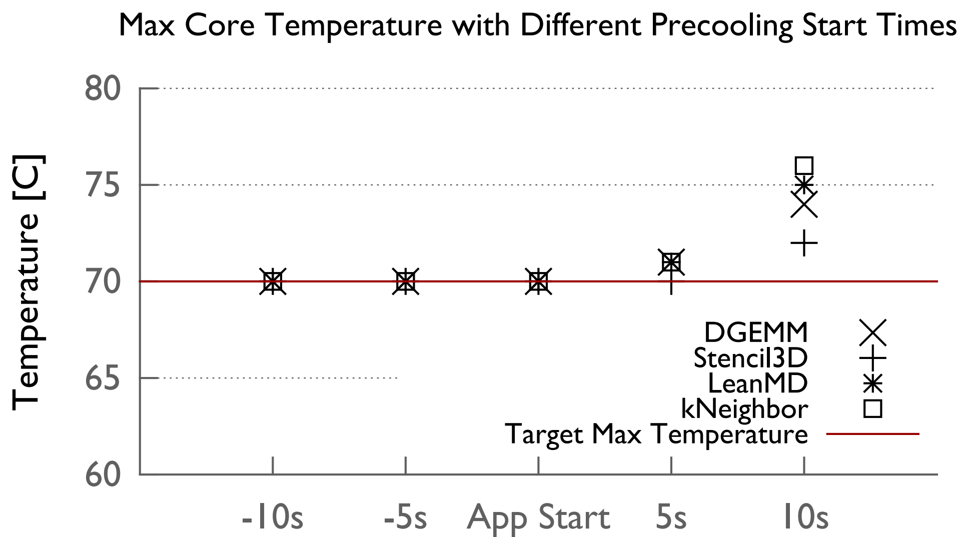


Figure 5.10: Precooling should start within a few seconds of application start at the latest.

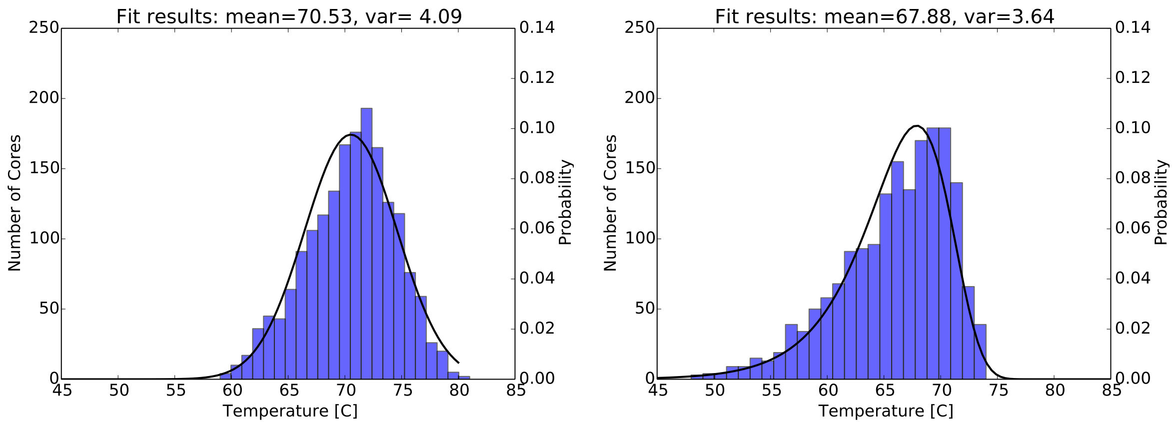


Figure 5.11: Steady-state temperature distribution of 1,800 cores running DGEMM kernel in two different architectures: Cori (left) with Intel Xeon Haswell processors and Minsky (right) with IBM POWER8 processors.

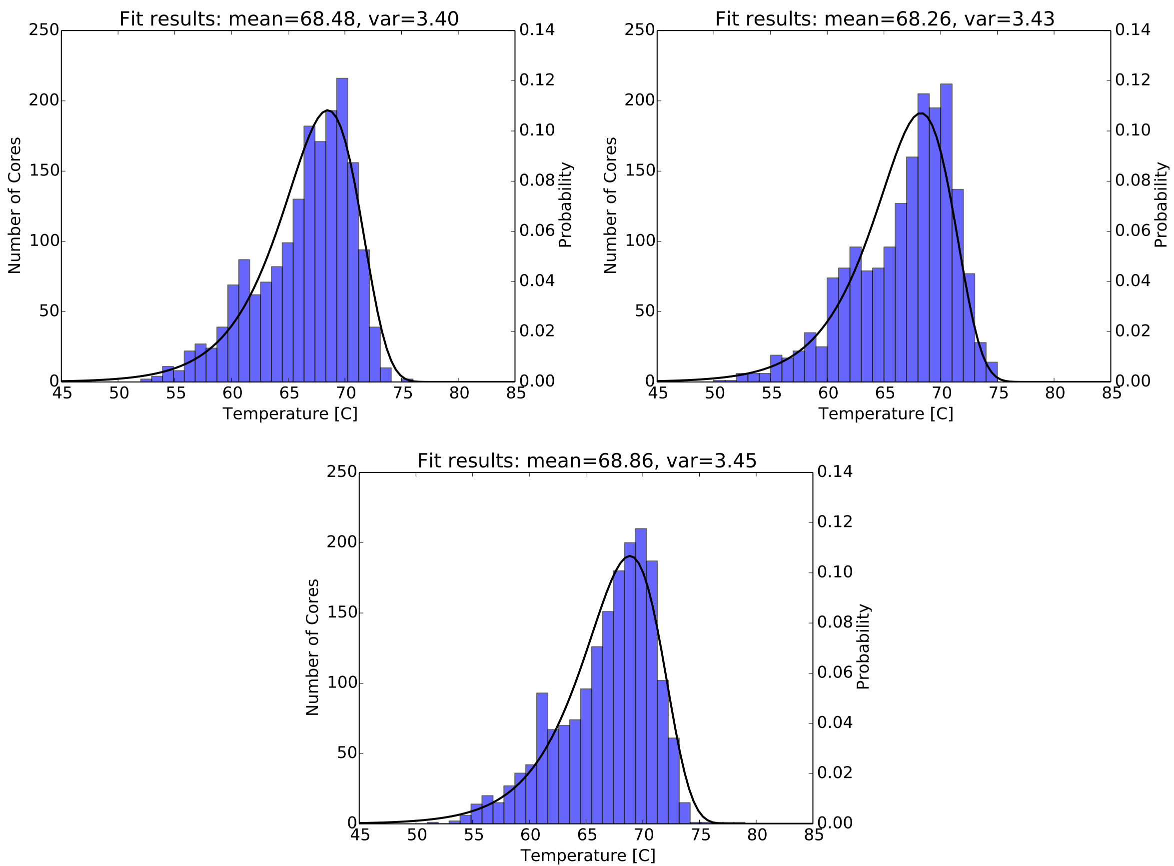


Figure 5.12: Steady-state temperature distribution of different applications on Minsky: kNeigbor, LeanMD, Stencil3D (in order). Despite their different characteristics, applications show almost the same temperature distribution when the cores are used in balance.

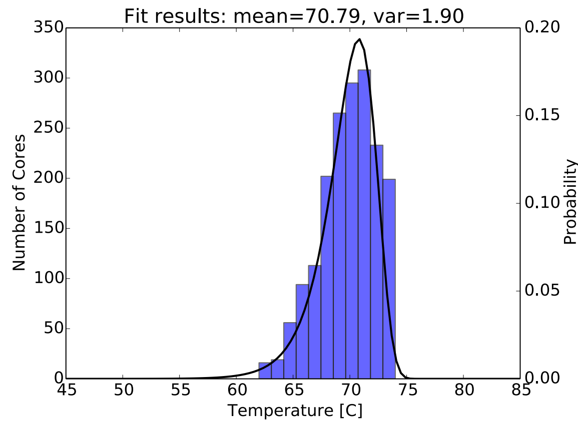


Figure 5.13: The top plot shows steady-state temperature distribution of the DGEMM benchmark with decoupled fans on Minsky. (Notice the range change in the y-axis.) Independent fan control cannot remove temperature variations fully overall, because of the intra-chip variations.

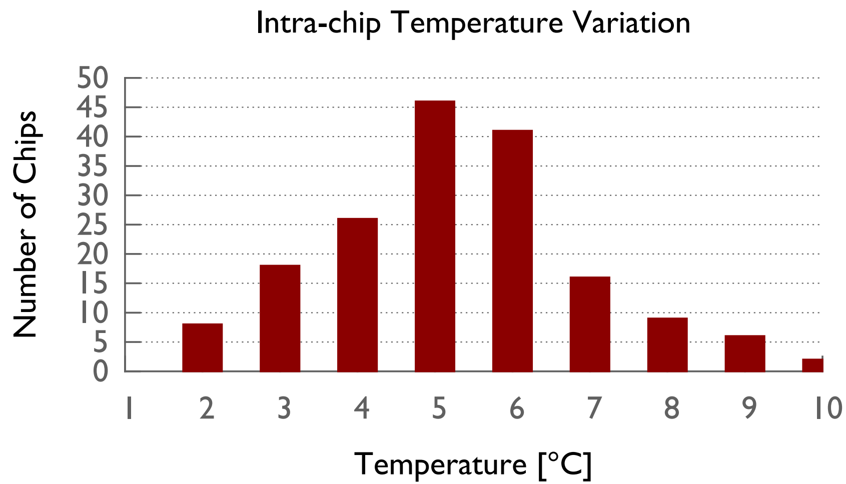


Figure 5.14: The distribution of intra-chip temperature variation among cores.

5.4 Mitigating Intra-Chip Temperature Variation

In this section, we investigate methods to solve intra-chip temperature variation problem that preemptive and independent fan control could not mitigate fully. Temperature-aware DVFS and load balancing are two potential solutions:

1) Temperature-Aware Frequency Control: The DVFS technique is commonly used in power and energy optimization and it often causes performance overhead that is undesired by HPC users. It can be used to mitigate the temperature variations and to do thermal-aware throttling as well. DVFS can be done in core-level or chip level, while the core-level one allows to provide finer-grained control. Most commercial processor architectures only support chip-level DVFS. Unlike many others, POWER8 chips do have per-core voltage regulators and support per-core DVFS [71], however it is not supported in production. For Intel processors, only Haswell generation supports per-core DVFS and the support has been discontinued for future generations [72].

Our temperature prediction model can be used to predict the temperatures of the cores after DFVS is applied. During the runtime, if any of the core temperatures are predicted to hit a certain threshold, then a new frequency can be determined and set. Chip-level DVFS can remove chip level temperature variations; however, it is ineffective in mitigating core level temperature variations. Therefore, we conclude, since core-level DVFS is not currently supported in many processors yet, this is not a viable solution. We look into load balancing next.

2) Temperature-Aware Load Balancing: Load balancing can be used to eliminate within chip temperature variations by moving the work units away from the hot cores. We implemented a new load balancing algorithm using Charm++'s load balancing framework. In Charm++, the workload is represented as C++ objects which can migrate from processor to processor [17]. In our algorithm, the runtime system moves objects from high-temperature (above average) cores to low-temperature (below average) cores in order to create temperature balance. The neural network model predicts core temperatures of potential object migration and it allows us to determine the best workload distribution. Since the neural network uses CPU utilization as input, we need to approximate the load of each

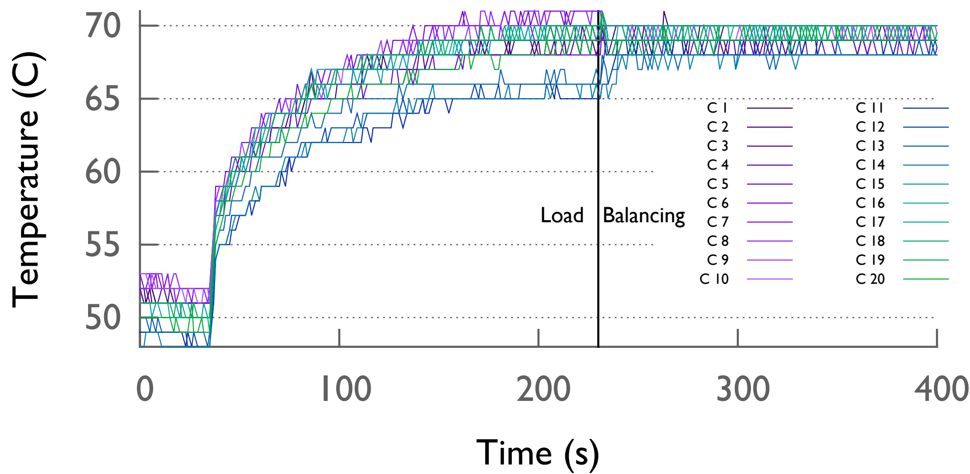


Figure 5.15: Load balancing reduces within chip temperature variations from 5°C to 2°C. (Inter-chip variation is mitigated by decoupled fans.)

task in terms of CPU utilization. We do this by calculating the ratio of a task's execution time and the total execution time of the tasks on the same processing core from historical information stored in Charm++'s runtime database.

Figure 5.15 shows how core temperatures change over time with load balancing triggered around 220s. It can be triggered earlier as well, it's done after the cores heat up only to show the difference between before and after load balancing temperatures clearly in the plot. Load balancing reduces the temperature variation from 5°C to 2°C, and the maximum core temperature from 71°C to 70°C. Around 5% of the total objects (or task units) needed to be migrated to achieve the balance. This results in a 7% performance overhead since the applications balance is distorted in order to create the temperature balance. Doing temperature balancing can allow the preemptive fan speed to be set at a higher level. In this particular case, a reduction of 1°C in the maximum temperature only gives minimal benefit in terms of the fan power. Load balancing may be beneficial only if the temperature difference is greater than or equal to 6°C. That still represents a significant portion of the chips, to be specific 41%. As we showed earlier in Figure 5.14, within-chip temperature variation can peak up to 10°C . For those chips that has an intra-chip variation greater than

6°C, the average difference between the maximum core temperature and the average core temperature is 3°C. 3°C reduction in maximum temperature in those chips reduces the peak fan power further by 3.3% and hence the average reduction in fan power becomes 56.6% compared to reactive fan control. This strategy comes at a cost of performance overhead of up to 10%. Given its marginal benefit and its performance overhead, this load balancing method may not be advisable if the CPU is at peak load like in our case. If the CPU is not fully utilized to their capacity (i.e., when the application application behavior creates the imbalance), then such thermal-aware mechanisms could be more useful. For example, when running at large scale, some applications use less cores than the number of physical cores in the node because of running out of memory. For a second example, some applications require to launch on power of two number of processes for performance reasons and some cores may have to be left idle. We have considered the worst-case scenario in our experiments.

5.5 Related Work

The neural network modeling approach has been gaining popularity in multiple areas of data center resource management. It has been used to predict performance of parallel applications [73, 74] in presence of multiple tunable parameters. Tiwari et al. uses neural networks to do power and energy modeling of HPC kernels with different code variants [75]. Moore et al. used neural networks to build an online thermal mapping, and management system for data centers [76, 77]. This prediction approach is coarse-grained (i.e., at the level of one or more servers), and it cannot predict individual core temperatures. Duy et al. used neural networks to predict user demand from the usage history to be able to turn servers on or off with their "green" scheduling algorithm for cloud data centers [60]. Neural networks have also been used to improve job scheduling decisions by predicting resource status (i.e., the load of the data center based on history information) [78]. Another study uses neural networks to predict the core, and Network-on-Chip component temperatures of the chip, and the predictions are used for energy efficient data exchange [79]. However, they heavily rely on the simulated data to evaluate their model, and their neural network models are not thoroughly validated in the literature. Other machine learning models have also been used

to predict data center temperatures [80]. However, this work does not incorporate server fan speed data in their model, and hence when the fans are triggered, accuracy of their temperature prediction model can drop significantly. Moreover, neural networks have shown many advantages over statistical methods, due to their capability in handling nonlinear behavior. Neural networks have been applied to many other areas, such as stock markets, pattern recognition, data mining, and medical diagnosis. Still, other prediction models such as least-square fitting model [81] can also be used to do proactive cooling mechanisms proposed in this paper.

Recent research supports the existence of thermal and manufacturing variations in supercomputing systems [6, 41]. Thermal-aware job scheduling strategies have been proposed to cope with this problem. However, because variations can be observed even within one node, workload scheduling strategies would not be sufficient to address the temperature variations, especially for large-scale HPC applications. Moreover, HPC applications may have communication patterns that allow them to benefit from topology-aware mapping strategies [82]. Job topology requirements can conflict with thermal-aware job scheduling techniques. Therefore, we propose a runtime-based technique for thermal-aware task scheduling that is less intrusive than job-level scheduling, allowing the job to continue to specify topology constraints. Only a small fraction of the tasks will be mapped to a location other than their "best" location.

Past research in thermal-aware load balancing studied reactive techniques, where the runtime system constantly monitors core temperatures and makes decisions based on the readings. For example, Sarood et al. proposed a temperature-restraining load balancing algorithm [13,61]. In this work, the runtime system applies DVFS to the processors that are exceeding a threshold temperature. However, because each processor can potentially have a different frequency level, a load imbalance could be created among processors. Then the runtime system applies load balancing to compensate for the disparate frequencies. This approach requires the system to monitor temperatures frequently and apply DVFS empirically when temperatures exceed a threshold. The frequency settings are found by trial-and-error. Another limitation of the work is that the object migrations are done without taking into account the temperature changes of the host and donor processors after load balancing. This will again result in the need for frequent temperature tracking and frequency control. In

our approach, we use neural networks to guide load balancing decisions. A neural network can precisely predict core temperatures for given workloads and for different workload-to-processor mappings. Therefore, it eliminates the need of monitoring core temperatures frequently and replaces empirical decision making with precise model-based decisions. The model can predict how the temperatures of the cores are going to be after doing load balancing. Another difference in our approach is that while our load balancer aims to balance the core temperatures, it does not try to bring the core temperatures under a certain temperature threshold.

A forward-looking fan control mechanism, where the system uses system utilization information to predict temperatures and controls the fan to dampen power peaks has been patented [83]. However, the effectiveness of the mechanism has not been published.

5.6 Summary

In this paper, we first analyze inefficiencies in air-cooling systems such as oscillations in fan speed and temperature variations. We show how proactive cooling mechanisms can be used to mitigate these inefficiencies and reduce cooling fan power by 53.3% and energy by 22.4% without any performance overhead. These solutions cannot be used in production without an accurate temperature prediction model. Yet, thermal-aware methods are often applied greedily. In this work, as a proof-of-concept model, we use a neural-network based approach that can predict steady-state core temperatures under different workloads, frequencies, and fan speed levels. We use the guidance from the model in designing our proactive cooling methods.

CHAPTER 6

Mitigating Across Component Power Variation

After finding solutions to mitigate frequency and temperature variations in chapters 4 and 5, this chapter proposes solutions to mitigate power variations. HPC architectures are beginning to have wide compute nodes with different components. For example, a single physical node in SummitDev supercomputer at ORNL and Sierra supercomputer (to be built in Lawrence Livermore National Laboratory) will contain two CPUs, four GPUs, two memory units, 2 network adapters [84]. A illustration of a compute node on SummitDev is shown in Figure 6.1. So far, we have solely shown the intrinsic manufacturing differences among CPUs. However, these differences causing power variations also exist in GPUs, memory components, network cards or disks. Yet, the assembly of the nodes are done randomly without taking into account these variations. Power aware node assembly technique can help mitigate the power variations and the side effects of power variations. In this chapter, we first show the power variation of different components within a node. Later we propose and analyze the feasibility of three different power aware physical node assembly techniques and compare it with the currently practiced method of random assembly. These three new assembly techniques are:

1. Categorized assembly
2. Application characteristics aware assembly
3. Balanced power node assembly

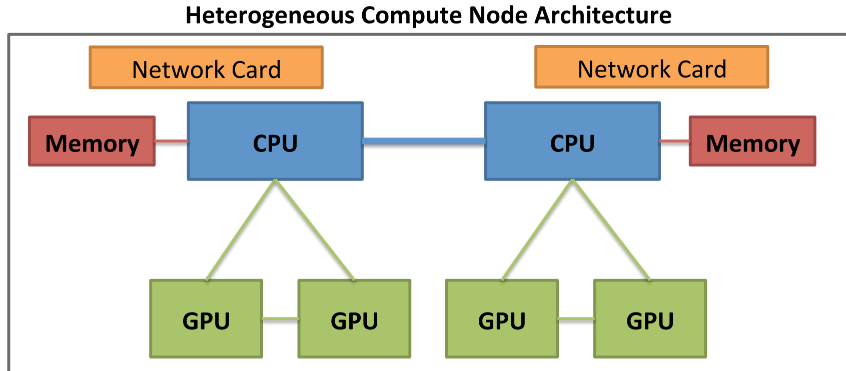


Figure 6.1: Node architecture of ORNL SummitDev Supercomputer with IBM Power8 CPUs, NVIDIA Tesla P100 GPUs, DDR4 memory and Mellanox EDR Infiniband network adapter.

Node assembly should be accompanied by a power-efficient job scheduler that takes into account the power variations of the individual hardware components within the node and the application characteristics.

The ideas proposed in this chapter are patent-pending [10, 85].

6.1 Understanding the Sources of the Power Variation

The CMOS transistor sizes becoming smaller and the threshold voltages becoming lower over time are two major causes of manufacturing related process variations. These process variations cause yield loss; the processors that do not satisfy the performance and power requirements have to be thrown out [86]. Even the ones that make it to production have variability in their power and operational frequency as we have shown in the previous chapters.

The are two main types of variations in CMOS very-large-scale integration (VLSI) process: variations in gate delays and leakage current. Both of these have required limits (i.e., the processor needs to sustain a frequency limit and it should not exceed a power limit, known as TDP). A processor might fail in satisfying one or both of the requirements as illustrated in Figure 6.2. Variations in gate delays cause variations in dynamic power of the chip (i.e., the power dissipation due to charging and discharging of load capacitance). Variations in leakage current cause variations in static power (i.e., power dissipation due to sub-threshold

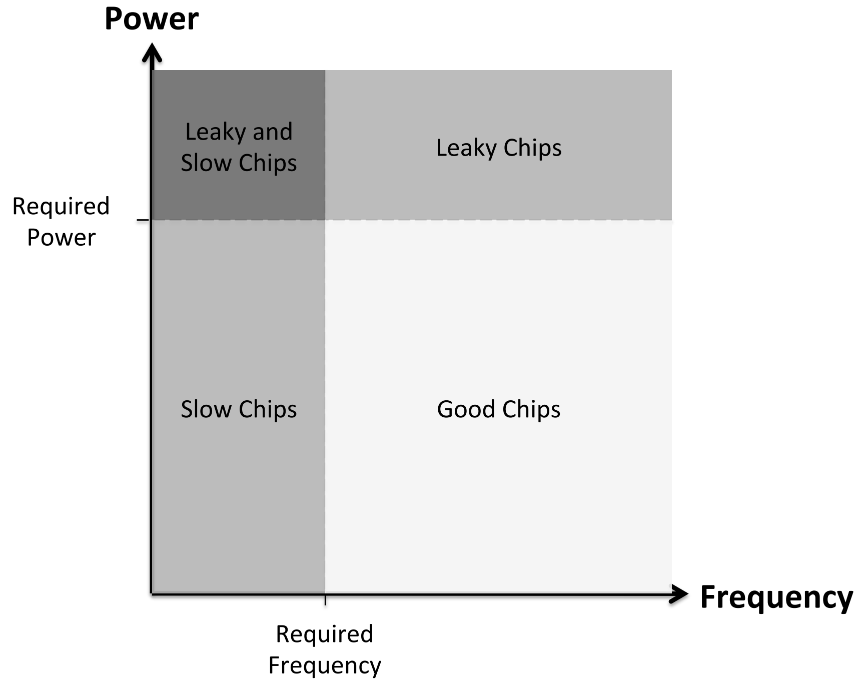


Figure 6.2: Power and frequency characterization of a manufacturing yield. Note that the distribution ratios in the figure do not represent actual numbers.

leakage). There is a third source of power consumption in chips, which is the short-circuit power dissipation from VDD to ground however it constitutes less than 10% of the total power and therefore it is not considered significant [87, 88].

It has been shown that the variation can be up to 30% in frequency and 20x in leakage power [89]. Such high variation in the manufacturing process with leaky and slow chips have introduced the concept of *Voltage binning*. *Voltage binning* refers to the technique of adjusting the voltage of the chip to satisfy the power or performance of the chip since changing the voltage of the chip effects both the performance and the power of the chip [90]. With this technique, some of the leaky or slow chips can be converted into good chips by respectively lowering or raising their supply voltage. Since leaky chips require lower and slow chips require higher voltage levels, the chips that are both leaky and slow have to be thrown out. Different binning techniques have been proposed to increase the yield and the profit of the chips [90, 91]. However, node level variations, where node components with each having different power variations assembled together, have not been yet studied in the literature.

6.2 Power Variation Analysis of Node Components

In this section, we analyze the power variation in different components of the nodes. Note that this data is collected from a production cluster with already-binned processors. In this chapter, we use Minsky platform located at the IBM T.J. Watson Research Center. Minsky is the smaller scale version of to SummitDev and Sierra supercomputers with the same node architecture as shown earlier in Figure 6.1.

In previous section, we identified the two independent sources of power consumption known as static and dynamic power consumption. Therefore, they need to analyzed independently. Figure 6.3 shows the static power distribution of the node components. The data represents the power of the components collected when they are idle. Chips show 13.9%, memory units show 25%, GPUs show 20%, and nodes show 23.5% variation in their idle power. The difference between maximum and minimum power consumption in the components are 139 W, 30 W, 20 W, 235 W respectively.

Figure 6.5 shows the active power of the chips running four different benchmarks. They all show a form of normal distribution. The power variation is 28% for DGEMM, 16% for KNeighbor, 20% for Stencil3D and LeanMD benchmarks. Although the power distribution of the applications are shifted or slightly different, the chips that consume high power versus low power are consistent throughout different benchmarks with the R-Squared correlation coefficient between 0.65 and 0.72. Note that this would not be the case if the applications had large load imbalances between the chips.

Next, we look at the correlation between idle and active power consumption of the chips. Figure 6.4 shows there is a weak correlation between them, with R-Squared value of 0.375. The average active power of the chips are 244.6 W, whereas the average idle chip power is 191.4 W. It is not straightforward to calculate the dynamic power from the active power since the idle power, which represents the static power is sensitive to temperature changes. So, when the workload starts running and chip temperature increases, the static power will also increase. However, dynamic power is not susceptible to temperature changes as much [92]. This can be seen in Figure 6.6. The R-Squared value between temperature and idle power is 0.740, whereas it is 0.287 between temperature and active power.

The trend shows that static power has been increasing as the gate sizes become smaller [93]. At least for this particular processor (Power8) and benckmarks, the static power is dominant. Ignoring the temperature increase effect on static power, dynamic power constitutes only 53.2 W of the 244.6 W, which is 21.7% of the total power.

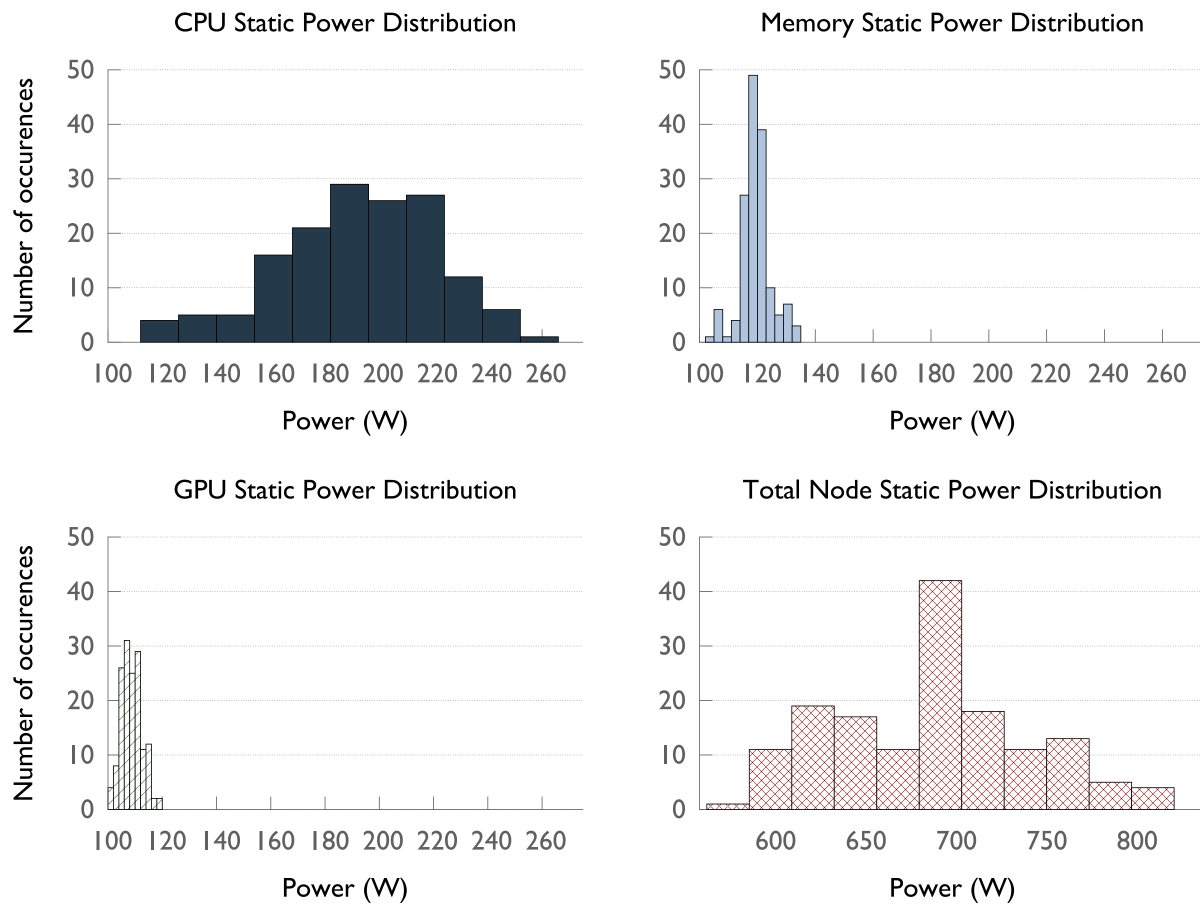


Figure 6.3: Static (idle) power distribution of node components (notice the x axis range change for the total node distribution in bottom right). Note that some of the difference in the idle power shown in this plot is caused by diffent instructions per second (IPS) number levels in the CPUs despite being idle. However, we still observe wide power differences among processors having similar IPS values as well.

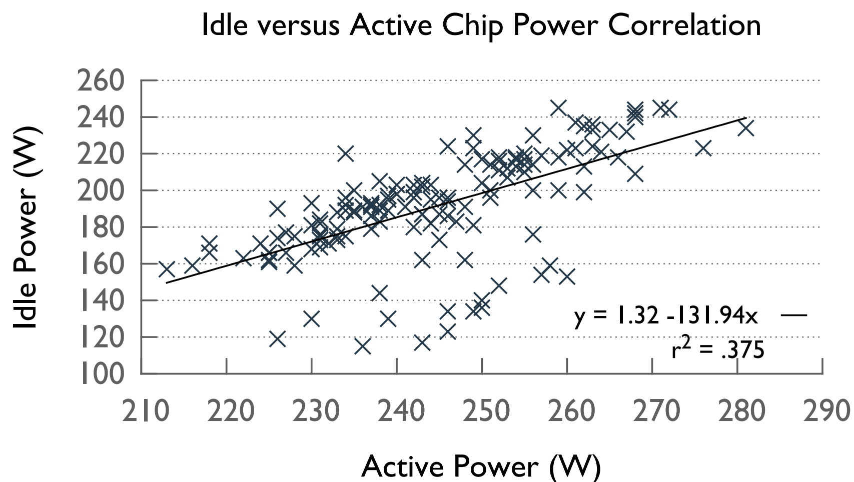


Figure 6.4: Idle versus active power of chips running DGEMM benchmark.

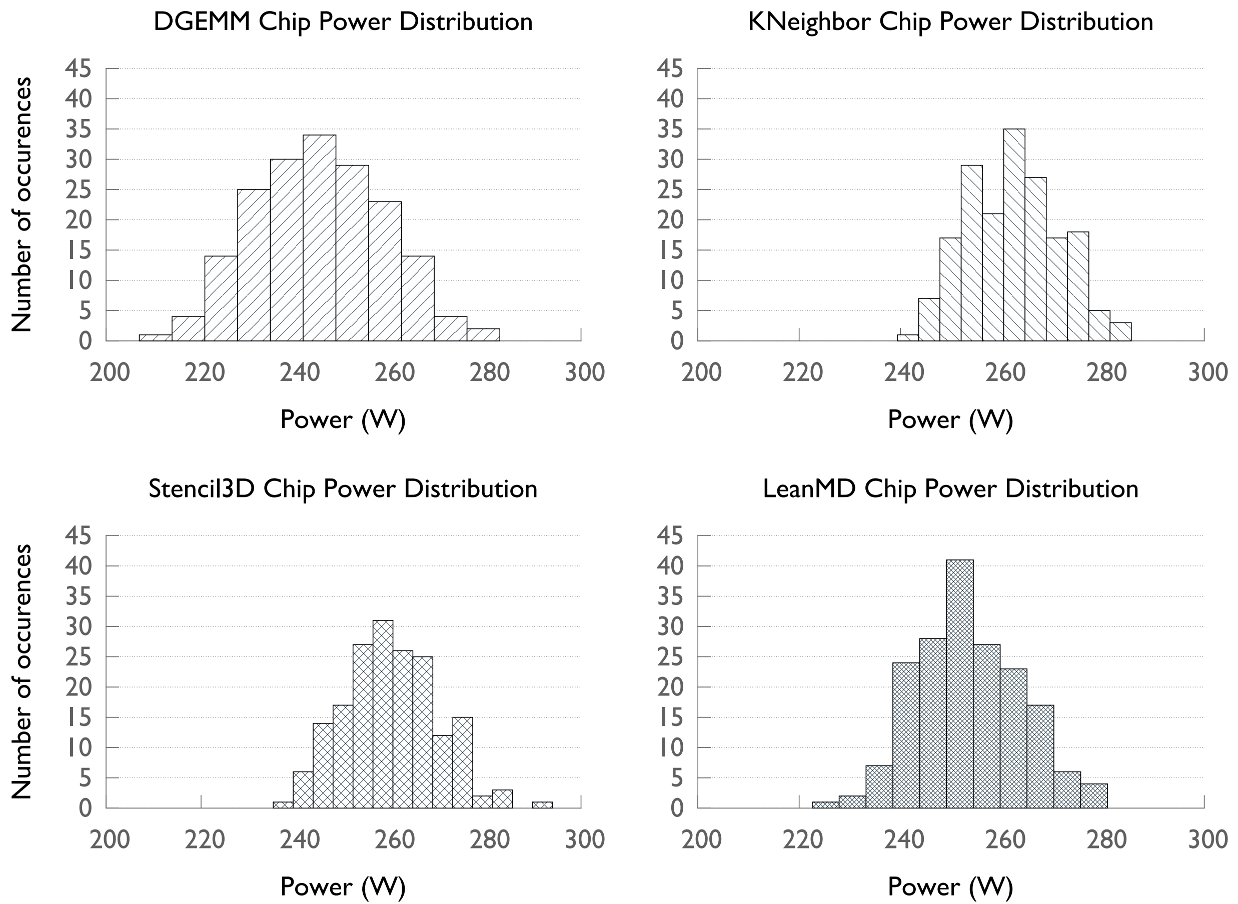


Figure 6.5: Power distribution of chips running different benchmarks.

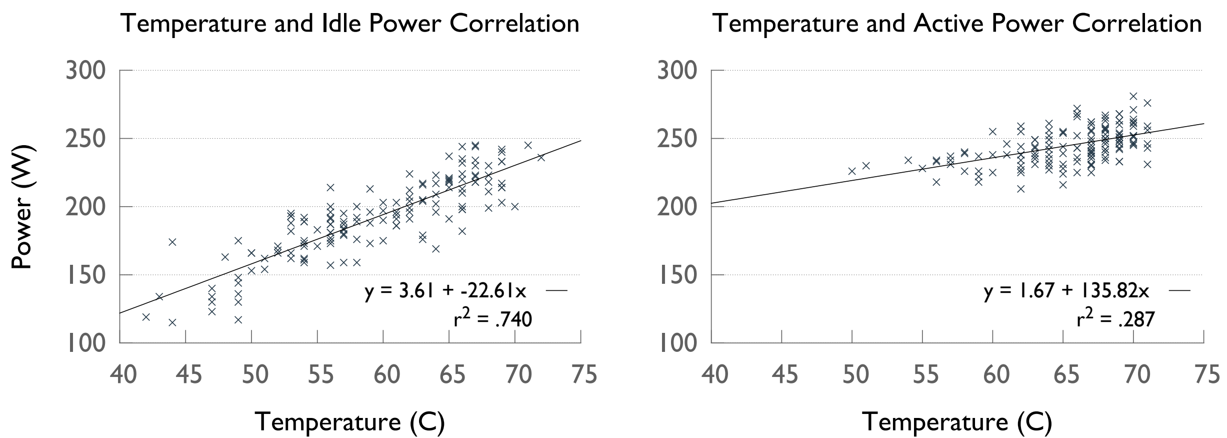


Figure 6.6: Temperature and power correlation of the chips.

6.3 Variation Aware Node Assembly

We now describe the random node assembly and the three new techniques that we propose. Each of the techniques has its own advantages and use cases. The three techniques we propose are: categorized assembly, application characteristics aware assembly, balanced power node assembly.

Random assembly

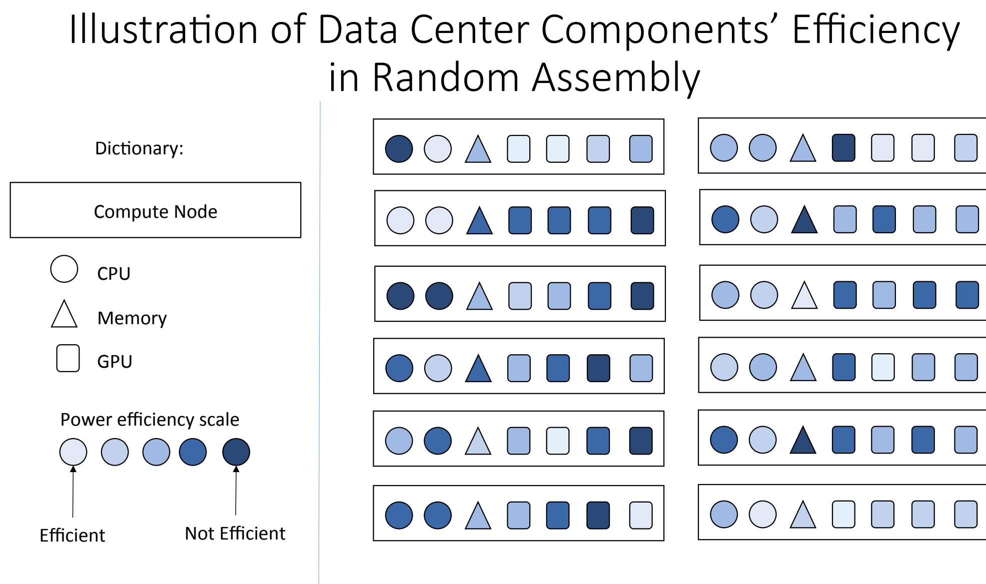


Figure 6.7: Illustration of node assembly types

This is the default assembly mechanism that is currently used in production assembly lines. In this mechanism, the components of a physical node are assembled without considering the characteristics of the individual components. Each component might have a different range of power variation and hence the variation range (max to min power) can be higher in the total node power. This is the cheapest mechanism in terms of the cost of assembly since it does not distinguish the components.

The illustration of this is shown in Figure 6.7. The components are divided into five different colors based on their power efficiency scale in the figure. The number five is selected arbitrarily, in fact it can be any number greater than or equal to two.

Type-1: Categorized assembly

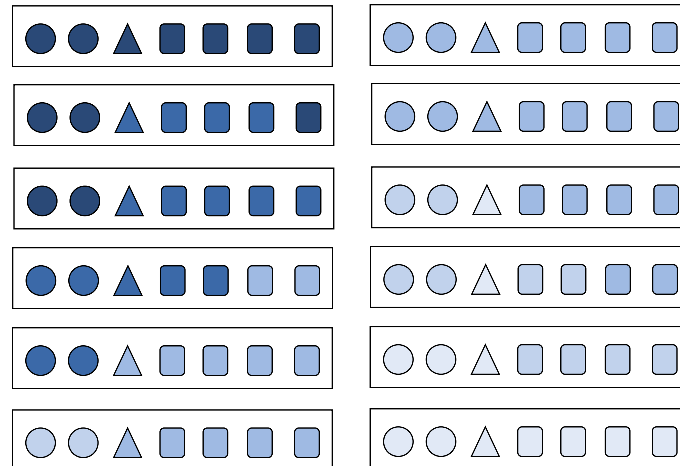


Figure 6.8: Illustration of node assembly Type-1: Categorized assembly

As shown in Figure 6.8, in this technique, each of the components within a node needs to be categorized into bins in terms of their power efficiency and each node contains sub-components that belong to same efficiency level. This method enables physical nodes to be classified easily in terms of their power efficiency (i.e., all efficient components will be gathered into an efficient node, or all inefficient components will be gathered into an inefficient node).

With the classification of the nodes, customers that have a limited power allocation (for example where the power infrastructure only allows a limited amount of draw), can now have an option to buy only the efficient nodes with the cost of a more expensive hardware. Other customers that do not have such limitation, if the hardware costs dominates the total cost of ownership (TCO) can buy the non-efficient nodes at a cheaper price. Customers can also buy a mixed batch of nodes of this type. Along with a smart job scheduler, this can still reduce the power and energy consumption compared with the random assembly if the data center is not at full load. For example, when the data center is at 90% load, the non-efficient nodes can be turned off first.

To calculate the power reduction using this method at a large scale, we first fit the power of the components in a Gaussian distribution as shown in Figure 6.9. Then we generate 5,000 random nodes based on the distributions of each component. We extrapolate to 5,000 since that is the rounded size of the Summit supercomputer. We evaluate the scenario where supercomputer contains mixed efficiency levels of Type-1 assembled nodes. Therefore, we can simply sort each of the components in terms of their power independently to get the node power with the Type-1 assembly. Figure 6.10 shows how much power reduction can be achieved when the data center is at different loads based on dynamic power. The power reduction in therms of KW shows like a dome-like curve due to the nature of Gaussian distribution in the power of the server components. The maximum power reduction in terms of KW happens when the data center is at a 50% load with a 110 KW reduction. To put this into scale of the whole machine, the power of the server components that are taken into picture in this calculation is around 6.5 MW. Note that the Summit supercomputer is expected to consume total of 13 MW power. In terms of percentage reduction, the maximum reduction happens when the data center is at least loads. This is because as the less nodes are used, only the most efficient nodes will be used and others will be turned-off.

Type-2: Application characteristics aware assembly

This method proposes customized assembly techniques depending on the applications' needs. For example, if the node is used to run GPU intensive workloads, then it should have efficient GPUs, but it can have inefficient CPUs. With this method, customers who know that they are going to run dominantly CPU intensive applications can only buy those types of nodes with efficient CPUs. A data center might have different types of workloads and hence can contain multiple varieties of nodes assembled using this method as illustrated in shown in Figure 6.11. In that case, the job scheduler of the data center needs to be aware of the application characteristics and the node characteristics to be able to place the applications to the appropriate nodes to obtain the best performance (i.e., place CPU intensive workloads to CPU efficient nodes).

In Chapter 3, we explained how differences in chips' power efficiency cause frequency varia-

tion as shown in Figure 3.2. This processor-to-processor variation is around 16%. Therefore, a 16% performance improvement can be achieved potentially using this assembly method .

Type-3: Balanced power node assembly

This technique aims to make the total node power of each node to be equal, or reduce the variation in the total node power. This would be helpful in estimating the total power of a node (i.e., when a data center needs to turn on or off a number of nodes, predicting the required power of that would be much easier if the variation is lower). Similarly, expected performance would also be similar, which can increase applications' performance predictability and reproducibility. An illustration of this technique is shown in Figure 6.12.

As we show earlier in Figure 6.3, the total node power can have a 235 Watts difference ranging from 587 Watts to 822 Watts. This technique would reduce the difference to almost none and the average expected node power would now be 687 Watts.

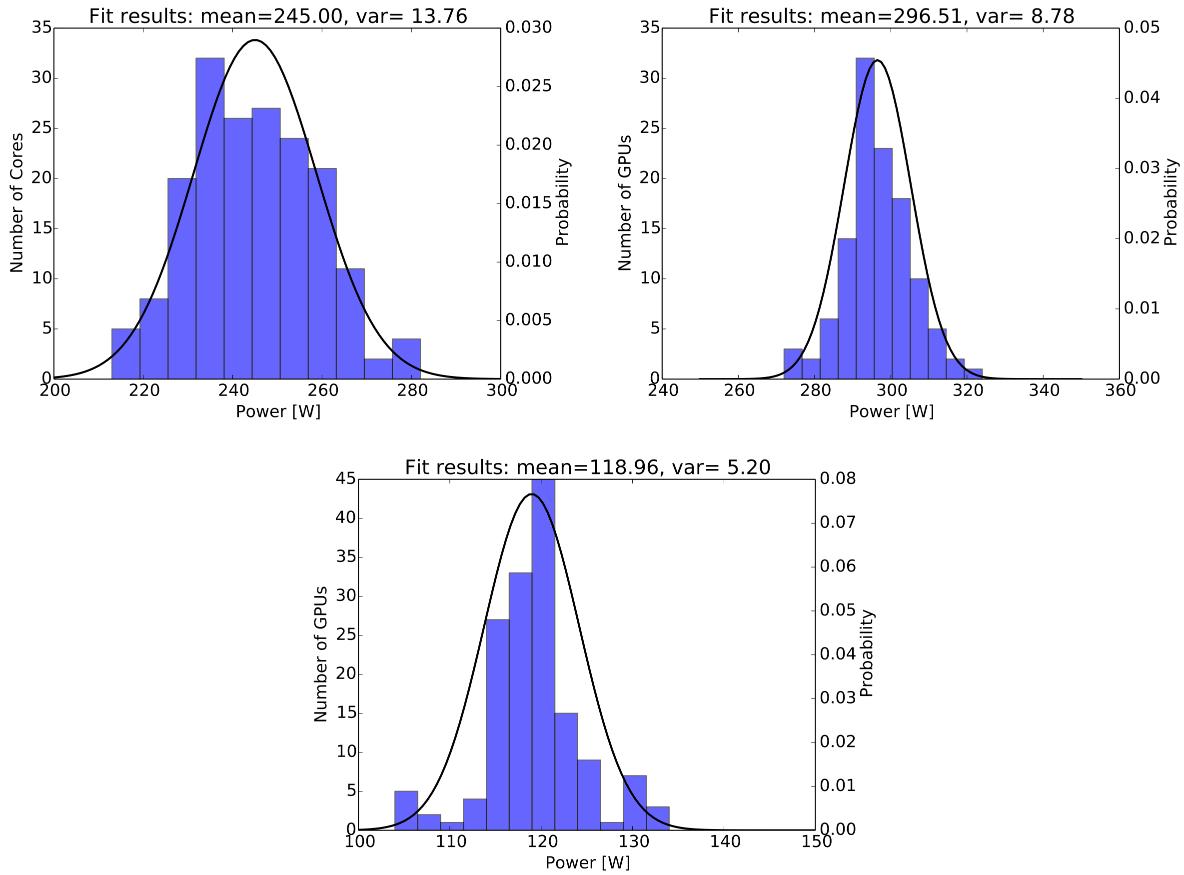


Figure 6.9: Distribution of the active power of node components: CPU, GPU, Memory (in order) fit to Gaussian distribution.

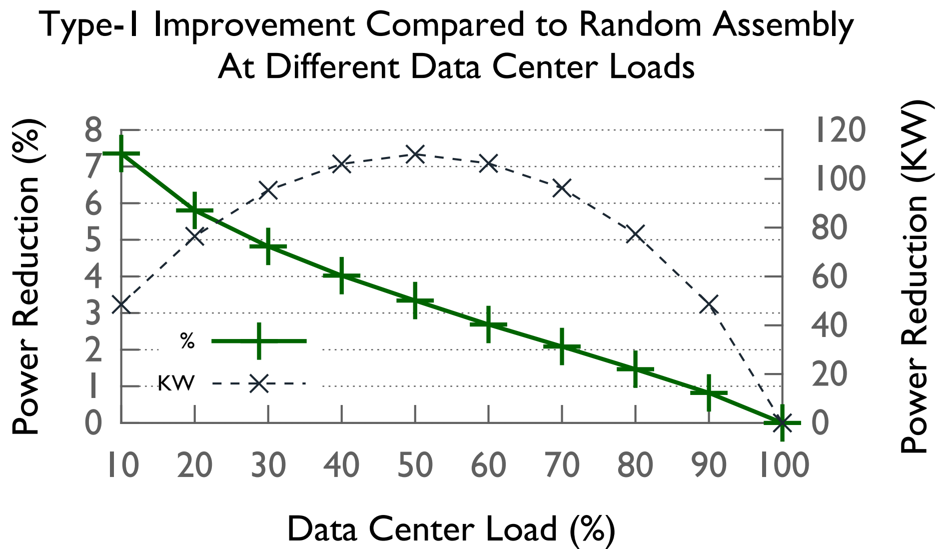


Figure 6.10: Power reduction with Type-1 node assembly compared to random assembly at different data center loads with a size of 5,000 nodes. The nodes that are not active are assumed to be turned-off.

Illustration of Type-2 Node Assembly

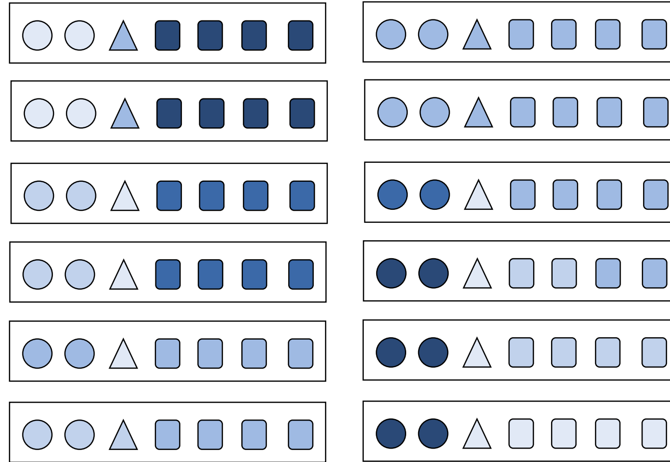


Figure 6.11: Illustration of node assembly Type-2: Application characteristics aware assembly

Illustration of Type-3 Node Assembly

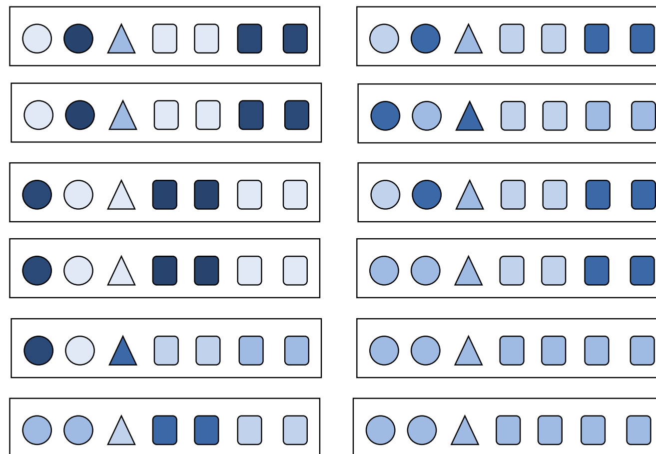


Figure 6.12: Illustration of node assembly Type-3: Balanced power node assembly

6.4 Summary

In this chapter, we first identified all of the different components within a physical node show variations in their power. Then, we proposed three new node assembly techniques in order to mitigate the negative effects of across component power variation in a physical node. Each of these techniques has its own merits and use cases. The first one makes component efficiencies at the same level as node efficiency. The second one assembles based on the applications' characteristics. Finally, the third one tries to achieve a balanced node power.

Implementability of the assembly approaches in real life depends on the the cost of the additional manpower required. We mention that different nodes can be sold at different prices (i.e., efficient ones at higher prices), more in depth pricing and profitability analysis are a recommended future directions.

The alternative common approach to exascale architecture model of fat heterogenous nodes with GPUs is fat "homogenous" nodes that have large amounts of small cores in a node like Intel's Xeon Phi architecture in next generation Aurora supercomputer [94]. This approach increases having large intra-chip core-to-core variations that cannot mitigated via assembly methods proposed in this chapter, since there is only one big chip in a physical node. Software-based dynamic scheduling approaches may become more important for such architectures to cope with variability.

CHAPTER 7

Mitigating Application-Level Variability

After addressing variability in frequency, temperature and power caused by manufacturing related reasons, this chapter looks into application level variability that are caused by functions that have different characteristics in parallel applications in order to do fine-grained energy optimizations.

DVFS is a well-known technique to reduce the power and/or energy consumption of various applications. While most processors provide chip-level DVFS, where the frequency of the cores in a chip can only be changed all together; core-level DVFS, where each core can be controlled independently, requires core-level voltage regulators in hardware and only is supported in production in Intel Haswell generation processors. The finer grained control that per-core DVFS provides can lead to higher energy efficiency compared to chip-level DVFS especially for the unsynchronized, unstructured parallel applications when carefully applied. Ability to do per-core DVFS opens up new doors for function or kernel level optimizations within runtime systems.

Many past research propose DVFS and RAPL [21] based solutions to optimize energy consumption [12,13]. These solutions have often been done at chip-level, i.e. changing the whole chip's frequency or capping the whole chip's power, not the individual core frequencies or core power. The reason for that is the lack of core level voltage regulators in the commercial processors. For the first time, Intel Haswell generation processors introduced the support for per-core DVFS in production [95,96]. Although this support have been discontinued on later generations Sky Lake and Kaby Lake, it is reported to return on Ice Lake generation [97].

The capability of doing per-core DVFS brings the premise of higher energy efficiency with finer-grained optimizations for a variety of applications.

While some HPC applications are regular or structured with a uniform behavior, i.e. all cores or processes within a chip execute similar type of work, such as stencil applications etc., some HPC applications are irregular or unstructured with dynamic behavior. In those unstructured applications, at a given time cores within a processors might do different types of work and execute different types of functions. Moreover, each function, kernel or phase might have a different optimal frequency level. (Note that energy optimal frequency is not necessarily the fastest frequency, many examples of this is shown in the past [98, 99]). For example, it has been shown that MiniFE application have loops that are affected differently by the frequency levels. [100]. Some applications for performance reasons have dedicated I/O or communication threads that inherently have different behavior than the rest of the application. Yet, many commercial processors used in HPC systems do not have support for per-core voltage and frequency scaling. With chip-level DVFS, different functions that happen simultaneously cannot run at their energy-optimal frequency levels.

The premise of per-core DVFS can be realized with a dynamic runtime that can do fine-grained energy optimizations. We implement a runtime based energy optimization module in a task-based programming model that can learn the optimal frequency for each task or function over time and show how per-core DVFS enables finer-grained optimization.

Main contributions of this chapter are:

- Analysis of per-core DVFS capabilities in three processors.
- Implementation of a fine-grained runtime technique which provides automated function-level energy efficiency (Two modes are provided: energy efficiency, and performance).
- Identification of use cases for per-core frequency and voltage regulation that motivates the need for implementing per-core voltage regulators in hardware.
- Up to 22% better performance or reduction in energy compared to per-chip frequency scaling.

To the best of our knowledge, we are not aware of any other work that does function level energy optimization using per-core DVFS for irregular applications (See related work in Section 7.4).

7.1 Motivation

In this section, first an analysis of the per-core DVFS support in the Haswell architecture is provided. Later, application use cases that can benefit from per-core DVFS are identified.

7.1.1 Per-Core Voltage and Frequency Scaling

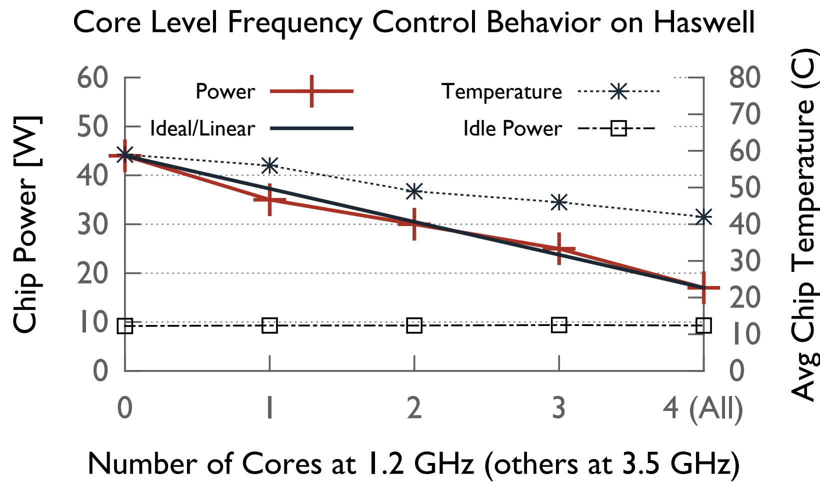
For the first time among commercial Intel processors, Haswell generation introduced per-core voltage regulators [95]. Per-core voltage regulation, also called Fully Integrated Voltage Regulator (FIVR) [72], give the premise of doing fine-grained power and energy control. In this section, we first show why per-core voltage regulars are important in doing core-level frequency scaling. We use an older Sandy Bridge generation processor which does not have FIVR to compare per-core DVFS behavior with the Haswell generation. The details of the processors that we use are given in Table 7.1.

Table 7.1: Platform hardware and software details

Processor	**Intel Haswell**	**Intel Sandy Bridge**	**IBM Power8**
Model	Xeon® E5-1620 v3	Xeon® E3-1245	Power8 8335-GTA
Cores	4		10
OS	Linux Ubuntu v 3.13.0		Red Hat Ent. Linux 7.2
Turbo Speed	3.6 GHz	3.7 GHz	3.6 GHz
Max Non-Turbo Speed	3.5 GHz	3.3 GHz	3.5 GHz
Min Non-Turbo Speed	1.2 GHz	1.6 GHz	2.0 GHz
Per-Core Voltage Regulators (FIVR)	Yes	No	Yes, not in production

Dynamic power of the CPU is proportional to the square of the CPU voltage and to the CPU frequency:

$$Power = C \times V^2 \times f$$



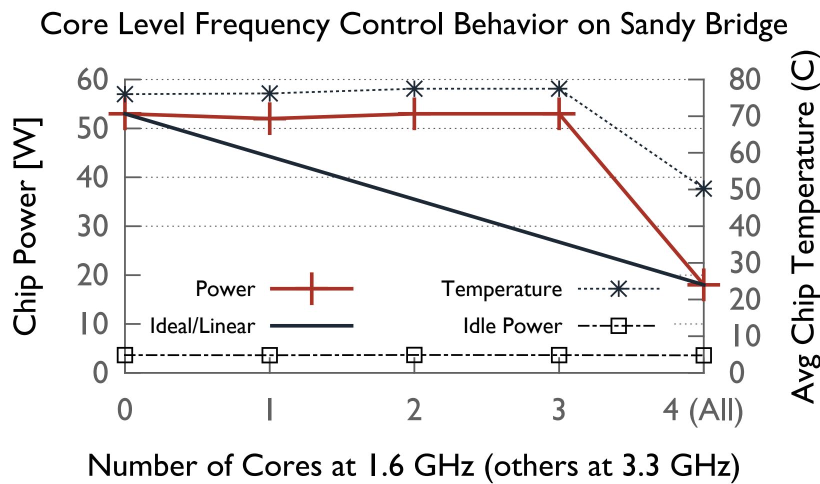


Figure 7.1: Core level DVFS on Haswell architecture shows proportional/linear decrease in power when core frequencies are dropped one by one. On the other hand, since Sandy Bridge do not have per core voltage regulators, all core frequencies needs to be dropped together to for a a reduction in power and temperature.

where C is capacitance, V is voltage, and f is frequency.

Voltage is a dominant factor in processor power and hence per-core frequency scaling without per-core voltage regulators is not effective in terms of performance-per-watt. Because, if the cores within the chip have different frequency levels, without per-core voltage regulators, the voltage level of the processor is determined by the highest frequency core.

Figure 7.1 compares per-core frequency scaling behavior of two processors; one with per-core voltage regulators (Intel Haswell) and one without (Intel Sandy Bridge). We use

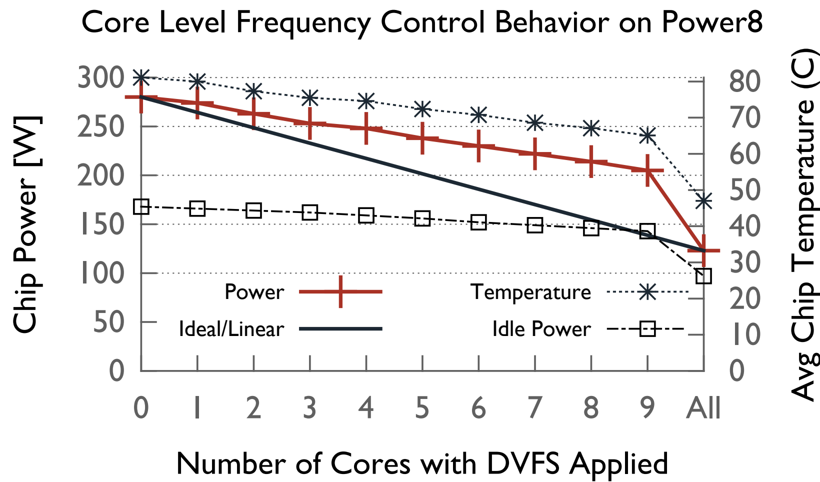


Figure 7.2: Core level frequency scaling on IBM Power8 chip.

`cpufreq` kernel module to control the frequency of the cores. In this experiment, we first set the frequency of all cores to the highest level and then decrease the frequency level to minimum one core at a time. For Haswell processor, as we decrease frequency of more number of cores to minimum the chip power drops linearly. On the other hand, for the Sandy Bridge processor, the chip power does not change until all of the core frequencies are lowered together. The reason for that is actually Sandy Bridge processor do not listen the per-core `cpufreq` commands and executes all of the cores at the maximum level until all of the core-frequencies are reduced together. Another observation from Figure 7.1 is that dynamic power of the Haswell chip constitutes 80% of the total chip power. Although this percentage is less than the 94% ratio in the earlier generation, dynamic power is still a major amount that can be optimized via DVFS, whereas static power is not effected by the frequency of the cores. Figure 7.2 shows the results of the same experiment on a 10 core IBM Power8 processor. Despite having per-core voltage regulators in hardware, per-core DVFS is not supported in production use with `cpufreq` module. Although, unlike Sandy Bridge, Power8 listens per-core DVFS commands and changes the frequency core by core. As a result, the chip power gradually decreases as shown in the figure. However, it is still not a linear reduction as in the case of Haswell – there is still a bigger drop when all of the core-frequencies are changed all together. This shows per-core frequency scaling without

per-core voltage regulation lacks effectiveness, i.e., it causes performance overhead without reducing the energy much.

7.1.2 Application Use Cases

Per-core DVFS would be beneficial for multiple different scenarios. We identify six categories of potential use cases that can benefit from per-core DVFS.

1. **Applications with Different Kernel Types**

   Applications may have different kernels which have different optimal frequencies. Moreover, those kernels do not necessarily execute at the same time. *MiniFE* application has shown to have two different kernels that have different characteristics [100]. *OpenATOM* is a quantum chemistry application which has many overlapping kernels and phases. Figure 7.3 shows the time profile graph of two processes mapped to two different cores running the benchmark. As each color represents a Charm++ entry method, it can be seen that there are many cases where each process is executing a different type of function.

2. **Applications with Dedicated I/O Threads**

   Dedicated I/O threads are used in I/O intensive parallel applications to improve the performance by offloading the I/O work to dedicated threads. Many past work uses this approach, including I/O frameworks [101, 102], fast asynchronous checkpoint/restart based fault tolerance mechanisms [103], machine-learning applications [104], high performance in-memory databases [105].

3. **Applications with Dedicated Communication Threads**

   It has been shown that using dedicated communication threads to drive the network communication helps improve the performance of communication intensive applications especially at scale. MPI Endpoints [106] and SMP mode of Charm++ [107] are two examples of this. MPI endpoints extension enables efficient multi-threaded communication by using dedicated cores that drive independent network communication.

Although losing a core for communication might cause performance loss in a computation dominant application, at large-scale when the communication becomes the large fraction, having dedicated cores for communication improves performance [106]. Charm++ implements a similar concept, called SMP mode. In the SMP mode, a logical node is formed by a custom number of threads and a dedicated communication thread to handle the communication between nodes. Commonly each physical node has one or more number of logical nodes.

4. **Applications Leaving Idle Cores**

   Parallel applications may leave idle cores for various reasons including, for performance reasons and for having core/count requirements.

   Some applications intentionally leave idle cores in the processors for performance reasons. We will give three examples to this. First, broadcast/reduction trees do not perform as well for numbers of processor counts that are not a power of two [108]. Second, some applications run out of memory when doing large scale simulations and might leave one or more cores idle. PDES [109] is an example to this. Third, some applications, like NAMD [107], suffers from OS interference. To remove the interference, all OS processes are be bind to specific core, which is then excluded, i.e., not used, by the application.

   There are several applications that requires to be run on specific core counts, i.e. such as power of two number of cores, cubic number of cores. In such scenarios, some cores have to be left idle depending on the processor core count. For example, LULESH [110] MPI version requires cubic number of ranks (hence cores) to map the 3D spatial domain. Graph500 benchmark [111] only supports graphs with power-of-2 vertex counts and hence MPI version is required to be to run on power-of-2 number of cores.

   Although these cases may seem to be ideal applications for per-core DVFS, in fact, if the idle core power is already optimized to be its lowest, more power reduction cannot be achieved by per-core DVFS. As we show in Figure 7.1, for Intel processors idle power is not effected by the frequency. For Power8 processors, frequency effects idle

power as shown in Figure 7.2, i.e., around 5 W per core can be reduced in application that leave cores idle if per-core DVFS is introduced. However, it is important to note that static power constitutes 21% of Intel Haswell processors 21%, whereas it is more than 50% for Power8 processors.

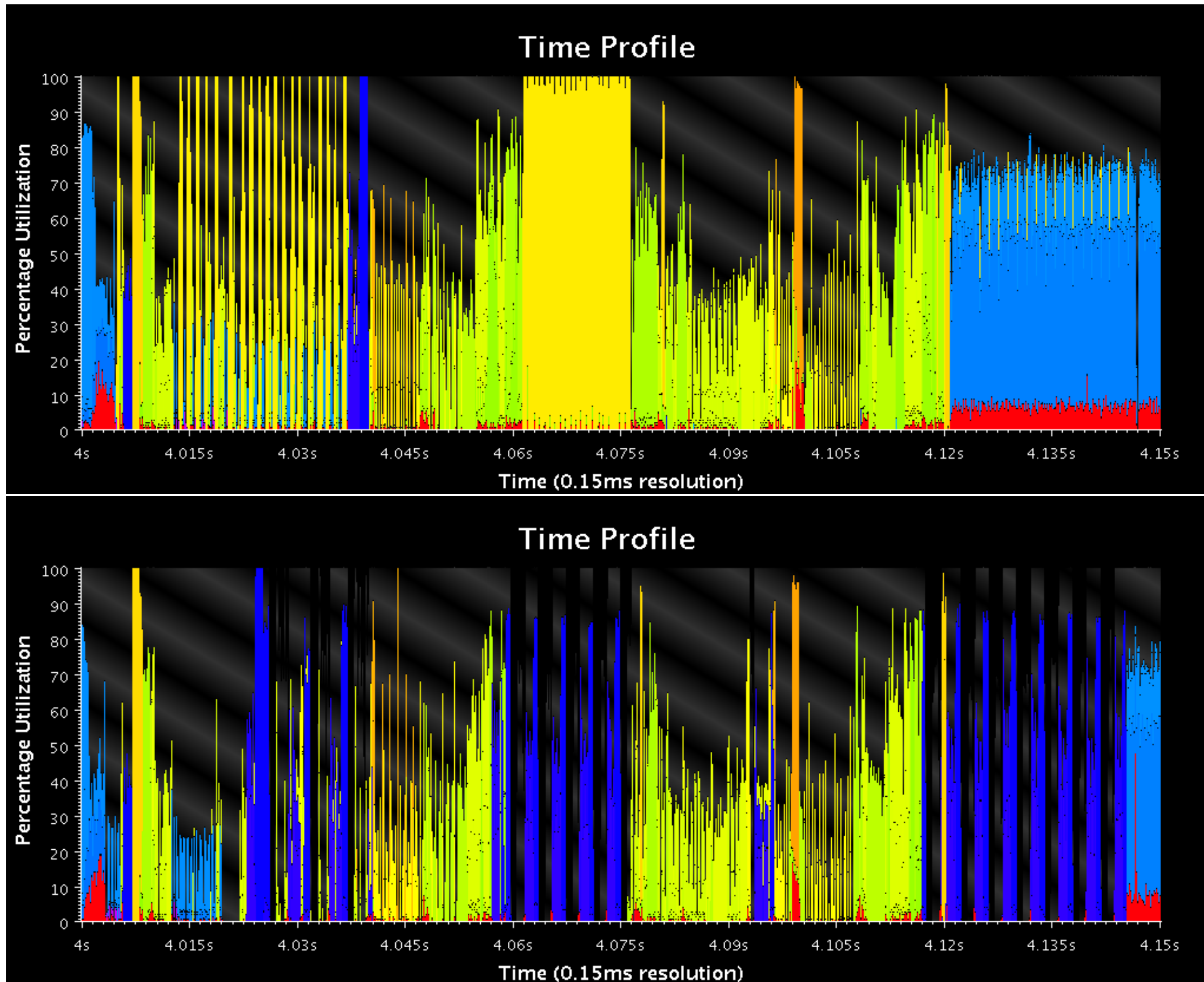


Figure 7.3: Timeline of two processors running the OpenATOM benchmarks. Each color represents a Charm++ entry method. Notice how different entry methods are executed on the two processes at the same time range.

7.2 Runtime Guided Frequency Regulation

In this section, we first give background information on the runtime system that we use as a proof-of-concept for our work. Then, we explain our runtime guided frequency regulation approach.

7.2.1 Charm++ Adaptive Runtime System

Charm++ is a parallel programming framework used by many large-scale applications including NAMD for molecular dynamics, OpenATOM for quantum chemistry, ChaNGa for cosmology, Episimdemics for epidemic simulations and many others [17]. In Charm++, the application data is decomposed into small task units (called *chares*) that communicate via asynchronous function calls (called *entry methods*). The runtime system is responsible for placement and execution of the task units.

Chares are C++ objects that represent the data and task units in Charm++. These objects can migrate from processor to processor by the runtime in order to create load balance. They can communicate with other objects via asynchronous function calls.

Entry Methods are asynchronous function calls on the chares. Chares can be located in a local or a remote processor, regardless of the location the runtime will deliver the function call as a message to its destination chase. Besides the application's entry methods, runtime itself also contains entry methods to do various tasks in the background such as communication, I/O, load balancing, tracing etc.

Charm++ RTS is responsible for the mapping of the objects to processors, sending the messages (entry method calls) to their destination chares, and executing the entry methods.

Charm++ programmer needs to write an interface file to define the class types and the corresponding entry methods so that the runtime system can generate the corresponding structures. Figure 7.4 shows a code snippet from the interface file of a stencil application. There is a natural division between the application phases. The rest of the code is common C++.

Charm++ would be a great fit to make a proof-of-concept implementation of our ideas for

```
1  array [3D] Jacobi {
2    // Normal Charm++ entry methods
3    entry Jacobi(void);
4    entry void begin_iteration(void);
5    entry void receiveGhosts(ghostMsg *gmsg);
6    entry void processGhosts(ghostMsg *gmsg);
7    entry void check_and_compute();
8    entry void doStep() {
9        serial "begin_iteration" {
10           begin_iteration();
11       }
12       for(imsg = 0; imsg < neighbors; imsg++) {
13           // "iterations" keeps track of messages across steps
14           when receiveGhosts[iterations] (ghostMsg *gmsg)
15               serial "process ghosts" { processGhosts(gmsg); }
16       }
17       serial "doWork" {
18           check_and_compute();
19       }
20   };
21 };
```

Figure 7.4: Code snippet from Charm++ stencil application.

two main reasons. First, programmer writes entry methods naturally in a way that different phases of the application are distinguished with different entry methods. This enables runtime to distinguish different types of work units automatically by simply controlling at each entry method-level. Second, the runtime have a transparent control over the full application from start to end. The approaches restricted to annotated loops or kernels do not give control over the whole application runtime which limits optimization scope and capabilities.

7.2.2 Fine-Grained Frequency Regulation in Runtime

Our approach fine-grained frequency regulation approach has three phases: 1) Statistics collection of power and performance for different frequency levels, 2) Calculating the optimal frequency based on the collected statistics, 3) Applying the optimal frequency on function basis in core-level. Next, we will explain each of there steps in detail.

Our module can simply be enabled by building Charm++ with `--enable-energyOpt` flag and linking the application with `-module energyOpt`.

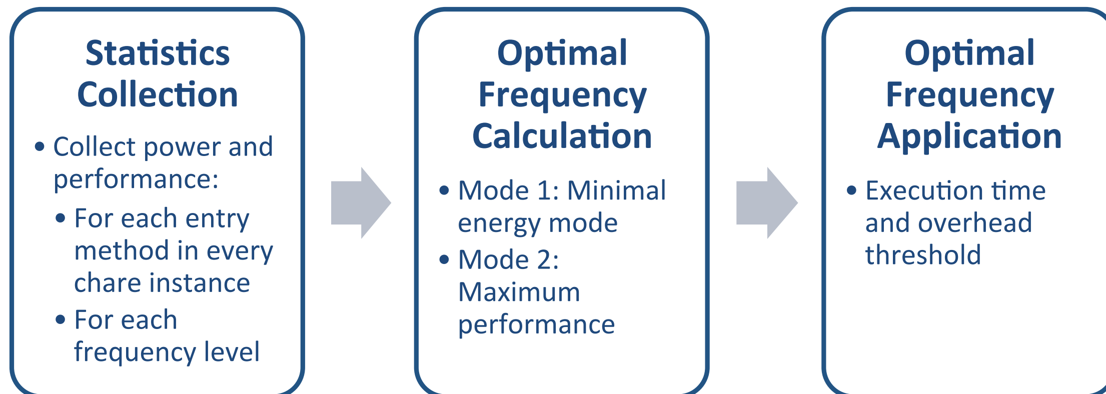


Figure 7.5: Runtime control flow.

Statistics Collection

The first phase from the start of the application is the statistics collection phase. First, the runtime collects the power and performance characteristics of each entry method in the application under different frequency levels. The frequency levels are set by the runtime system as the application moved forward.

The entry method statistics are collected and stored per each instance of a chare element. Although every instance of a chare element have the same functions, different data size might cause different characteristics depending on the instance. Therefore, it is more accurate to collect the statistics per instance of chare elements. Particularly, the execution time and the energy consumption of each entry methods are collected under every supported frequency level.

There is, however, one obstacle in collecting power statistics of the functions. Despite the support of per-core DVFS, Haswell processors do not provide core-level power counters – only chip level power information is available through model-specific registers (MSRs). Therefore, a workaround is necessary. To be able to calculate the correct power information in core level, we execute the entry methods on the cores exclusively via locks during the statistics collection phase so that only one core is running at a time. The positive side of this approach is that it can capture core-to-core power or performance variations that might happen since profiling of the kernels are done on the same cores that are going to execute them after the profiling phase. The negative side of this approach is that the measurements on single core

may not represent the performance and power when other cores are active. First, the static power of the other three cores needs to be subtracted from the total chip power. The static power of the three cores can simply be calculated as three quarters of the the idle power of the processor. Since there are four cores in the processor we use, exclusive entry methods execution causes a 4x slowdown in the application during this phase. This is a limitation of the processor and if there were core-level power counters, there would not be any need for a locking mechanism. Since the statistics collection phase constitutes a small fraction of the total application run, use of this exclusive execution method is still viable despite its overhead. Second, we need to make sure the measurements we collect when kernels are running exclusively are correct when they run together with other cores. Figure 7.6 shows that for compute intensive benchmarks, like DGEMM, the optimal frequency calculated using one core is consistent with the results when other cores are active. However for memory intensive kernels, like MEMOPS, as more cores are running the kernel, the energy optimal frequency drops significantly as more cores are competing for the same memory bandwidth. Figure 7.7 shows the optimal frequency of MEMOPS kernel again, but with varying allocated data sizes. From 128 MB to 1.5 GB data per process, the trend is more or less consistent: as more processes are active the optimal frequency drops. This shows that for memory intensive applications, what other cores are running effects the performance of the kernel, therefore exclusive measurements on a single core cannot represent the general case. Therefore, the count of memory operations (such as reads, writes, cache misses) needs to be collected for each kernel in addition to the execution time and the power consumption during the statistics collection phase. A method to prevent this problem could be using a power prediction model to predict the core-power consumption [112, 113]. However, such model can be error prone and it would increase the runtime algorithm complexity. The best way to overcome these limitations is to provide core-level power counters and perhaps this chapter provides a motivation for hardware developers to enable that support.

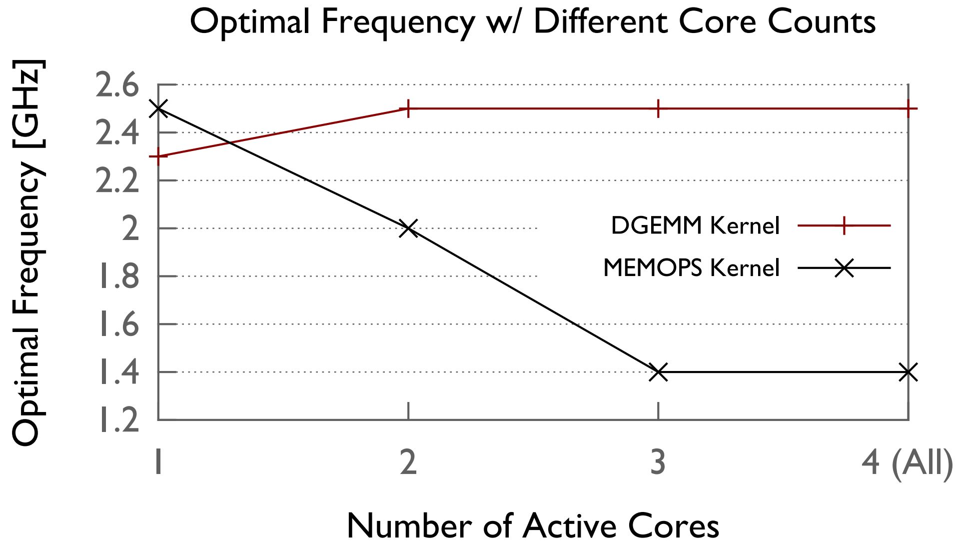


Figure 7.6: Optimal frequency of DGEMM kernel remains more or less stable despite the number of active cores running the kernel, wheres optimal frequency of the MEMOPS kernel drops significantly as more cores are activated.

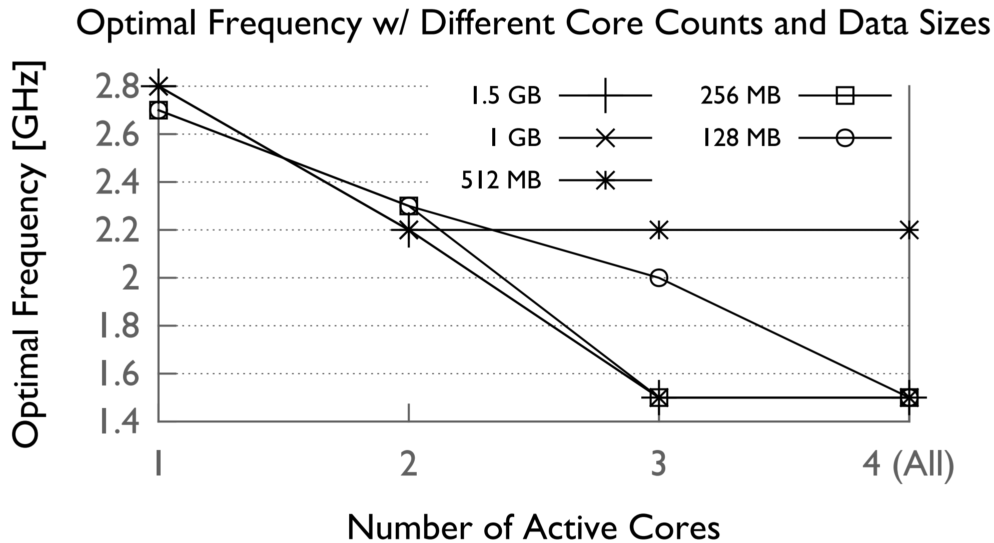


Figure 7.7: Optimal frequency of the MEMOPS kernel depends more on the number of active cores than the data size.

Optimal Frequency Calculation

After the statistics collection phase is done, optimal frequency is calculated per each entry method in each chare instance. We implement two options: the frequency that provides

minimal energy ($MinE$) and the lowest frequency without sacrificing performance ($MaxP$). $MaxP$ mode can also be used to find the lowest frequency with sacrificing from performance no more than a specified percentage (i.e, what is the lowest frequency that an application can use with a maximum of x% overhead?)

Optimal Frequency Application

After the optimal frequency levels are determined, the next phase is to apply the optimal frequency before the each method gets executed. There are some important issues that needs to be considered when applying DVFS (p-state change) on per function basis. There is a delay between sending p-state change request and the processor switching its frequency, this delay is called transition delay. This delay is important in making function level transition decisions because if the function duration is smaller than the delay, switching to a different level may not be useful. We have observed this delay can be up to around 500 µs. This confirms the earlier reported results on the transition delay [96]. Although the actual p-state transition may take much shorter than 500 µs, the p-state transition requests are not executed immediately, but in 500 µs periods in Haswell processors unlike previous generations (including Haswell-HE) [96].

Another important concern is the overhead of applying DVFS frequently. We have measured the overhead to be between 2 to 5 µs. The overhead is mainly caused by writing the new frequency level to the appropriate system file (two file writes for the two SMT threads). Although this is not a significant overhead, our algorithm takes this effect into account when making frequency scaling decisions and and tries to minimize the overhead.

Figure 7.8 shows the timeline where two kernels (Kernel-1 and Kernel-2) are executing one after another in a core. Let's define:

$F1$: energy optimal frequency for Kernel-1
$F2$: energy optimal Frequency for Kernel-2.

Applying the optimal frequency for Kernel-2 after the execution of Kernel-1 would be

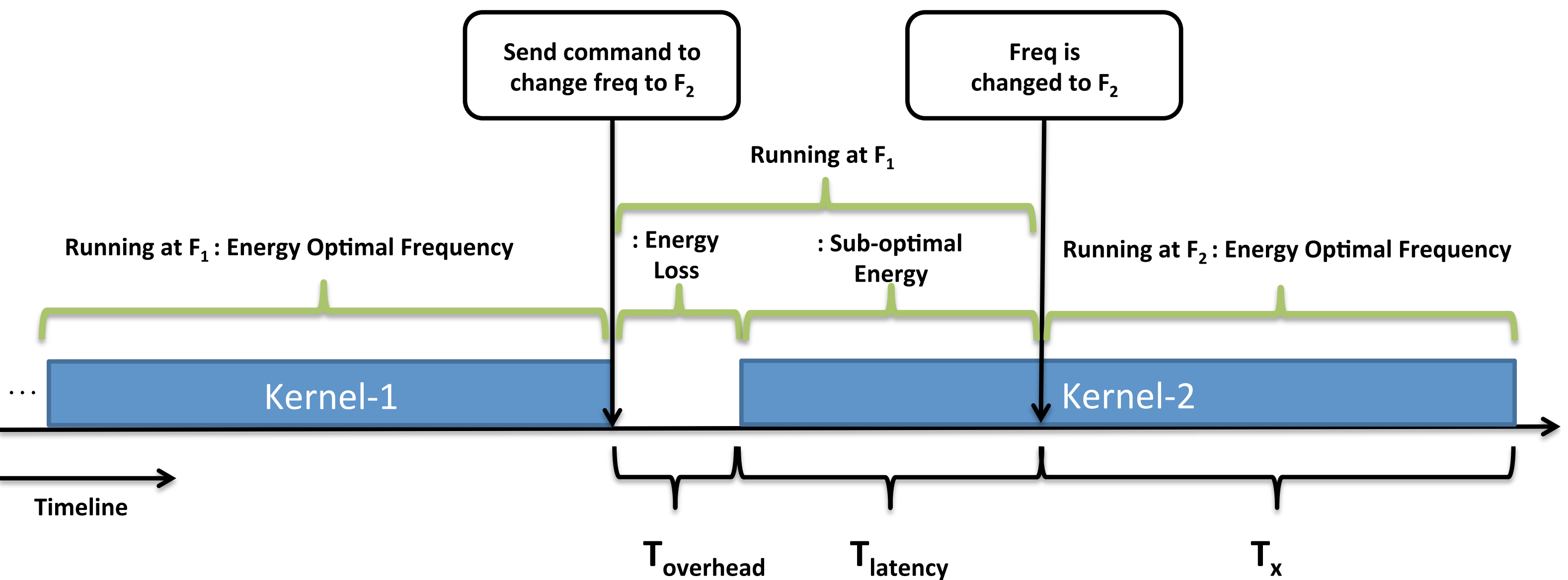


Figure 7.8: Timeline of two kernels executing one another where the runtime applies optimal frequency.

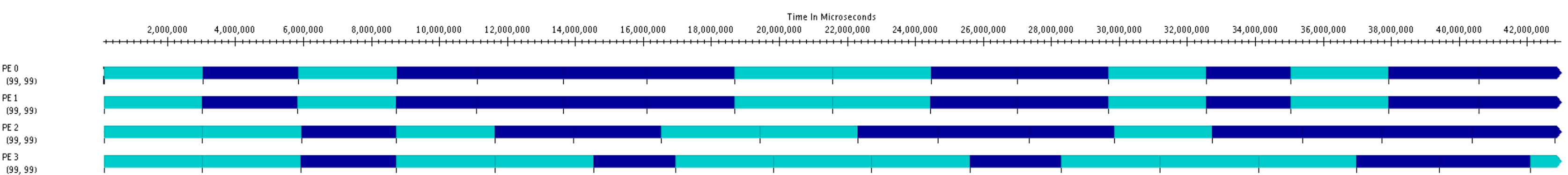


Figure 7.9: Timeline of the synthetic benchmark having two kernels (represented by light blue and dark blue colors) randomly overlapping. Each line represents a process, in this case four processes are running in parallel.

beneficial only if the following condition is satisfied:

$$E_{loss} + E_{F1_portion} + E_{F2_portion} < E_{F1}$$

E_{F1}: Energy consumption when whole Kernel-2 runs at *F1*

E_{loss}: Energy loss occurred during DVFS overhead

$E_{F1_portion}$: Energy consumption when Kernel-2 runs at *F1* during the transition latency phase

$E_{F2_portion}$: Energy consumption when a portion of Kernel-2 runs at *F2* after the transition is complete

Expanding the above formula in terms of power and execution time, we get:

$$T_{overhead} \times P_{F1} + T_{latency} \times P_{F1} + T_X \times P_{F2} < T_1 \times P_{F1}$$

E_{F1}: Energy consumption when all portion of Kernel-2 runs at *F1*

P_{F1}: Power consumption when Kernel-2 runs at *F1*

P_{F2}: Power consumption when Kernel-2 runs at *F2*

$T_{overhead}$: Constant, 5 microseconds

$T_{latency}$: Constant, 500 microseconds

T_X: Duration of the target kernel

Value of T_X depends largely on $F1$ and $F2$. In Figure 7.10, we show experimental results with a compute kernel where $F2$ is 2.3 GHz and $F1$ is labeled as transition frequency on the x-axis. The first observation is that the further away the transition latency is from 2.3 GHz, the more energy reduction is expected to happen. Second observation is that the smaller the kernel duration is, the less energy reduction happens because of the transition latency and overhead as described earlier in this section. Kernel duration label on each line in the plot represents the duration when run on the highest frequency. The duration of the kernel is adjusted simply by changing the loop counter to set how many times the kernel is executed.

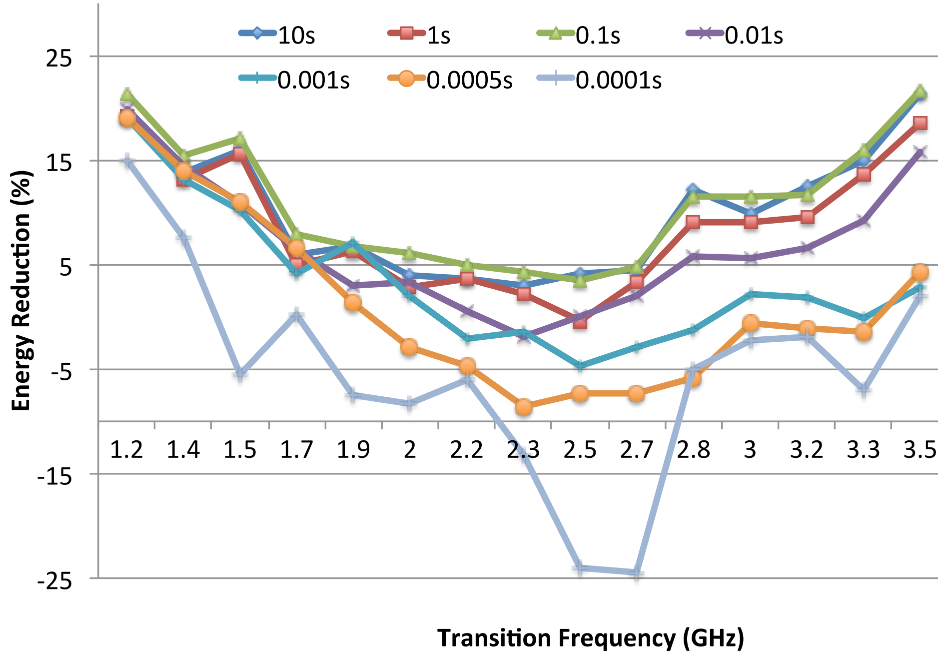
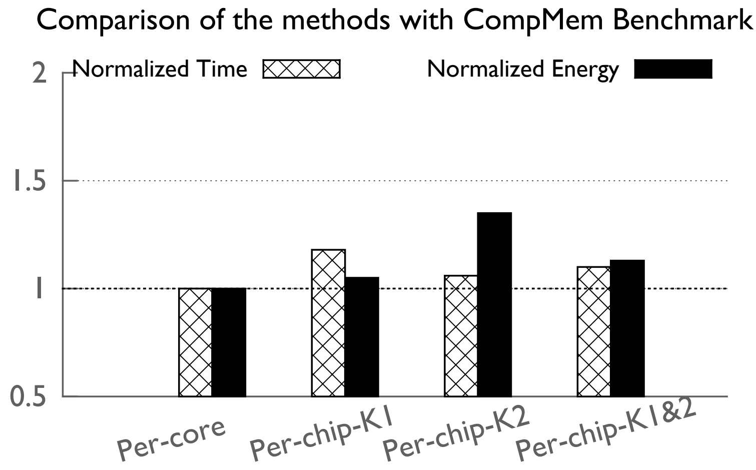


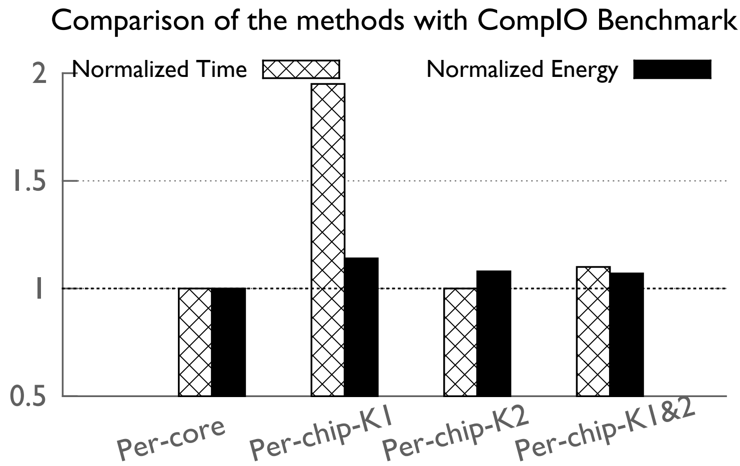
Figure 7.10: Plots shows how much energy can be reduced if a kernel that has an energy optimal frequency of 2.3 GHz is transitioned from different frequency levels. Different lines represent different kernel durations.

7.3 Experimental Results

In this section, we evaluate the effectiveness of our runtime module, mainly comparing with per-chip DVFS. We first use a synthetic benchmark, CompMem, that has two different kernels that are randomly called one after the other in every process. The goal of this benchmark is to have processes execute functions that have different characteristics at a given time. One kernel is a compute intensive one (i.e., a DGEMM kernel with a matrix size that fits in cache). The second kernel is a memory intensive kernel, MEMOPS, that consists of allocating memory, copying data into allocated memory via *memcpy*, and deallocating the memory. We use an array of size approximately 500MB per process. Figure 7.9 illustrates the timeline of this benchmark. We also implement two other versions of this benchmark; CompIO and CompComm. CompIO benchmark has one dedicated thread doing IO operations such as file reads and writes to the local disk. The other threads are running a compute intensive kernel. CompComm benchmark has one dedicated thread responsible for external communication (i.e., sending and receiving messages to and from other processors).

Figure 7.11 compares the time and energy measurements of our fine-grained runtime based solution (labeled as *Per-core*) with other chip level solutions (labeled as *Per-chip-**). *Per-core* always gives the minimum energy and minimum execution time. *Per-chip-K1* and *Per-chip-K2* optimizes only for kernel 1 and 2 respectively, therefore they have high execution time overhead or high energy. *Per-chip-K1&2* uses the mid-point of *Per-chip-K1* and *Per-chip-K2*, therefore creates a balance between the high execution time and the high energy of those methods, however it still is not better than *Per-core*.





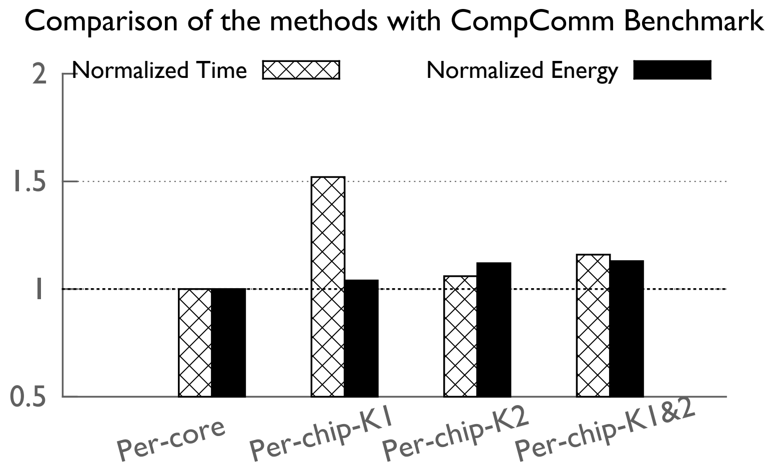


Figure 7.11: **Per-core:** Uses energy optimal frequency for each kernel in core level. **Per-chip-K1:** Uses per-chip DVFS with optimal frequency of kernel-1 which is 2.2 GHz, **Per-chip-K2:** Uses per-chip DVFS with optimal frequency of kernel-2 which is 3.3 GHz, **Per-chip-K1&2:** Uses per-chip DVFS with optimal frequencies of kernel 1 and 2 running together which is 2.7 GHz.

7.4 Related Work

Many of the past research uses chip-level DVFS to do energy or power optimizations [114–117]. Sarood et. al. proposes a thermal aware load balancer [12, 13] which uses chip-level DVFS technique to restrain the temperature of the processors. Padoin et al. extends this approach and proposes a load balancer which utilizes per-core DFVS [118]. However this work is not done in a hardware that supports per-core DVFS, instead it simulates the environment of a per-core DVFS by placing only one process on each chip.

Another chip-level adaptive frequency selection approach has been proposed for GPU and CPU kernels [119]. This approach can select a kernel to run on GPU or CPU and at which frequency level based on Pareto optimality. The first drawback of this work is that it works at individual kernel level and assumes all threads within the processor are executing the same kernel, and does not do any optimization in between the kernels. Our approach does optimization for whole application and in a transparent way taking into account frequency switching latency and overheads. This approach also proposes a kernel classification using machine learning algorithms so that the ideal configuration of a kernel can be applied directly. A similar approach can be applied in our runtime to make decision making easy. One drawback of such approach is that the learning algorithm needs to be trained for each processor type from scratch since the performance and power characteristics can change based on the platform. The approach we propose in this chapter is not platform specific and can be used on a new platform by only porting the power or energy reading measurements if necessary. Therefore, despite the minimal statistics collection overhead in the beginning of the application, our runtime based approach is much more practical.

Lim et al. proposes using DVFS in communication phases of MPI applications, such as `MPI_Send, MPI_Recv` etc., to reduce the energy consumption [120]. Their approach is implemented within MPI and transparent to the application like ours. However, their approach is also inherently limited to communication phases within MPI. On the other hand, our scope is the whole application and can optimize for all phases and kernels of the application. Bhalachandra et al. uses a similar approach for MPI; when there is slack before an `MPI_Barrier`, it applies DVFS to balance the arrival time of the processes to the barrier [121]. The same

drawbacks of Lim et al.'s paper applies to this one as well.

7.5 Summary

This chapter proposes a fine grained runtime approach to fully optimize the energy efficiency of applications at function level considering function to function variations within the applications. Per-core DVFS support from the architecture is essential in our method since cores in the processor might be executing different functions at any given time. Per-core power measurement support is also essential to put our method effectively in a practical use.

There are some limitations with this approach as well. The method does not support hyper-threading since hyper-threads can share the same physical core, i.e. execute instructions from different threads in the same core, and is the hyper-threads within the same physical core is executing different application kernels or functions, there is no way to change their frequency for each of threads separately. Another limitations is with fat nodes having lots of cores if core-level power data is not provided by the architecture. This means for the performance and power collection phase, the locking mechanism to ensure only one core is executing at a time and this cause number of cores times slowdown for the data collection phase. A future direction with this method is to understand how local optimal frequency selection decisions effects the global application performance and combine the method with a frequency aware load balancer.

CHAPTER 8

Conclusion

This dissertation examined several different types of variations that are found in large scale HPC systems. Detailed analyses of frequency, temperature, and power variations have been presented, along with the identification of the sources of these variations and novel ways to mitigate them.

Several contributions have been made in this dissertation. These contributions include:

- A comprehensive system within which the data center resource manager dynamically interacts with the individual runtime systems of jobs, to optimize performance and power consumption in an environment with system failures under constraints supplied by users or administrators is presented.

- Measurement and analysis of performance variation of up to 16% between processors in top supercomputing platforms including: Cori, Edison, Cab, Stampede, and Blue Waters on 1K chips is made.

- Measurement and analysis of frequency, power, and temperature of processors on Edison, Cori and Minsky is made.

- Analysis of potential solutions to mitigate the effects of frequency variation including: disabling Turbo Boost, replacing slow chips, idling cores, and dynamic task redistribution is evaluated.

- A speed-aware dynamic task redistribution technique which improves performance up to 18% is demonstrated.
- A simple yet powerful neural network-based temperature prediction model that can predict steady state core temperatures accurately under different core utilization levels, frequency levels, and fan speeds is implemented.
- A proactive fan control mechanism, called "precooling", that removes power oscillations and can reduce the maximum fan power by 45.6% on average as well as energy consumption by 9.4% by predicting core temperatures is proposed.
- An analysis of inter-chip and intra-chip temperature variations in two different architectures using 1,000 cores with 25°C variation is made. It is demonstrated that decoupling the fans reduces the variation down to 10°C, the power by an additional 7.7%, and the energy by 13%. Moreover, it is shown that the remaining 10°C is intra-chip variation and can be reduced to 2°C via thermal-aware load balancing.
- Patent-pending power aware node assembly techniques that can help mitigate power variations and side effects of the power variations are invented.
- An analysis of per-core DVFS capabilities in three different processors.
- A fine-grained runtime technique which provides automated function-level energy efficiency providing 5-10% more reduction in energy compared to per-chip frequency scaling is implemented.
- Identification of use cases for per-core frequency and voltage regulation that motivates the need for implementing per-core voltage regulators in hardware is made.

8.1 Future Directions

Different optimization approaches under the presence of variability presented in this dissertation may not all comply with each other. For example, speed-aware load balancing decisions may conflict with temperature-aware load balancing decisions considering that temperature

does not directly correlate with speed. Another example is that the node assembly method can potentially reduce power variations and remove the need for a speed-aware load balancer.

The variability in HPC data centers are not limited to the ones which are described in this dissertation. This dissertation isolated the other sources and focused on manufacturing related variability. In fact, there are multiple different sources of variability that effect the performance of applications more directly. For example, network variability due to traffic or cross application contention, systems software variability due different versions of compilers or modules, SMP resource variability due to contention in shared caches, and operating system related variability due to kernel process scheduling are some of the important ones [122]. Variability is an important concern for performance portability and reproducibility. Further research is necessary to understand different sources of variability and mitigate their negative effects which manifest in applications over time, run-to-run, or platform-to-platform in a holistic manner not only from performance but also from power and energy efficiency perspective as well.

REFERENCES

[1] "Top 500 list, June 2017," https://www.top500.org/lists/2017/06/.

[2] ASCAC Subcommittee, "Top ten exascale research challenges," *US Department Of Energy Report*, 2014.

[3] A. Shehabi, S. Smith, N. Horner, I. Azevedo, R. Brown, J. Koomey, E. Masanet, D. Sartor, M. Herrlin, and W. Lintner, "United states data center energy usage report," *Lawrence Berkeley National Laboratory, Berkeley, California. LBNL-1005775*, vol. 4, 2016.

[4] T. Patki, N. Bates, G. Ghatikar, A. Clausen, S. Klingert, G. Abdulla, and M. Sheikhalishahi, "Supercomputing centers and electricity service providers: A geographically distributed perspective on demand management in europe and the united states," in *International Conference on High Performance Computing.* Springer, 2016, pp. 243–260.

[5] B. Acun, A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni, and L. V. Kalé, "Power, reliability, and performance: One system to rule them all," *Computer*, vol. 49, no. 10, pp. 30–37, 2016.

[6] B. Acun, P. Miller, and L. V. Kale, "Variation among processors under turbo boost in hpc systems," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926289 pp. 6:1–6:12.

[7] B. Acun and L. V. Kale, "Mitigating processor variation through dynamic load balancing," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, May 2016, pp. 1073–1076.

[8] B. Acun, E. K. Lee, Y. Park, and L. V. Kale, "Support for power efficient proactive cooling mechanisms," in *Proceedings of the International Conference On High Performance Computing, Data and Analytics.* IEEE, 2017.

[9] B. Acun, E. K. Lee, Y. Park, and L. V. Kalé, "Neural network-based task scheduling with preemptive fan control," in *International Workshop on Energy Efficient Supercomputing (E2SC).* ACM, 2016.

[10] B. Acun, E. K. Lee, and Y. Park, "Power efficiency aware node component assembly," July 2017, US Patent Application No: 15/658,494.

[11] O. Sarood, A. Langer, A. Gupta, and L. 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*, ser. SC '14. New York, NY, USA: ACM, 2014.

[12] O. Sarood, P. Miller, E. Totoni, and L. V. Kale, "'Cool' Load Balancing for High Performance Computing Data Centers," in *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[13] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.

[14] O. Sarood, E. 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.

[15] E. Totoni, J. Torrellas, and L. V. Kale, "Using an adaptive hpc runtime system to reconfigure the cache hierarchy," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[16] E. Totoni, N. Jain, and L. Kale, "Power management of extreme-scale networks with on/off links in runtime systems," *ACM Transactions on Parallel Computing*, 2014.

[17] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[18] H. Menon, N. Jain, G. Zheng, and L. V. Kalé, "Automated load balancing invocation based on application characteristics," in *IEEE Cluster 12*, Beijing, China, September 2012.

[19] E. Meneses, X. Ni, G. Zheng, C. Mendes, and L. Kale, "Using migratable objects to enhance fault tolerance schemes in supercomputers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 2061–2074, July 2015.

[20] A. Gupta, B. Acun, O. Sarood, and L. V. Kale, "Towards Realizing the Potential of Malleable Parallel Jobs," in *Proceedings of the IEEE International Conference on High Performance Computing*, ser. HiPC '14, Goa, India, December 2014.

[21] Intel, "Intel-64 and IA-32 Architectures Software Developer's Manual , Volume 3A and 3B: System Programming Guide, 2011."

[22] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 173–182.

[23] E. Totoni, N. Jain, and L. V. Kale, "Toward runtime power management of exascale networks by on/off control of links," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2013 IEEE International Symposium on*, 2013.

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

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

[26] "Cori Supercomputer at NERSC," http://www.nersc.gov/users/computational-systems/cori/.

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

[28] "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.

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

[30] "Stampede supercomputer at TACC," https://www.tacc.utexas.edu/stampede/.

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

[32] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[33] 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.*, vol. 14, no. 3, pp. 189–204, 2000.

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

[35] F. Petrini, D. Kerbyson, and S. 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, Nov. 10–16, 2003.

[36] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, no. 2, pp. 20–27, 2012.

[37] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel® Core i7 Turbo Boost feature," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* IEEE, 2009, pp. 188–197.

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

[39] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, D. 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.* IEEE, 2003, pp. 81–92.

[40] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. 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.

[41] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. 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.* ACM, 2015, p. 78.

[42] B. Austin and N. 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*, ser. E2SC '14. Piscataway, NJ, USA: IEEE, 2014. [Online]. Available: http://dx.doi.org/10.1109/E2SC.2014.7 pp. 51–59.

[43] S. Dighe, S. R. Vangal, P. Aseron, S. Kumar, T. Jacob, K. A. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. 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*, vol. 46, no. 1, pp. 184–193, Jan 2011.

[44] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 363–374.

[45] E. Totoni, "Power and energy management of modern architectures in adaptive HPC runtime systems," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2014.

[46] A. Langer, E. Totoni, U. S. Palekar, and L. 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.

[47] A. Hammouda, A. R. Siegel, and S. F. Siegel, "Noise-tolerant explicit stencil computations for nonuniform process execution rates," *ACM Trans. Parallel Comput.*, vol. 2, no. 1, pp. 7:1–7:33, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2742351

[48] L. Kale, A. Langer, and O. 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.

[49] K. Zhang, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Minimizing thermal variation across system components," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1139–1148.

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

[51] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-aware task scheduling at the system software level," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ser. ISLPED '07. ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1283780.1283826 pp. 213–218.

[52] G. Shipman, P. McCormick, K. Pedretti, S. L. Olivier, K. B. Ferreira, R. Sankaran, S. Treichler, A. Aiken, and M. Bauer, "Analysis of application sensitivity to system performance variability in a dynamic task based runtime." Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[53] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[54] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[55] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained hpc applications," in *International Conference on High Performance Computing*. Springer International Publishing, 2015, pp. 394–408.

[56] L. Wang, S. U. Khan, and J. Dayal, "Thermal aware workload placement with task-temperature profiles in a data center," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 780–803, 2012.

[57] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, "Temperature management in data centers: Why some (might) like it hot," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254778 pp. 163–174.

[58] M. K. Patterson, "The effect of data center temperature on energy efficiency," in *Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008. 11th Intersociety Conference on*, May 2008, pp. 1167–1174.

[59] N. Ahuja, C. Rego, S. Ahuja, M. Warner, and A. Docca, "Data center efficiency with higher ambient temperatures and optimized cooling control," in *Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM), 2011 27th Annual IEEE*, March 2011, pp. 105–109.

[60] T. V. T. Duy, Y. Sato, and Y. Inoguchi, "Performance evaluation of a green scheduling algorithm for energy savings in cloud computing," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–8.

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

[62] W. Huang, C. Lefurgy, W. Kuk, A. Buyuktosunoglu, M. Floyd, K. Rajamani, M. Allen-Ware, and B. Brock, "Accurate fine-grained processor power proxies," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 224–234.

[63] E. K. Lee, H. Viswanathan, and D. Pompili, "Vmap: Proactive thermal-aware virtual machine allocation in hpc cloud datacenters," in *High Performance Computing (HiPC), 2012 19th International Conference on*, 2012, pp. 1–10.

[64] Z. Wang, C. Bash, N. Tolia, M. Marwah, X. Zhu, and P. Ranganathan, "Optimal fan speed control for thermal management of servers," in *ASME 2009 InterPACK Conference collocated with the ASME 2009 Summer Heat Transfer Conference and the ASME 2009 3rd International Conference on Energy Sustainability.* American Society of Mechanical Engineers, 2009, pp. 709–719.

[65] H. Demuth and M. Beale, "Neural Network Toolbox for Use with MATLAB," MathWorks, 2017, http://www.mathworks.com/help/nnet/.

[66] 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*, Baltimore, MD, September 2002, pp. 1–18.

[67] D. Carraway, "lookbusy – a synthetic load generator," 2017, https://www.devin.com/lookbusy/.

[68] J. J. More, "The levenberg-marquardt algorithm: Implementation and theory," *Numerical Analysis, ed. G. A. Watson, Lecture Notes in Mathematics 630, Springer Verlag*, pp. 105–116, 1977.

[69] M. F. Moller, "A scaled conjugate gradient algorithm for fast supervised learning," *NEURAL NETWORKS*, vol. 6, no. 4, pp. 525–533, 1993.

[70] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the rprop algorithm," in *Neural Networks, 1993., IEEE International Conference on*, 1993, pp. 586–591.

[71] Z. Toprak-Deniz, M. Sperling, J. Bulzacchelli, G. Still, R. Kruse, S. Kim, D. Boerstler, T. Gloekler, R. Robertazzi, K. Stawiasz, T. Diemoz, G. English, D. Hui, P. Muench, and J. Friedrich, "5.2 distributed system of digitally controlled microregulators enabling per-core dvfs for the power8tm microprocessor," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 98–99.

[72] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill, "Fivrfully integrated voltage regulators on 4th generation intel® core socs," in *Applied Power Electronics Conference and Exposition (APEC), 2014 Twenty-Ninth Annual IEEE*. IEEE, 2014, pp. 432–439.

[73] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: https://doi.org/10.1007/11549468_24 pp. 196–205.

[74] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229479 pp. 249–258.

[75] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling power and energy usage of hpc kernels," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 990–998.

[76] J. Moore, J. S. Chase, and P. Ranganathan, "Weatherman: Automated, online and predictive thermal mapping and management for data centers," in *2006 IEEE International Conference on Autonomic Computing*. IEEE, 2006, pp. 155–164.

[77] L. Wang, G. von Laszewski, F. Huang, J. Dayal, T. Frulani, and G. Fox, "Task scheduling with ann-based temperature prediction in a data center: a simulation-based study," *Engineering with Computers*, vol. 27, no. 4, pp. 381–391, 2011.

[78] J. Huang, H. Jin, X. Xie, and Q. Zhang, "Using narx neural network based load prediction to improve scheduling decision in grid environments," in *Third International Conference on Natural Computation (ICNC 2007)*, vol. 5. IEEE, 2007, pp. 718–724.

[79] S. Aswath Narayana, "An artificial neural networks based temperature prediction framework for network-on-chip based multicore platform," M.S. thesis, Rochester Institute of Technology, 2016.

[80] K. Zhang, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Minimizing thermal variation across system components," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1139–1148.

[81] F. Beneventi, A. Bartolini, and L. Benini, "Static thermal model learning for high-performance multicore servers," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011, pp. 1–6.

[82] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, http://hdl.handle.net/2142/16578.

[83] D. Tobias, "Forward-looking fan control using system operation information," Feb. 7 2006, uS Patent 6,996,441. [Online]. Available: https://www.google.com/patents/US6996441

[84] "Summitdev Supercomputer at ORNL," https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/.

[85] B. Acun, E. K. Lee, and Y. Park, "Multi-component power-aware job scheduling based on node and application characteristics," Aug. 2017, US Patent App.

[86] R. R. Rao, D. Blaauw, D. Sylvester, and A. Devgan, "Modeling and analysis of parametric yield under power and performance constraints," *IEEE Design & Test of Computers*, vol. 22, no. 4, pp. 376–385, 2005.

[87] B. Jovanović and M. Jevtić, "Static and dynamic power consumption of arithmetic circuits in modern technologies," in *ETRAN, National Conference*, 2011, pp. EL3–6.

[88] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[89] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 338–342.

[90] V. Zolotov, C. Visweswariah, and J. Xiong, "Voltage binning under process variation," in *Proceedings of the 2009 International Conference on Computer-Aided Design*. ACM, 2009, pp. 425–432.

[91] A. Datta, S. Bhunia, J. H. Choi, S. Mukhopadhyay, and K. Roy, "Profit aware circuit design under process variations considering speed binning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 7, pp. 806–815, 2008.

[92] Y.-P. You, C. Lee, and J. K. Lee, "Compiler analysis and supports for leakage power reduction on microprocessors," *Lecture notes in computer science*, vol. 2481, p. 45, 2005.

[93] A. Vassighi and M. Sachdev, "Power, junction temperature, and reliability," *Thermal and Power Management of Integrated Circuits*, pp. 13–49, 2006.

[94] "Aurora Supercomputer at Argonne National Laboratory," http://aurora.alcf.anl.gov/.

[95] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan et al., "Haswell: The fourth-generation intel core processor," *IEEE Micro*, no. 2, pp. 6–20, 2014.

[96] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," 2015.

[97] "Intel to bring back fivr after skylake and kaby lake cpus," https://www.overclock3d.net/news/cpu_mainboard/intel_to_bring_back_fivr_after_skylake_and_kaby_lake_cpus/1.

[98] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya, "Some Observations on Optimal Frequency Selection in DVFS-based Energy Consumption Minimization," *Journal of Parallel and Distributed Computing*, vol. 71, no. 8, pp. 1154–1164, 2011.

[99] B. Austin and N. 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*. IEEE Press, 2014, pp. 51–59.

[100] W. Wang and E. A. Leon, "Evaluating dvfs and concurrency throttling on ibms power8 architecture."

[101] F. Isaila, J. Blas, J. Carretero, R. Latham, and R. Ross, "Design and evaluation of multiple-level data staging for blue gene systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 946–959, 2011.

[102] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–11.

[103] J. Fu, R. Latham, M. Min, and C. D. Carothers, "I/o threads to reduce checkpoint blocking for an electromagnetics solver on blue gene/p and cray xk6," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2012, p. 2.

[104] M. Grossman, M. Breternitz, and V. Sarkar, "Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 762–775, 2016.

[105] J. Bomfim and R. Rothstein, "In-memory database for high performance, parallel transaction processing," July 12 2002, uS Patent App. 10/193,672.

[106] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 487–498.

[107] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[108] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," *ACM Sigplan Notices*, vol. 34, no. 8, pp. 131–140, 1999.

[109] E. Mikida, N. Jain, L. Kale, E. Gonsiorowski, C. D. Carothers, P. D. Barnes Jr, and D. Jefferson, "Towards pdes in a message-driven paradigm: A preliminary case study using charm++," in *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*. ACM, 2016, pp. 99–110.

[110] "Lulesh," http://computation.llnl.gov/casc/ShockHydro/.

[111] "Graph 500 benchmark," http://www.graph500.org/.

[112] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao, "Performance and power modeling in a multi-programmed multi-core environment," in *Proceedings of the 47th Design Automation Conference*. ACM, 2010, pp. 813–818.

[113] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, "Portable, scalable, per-core power estimation for intelligent resource management," in *Green Computing Conference, 2010 International*. IEEE, 2010, pp. 135–146.

[114] L. Wang, G. Von Laszewski, J. Dayal, and F. Wang, "Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 368–377.

[115] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive dvfs and thread packing under power caps," in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*. ACM, 2011, pp. 175–185.

[116] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 143–154.

[117] C.-M. Wu, R.-S. Chang, and H.-Y. Chan, "A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters," *Future Generation Computer Systems*, vol. 37, pp. 141–147, 2014.

[118] E. L. Padoin, M. Castro, L. L. Pilla, P. O. Navaux, and J.-F. Méhaut, "Saving energy by exploiting residual imbalances on iterative applications," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

[119] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. De Supinski, "Adaptive configuration selection for power-constrained heterogeneous systems," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 371–380.

[120] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs," in *SC 2006 conference, proceedings of the ACM/IEEE*. IEEE, 2006, pp. 14–14.

[121] S. Bhalachandra, A. Porterfield, S. Olivier, and J. Prins, "An adaptive core-specific runtime for energy efficiency," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2017.

[122] D. Skinner and W. Kramer, "Understanding the causes of performance variability in hpc workloads," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2005, pp. 137–149.