THE CASE FOR HARDWARE OVERPROVISIONED SUPERCOMPUTERS

by

Tapasya Patki

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2015

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Tapasya Patki
entitled The Case for Hardware Overprovisioned Supercomputers
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: 7 July 2015
David K.Lowenthal

Date: 7 July 2015
Saumya K. Debray

Date: 7 July 2015
John H. Hartman

Date: 7 July 2015
Barry L. Rountree

Date: 7 July 2015
Martin W. J. Schulz

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: 7 July 2015
Dissertation Director: David K. Lowenthal

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

SIGNED: Tapasya Patki

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. David Lowenthal for his technical guidance, encouragement and relentless support over the years. Dave gave me the freedom to explore new ideas, motivated me to seek depth in my research, taught me how to ask the right questions, and helped me overcome challenges. He always inspired me to not lose sight of the big picture, and pushed me to seek simplicity when solving difficult problems. Dave patiently and tirelessly read all my drafts, sat through every single practice talk, and gave constructive feedback while preserving his great sense of humor. Thank you, Dave, for being such an inspiring advisor.

I wish to thank Dr. Martin Schulz and Dr. Barry Rountree for providing me with the opportunity to intern at Lawrence Livermore National Laboratory and for serving on my committee. Martin always gave insightful comments on research, edited my paper drafts, and taught me how to develop better academic and industrial collaborations. I also know a lot more about beer today than I did before because of Martin.

I remember being stuck with a script problem when I first met Barry at a conference. He helped me by giving me a three-minute bash tutorial, followed by a fifteen-minute introduction to R, further followed by a discussion about his favorite books on evolution. I have learned a lot from my conversations with Barry. His unique way of visualizing data has enabled me to present my research better, and his unconventional approach to problem solving has pushed me to think outside the box.

I would like to thank my committee members, Dr. Saumya Debray and Dr. John Hartman, for their timely feedback and suggestions on my research. I also want to thank Dr. Bronis R. de Supinski for his valuable advice over the years.

I had the opportunity to work with some amazing fellow labmates. Thank you, Gregory Striemer, Peter Bailey, Anjana Sasidharan, Kathleen Shoga, Timothy Meyer, Neha Gholkar, Matthias Maiterth, Joachim Protze and Rogelio Long, for the intense late-night debugging sessions, technical discussions and funny conversations. A special thanks to my friends Saravanakrishnan Krishnamoorthy, Jinyan Guan, Kate Kharitonova, Sabrina Nusrat, Aarthi Goverdhan, Aditi Kapoor, Samitha Manocha, Parul Shukla, and Pratibha Pandey for their support and motivating words.

I also wish to thank Janhavi Sabnis, Akshat Gupta, Saurabh Maniktala, Anisha Goel, Truc Nyugen and Shloka Desai, for being as crazy as me. Thank you for making me laugh hysterically, for cheering me up when things got rough, and for being my family in the US. You all are the best friends anyone could ever ask for. And thank you, Shloka, for help with proofreading this dissertation.

Finally, I want to thank my family for always believing in me and for their unconditional love and support. Thank you, Varsha Patki, for putting up with this younger cousin of yours, and for being there for me. I will always look up to you.

DEDICATION

To my parents, Anuradha and Arunkumar Patki, for everything.
Words can never express how much I love you.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

ABSTRACT

Power management is one of the most critical challenges on the path to exascale supercomputing. High Performance Computing (HPC) centers today are designed to be worst-case power provisioned, leading to two main problems: limited application performance and under-utilization of procured power. In this dissertation we introduce *hardware overprovisioning*: a novel, flexible design methodology for future HPC systems that addresses the aforementioned problems and leads to significant improvements in application and system performance under a power constraint.

We first establish that choosing the right *configuration* based on application characteristics when using hardware overprovisioning can improve application performance under a power constraint by up to 62%. We conduct a detailed analysis of the infrastructure costs associated with hardware overprovisioning and show that it is an economically viable supercomputing design approach. We then develop RMAP (Resource MAnager for Power), a power-aware, low-overhead, scalable resource manager for future hardware overprovisioned HPC systems. RMAP addresses the issue of under-utilized power by using *power-aware backfilling* and improves job turnaround times by up to 31%. This dissertation opens up several new avenues for research in power-constrained supercomputing as we venture toward exascale, and we conclude by enumerating these.

CHAPTER 1

INTRODUCTION

As of June 2015, the world's fastest supercomputer, Tianhe-2, delivers 33.8 petaflops[1] of sustained performance while using 17.6 MW of power. As we push the boundaries of supercomputing further, power becomes a limiting and expensive resource. Effectively translating this limited power resource into performance and utilizing it fully are thus critical challenges in the next generation of supercomputing. The U.S. Department of Energy has set an ambitious target of achieving an exaflop[2] under a constraint of 20 MW by 2023 (Department of Energy, 2014)—thus requiring a 30-fold improvement in performance with merely 14% increase in power and rendering existing approaches obsolete. This has triggered research both in the semiconductor industry and the supercomputing community, with the former focusing on designing efficient processor, memory and network technologies, and the latter focusing on utilizing this new hardware and developing better system software to manage power and improve performance while ensuring reliable operation. Both these directions for research are complementary and are essential in order to achieve exascale.

The focus of the research conducted in this dissertation is to optimize application as well as system performance in power-constrained supercomputing. The key contributions include the idea of *hardware overprovisioning* and the design and implementation of associated system software necessary to realize its benefits in supercomputing. Hardware overprovisioning is a novel technique that improves performance significantly under an application-level power bound. We study its performance benefits and analyze its economic viability so as to understand the associated tradeoffs. We also present RMAP (Resource MAnager for Power), a resource manager targeted at future hardware overprovisioned and power-limited systems. We

[1] A supercomputer that can perform 10^{15} floating point operations per second

[2] Or, 10^{18} floating point operations per second

analyze how RMAP can be deployed on a real system with multiple users and show how it can improve system throughput and power utilization. The following subsections provide an overview of the contributions of this dissertation.

1.1 Hardware Overprovisioning

Supercomputers today are designed to be *worst-case power provisioned*, implying that all nodes in the system can run at peak power *simultaneously*. While this power allocation strategy is useful for a few power-hungry benchmarks such as High-Performance Linpack, it leads to inefficient use of power for most HPC production applications that fail to utilize the allocated peak power on a node.

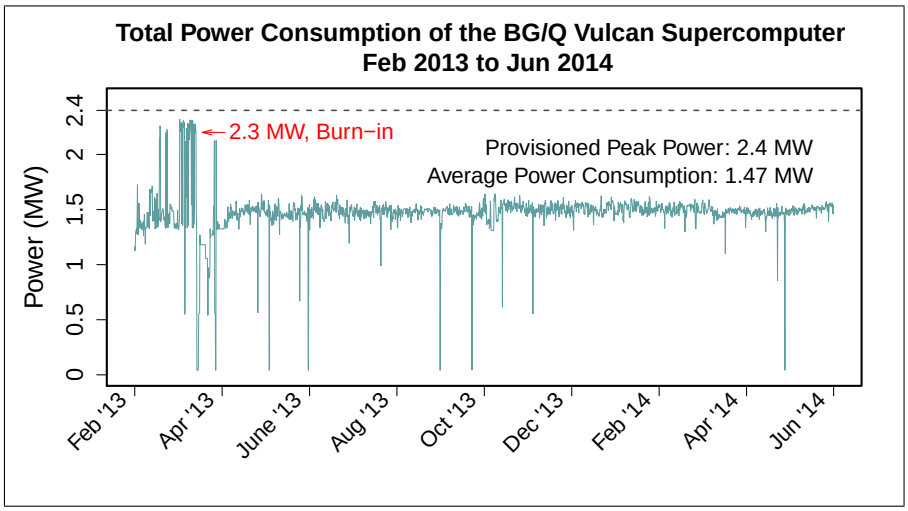


Figure 1.1: Power Consumption on Vulcan

One of the key limitations of worst-case provisioning in supercomputing design is underutilization of the power infrastructure. An example of this can be found in data collected on Vulcan (see Figure 1.1), which is a BlueGene/Q system located at Lawrence Livermore National Laboratory (LLNL). As of June 2015, Vulcan is the tenth-fastest supercomputer in the world with 393,216 compute cores and has been provisioned with 2.4 MW of power. Power consumption data was gathered from Vulcan every three minutes by LLNL's Livermore Computing division for over 16 months. This data

shows that applications used only 1.47 MW on average, with the only exception being the burn-in phase. This under-utilization of power has many negative ramifications, such as the use of lower water temperatures than needed for cooling—which leads to additional power wasted on water chillers.

Ideally, supercomputing centers should utilize the procured power fully to accomplish more science, especially because power is a limited and expensive resource. Thus, a more flexible and efficient design approach is to build a *reconfigurable* system that has more capacity (nodes) under the same site-level power constraint and can adapt to requirements of the applications. Such a system can provide limited power to a large number of nodes, peak power to a smaller number of nodes, or use an alternative allocation in between based on application characteristics. This approach is referred to as *hardware overprovisioning* (or overprovisioning for short).

Good performance on an overprovisioned system relies on choosing a *configuration* that takes the application's characteristics into consideration. We define a configuration as a tuple consisting of three values—number of nodes, number of cores per node, and power bound per node. Our research demonstrates that choosing the right configuration on an overprovisioned system can reduce individual application execution time under a power bound by up to 62% and by 32% on average when compared to worst-case provisioning (Patki et al., 2013), thus improving performance by 2.63x (1.47x on average). For example, for applications that are highly scalable, running more nodes at lower power per node might result in shorter execution times for the same amount of power. Similarly, for applications that are memory-bound, running fewer cores per node may result in the best performance.

1.2 Economic Viability of Overprovisioning

An overprovisioned system has more capacity than it can simultaneously fully power. Such a system can provide significant performance improvements in power-constrained scenarios. However, these benefits come with an additional hardware and infrastructure cost, and it is essential to conduct a thorough cost-benefit analysis before investing in

large-scale overprovisioned systems. The hardware cost for an overprovisioned system depends on several factors, such as the cost of the underlying processor architecture and the high-speed interconnection network. A hardware cost budget thus provides a choice between purchasing fewer, high-end processors that result in a worst-case provisioned system, or more low-cost, previous generation processors that can be overprovisioned. We develop a model to compare the costs and benefits of worst-case provisioning with overprovisioning, thus providing an analytical tool for future HPC system designers. We show that it is possible to achieve better performance with overprovisioned systems while keeping the system acquisition cost constant.

1.3 Power-Aware Resource Management

For overprovisioning to be practical, system software such as resource managers and runtime systems need to be made power-conscious. With this goal in mind, we developed RMAP (Resource MAnager for Power) (Patki et al., 2015), a low-overhead, scalable resource manager targeted at future power-constrained systems. Similar to existing schedulers, users request a node count and time limit for their job. Traditional resource managers use this information to determine a physical node allocation for the job based on a scheduling policy. Jobs have to wait in the job queue until enough nodes are available, and are terminated if they exceed their requested time limit. The traditional assumption is that the job will be allocated all the power (and cores) on the requested nodes.

Future power-limited systems, however, will have to enforce job-level and node-level power bounds. In order to address this, RMAP translates the user's node request to a job-level power bound in a fair-share manner by allocating the job power proportional to the fraction of nodes of the total system that it requested. Within RMAP, we propose and implement an *adaptive* policy based on overprovisioning and *power-aware backfilling*. The main goals for this policy are to improve application performance as well as system power utilization while adhering to the derived job-level power bound. Additionally, the policy also strives to minimize the average turnaround time for all

jobs in the job queue. Power-aware backfilling is a simple, greedy algorithm that trades some of the performance benefits (raw execution time) obtained from overprovisioning for faster turnaround times and shorter queue wait times. Additionally, two other baseline policies that represent naive overprovisioning and worst-case provisioning are implemented within RMAP. The naive overprovisioning policy finds the best performing configuration under the derived job-level power bound; and the policy based on worst-case provisioning guarantees safe, correct execution by allocating the requested nodes at peak power to the job while adhering to the power constraint.

We implement RMAP by extending SLURM, a popular, open-source HPC resource manager that is deployed on several of the Top500 supercomputers (Yoo et al., 2003). Our results indicate that the adaptive policy can increase system power utilization while adhering to strict job-level power bounds and can lead to 31% (19% on average) and 54% (36% on average) faster average job turnaround times when compared to the policies based on worst-case provisioning and naive overprovisioning, respectively (Patki et al., 2015). It is important to note that naive overprovisioning may result in worse turnaround times, and that future supercomputing system software needs to be intelligent and adaptive.

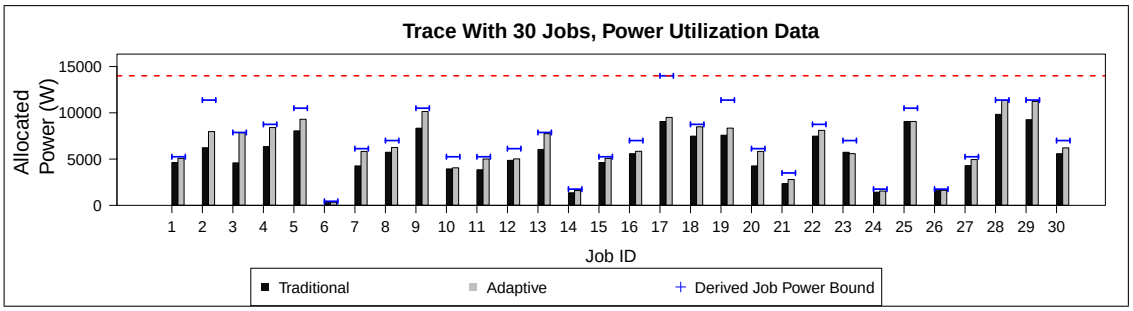


Figure 1.2: Power Utilization

Figure 1.2 depicts power results of the adaptive and traditional policies within RMAP on a random job trace in a scenario with unconstrained power (14,800 W on a 64-node system, represented by a dashed red line). The y-axis is the power allocated to the job for execution, the x-axis represents the Job IDs, and the blue lines on the

top indicate the derived job-level power bound using fair-share. As can be observed from this graph, the adaptive policy always utilizes power better than the traditional policy. For some jobs, such as Job 3, the adaptive policy utilizes 70% more power than the traditional policy. Overall, the adaptive policy utilizes 17% more per-job power, leading to significantly less wasted power and faster turnaround times.

1.4 Summary of Contributions

In summary, this dissertation makes the following contributions:

- We introduce hardware overprovisioning and show that it can improve performance under an application-level power bound by up to 62% (32% on average).
- We present a detailed study on the cost-benefit analysis of hardware overprovisioning and develop a model that HPC system designers can use to determine whether overprovisioning results in a net benefit for their site.
- We describe RMAP, a resource manager targeted at future power-limited systems. Within RMAP, we design the *Adaptive* policy which uses hardware overprovisioning and power-aware backfilling. We show that this novel policy results in up to 31% (19% on average) faster turnaround times for a job queue when compared to a simple, traditional policy based on worst-case provisioning, and that it also improves overall system power utilization.

The rest of this dissertation is organized as follows. Background information about power management and scheduling in high-performance computing systems is provided in Chapter 2. Chapter 3 introduces hardware overprovisioning and highlights its benefits. Chapter 4 analyzes the economic viability of hardware overprovisioned supercomputers. Chapter 5 introduces RMAP (Resource MAnager for Power), which is a job scheduling system targeted toward future power-limited supercomputers. Chapter 6 presents related work in the area, and finally, Chapter 7 concludes this dissertation and describes future research.

CHAPTER 2

BACKGROUND

In this chapter, we present background information about power, scheduling and economic terminology used in the context of supercomputing. Section 2.1 provides an overview of the organization of large-scale HPC facilities. Sections 2.2 and 2.3 focus on processor power measurement and management techniques. Section 2.4 introduces economic terms that are used in Chapter 4. Section 2.5 discusses backfilling, a common job scheduling technique used to improve system utilization in current supercomputing systems.

2.1 High Performance Computing (HPC) Basics

A high-performance computing facility consists of parallel computing systems with several thousand nodes linked together to provide high computational capacity in order to solve some of the most pressing problems in science and engineering. A typical system is laid out as a combination of login, compute and gateway nodes. Login nodes generally provide access to the main system and are shared by multiple users. They are interactive and can be used for tasks such as compiling applications, submitting job requests, launching GUI interfaces and sometimes debugging. Only a small percentage of the system is comprised of login nodes. Users log in to the HPC system with these nodes and submit requests to access compute nodes. These requests are granted by system resource managers.

Compute nodes determine the computational strength of an HPC system and are used to run critical applications. In general, compute nodes are not shared and are not interactive. Users request a set of compute nodes to execute their applications with the help of a resource management system (such as SLURM, discussed in Section 2.5.2). Compute nodes typically consist of multiple sockets (each with multiple cores)

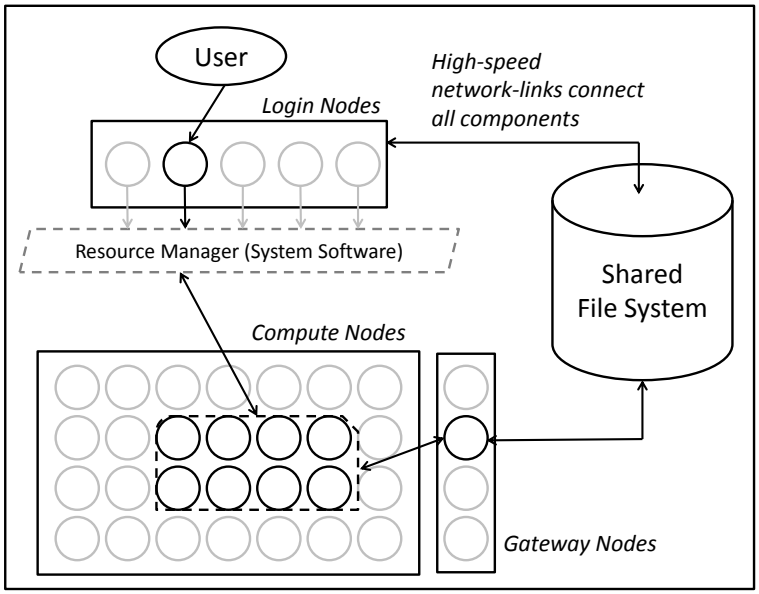


Figure 2.1: Typical HPC System

and associated memory. They may also have additional processing components or accelerators, such as General Purpose Graphics Processing Units (GPGPUs) or Xeon Phis. Login nodes and compute nodes may have similar architectures.

Compute nodes are organized in racks and are connected with other compute nodes as well as other components of the HPC system with the help of a high-speed interconnect (typically InfiniBand). The layout of the underlying network topology (mesh, tree, or torus) usually determines the layout of the compute nodes in the HPC system. HPC compute nodes are usually diskless and do not have local storage. They are connected to persistent storage servers with the help of gateway nodes. Storage servers host parallel file systems, which provide fast, high-bandwidth access and are also fault-tolerant. An example of such a file system is Lustre (Koeninger, 2003). Figure 2.1 shows this layout and also depicts how a user interacts with a typical HPC system.

HPC applications can be *strongly-scaled* or *weakly-scaled*. For strong scaling, the total input problem size stays fixed as more compute nodes are added to the application. In the case of weak scaling, the problem size per node stays constant. Thus, when more nodes are added, the total input problem size changes.

2.2 Power Basics

A typical supercomputing site is organized in a hierarchical manner. Each site has a few machine rooms, and each machine room hosts a few HPC systems. Each of these HPC systems have various components as described in Section 2.1. In the future, power-constrained supercomputing facilities will have to enforce power bounds at each level in the aforementioned hierarchy to ensure reliable operation. Site-level power constraints will apply across multiple machine rooms. Electricity service providers as well as the infrastructure availability at the facility will determine the amount of power that can be brought into each machine room. These machine-room level power bounds will then be translated to system-level power bounds. Each HPC system will have its own power bound, so that all these systems can function effectively and simultaneously within the power budget of the machine room. These system-level power bounds will further be inherited by jobs and applications, which in turn will translate to component (or node) level power bounds. Most of these power bounds, including the site-level constraints will be dynamic owing to different sources of energy generation (Bates et al., 2015).

Figure 2.2 shows these inherited power bounds. Techniques such as hardware overprovisioning and power-aware job scheduling, which are the focus of this dissertation, will be used at the system-level. Additionally, runtime systems will need to balance power (in addition to load-balancing) and improve performance at the application-level. These steps can be accomplished by using node-level power measurement and management techniques that are typically supported by the underlying hardware.

2.2.1 Processor Power

All the components described in Section 2.1 (nodes, interconnects, storage servers etc.) contribute to the total power draw of an HPC system, which has two key components: *base* power and *dynamic* power. Base power (sometimes referred to as idle or static power) is the amount of power required to operate the HPC system without an active

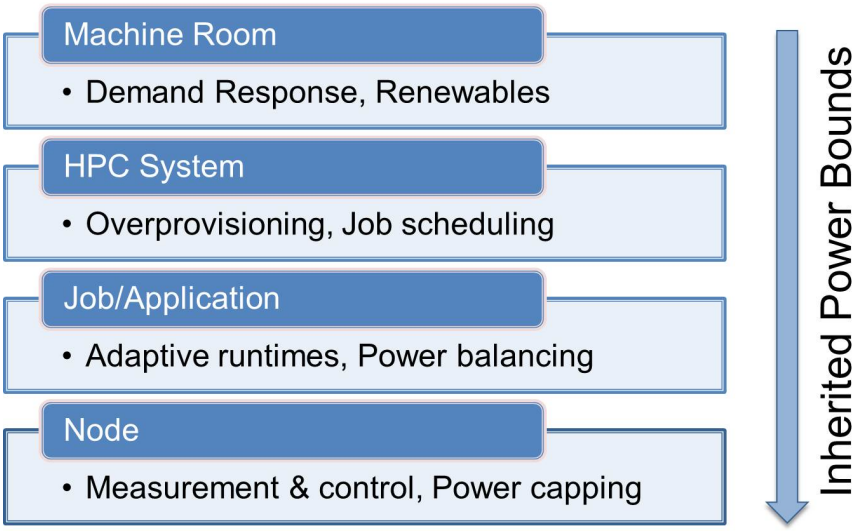


Figure 2.2: Inherited Power Bounds

workload (that is, no jobs are running and the system is idle). It is thus the minimum amount of power that the HPC facility needs to be able turn on all the components in the system and leave them in idle state. This typically includes the storage server and interconnect power and the power consumed by the nodes when they are inactive. Dynamic power, on the other hand, depends on the workload or application. The main contributors to dynamic power are nodes that are executing active applications (thus their CPU(s) and memory are in use).

Traditionally, HPC systems are designed to be *worst-case power provisioned*. Worst-case provisioned systems are designed to account for the maximum possible power consumption. Node components (CPU and memory) typically have a vendor-specified Thermal Design Point (TDP) that indicates the maximum power draw associated with the component. The maximum power consumption for an HPC system can thus be determined by the sum of the TDPs of all the nodes and other system power (such as disks and interconnects). In reality, though, nodes rarely reach their TDP limits when running real applications. Allocating all the power on a node to an application is thus unnecessary.

In order to minimize the wastage of procured power (refer to Figure 1.1 from Chapter 1), it is critical to understand the power profiles of scientific applications and their contributions to dynamic power. The focus of this dissertation is thus on CPU and memory power consumption. In general, CPU power consists of the power for cores and for on-chip caches and depends on the operating frequency of the processor (Mudge, 2001). More specifically, CPU power can be defined as shown in Equation 2.1 and has three parts. Dynamic or switching power, P_{dyn}, is associated with charging and discharging of capacitance and corresponds to switching from a 0 to 1 and vice versa. Short-circuit power, or P_{sc}, is a small amount of power that is consumed during brief, momentary transitions. Finally, leakage power, or P_{leak}, is the power *lost* due to transistor leakage current. In current circuits, the main component is P_{dyn}, which depends on the frequency of operation, f, and the supply voltage, V_{dd}, as shown in Equation 2.2.

$$P_{cpu} = P_{dyn} + P_{sc} + P_{leak} \tag{2.1}$$

$$P_{dyn} = CV_{dd}^2 f \tag{2.2}$$

Thus, power consumption in processors is controlled by changing the frequency of operation or by scaling the supply voltage. Doing so during the execution of a user application is commonly referred to as Dynamic Voltage and Frequency Scaling (DVFS). DVFS has been widely applied to power and energy research in HPC (Cameron et al., 2005; Hsu and Feng, 2005; Springer et al., 2006; Ge et al., 2007; Li and Martìnez, 2006; Li et al., 2004; Ge et al., 2007; Kappiah et al., 2005; Rountree et al., 2007, 2009). Memory power consumption also depends on the operating frequency of the associated DRAM (David et al., 2010). Techniques such as DVFS, however, are not available for current memory technologies due to the volatile nature of random-access memories (Mittal, 2012).

2.2.2 Power and Energy

Research in power-constrained supercomputing is often confused with energy efficiency research for data centers. It is important to thus clarify the difference between the two. The main goal for HPC centers is to accomplish useful science. This can be realized by utilizing all the procured power efficiently and by translating it into application performance and system throughput. Saving energy or power is typically considered secondary as long as a site-wide power constraint is met. HPC facilities also have symbiotic contractual relationships with their electricity providers that support these goals.

For data centers, on the other hand, the focus typically is to meet a service-level agreement. Performance can be compromised in order to minimize the operating cost as long as the desired service level is met. Thus saving energy and reducing electricity utility bills at the cost of reduced application performance is a viable option for data centers.

Saving energy for HPC systems subject to a small, acceptable delay has already been studied widely (Cameron et al., 2005; Hsu and Feng, 2005; Springer et al., 2006; Ge et al., 2007; Li and Martìnez, 2006; Li et al., 2004; Ge et al., 2007; Kappiah et al., 2005; Springer et al., 2006). It has however been demonstrated that it is possible to achieve significant energy savings without incurring any performance penalties in HPC systems (Rountree et al., 2007, 2009). As a result, the goal for future HPC system designers is to maximize performance under a power constraint.

2.3 Power Measurement and Control Techniques

Several vendor-specific and agnostic mechanisms to measure and manage node power have been developed. Some of these mechanisms use sensors and some others use performance counter based models to estimate component power. The techniques used in this dissertation include Intel's Running Average Power Limit (RAPL), Penguin Computing's PowerInsight (PI) and IBM's Environmental Monitor (EMON). We discuss these techniques in detail below. Of these three, RAPL is the only technique that

allows us to constrain and manage power at present. It is expected that more systems will provide such capabilities in the future. Table 2.1 summarizes these techniques and shows the granularity of measurement available in each case. For most of our work, we use RAPL, with the exception of the processor manufacturing variability study presented in Chapter 7. We primarily focus on module power (CPU sockets and DRAM) in this work, mostly because it can be controlled dynamically in HPC systems and can be used to constrain and manage power at a fine granularity. Other components, such as interconnects, do not provide this dynamic control and account for static or base power consumption.

2.3.1 Running Average Power Limit (RAPL)

RAPL is a power management interface that has been introduced with the Intel Sandy Bridge microarchitecture (Intel, 2011; David et al., 2010). It supports on-board power measurement and *hardware power capping* across two main domains—*Package* (PKG or CPU) and *Memory* (DRAM). The PKG domain represents the processor die (cores and on-chip caches), and the DRAM domain includes directly attached memory. Additionally, power measurement is supported across the *Power Plane 0* (PP0) and *Power Plane 1* (PP1) domains. PP0 covers mostly the cores, while PP1 covers so-called *uncore* devices (for example, the off-chip last-level cache, the Intel QuickPath Interconnect or an integrated graphics card). For the Intel Sandy Bridge microarchitecture, there are two processor models, the *client* (family: 0x06, model: 0x2A) and the *server* (family: 0x06, model: 0x2D). The client model supports the PKG, PP0 and the PP1 domains, and the server model supports the PKG, PP0 and DRAM domains. We use the Sandy Bridge server model in our work.

The interface to RAPL is implemented with the help of programmable Machine Specific Registers (MSRs). To access these MSRs, developers can use the Linux *msr* kernel module. This kernel module exports a file interface at `/dev/cpu/N/msr` that can be used to read from or write to the MSRs given appropriate permissions using the `rdmsr` and `wrmsr` instructions. Note that `rdmsr` and `wrmsr` are privileged instructions that have to execute from within the kernel space on protection ring 0. Each RAPL

domain has a 32-bit, read-only `ENERGY_STATUS` MSR for measurement, which is updated approximately every millisecond. This MSR is expected to roll over within hours owing to the precision of the unit used (joules). Average power is calculated by using this MSR and by dividing the accumulated joules by elapsed time.

RAPL also supports power capping across the PKG and DRAM domains. To enforce a power cap using RAPL, users can specify a power bound and a time window in the `POWER_LIMIT` MSR, and the hardware ensures that the average power over the time window does not exceed the specified bound in the requested power domain. The `POWER_INFO` set of registers provides information on the *thermal specification power*, the lowest power bound and the largest time window supported. For the Sandy Bridge server model, the lowest PKG power bound is 51 W, the thermal specification is 115 W, the largest possible time window is 20 seconds, and the maximum power rating is 180 W. Additionally, the RAPL interface includes the `MSR_RAPL_POWER_UNIT` read-only register that lists the units for power, energy and time in Watts, Joules and Seconds respectively, at architecture-specific precision. The Sandy Bridge server model has units of 0.125W, 0.0000152J and 0.000977 seconds. DRAM power capping is often not supported in mainstream processors and is considered a test feature. Techniques other than DVFS, such as controlling the burstiness and flow of memory traffic, are used to control DRAM power with RAPL.

Intel also has a Turbo Boost feature, which accelerates processor performance dynamically by allowing cores to run faster than the rated operating frequency in some scenarios. Whether Turbo Boost will be used depends on the workload and the ambient temperature. An MSR can be used to enable or disable the Turbo mode, but the operating frequency of the processor in the Turbo mode cannot be directly controlled in software. While the maximum Turbo frequency can be set in software, the actual frequencies of operation are determined at runtime.

To facilitate power measurement of HPC applications, we developed the *librapl* library to ensure safe user-space access of Intel's RAPL MSRs. This library uses the MPI profiling interface to intercept `MPI_Init()` and `MPI_Finalize()` calls to set up the necessary MSRs. Thus, we do not have to modify application source code to measure

or cap power. The library can also sample MPI programs at a desired time interval and intercept every MPI call in the application to gather timing and energy/power information. It further infers the operating frequency for each core using the `APERF` and `MPERF` MSRs (Intel, 2011). Our library can be downloaded from `https://github.com/tpatki/librapl`. A modified version of this library with an improved interface has been recently released by Lawrence Livermore National Laboratory (Shoga et al., 2014) in production form (now referred to as `libmsr`).

2.3.2 IBM BlueGene/Q Environmental Monitor (EMON)

BlueGene is an IBM project for designing power-efficient petaflop supercomputers. Three system architectures—BlueGene/L, BlueGene/P, and BlueGene/Q—have been designed in this project. BlueGene/Q (BG/Q) is the third (and current) generation in the BlueGene series of supercomputers. Each rack of a BG/Q machine houses two midplanes, eight link cards, and two service cards. Each midplane has 16 node boards, each of which has 32 compute cards (nodes) connected by a 5D-torus. Power measurement is supported at the node board level. A compute card has 17 active cores, 16 of which are application cores. The 17th core is used for system software. There is an additional 18th core that has been deactivated and is only used to increase manufacturing yield. Measurements are supported across seven different domains, the main ones being chip cores (CPU) and chip memory. Other domains include HSS network transceiver and compute/link chip, Chip SRAM, Optics, Optics/PCIExpress and Link chip core.

To facilitate power measurement, each node board is connected to an FPGA over the EMON bus, which in turn is connected to two Direct-Current Assemblies (DCAs). The DCAs have a microcontroller that periodically calculates instantaneous power (Wallace et al., 2013a,b; Yoshii et al., 2012) and communicates with the FPGA over the I2C bus. The FPGA can relay data to the BG/Q compute nodes over the EMON bus. IBM provides an EMON API to access power consumption data.

Table 2.1: Power Measurement Techniques

Technique	**Reported**	**Granularity**	**Power Capping**
RAPL	Average	1 ms	Yes
PowerInsight	Instantaneous	1 ms (or less)	No
BGQ EMON	Instantaneous	300 ms	No

2.3.3 PowerInsight (PI)

PI is an architecture-independent, sensor-based technique to monitor node power (DeBonis et al., 2012; Laros et al., 2013). The PI architecture consists of three components: (1) a harness with sensor modules for measurement; (2) a cape, or carrier board, with three Analog-to-Digital-Converters (ADC) connected to the sensor modules, and; (3) a BeagleBone core with an ARM Cortex A8 processor, 256 MB of memory, and USB/Ethernet connectivity to the primary node being measured.

Each sensor module is an Allegro ACS713 Hall Effect current sensor and a voltage divider. Typically, the BeagleBone is plugged in to the motherboard using the USB port which provides console connectivity to the node. CPU and DRAM power consumption data can be measured by using the *getRawPower* software utility or with the provided PI-API.

2.3.4 Limitations

At present, a general standard for measuring and managing power does not exist. The Energy Efficient High Performance Computing Working Group (EE HPC WG) has started working towards this goal recently (EE HPC WG, 2014). Current vendor-specific techniques, such as Intel's RAPL, are targeted at certain specific architectures and are not portable. While PI is an architecture-independent mechanism, it has not been studied widely yet and is difficult to deploy at scale.

Another key concern with current measurement and control techniques is their accuracy. Model-based techniques (such as RAPL on the Sandy Bridge

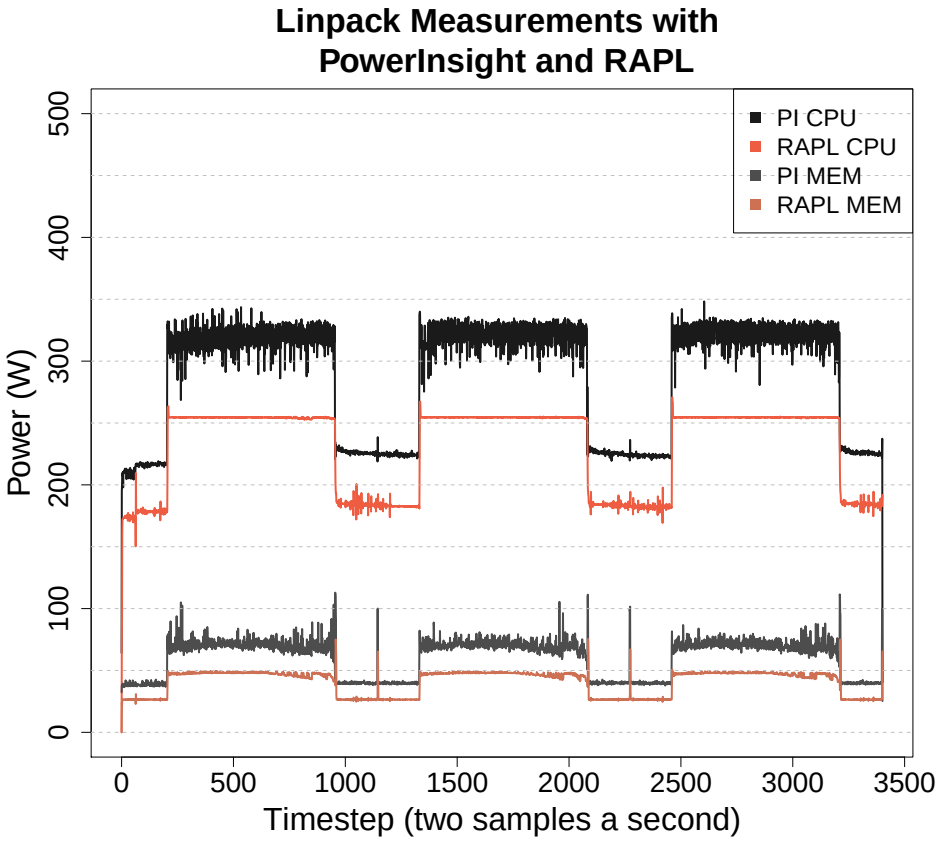


Figure 2.3: Comparison of RAPL and PI on High-Performance Linpack

micro-architecture) are subject to prediction errors, but these error margins are not publicly available. The general agreement in the community is that the error is under 10% (Rotem et al., 2012). Sensor-based and power meter-based measurements are also subject to errors and need to be analyzed carefully. For some measurements, it can be unclear as to what exactly is being measured. An example of this can be seen in Figure 2.3. This figure depicts power measurements using RAPL and PI when High-Performance Linpack was running on a 24-core Ivy Bridge node. As discussed earlier, RAPL and PI have similar granularity of measurement (about 1 ms). RAPL reports average power over the 1 ms time window, and PI reports instantaneous power. In the figure, we show power measurements reported by RAPL and PI for both CPU and memory. As can be observed, these measurements do not coincide and there is a

wide gap in the reported data. One explanation for this gap from Intel is that the PI sensors are measuring the voltage regulator power in addition to the component power. This hypothesis, however, has not been tested yet. It is thus critical to understand what is being measured in each case, how this measured information is being reported, and what the error bounds are. In this dissertation, we use Intel's RAPL, primarily because it gives us the ability to control and cap power.

2.4 Basic Economic Terminology

For any HPC system, two main expenditures need to be considered for procurement—*capital* expenditures (or, CapEx) and *operational* expenditures (or, OpEx). CapEx refer to initial investment costs that are made upfront. For example, the purchasing cost of servers or the construction cost of the machine room (including space and cooling infrastructure). These costs depreciate over time. OpEx, on the other hand, represent recurring costs that are necessary to successfully run the HPC system. These may include maintenance and support costs for the hardware, repair costs for any failed components, electricity costs and human resources costs. Power is a critical component of both CapEx and OpEx. With power, we need to include hardware costs, conditioning and delivery costs, electricity costs and cooling costs. HPC system procurement costs are typically business sensitive and limited information is available in the public domain. The Magellan Report provides information for the Hopper supercomputer (Department of Energy, 2011), and we discuss this in Chapter 4.

2.5 Resource Management Basics

On a traditional HPC system, users submit jobs by specifying a node count and an estimated runtime. These resource requests are maintained in a *job queue*, and users are allocated dedicated nodes based on a scheduling policy. The job executes when the resource manager acquires the specified number of nodes. If the job exceeds the estimated runtime, it is terminated. Depending on the size (node count) of their request, most HPC users are required to use specific partitions within the system. For example,

most high-end clusters have a *small* debug partition that specifically targets small-sized jobs and a general-purpose *batch* partition for medium and large-sized jobs.

One example of a scheduling policy is First-Come First-Serve (FCFS), which is non-preemptive and services jobs strictly in the order that they arrive. Although FCFS is simple to implement and ensures fairness, it can block several small jobs in the job queue when a job requesting more resources (large node count) is waiting for its request to be granted. FCFS also leads to fragmentation and longer wait times as nodes are not utilized efficiently (several nodes remain idle as the requirements of the next job cannot be met immediately).

There are also policies that do not dedicate nodes to jobs, such as *gang scheduling* (Feitelson and Jette, 1997; Setia et al., 1999; Batat and Feitelson, 2000). With gang scheduling, jobs can time-share nodes in a coordinated manner. That is, nodes are not exclusively allocated to a single job. Gang scheduling policies use explicit communication to relay global information and system state among participating jobs, leading to multiple points of failure and high overheads. Implicit coordinated scheduling policies that infer system state by observing local events have been proposed to address some of the aforementioned problems (Andrea Apraci-Dusseau, 1998). While these policies may improve utilization and overall job turnaround time in some cases, they are not considered to be a feasible option in supercomputing due to the high context-switching and paging costs involved.

Resource management policies also have to consider other aspects such as job priorities and user fairness. Scheduling policies may assign priorities to jobs, and provide these jobs a higher quality of service and faster turnaround times. Typically, in HPC systems, priorities are implemented with the help of separate job queues. All jobs in a particular job queue have the same scheduling priority. Queues with higher priorities are thus given preference. Scheduling policies also need to ensure fairness to all the users of the HPC facility. A policy is *fair-share* when it ensures that all the resources in the HPC system (nodes and power, for example) are equally distributed among all users of the system, and no users or jobs are given preferential treatment, unless they are associated with a higher priority.

2.5.1 Backfilling

Backfilling algorithms (Lifka, 1995; Mu'alem and Feitelson, 2001; Skovira et al., 1996) address the problems of FCFS by executing smaller jobs out of order on the idle nodes while the next job (typically large) is waiting for resources. Backfilling is thus an optimization of the FCFS algorithm that improves utilization by moving smaller jobs forward, thus reducing job wait times for some jobs and in turn reducing the overall average turnaround time. Backfilling frequently uses a greedy algorithm that picks the *first-fit* from the job queue. The *first-fit* might not always be the *best-fit*, and a job further down the queue may be a better fit for the hole being backfilled. Finding the *best-fit* involves scanning the entire job queue, which increases job scheduling overhead significantly (Shmueli and Feitelson, 2003). Because of this, most practical resource managers use *first-fit* based backfilling.

A typical job queue for an HPC system is shown in Figure 2.4. Each job is represented by a box that depicts its requested node count and estimated runtime and the box area represents the size of a job. In our example, `A` is a modestly-sized job, and `B` and `C` are examples of large and small jobs respectively. Figure 2.4 also compares FCFS and backfilling. The y-axis in the figure represents available nodes, and the x-axis denotes real time (starting with the current time). While FCFS follows the order in which jobs arrive strictly, backfilling identifies opportunities to utilize the idle nodes by moving jobs `C` and `D` forward when job `A` is waiting for resources. This improves both node utilization as well as overall turnaround times.

There are two variants of backfilling— *easy* and *conservative*. Easy backfilling allows short jobs to move ahead and execute out of order as long as they do not delay the *first* queued job. Other jobs in the job queue may experience a delay with this scheme. Conservative backfilling, on the other hand, only lets short jobs to move ahead if they do not delay *any* queued job. Both variants make reservations for the queued jobs to avoid starvation (easy for the *first* and conservative for *all*). Easy backfilling is aggressive and greedy and attempts to maximize utilization at the current instant. Mu'alem et. al. (Mu'alem and Feitelson, 2001) compared and evaluated these two variants of backfilling

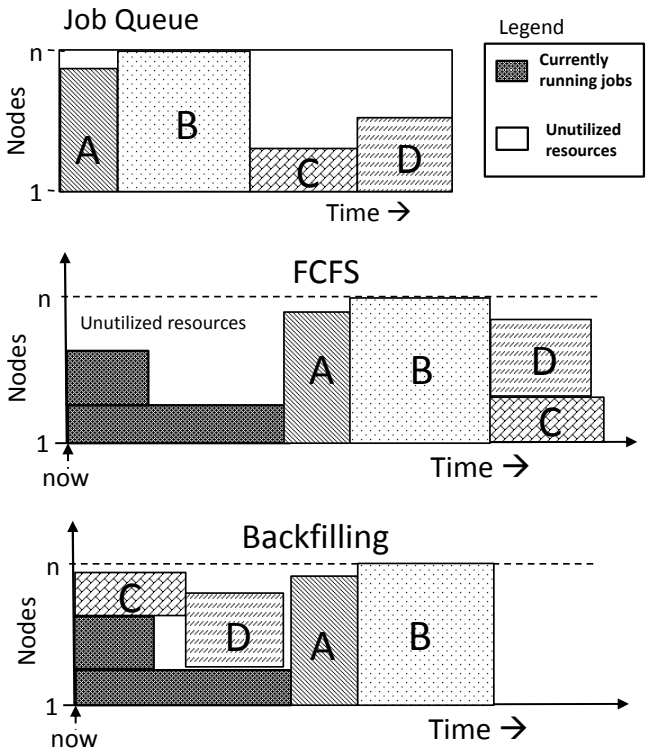


Figure 2.4: Comparison of FCFS and Backfilling

and concluded that they are similar in terms of performance, with easy backfilling doing slightly better than conservative backfilling for most workloads. Easy backfilling is thus preferred over conservative backfilling in practice. Researchers have also observed that for typical large HPC systems, enabling backfilling improves utilization by about 20% (Jackson et al., 2001).

Backfilling, though usually beneficial, does have some minor limitations. Because it runs jobs out of order, it may sometimes de-emphasize job priorities and user fairness in order to improve utilization. Also, in some scenarios, it introduces a *pseudo-delay* by adhering to a fixed reserved start time for a job and not letting the job run sooner as it normally would with FCFS. Pseudo-delay happens because users over-estimate job runtimes and provide poor inputs, and backfilling depends on these estimates for making reservations for the queued jobs.

2.5.2 Simple Linux Utility for Resource Management (SLURM)

Simple Linux Utility for Resource Management, or SLURM, is an open source job scheduler that was primarily designed by Lawrence Livermore National Laboratory and SchedMD in 2002 (Yoo et al., 2003). Its main purpose is to simplify accessing resources on HPC clusters and to provide a standard framework for launching, managing and monitoring jobs executing on parallel machines. SLURM is a sophisticated batch scheduling system with half a million lines of code that is scalable, fault-tolerant, and portable. It is currently deployed on many of the world's fastest supercomputers and can be configured to use a particular scheduling algorithm with the help of plugins. The SLURM codebase includes several scheduling algorithms, such as FCFS, Backfilling, Gang Scheduling, and Multifactor Priority Scheduling.

For users, SLURM provides a set of tools such as `salloc`, `srun`, `sacct`, and `scancel`. These tools allow users to create, submit and modify job requests. For example, `srun` allows users to specify resource requirements for their job, such as the number of nodes, number of cores per node, job steps, memory usage, etc. For administrators, SLURM supports utilities such as `scontrol` and `sacctmgr` and provides various authentication, configuration and build options.

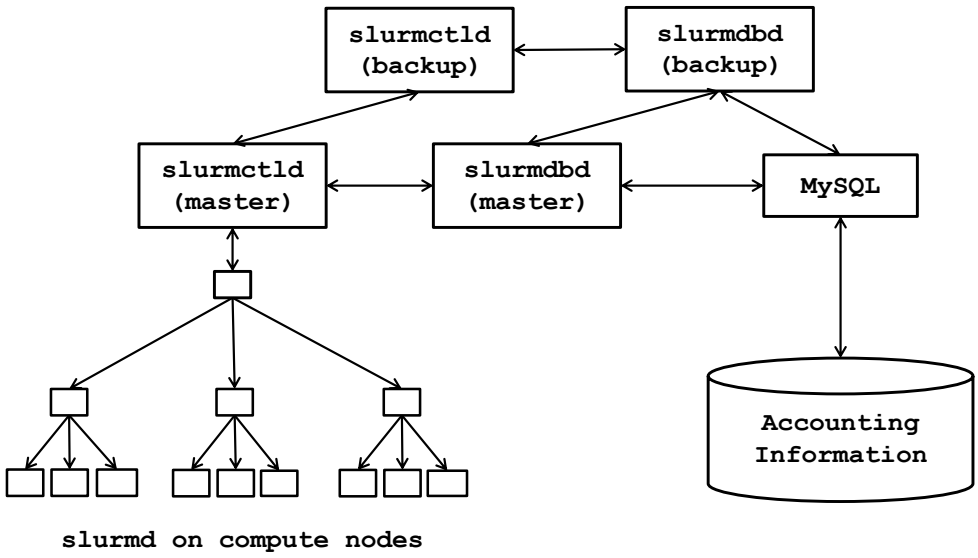


Figure 2.5: SLURM Architecture

Figure 2.5 depicts SLURM's architecture (Source: SchedMD). SLURM uses the `slurmctld` daemon ('controller') to monitor job queues and manage resource allocations. A single instance of this daemon typically runs on the head node of a cluster, along with the SLURM database daemon, `slurmdbd`. `Slurmdbd` is used for collecting accounting information (such as fair-share policies and priorities) with the help of a MySQL interface. Optional backup daemons for `slurmctld` and `slurmdbd` make the system fail-safe. Each compute node runs the `slurmd` daemon, which is primary used for launching tasks on a particular node. It communicates with the `slurmctld` daemon using RPC calls to indicate job status (for example: completed successfully, failed, killed by the user). `Slurmd` daemons support hierarchical communications with configurable fanout.

CHAPTER 3

HARDWARE OVERPROVISIONING

Several supercomputers today are targeted toward High Performance Linpack-like applications (Petitet et al., 2004) and designed to be *worst-case provisioned*—such that all nodes in the system can run at peak power simultaneously and HPC applications are assigned all the available power on a node. While this design strategy applies well for Linpack-like applications, several other scientific applications are unable to utilize the assigned peak power per node, resulting in limited performance under a power constraint. This chapter introduces *hardware overprovisioning* (or, overprovisioning for short) and explores its potential benefits when optimizing for application performance (execution time) under a power constraint. We analyze the performance of eight individual applications on a dedicated, 32-node system, and show that overprovisioning can improve performance (that is, execution time) under a power bound by up to 62% (2.63x) when compared to worst-case provisioning. The average improvement that we observe is 32% (1.47x). Section 3.1 introduces the motivation behind overprovisioning. Section 3.2 provides the details of the benchmarks and the system used in this study. Section 3.3 presents baseline results on a single node, and Section 3.4 presents our multi-node results at various system-level, global power bounds. Section 3.5 summarizes and concludes this chapter.

3.1 Motivation for Overprovisioning

Traditionally, HPC systems are designed in a manner that all components (nodes) can operate at peak power simultaneously. Multiple-node scientific applications executing on such systems are assigned peak power on each active node by default. Going forward, when a global power constraint is enforced, such a worst-case provisioned design will limit the total number of nodes that the HPC system can use. As a

result, HPC application performance will also be limited as the application will have to possibly execute on fewer nodes. Also, because HPC application power and performance profiles can vary significantly, some applications may have different per-node power requirements. Such applications might not be able to utilize the assigned peak power per node and will result in underutilized and wasted power.

An overprovisioned system is a system with more capacity (nodes), with the caveat that we cannot simultaneously power all components at peak power because of strict power constraints. Such a system will be more flexible with regards to the total number of nodes that it can use, and can be reconfigured based on the characteristics of different applications.

The motivation for overprovisioning can be drawn from the processor architecture community. Modern multicore processor architectures support dynamic overclocking features such as Intel's Turbo Boost (Intel, 2008) and AMD's Turbo CORE (AMD, 2015). In such processors, the CPU frequency at which a core executes depends on the number of active cores, and not all cores can simultaneously run at the highest CPU frequency. Such processors are dynamically reconfigurable and adapt to the currently executing workload.

The same idea can be extended to multiple nodes in HPC systems. Exascale systems will have a power budget; the current bound set by the US Department of Energy is 20 MW. Overprovisioning in the supercomputing context means that not all nodes in the facility can execute at peak power simultaneously.

While overprovisioning means that the supercomputing center buys more compute capacity than can be used, it allows the user to customize the system to an application and thereby to achieve better performance under a power constraint by scheduling power carefully among resources. These resources include the racks in the machine room, the nodes within the racks, the components within a node, and the interconnect. A fixed system-level power-constraint on the cluster will hierarchically translate to application-level and node-level power-constraints.

As opposed to worst-case provisioning, overprovisioning is advantageous because it can allow both highly scalable and less scalable applications to perform well. This

is because it supports reconfiguration and allocation of component power based on the application's characteristics. For example, an application's scalability determines whether we should use fewer nodes at higher power per node or more nodes at lower power per node under an application-level power constraint. In addition, depending on an application's CPU and memory usage, one can choose to use fewer cores per node or to allocate component power within a node (that is, power to the packages and the memory subsystem) based on utilization. Also, when only a few nodes in the application are on the critical path, running all nodes at peak power might be wasteful, and a different power allocation scheme might work better.

In order to explore overprovisioning and its effect on application performance, we solve the following question: given a machine with n nodes and c cores per node, a cluster-level power bound P on the machine, and a strongly-scaled HPC application, how important is choosing the *configuration* in minimizing execution time? We define a configuration as: (1) a value for n, (2) a value for c, (3) an amount of power p to be allocated to each node. The constraint is that the total power consumed must be no more than the bound P, and the goal is to minimize application runtime.

We explore this issue through a series of experiments on an Intel Sandy Bridge cluster at LLNL and use Intel's RAPL interface to enforce a range of power bounds on a diverse set of HPC applications and benchmarks. Thus, the cluster is effectively emulating an overprovisioned system.

3.2 Experimental Setup

We now discuss the experiments used in our analysis. In this work, we report power values in Watts and timing information in seconds. We conduct our experiments on the *rzmerl* cluster at LLNL, which is a 162-node Sandy Bridge cluster. Each node is a Intel Sandy Bridge 062D server model (Intel Xeon E5-2690) with two PKG domains, and 8 cores per package. The memory per node is 32GB. The clock speed is 2.6 GHz, and the maximum turbo frequency is 3.3 GHz. The socket TDP is 115 W, and the minimum recommended power cap is 51 W. The cluster has a 32-node per job

limit. We use MVAPICH2 version 1.7 and compile all codes with the Intel compiler version 12.1.5. OpenMP threads are scheduled using the scatter policy by setting the `KMP_AFFINITY` environment variable. While measurements are possible, power capping for PP0 and DRAM has been disabled by default by the system administrators. We run all experiments with power capping on the PKG domain with a single capping window and the shortest possible time window (0.000977 seconds). Thus, we avoid any potential large power spikes that might result when using larger time windows. We use four PKG power caps: 51 W; 65 W; 80 W; and 95 W. We run each configuration at least three times to mitigate noise. We disable Turbo Boost when power capping. We also run experiments with turbo enabled without using any RAPL-enforced power bounds. A cap of 115 W in our results represents this *turbo mode*. 115 W corresponds to the thermal limit on the PKG and, when we use turbo mode, this thermal limit becomes our power cap by default.

Our experiments use a hybrid MPI/OpenMP model. Hybrid models have low intra-node communication overhead and are more flexible in terms of configurations that we can test. Also, future architectures are likely to have many integrated cores on a single chip (for instance, Intel's MIC (Intel, 2012)). Because MPI processes have significant memory and communication overhead, the hybrid model is more likely than a flat MPI model. All applications are strongly scaled.

HPC Applications We use four HPC applications: SPhot (Lawrence Livermore National Laboratory, 2001) from the ASC Purple suite (Lawrence Livermore National Laboratory, 2002) and BT-MZ, SP-MZ, and LU-MZ from the NAS suite (Bailey, 2006). SPhot is a 2D photon transport code that uses a Monte Carlo approach to solve the Boltzmann transport equation by mimicking the behavior of photons as they move through different materials. SPhot is a CPU-bound, embarrassingly parallel application. For our multiple-node experiments, the input parameter `nruns` was set to 8192. For the single-node experiments, `nruns` was set to 1024. The NAS Multi-zone parallel benchmarks (NAS-MZ) are derived from Computational Fluid Dynamics (CFD) applications and are designed to evaluate the hybrid model. We use all three NAS-MZ

benchmarks: Block Tri-diagonal solver, or BT-MZ; Scalar Penta-diagonal solver, or SP-MZ; and Lower-Upper Gauss-Seidel solver, or LU-MZ. We use the Class C inputs.

Synthetic Benchmarks In order to cover the extreme cases in the application space, we also develop four MPI/OpenMP synthetic benchmarks. These tests are (1) CPU-bound and scalable (SC); (2) CPU-bound and not scalable (NSC); (3) Memory-bound and scalable (SM); (4) Memory-bound and not scalable (NSM). The CPU-bound benchmarks run a simple spin loop, and the Memory-bound benchmarks conduct a vector copy in reverse order. We control scalability by adding communication using `MPI_Alltoall()` (i.e., fewer calls to `MPI_Alltoall()` means better scalability).

3.3 Single-Node, Baseline Results

We first present details on the effect of power on our applications to understand the impact of power capping. We run our benchmarks on one node, varying the core count from 4 to 16. characteristics and their impact on configurations. Figure 3.1 shows the average PKG and DRAM power measured across the two sockets of the node for our benchmarks, at 4 and 16 cores and in turbo mode. We observe that some applications are more memory-intensive than others and hence consume more DRAM power. Examples of these are BT-MZ, SP-MZ, LU-MZ, and SM. At 16 cores per node, SP-MZ used 10.3% of its socket power for memory. Similarly, BT-MZ and LU-MZ use about 7-8% of their socket power for memory. SPhot, on the other hand, is relatively CPU intensive. We also observe that moving from 4 to 16 cores affects PKG power more than DRAM power. For instance, SPhot used 44.6 W of PKG power at 4 cores and 82.1 W of PKG power at 16 cores; DRAM power increased from 4.9 W to 5.7 W. Similarly, for BT-MZ, PKG power increased by 93% from 54.5 W to 105.7 W, but DRAM power only increased by 31%.

Figure 3.1 also shows that applications do not always use their allocated power. While the thermal limit on the PKG in turbo mode is 115 W, none of the applications actually use this much power, even when using all 16 cores. Most applications need between 80 and 90 W, except for BT-MZ, which uses nearly 106 W when running 16

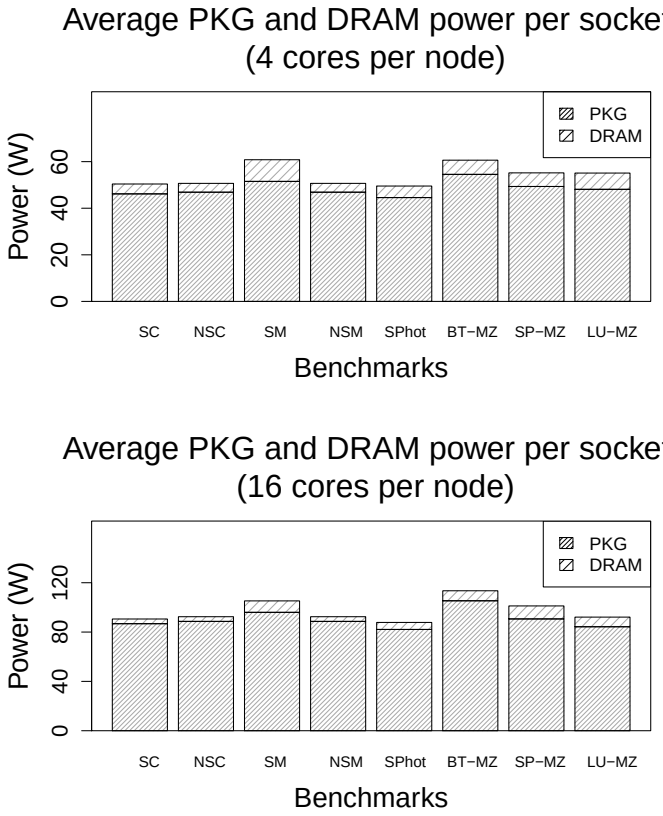


Figure 3.1: Average PKG and DRAM Power Consumption Per Socket

cores. This results in both wasted power and reduced application performance. This important observation calls for efficient node-level power allocations.

Figure 3.2 shows the impact of varying the PKG power bound on the node on application performance at 4 and 16 cores per node for SPhot and SP-MZ. At 4 cores per node, neither of the applications uses more than 51 W of PKG power; thus, in an ideal situation, when Turbo Boost is disabled, application performance should be unaffected as we decrease power from 100 W to 51 W when running 4 cores. However, we observe a slight slowdown in performance of about 1-2% for our applications from 51 W to 100 W with 4 cores per node. Since we run each experiment at least three times to mitigate noise, we believe that the overhead introduced by Intel's RAPL algorithm (which is different at different power bounds) may cause this slowdown.

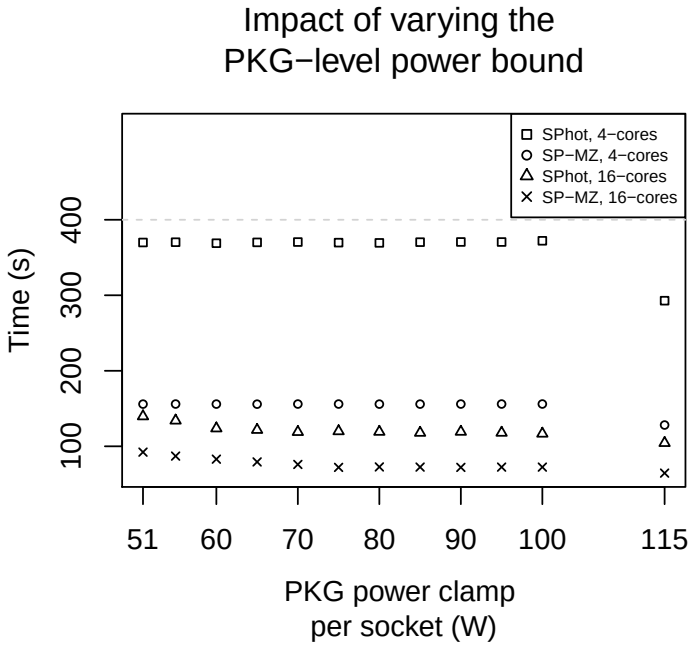


Figure 3.2: Impact of Varying the PKG Power Cap on Execution Time

At 16 cores per node, our applications benefit from more power up to 85 W. Performance improved by 21.3% for SP-MZ and by 17.1% for SPhot. In turbo mode (115 W), all applications perform significantly better as they run at a higher CPU frequency. Performance for SPhot improved more than SP-MZ, primarily because SPhot is CPU-bound and less memory intensive.

Next, we run single-node and multiple-node experiments with Turbo Boost to determine its effect on configurations. We collect samples every second with *librapl* and measure the frequency ratio by reading the `APERF` and `MPERF` MSRs (Intel, 2011). Figure 3.3 shows our results on a single-node with varying core count on our CPU-bound, scalable synthetic benchmark. We multiply the median frequency ratio value from the samples with the maximum non-turbo frequency (2.6 GHz) to determine the effective turbo frequency. Frequency ratios vary little across our samples.

Our results indicate that the effective turbo frequency depends on the number of active cores. Intel documentation (Intel, 2008) confirms this observation and also

mentions that the turbo frequency varies with temperature. We run our experiments at LLNL, where the machine room temperature is fairly constant over time and do not encounter any variation in the turbo frequency that could be attributed to temperature.

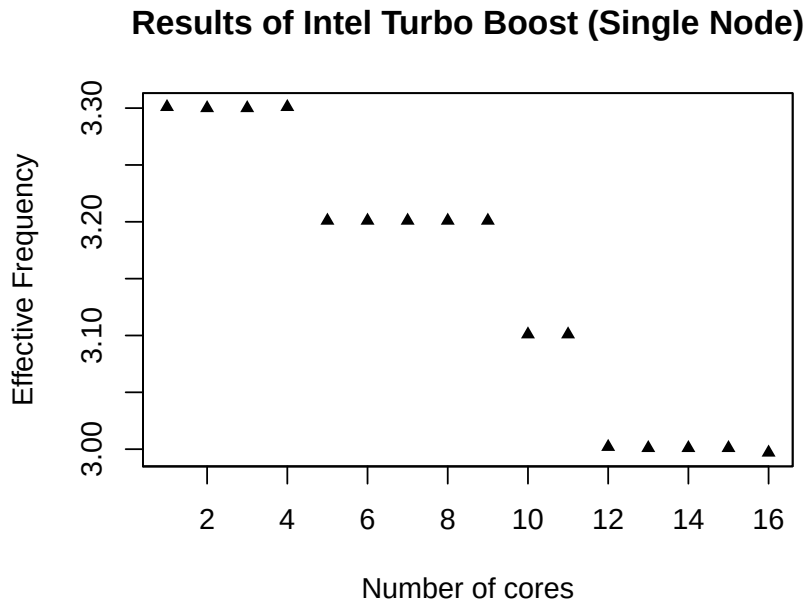


Figure 3.3: Turbo Boost Example on a Single Node: CPU-bound and Scalable Micro-benchmark

We also run multiple-node experiments with Turbo Boost enabled and collect frequency samples every second with *librapl*. In these experiments, we use the same number of cores per node. All nodes reach the same turbo frequency, which is stable throughout the application's execution. All nodes engage in turbo mode similarly.

3.4 Multi-node, Overprovisioning Results

This section presents and analyzes multiple-node results of the HPC applications and synthetic benchmarks. Because we could not feasibly run every possible configuration and because *rzmerl* has a 32-node limit per job, we run experiments with 8 to 32 nodes and 4 to 16 cores per node, in increments of 2. We assume uniform power allocation per node and that the applications are perfectly load-balanced.

3.4.1 Configurations

As discussed previously, we define a configuration as: (1) a value for number of nodes, n, (2) a value for number of cores per node, c, and (3) power allocated per node, p, in Watts. We denote a configuration as $(n \times c, p)$. Because power capping is unavailable on DRAM and PP0, we use p to represent the PKG power that is allocated to the node. We measure DRAM power along with PKG power, and our results report their sum on each socket across all nodes when we report total power.

For comparison purposes, we define four special configurations: *packed-max*, *packed-min*, *spread-max*, and *spread-min*. The term *packed* denotes that a configuration uses all cores on one node before using an additional node, while *spread* denotes that processes are spread as evenly as possible across all nodes, with 4 being the fewest cores per node used. When *max* is appended, we use the maximum power (that is, turbo mode) on each node, while *min* means we use the minimum rated power (that is, cap at 51 Watts). In all four special configurations, we continue to add cores and/or nodes until we reach the global power bound (or until we cannot add more nodes).

We run our benchmarks on multiple nodes under various overprovisioned scenarios. Our cluster of 32 nodes consumed about 6350 W of power when running all cores as fast as possible (based on BT-MZ data, which had the maximum power consumption). We investigate four overprovisioned scenarios; in each we have up to 32 nodes at our disposal, but power limits of 2500 W, 3000 W, 3500 W, and 4000 W. For comparison purposes, we also look at the case with unlimited power.

Figure 3.4 displays, at several global power bounds, the performance improvement for each of our benchmarks (in execution time) that overprovisioning can provide when compared to worst-case provisioning. This graph compares the best performing configuration under the respective power bound with *packed-max* (which is worst-case provisioning). If no bar exists then *packed-max* is optimal.

The average speedup of BT-MZ, LU-MZ, SP-MZ, and SPhot when using overprovisioning compared to worst-case provisioning is 73.8%, 55.6%, 67.2%, and 50.9% for a 2500 W, 3000 W, 3500 W, and 4000 W bound, respectively.

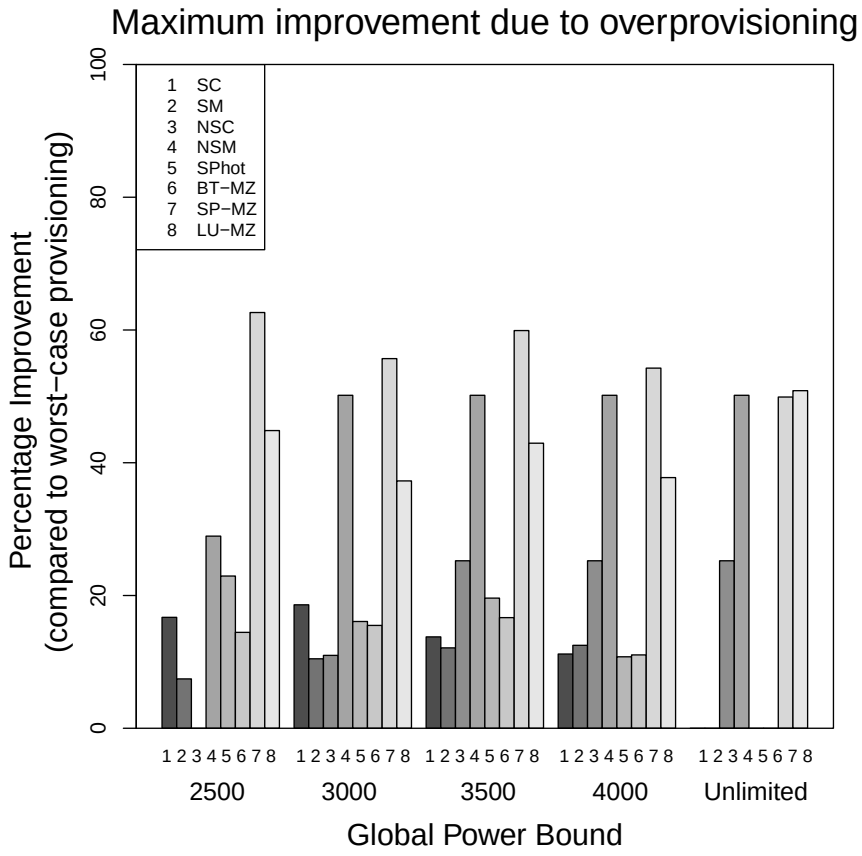
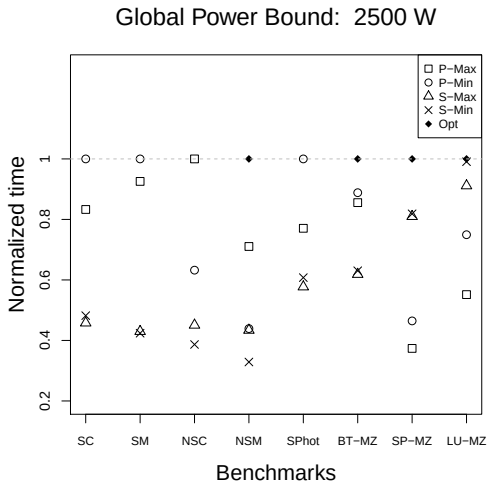


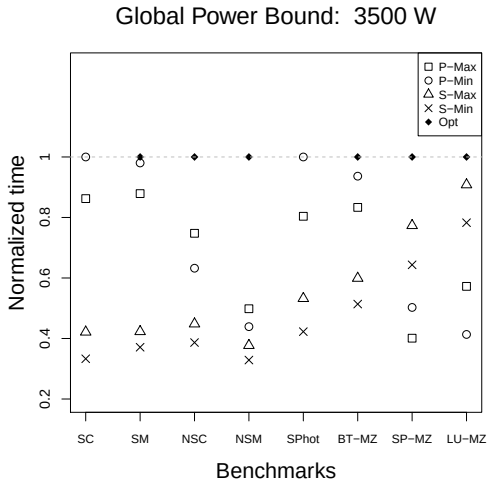
Figure 3.4: Performance Improvement due to Overprovisioning

Overprovisioning can lead to large performance improvements. For three cases, using a configuration other than *packed-max* is best even with unlimited power due to memory contention that degrades performance when using all cores.

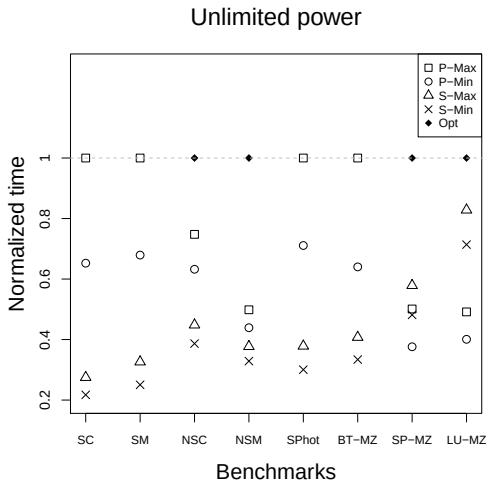
Overprovisioning essentially allows applications to utilize the machine as a reconfigurable architecture, which allows for better performance. For example, consider the global power bound of 3500 W. Here, SP-MZ executes 2.49 times faster when using the optimal configuration of $(26 \times 12, 80)$ than the *packed-max* configuration associated with worst-case provisioning, $(20 \times 16, 115)$. With worst-case provisioning, the facility would only have 20 nodes. In this work, we study the *potential* impovements that overprovisioning can provide. We leave the (orthogonal) question of *how* to determine the best configuration for an application under a power bound for future work.



Bmark	Configuration $(n \times c, p)$	Total Power (W)	Time (s)
SPhot	$(12 \times 16, 115)$	2181.29	74.27
	$(22 \times 16, 51)$	2388.51	57.24
	$(24 \times 4, 115)$	2459.74	99.18
	$(32 \times 4, 51)$	2472.33	94.19
	$(22 \times 16, 51)$	2388.51	57.24
SP-MZ	$(12 \times 16, 115)$	1974.66	13.88
	$(20 \times 16, 51)$	2179.92	11.16
	$(22 \times 4, 115)$	2449.71	6.40
	$(28 \times 4, 51)$	2337.83	6.34
	$(22 \times 8, 80)$	2452.81	5.19
LU-MZ	$(12 \times 16, 115)$	2227.95	25.29
	$(20 \times 16, 51)$	2350.36	18.61
	$(22 \times 4, 115)$	2409.92	15.31
	$(28 \times 4, 51)$	2383.34	14.08
	$(22 \times 8, 80)$	2412.42	13.95



Bmark	Configuration $(n \times c, p)$	Total Power (W)	Time (s)
SPhot	$(18 \times 16, 115)$	3263.74	49.52
	$(32 \times 16, 51)$	3477.62	39.81
	$(32 \times 4, 115)$	3252.82	74.79
	$(32 \times 4, 51)$	2472.33	94.19
	$(32 \times 16, 51)$	3477.62	39.81
SP-MZ	$(20 \times 16, 115)$	3240.22	9.10
	$(32 \times 16, 51)$	3494.16	7.26
	$(32 \times 4, 115)$	3431.63	4.72
	$(32 \times 4, 51)$	2647.40	5.67
	$(26 \times 12, 80)$	3497.16	3.65
LU-MZ	$(18 \times 16, 115)$	3308.11	16.46
	$(30 \times 16, 51)$	3379.74	22.78
	$(32 \times 4, 115)$	3492.11	10.37
	$(32 \times 4, 51)$	2730.48	12.03
	$(32 \times 8, 95)$	3497.34	9.42



Bmark	Configuration $(n \times c, p)$	Total Power (W)	Time (s)
SPhot	$(32 \times 16, 115)$	5768.41	28.30
	$(32 \times 16, 51)$	3477.62	39.81
	$(32 \times 4, 115)$	3252.82	74.79
	$(32 \times 4, 51)$	2472.33	94.19
	$(32 \times 16, 115)$	5768.41	28.30
SP-MZ	$(32 \times 16, 115)$	4978.38	5.45
	$(32 \times 16, 51)$	3494.16	7.26
	$(32 \times 4, 115)$	3431.63	4.72
	$(32 \times 4, 51)$	2647.40	5.67
	$(32 \times 14, 115)$	5590.13	2.73
LU-MZ	$(32 \times 16, 115)$	5487.03	17.48
	$(32 \times 16, 51)$	3608.65	21.43
	$(32 \times 4, 115)$	3492.11	10.37
	$(32 \times 4, 51)$	2730.48	12.03
	$(32 \times 8, 115)$	4458.88	8.59

Figure 3.5: Detailed Overprovisioning Results

Comparing Different Configurations

Figure 3.5 shows in-depth results for three of our power bounds (2500 W, 3500 W, and unlimited) for all applications. The y-axis represents performance of the four canonical configurations, normalized to the performance of the optimal configuration under the global power bound (higher is better). The x-axis lists our eight benchmarks. Each figure includes a table that provides the actual configurations. The figure also contains the total power consumed (packages and memory power) and the time taken.

We make three important observations for Figure 3.5. First, the figure shows some applications perform best using *packed* configurations compared to the *spread* configurations and that execution time can vary substantially between *packed* and *spread* configurations. This trend can be observed across all the benchmarks as well as power bounds. For example, at a global power bound of 2500 W, the *spread-min* configuration for SPhot runs 64.5% slower than *packed-min*. For SP-MZ at the same power bound, the *spread-max* configuration runs 2.16 times faster than the corresponding *packed-max* configuration. For SC, the *spread-min* runs more than twice as slow when compared to the *packed-min* configuration. In addition, for NSC, *spread-max* runs 2.36 times slower than *packed-max*.

The difference between the *packed-max* and *packed-min* configurations can also be significant. For example, with SPhot the *packed-max* configuration runs 29.8% slower than *packed-min* at 2500 W because the *packed-min* configuration utilizes more nodes and thus more total cores at lower power per node, as can be seen from the corresponding table. For SP-MZ and BT-MZ, the execution time difference between the *spread-max* and *spread-min* configurations is negligible. However, for LU-MZ this difference is 8% under a power bound of 2500 W and 16% under a power bound of 3500 W. When we vary the power bound, the execution time difference as well as the trend followed by the canonical and optimal configurations varies. At each global power bound, the configurations associated with the same canonical form can be different with an increase in the nodes and cores. For example, with SP-MZ *packed-max* is $(12{\times}16, 115)$ at 2500 W and $(20 \times 16, 115)$ at 3500 W.

Second, the best configuration is not always one of *packed-max*, *packed-min*, *spread-max*, or *spread-min*. For example, at 2500 W, BT-MZ, SP-MZ, LU-MZ and NSM have an optimal configuration that is different than the four canonical configurations. As the corresponding table shows, the optimal configuration for SP-MZ was $(22 \times 8, 80)$, which was 22% faster than the fastest canonical configuration, $(28 \times 4, 51)$. Running fewer nodes at higher power per node performs better than running more nodes at lower power per node in this case, as opposed to SPhot.

Third, the best configuration for an application depends on the particular global power bound. For instance, at a power bound of 2500 W, the best configuration for SP-MZ is $(22 \times 8, 80)$. On the other hand, when the power bound is 3500 W, it is $(26 \times 12, 80)$. It is important to note that the number of cores per node in the latter configuration are different than those in the former. While one expects to have a configuration with more nodes at a higher power bound, the observation that the number of cores per node can be different is not intuitive.

Application characteristics determine whether better execution times result from using more nodes, with fewer cores per node and at lower power; or fewer nodes, with more cores per node, and at a higher power. For instance, SPhot and BT-MZ always perform better when running more cores per node, and fewer nodes at higher power (i.e., *packed*), primarily because they are more CPU intensive than SP-MZ and LU-MZ. The *spread* configurations perform better for SP-MZ and LU-MZ because they are more memory intensive, and using more cores per node for such applications causes memory contention. For example, at 2500 W and 3500 W, the *spread* configurations are close to optimal for LU-MZ. Application scalability also plays an important role in determining the right configuration. SP-MZ, LU-MZ, NSC, and NSM have an optimal configuration that is neither *packed* nor *spread*, but somewhere in between because running as few as 4 cores per node and as many nodes as possible under the power bound increases communication.

To further illustrate how configurations can differ under a power bound, Figure 3.6 shows the entire configuration space for SP-MZ at 4500 W (Graph Courtesy: Dr. Rountree). This figure has 5 dimensions, three of which are input dimensions from

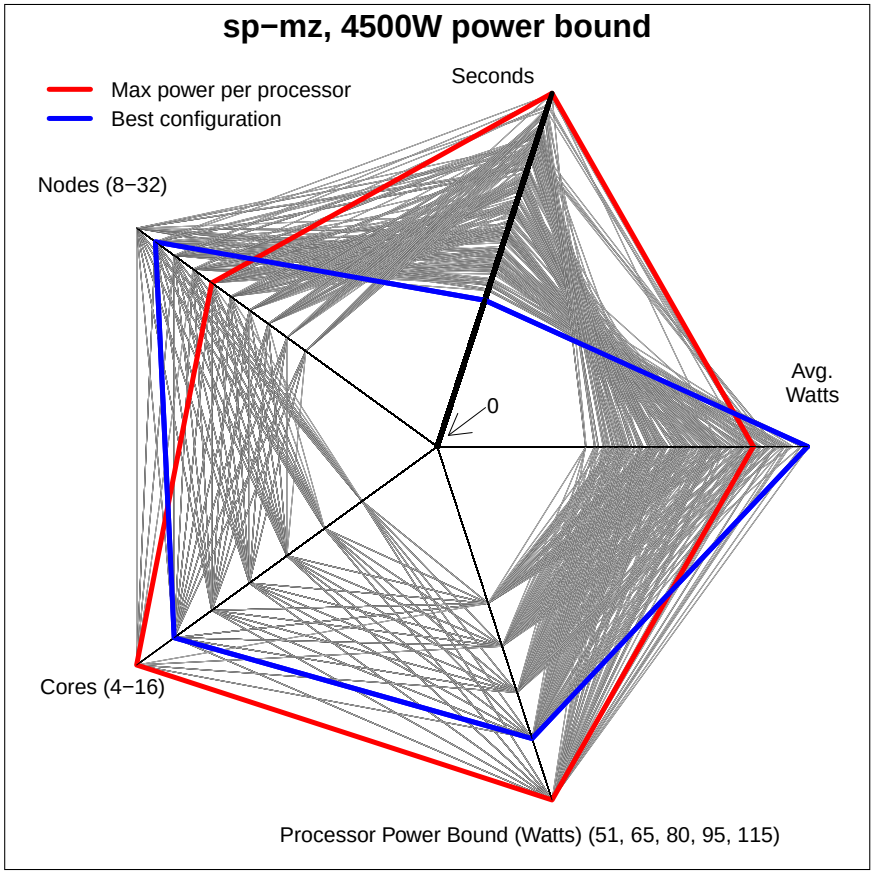


Figure 3.6: Example of SP-MZ at 4500 W

the application's configuration (nodes, cores, processor power cap), and two of which are output dimensions (performance in seconds, and application power consumption as average watts). The origin (zero) is at the center of this radar chart. To improve performance and to utilize power effectively, we want to "pull-in" toward the center for the performance dimension.

The *packed-max* and the best configuration under 4500 W for SP-MZ have been highlighted in red and blue, respectively. As can be clearly observed, the best configuration uses more nodes, less power per node, and fewer cores per node; and improves execution time by more than 2x. Additionally, it utilizes the allocated power better.

Using Fewer Cores

Cases exist in which using *fewer* total cores results in better execution time. This result is not unexpected, since similar results have been shown for worst-case provisioning (Curtis-Maury et al., 2008b). For example, as the table corresponding to unlimited power depicts (Figure 3.5), both SP-MZ and LU-MZ perform better with fewer total cores with the same PKG-level power bound. With SP-MZ, the *spread-max* configuration, $(32 \times 4, 115)$ is about 15% faster than the *packed-max*, $(32 \times 16, 115)$ (which runs four times as many cores). Two reasons lead to this behavior: first, in turbo mode, running fewer cores per node results in a higher effective turbo frequency; and, second, memory contention at 16 cores causes performance degradation. However, the optimal configuration is $(32 \times 14, 115)$, which uses more total cores than the *spread-max* configuration, but fewer cores per node than the *packed-max* configuration. The result is significantly better memory performance. The optimal configuration runs 72% faster than the fastest canonical one, which cannot be attributed to turbo mode because having 14 or 16 active cores per node results in the same effective turbo frequency.

In the case of LU-MZ, though, the effect of turbo mode as well as reduced memory contention can be seen more prominently. The optimal configuration, $(32 \times 8, 115)$ runs more than twice as fast as the *packed-max*, and 17% faster than the fastest canonical configuration, *spread-max*.

3.5 Summary

In this work we identified an emerging problem in the HPC arena: how to leverage overprovisioning in HPC installations. With power becoming a first order design constraint on our road to exascale, such overprovisioned systems will be commonplace, that is, we will no longer be able to fully power all nodes simultaneously. In such a scenario, we need to understand how we can configure our applications to best exploit the overall system while adhering to a global power bound. Our experiments show that the optimal configuration under a cluster power bound for a strongly-scaled HPC application performs much better than the configuration corresponding to the naive,

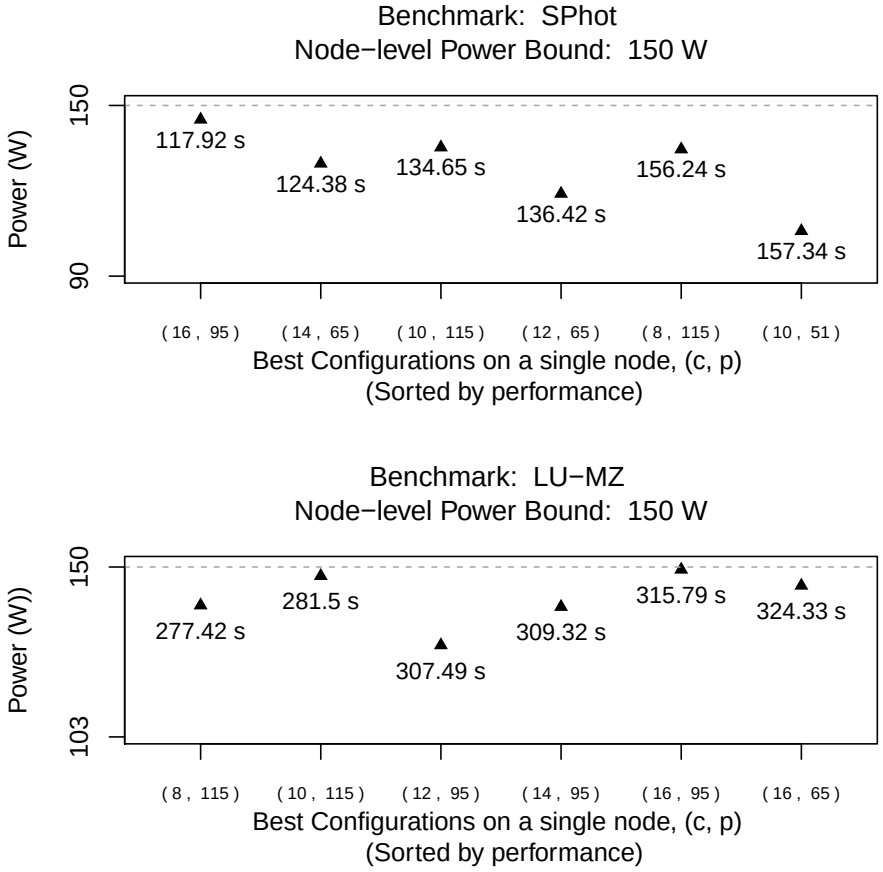


Figure 3.7: Configurations for SPhot and LU-MZ on a Single Node

worst-case provisioned scenario. We also presented results on package power capping and discussed its impact on application performance as well as on configurations. Overall, our experiments show that we can obtain substantial application speedup when carefully exploiting overprovisioned systems—over 32% (1.47x) on average.

Future work can proceed in multiple directions. One avenue is to investigate non-uniformity in power allocation. In this work we studied *uniform* overprovisioned supercomputing. That is, while we showed that using more nodes with reduced power often leads to better performance, each node has a uniform configuration.

In particular, assigning nonuniform power per node may improve performance, especially if the application exhibits load imbalance. However, a naive power assignment

to each node (that is, proportional increase in power based on amount of work) may produce poor results. For example, consider results on a single node in Figure 3.7. For SPhot, using more cores at lower power is profitable. On the other hand, for LU-MZ, running fewer cores at higher power is better.

Clearly, if we allocate power to nodes in a nonuniform manner, we should understand how power bounds impact performance through a node-level power-performance model. Previous related work has used nonuniform DVFS settings to save energy with negligible performance loss (Rountree et al., 2009).

A second avenue is to develop models to predict the optimal configuration given a system-level power-constraint and a strongly-scaled application. This involves multiple steps, including (1) developing a single-node model to predict the optimal number of cores and power allocation for the components within a node (packages and memory subsystem) and (2) a model to allocate inter-node power based on the critical path of the application and its load imbalance.

A third avenue is to study the impact of using dynamic overclocking techniques such as Turbo Boost. We also plan on experimenting with capping DRAM power. RAPL is one of the technologies that will eventually enable us to schedule power more intelligently on a power-constrained cluster.

CHAPTER 4

ECONOMIC VIABILITY OF OVERPROVISIONING

An overprovisioned system has more capacity than it can simultaneously fully power and can provide significant performance improvements in power-constrained scenarios. However, these benefits come with an additional hardware and infrastructure cost, and it is essential to conduct a thorough cost-benefit analysis before investing in large-scale overprovisioned systems. In this chapter, we develop a simple model to conduct this analysis, and we show that overprovisioned systems can be a net benefit. Section 4.1 discusses how supercomputers are typically procured and priced. Section 4.2 presents data on the performance benefits of adding more nodes and choosing the right configuration at that node count. In Sections 4.3 and 4.4, we develop a model to analyze the tradeoffs between performance and infrastructure cost of an overprovisioned system and establish that we can reap the benefits of overprovisioning without any additional financial investments. We conclude with a summary in Section 4.5.

4.1 Understanding Supercomputing Procurement

The typical process for procuring a supercomputing system involves releasing a public Request For Proposals (RFP) to which vendors can respond with a quote. The key part of an RFP is a technical proposal that requests details of both the hardware and associated system software. Additionally, information about testing and scaling the proposed system is requested. The hardware components include the nodes (processors, memory and accelerators), the interconnection network, and the I/O subsystem. A benchmark suite to rank performance of each of these components is provided by the purchaser, and vendors respond with performance results and price quotes. Additional details such as a business proposal and a price proposal are also requested from the vendors.

Table 4.1: Cost of a DOE HPC System (Source: Magellan Report)

Total Acquisition Cost	$52,000,000
Annual Acquisition Cost (divided over 4 years)	$13,000,000
Maintenance and Support	$5,200,000
Power and Cooling	$2,600,000
FTEs	$300,000
Total Annual Costs	$21,100,000
Cores	153,408
Utilization	85%
Annual Utilized Core Hours	1,142,275,968
Cost Per Utilized Core Hour (includes storage subsystem)	$0.018

An example of a recent RFP is for the CORAL systems to be hosted at Oak Ridge National Laboratory, Argonne National Laboratory, and Lawrence Livermore National Laboratory. More information can be found at `https://asc.llnl.gov/CORAL/`. Over the lifetime of a system, there are two key expenditures involved: the infrastructure investment cost, or the capital expenditure; and the maintenance and support cost, or the operational expenditure. Power costs are typically considered separately, because they can involve both capital as well as operational expenditures. Little public information is available on the breakdown of these costs, as the process is typically business sensitive. The Magellan Report provides some details on the cost of the Hopper supercomputer (Department of Energy, 2011). Other cost models have been studied as well, especially in the cloud computing and data centers community (Tak et al., 2011; Heikkurinen et al., 2015; Walker, 2009; Koomey et al., 2007). However, these models focus on determining the total cost of ownership. The Magellan Report is the closest to our research and we discuss data from this report below.

The goal of the analysis in the Magellan Report was to determine the cost per hour for a core on a DOE HPC system. The report discussed this cost for Hopper, a Cray XE-6 system at NERSC that delivers 1.28 petaflops. Hopper was placed at number 5 in the list of the world's fastest supercomputers in 2010; the contract for Hopper was valued at $52M. Table 4.1 shows the data from this report. The system had a

power budget of 3 MW, but used only 2.6 MW. The power cost assumed was 10 cents per KWhour, which has been reported to be a high estimate. The annual cost for maintenance and support is 10% of the contract cost, which amounts to $5.2M. The contract includes both hardware and software costs, which requires additional FTEs to be supported. The FTE cost was estimated based on DOE staff rates and amounts to $300K. Thus, the total cost of operation is $8.1M per year. The annual cost of acquisition amounts to $13M, assuming a 4-year lifetime. Finally, the cost per core hour was calculated by assuming an 85% utilization of cores. The Hopper system has 153,408 cores. Approximately 10% of the cost per core hour accounts for the storage subsystem. Overall, the determined cost per utilized core hour for the Hopper system has been calculated as $0.018, as shown in Table 4.1.

This data gives us an estimate of how the acquisition, power and maintenance costs come together in a real system. When designing an overprovisioned system, the goal is to improve performance *without* increasing the cost of computation. The key here is in the acquisition step, where the choice is usually between expensive, efficient high-end nodes and older-generation, slightly inefficient nodes. We discuss this in Section 4.3.

4.2 Degree of Overprovisioning

One of the key things to analyze when overprovisioning is the degree to which we should overprovision. It is important to determine this degree because of the law of diminishing returns—adding more capacity (nodes) beyond a certain limit while keeping the same total system power bound will not result in proportional performance benefits. We define the degree of overprovisioning to be the ratio between the number of nodes in the overprovisioned system to the number of nodes in *absolute* worst-case provisioning (discussed below) that leads to the maximum performance improvement under a specified power bound.

The optimal degree of overprovisioning is difficult to determine because it depends on the global power bound as well as workload characteristics. Based on our emperical measurements and the data presented in Chapter 3, typically adding about 30-50%

more nodes than worst-case provisioning and choosing the correct configuration resulted in performance improvements across all eight applications that we considered.

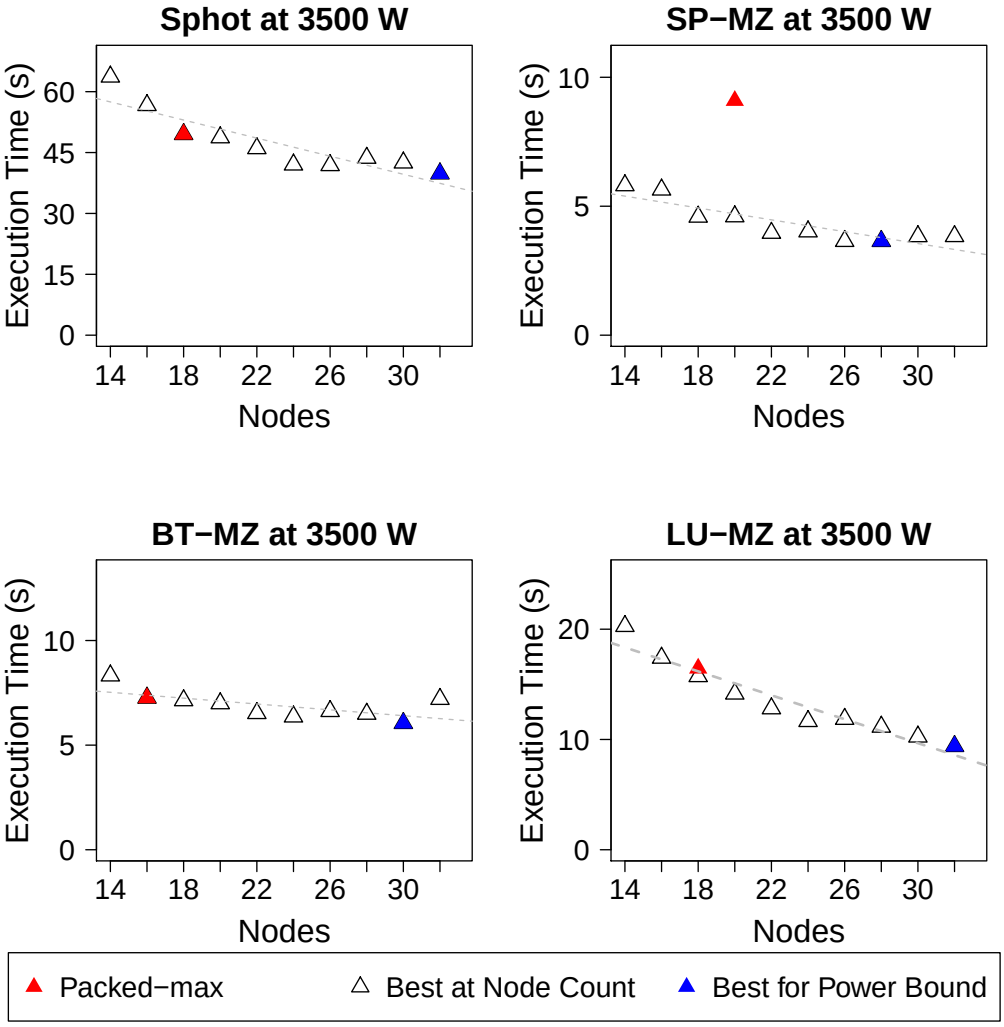


Figure 4.1: Benefits of Adding More Nodes, 3500 W

In order to understand how the degree of overprovisioning affects performance, we analyze the performance of configurations at two fixed global power bounds when the total number of nodes available in the HPC system is varied. Recall that a configuration consists of three values—number of nodes, number of cores per node, and power per node. Figures 4.1 and 4.2 show data for the four HPC applications at 3500 W and

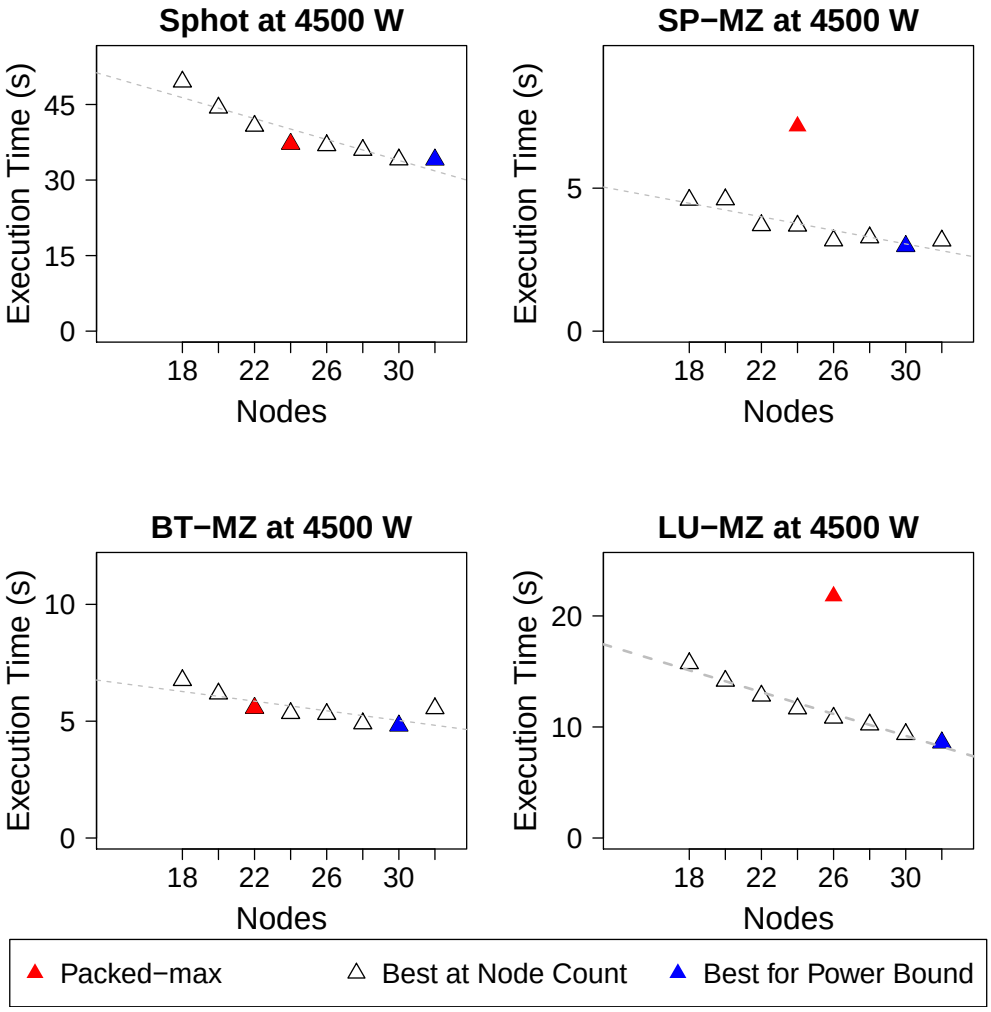


Figure 4.2: Benefits of Adding More Nodes, 4500 W

4500 W. This dataset is from Chapter 3 and has been collected on node counts ranging from 4 to 32, cores per node ranging from 4 to 16, and 5 package power bounds ranging from 51 W to 115 W. The y-axis in each subgraph shows the raw execution time of the application, and the x-axis represents a node count. For each node count, the best *valid* configuration (one that does not exceed the specified power bound) at that particular node count has been plotted (black empty triangle). The best performing configuration under the power bound has been marked with a blue triangle. Additionally, the *application-specific worst-case* configuration (all cores on a node, with no power caps and Turbo Boost enabled at 115 W) has been plotted (marked with a red triangle).

The actual power consumed by the application was used to determine this worst-case configuration. As a result, this is the *packed-max* configuration under the specified power bound, which allocates as many nodes as possible, with each node using all cores and peak power.

Depending on the application, the benefits of adding more nodes under the same power bound vary. For example, the benefits for an application such as BT-MZ are limited. On the other hand, adding more nodes is beneficial for applications such as SPhot and LU-MZ. For some applications, choosing the right configuration might be more important than adding more resources (nodes). For example, consider SP-MZ at 3500 W. The *packed-max* configuration performs much worse than the best configuration at the same node count. A similar trend can be observed with LU-MZ at 4500 W.

In Chapter 3, we reported the benefits of overprovisioning as conservatively as possible, which is why we chose to look at application-specific, *packed-max* configurations instead of the *absolute worst-case* scenarios. For procurement, however, we have to follow the *absolute* worst-case configuration instead of the application-specific, *packed-max* configuration, because we are analyzing system design under a strict power constraint. The absolute worst-case configuration can be obtained by deriving the maximum possible node power from the TDP values of the node components and is not application-specific. In our example, the absolute worst-case *node count* is 12 nodes for 3500 W and 16 nodes for 4500 W. This has been derived by calculating the maximum possible power on each node, assuming that each socket takes 115 W and each memory unit takes 30 W of maximum power. It is important to note that we still have the option of picking the right number of cores and the correct power cap for the application at this fixed, absolute worst-case node count.

As discussed in Chapter 1, traditional systems are designed to accommodate the absolute worst-case, which leads to limited performance as well as wasted power. As an example, we list the performance and power details associated with the absolute worst-case, the application-specific *packed-max*, and the best overprovisioned configurations for a global power bound of 3500 W in Table 4.2.

Table 4.2: Power and Performance Data, 3500 W

App.	**Absolute Worst-case** $(n \times c, p) : \{secs, W\}$	**App-specific Packed-Max** $(n \times c, p) : \{secs, W\}$	**Best Configuration** $(n \times c, p) : \{secs, W\}$
SPhot	$(12 \times 16, 115) : \{74.3, 2183\}$	$(18 \times 16, 115) : \{49.5, 3264\}$	$(32 \times 16, 51) : \{39.8, 3478\}$
SP-MZ	$(12 \times 16, 115) : \{13.9, 1975\}$	$(20 \times 16, 115) : \{9.1, 3240\}$	$(26 \times 12, 80) : \{3.6, 3497\}$
BT-MZ	$(12 \times 16, 115) : \{9.9, 2398\}$	$(16 \times 16, 115) : \{7.3, 3232\}$	$(30 \times 14, 51) : \{6.1, 3418\}$
LU-MZ	$(12 \times 16, 115) : \{25.3, 2228\}$	$(18 \times 16, 115) : \{16.5, 3308\}$	$(32 \times 8, 95) : \{9.4, 3497\}$

This data is in line with our measurements shown on the Vulcan system in Chapter 1 (Figure 1.1). In the absolute worst-case, where we can only power up 12 nodes based on the TDP measurements, 37% of procured power is wasted on average. Instead, if we buy more capacity, we see significant benefits in performance, even when we choose the application-specific, *packed-max* configuration. Being able to overprovision to a larger degree and reconfigure further for performance based on application characteristics can result in a maximum speedup of 3.0x (average speedup of 2.8x) when compared to the absolute worst-case provisioning scenario. Also, less than 1% of procured power (3500 W) is wasted. Achieving these benefits of overprovisioning without adding any infrastructure cost is possible, and we discuss this in Sections 4.3 and 4.4.

4.3 Projecting the Cost of Overprovisioning

In this section, we develop a model that system designers can use to determine whether or not overprovisioning will be of benefit. The model takes into account four main aspects: node price, node performance, system power budget, and maximum node power. We first provide the intuition and motivation for this model in this section, and then discuss its formulation in Section 4.4.

An overprovisioned system can lead to better utilization of power and performance benefits. However, buying more capacity comes at additional cost when compared to a worst-case provisioned system. Overprovisioning might not be attractive for adoption if the acquisition cost increases. It is possible for the hardware cost of an overprovisioned system to remain fixed. Processors that are a generation older may have fairly similar features and are typically offered at a significantly lower unit price. A hardware cost

budget thus provides a choice between more inefficient, older-generation processors leading to better job throughput, utilization and performance under a power constraint, or fewer, high-end, efficient nodes on a non-overprovisioned cluster (which likely leads to wasted power). It is important to note that the individual older-generation processors can be less power efficient than the high-end processors. However, reconfiguring multiple such processors in an HPC system under a global power constraint can lead to better overall performance and system power efficiency.

As an example of individual processor features, let us compare the Intel Xeon E5 2697 v2 (Ivy Bridge, 22nm) processor with the Intel Xeon E5-2670 (Sandy Bridge, 32nm) processor. The former is a high-end, dodeca-core processor operating at 2.7 GHz priced at about $3300 (as of June 15, 2015 on Amazon, Inc.). The latter is an older-generation octa-core processor operating at 2.6 GHz, priced at about $1700 (as of June 15, 2015 on Amazon, Inc.). These list prices include DDR3 memory and are reported in USD. Table 4.3 shows the details of these two server processors.

PassMark Software Private Ltd. is a performance analysis and consulting company that maintains the world's largest CPU benchmarking website, `cpubenchmark.net`. The website provides users access to benchmarking results for over 600,000 systems spanning more that 1200 different types of CPUs and is considered to be the leading authority in CPU benchmarking. The CPU benchmarking suite from PassMark software consists of eight computational workloads that can be used to rank multicore CPUs—Integer Math Test, Floating Point Math Test, Prime Number Test, Compression Test, Encryption Test, Physics Test, String Sorting Test and a Multimedia Instructions Test. There are also single-core tests. A Passmark score is obtained by running these tests on all the cores and then taking an average. The higher the PassMark score, the better. Details about PassMark benchmarking tests can be found at `http://www.cpubenchmark.net/cpu_test_info.html`. The PassMark scores for the two server processors under consideration were obtained from the popular CPUBoss website, which is a website created by a CPU (as well as GPU and SSD) performance research team in Canada (`http://cpuboss.com/`).

Table 4.3: Comparing 32nm and 22nm processor technologies

Features	**Intel Xeon E5 2697 v2**	**Intel Xeon E5-2690**
Manufacturing Technology	Ivy Bridge 22nm	Sandy Bridge 32nm
Cores Per Processor [Threads]	12 [24]	8 [16]
Processors Per Node	2	2
Clock Speed [Turbo]	2.7 GHz [3.5 GHz]	2.6 GHz [3.3 GHz]
L2 Cache	3 GB	2 GB
L3 Cache	30 GB	20 GB
Maximum Memory Support	786,432 MB	393,216 MB
PassMark Performance Test Result	17,812	13,985
TDP	130 W	115 W
List Price (USD)	$3300	$1700

Based on the PassMark score, it is expected that a system designed with the former will be about 27% faster (based on the PassMark values of 17,812 and 13,985) than one designed with the latter on most computational workloads. For applications that are highly-scalable and do not utilize peak node power, however, the former will lead to significant wasted power and limited performance due to the inability to scale out (based on the power efficiency trends on the CPUBoss website). For a given node cost budget, it is possible to buy 94% more nodes of the latter Sandy Bridge processor and design a hardware overprovisioned machine that results in significantly more performance improvements, primarily because it utilizes power better and leverages application characteristics. Based on this intuitive analysis, we propose a model that system designers can utilize to determine whether or not overprovisioning will result in net benefit in their case.

4.4 Cost Model Formulation and Analysis

The main goal of our model is to determine whether or not it is possible to use overprovisioning to improve performance without added infrastructure cost. As shown in Section 4.3, it is possible to buy (significantly) more, older-generation nodes with similar performance characteristics for a given node cost budget, even when considering

the costs of other hardware such as the interconnect. While compute servers contribute to a significant portion of the total hardware acquisition cost of a system, it is important to consider the cost of other components such as the interconnect and the I/O subsystem. For example, a worst-case provisioned system might need fewer total racks than an overprovisioned system and will thus have a lower interconnect cost. Public information on the breakup of the hardware costs for the nodes, interconnect and the I/O subsystem is unavailable, so we translate the other costs to the node level in order to correctly estimate their influence.

Similarly, it is important to analyze the performance difference between the high-end and the older-generation nodes. In the example in Section 4.3, we observed that the high-end node is expected to be 27% faster for the single-node case. However, performance at a different node count greatly depends on the application's scalability and memory intensity characteristics. In order to accommodate this, we create a model to predict performance (execution time) on the two architectures based on the four HPC applications that we considered in this dissertation (SPhot, SP-MZ, BT-MZ, and LU-MZ). Another important factor that contributes to the cost of overprovisioning is the system wide power budget and the maximum node power of the high-end node. This is important because it helps constrain the computational cost of the overprovisioned system that is being designed to that of the worst-case provisioned system. We use these main contributing factors as input parameters for our model, and explain these in detail in the following subsections (Sections 4.4.1-4.4.3). Table 4.4 summarizes these input parameters. We design a worst-case provisioned system with a high-end node and an overprovisioned system with an older-generation node in our model, and do so under the same cost and system power constraint. We then predict performance on these two systems in order to conduct a cost-benefit analysis. The details of the model are presented after the explanation of the input parameters, in Section 4.4.4.

4.4.1 Input Parameters: Power Budget and Node Power Values

The most critical input parameter when designing a worst-case provisioned or an overprovisioned system is the global power budget to which the system must adhere.

Table 4.4: Model Input Parameters

Parameter	**ID**	**Description**
Power Bound	P_{sys}	Power Budget allocated to the *computational* components of the system
Maximum Node Power	P_{n_max}	Maximum possible node power for the *high-end node* based on its overall TDP.
Minimum Node Power	P_{n_min}	Minimum possible node power for the *older-generation node* based on its idle power.
Effective cost ratio	r_c	Ratio of the *effective* per-node cost of the high-end node to that of the older-generation node
Performance	r_p	Percentage by which the high-end node is faster than the older-generation node
Workload Scalability Model	$\{m, c\}$	The slope and intercept of the linear performance model based on workload scalability characteristics (obtained on the older-generation node)

For the scope of our model, we are limiting this to the total power associated with computation, the source of which is compute nodes and other components on the rack, such as the interconnect. This is because we are studying the impact of dynamic power on application performance, as discussed in Chapters 2 and 3. The total site-wide power budget, of course, will include power from other static components, such as the power of the I/O subsystem. Going forward, when we talk about a power budget, we assume it is the dynamic power budget for the compute nodes, and we refer to this as P_{sys}.

In order to ensure that the system power budget is always met, we need to account for the maximum power on each node for a worst-case provisioned system. Similarly, for an overprovisioned system, we need to account for the minimum power required for each node (idle power). We thus need two input parameters, P_{n_max} and P_{n_min}, which correspond to the maximum node power for the high-end, worst-case system node, and the minimum node power for the older-generation, overprovisioned system node. The node power is determined based on the TDP of the processor and memory, as well as the associated interconnect power. These values help us determine the maximum number of nodes we can power up successfully without exceeding the system power budget for both the worst-case provisioned and the overprovisioned scenarios.

4.4.2 Input Parameter: Effective Cost Ratio

In order to understand the cost implications of the processors under consideration, we define the effective cost ratio, which we then use as input parameter. To help understand this parameter, we begin with an example and design a worst-case provisioned system rack with the high-end, Ivy Bridge node described in Section 4.3. Let us assume that each rack in the machine room can fit 16 server nodes that are connected via one state-of-the-art InfiniBand FDR 36-port switch. The typical cost for the aforementioned InfiniBand switch is about $10,000 (as of June 15, 2015 on Amazon Inc.). Other necessary components for a rack, such as the power supplies, amount to another $2,000. Thus, the cost of the base components of an empty rack (that is, excluding the actual compute servers) is about $12,000. As a result, the total cost for a single rack with the high-end nodes is $16 \times \$3,300 + \$12,000$, which equals $64,800. This translates to an effective per-node cost of $\$64,800/16$, which amounts to $4,050. Let us assume that our total cost budget for this example is $64,800.

We now analyze the design of an overprovisioned system by using the cheaper, older-generation nodes. If we only accounted for the node costs in isolation, we will be able to add 94% more nodes of the older-generation, Sandy Bridge processor (that is, $\lfloor 16 + 0.94 * 16 \rfloor$, for a total of 31 nodes), as discussed in Section 4.3. However, adding even one more node will mean that we need a new rack[1]. With each new rack, we need to add in the base cost for an empty rack, which is $12,000. The cost of a single *full* rack for the overprovisioned system is thus, $16 \times \$1,700 + \$12,000$, which equals $39,200, which translates to a per-node cost of $2,450. Thus, with a cost budget of $64,800, we can only add $\frac{\$64,800-(\$39,200+\$12,000)}{\$1,700}$ more nodes, which amounts to 8 nodes.

The second rack in this example is not fully utilized because it has a capacity of 16 nodes and we can only add 8 nodes to it. If we consider this in our design, the per-node cost for such a rack will be higher. In this example, the per-node cost for the second rack is $3,200, because it has fewer nodes ($\frac{\$12,000+8\times\$1,700}{8}$). It is important to note that the fewer nodes scenario will only apply to the last rack in the system, and depending

[1]This applies for a simple single-switch level HPC system. There might be scenarios where multiple levels of network switches are used an the HPC system.

on the total number of racks in the HPC system, it may not may not be important to consider the *average* per-node cost across all racks in the system. In this particular example, if we only consider two racks, this average is $2,800.

Because of this, in our model, we use *effective cost ratio* as an input parameter to represent the costs of the two types of nodes under consideration. The effective cost ratio, r_c, is defined as the ratio of the price of the high-end node to that of the cheaper, older-generation node. The effective cost ratio is always greater than 1, because the high-end nodes are more expensive than the older-generation nodes. In this example, the effective cost ratio will be $\$4,050/\$2,800$, which equals 1.45. This will mean that we can add $\lfloor 0.45 \times 16 \rfloor$, or 7 more nodes to the overprovisioned system. It is important to note that this is always a conservative estimate of the degree of overprovisioning because we are accommodating for potentially unutilized racks. For a real HPC system with several hundred racks, the effective ratio is higher and closer to the ratio of the effective per-node costs of the high-end and the older-generation node, which in this scenario is $\$4,050/\$2,400$ and amounts to 1.7.

4.4.3 Input Parameters: Performance Parameter and Workload Scalability Model

The difference in single-node performance of the high-end node to that of the older-generation node is captured as a percentage value, r_p. The parameter r_p represents the percentage amount by which the high-end node is faster than the older-generation node. This parameter can be obtained by using a standardized benchmarking tests such as with PassMark benchmarks or SPEC CPU benchmarks. A typical RFP response from the vendor may also be used for obtaining this ratio.

In order to predict performance of applications at scale, we also need a scalability model for the workloads under consideration. In our case, we obtain this scalability model on the system with older-generation nodes, and project the performance using the r_p parameter to the system with high-end nodes. We assume that applications will scale similarly, and this is a fair assumption to make when the nodes under consideration are separated by one or two generations (such as the Ivy Bridge and the Sandy Bridge nodes from Section 4.3). This is because processors separated by one or two generations

have similar features, such as clock rates, vector units, and cache and memory support. As we discussed in Section 4.1, an RFP includes a few real scientific code kernels on which the vendors need to report performance. Generally, these are publicly available benchmarking suites, such as the Mantevo suite, which lists a few *miniapps* for vendors (`https://mantevo.org/`). The scalability model can be obtained by studying the performance of such miniapps at different configurations (that is, nodes, cores per node and power bounds).

Application-specific as well as general scalability models can be developed (Barnes et al., 2008; Gahvari et al., 2011; Gahvari, 2014). HPC systems may have different purposes, for example, some systems are mission-critical, while others are developed for academic research. The workload scalability model should accurately represent the application characteristics on the older-generation node that are being considered for the overprovisioned system. Developing application-specific models and then extracting characteristics to generalize them is an orthogonal problem to our work.

We assume a linear workload scalability model in this work and require two inputs, the slope and intercept (m and b). These inputs are determined by gathering data for the application at several node counts, considering the best configuration at each node count for the application and fitting a line through these best configurations. This is shown in Figures 4.1 and 4.2.

For the scope of this chapter, we focus on analyzing data on the four strongly-scaled HPC applications. For each application, we first set a range of nodes for valid strong-scaling operation based on experimentation. Then, given a system power bound, we choose the best performing configuration at each valid node count and develop a linear scalability model.

In our data, the coefficients of the linear model (slope and intercept) were nearly constant across all the global system power bounds under consideration for a particular application (less than 2% difference across 6 different system power bounds ranging from 2500W to 6500 W). Thus, we assume that for a given application, the input parameters m and b are identical across different power bounds. Our average prediction error across all applications is under 7%. Figures 4.1 and 4.2 show the fit for the four applications at

two different system power bounds. We do not develop a general scalability model that captures the characteristics of all four applications in this work, as this is an orthogonal problem beyond the scope of this dissertation.

4.4.4 Model Formulation

We now describe the formulation of our model. The input parameters are listed in Table 4.4. A summary of variables used for the formulation and the output parameters is provided in Tables 4.5 and 4.6. The goal of the model is to determine whether or not the overprovisioned system designed with the same cost budget as a worst-case provisioned system will result in performance benefits under a system-level power constraint. In order to accomplish this, we design two systems in the model, one with worst-case provisioning using high-end nodes and another with overprovisioning using older-generation nodes. In both cases we assume the same cost budget, and we ensure that the system-wide power budget is met. We then predict application performance on both these systems.

Table 4.5: IDs used in Model Formulation

Parameter	**ID**	**Description**
Worst-case Nodes	n_{wc}	Maximum number of nodes for worst-case provisioning based on the system power budget
Node Limit	n_{lim}	Maximum number of nodes for overprovisioning based on the system power budget
Overprovisioned Nodes	n_{ovp}	Actual number of nodes used for overprovisioning based on the computational cost constraint as well as the system power budget
Worst-case Cost Constraint	c_{wc}	Computational cost constraint derived from the worst-case provisioning system. The overprovisioned system has the same cost constraint.

A key constraint in the model is to ensure that the specified computational power budget, P_{sys}, is honored. When designing the worst-case system, P_{sys} constrains the maximum number of high-end nodes we can run at full power. Similarly, for the overprovisioned system, this constrains the maximum number of older-generation

Table 4.6: Model Output Parameters

Parameter	**ID**	**Description**
Worst-case Time	t_{wc}	Predicted application execution time for the high-end, worst-case provisioned system
Overprovisioned Time	t_{ovp}	Predicted application execution time for the overprovisioned system with same cost constraint
Performance Ratio	s_{ovp}	Ratio of the performance of the high-end, worst-case provisioned system to that of the overprovisioned system. That is, t_{wc}/t_{ovp}

nodes that we can run at idle power and effectively controls the maximum degree of overprovisioning. The input parameters associated with this step were discussed in Section 4.4.1.

The worst-case provisioned node count is determined by the maximum power on the high-end node (P_{n_max}), and the node limit for the overprovisioned system is determined by the minimum power on the cheaper node (P_{n_min}). Both the node-level power values are calculated by considering the power consumption of the processor, memory and associated interconnect. We thus derive the maximum number of nodes that can meet the computational power budget for both the systems as shown in Equations 4.1-4.2.

$$n_{wc} = \frac{P_{sys}}{P_{n\_max}} \tag{4.1}$$

$$n_{lim} = \frac{P_{sys}}{P_{n\_min}} \tag{4.2}$$

We now derive a fixed computational cost constraint for both systems. We use effective cost ratio, r_c, and the number of nodes obtained from the worst-case provisioned system from Equation 4.1 to determine the cost constraint.

Using the effective cost ratio allows us to assume that the base cost for the older-generation node is 1. We can thus look at the impact of the cost difference between the high-end and the older-generation nodes more clearly. The effective cost ratio was described in detail in Section 4.4.2. The dollar amount associated with the cost constraint can be determined by the product of n_{wc} and the unit cost of the high-end

node, which denotes the cost of the worst-case provisioned system. This step is shown in Equation 4.3.

$$c_{wc} = n_{wc} \times r_c \tag{4.3}$$

We determine the maximum degree of overprovisioning using both the power and cost constraints. Equation 4.4 represents this. The unit cost for the older-generation node is assumed to be 1, so the number of nodes that we can buy within the cost constraint is $\lfloor c_{wc} \rfloor$. If this exceeds the number of nodes that are supported by the power budget (n_{lim}), we will end up buying a cheaper overprovisioned machine and will have some part of the cost budget left over.

$$n_{ovp} = min(n_{lim}, \lfloor c_{wc} \rfloor) \tag{4.4}$$

Finally, we predict performance on both the systems. We use the performance parameter, r_p, and the parameters $\{m, c\}$ (slope, intercept) from the workload-specific scalability model to accomplish this. These parameters were described in detail in Section 4.4.3. If the execution time taken on a single older-generation node is 1, and the high-end node is faster by r_p percent, the execution time for the high-end node is $1 - r_p$.

We then determine the ratio s_{ovp}, which is representative of speedup due to overprovisioning and is defined as the ratio of the workload's execution time on the worst-case provisioned system to that of the overprovisioned system. The node range specified in the workload model is used here to determine the linear region of operation for the model, and slope and intercept values are used to determine performance. Equations 4.6-4.7 depict these steps. For overprovisioning to have a net benefit, s_{ovp} should be greater than 1.

$$t_{ovp} = m \times n_{ovp} + c \tag{4.5}$$

$$t_{wc} = (m \times n_{wc} + c) \times (1 - r_p) \tag{4.6}$$

$$s_{ovp} = \frac{t_{wc}}{t_{ovp}} \tag{4.7}$$

4.4.5 Analysis

In Section 4.4.4, we presented our model and derived a ratio (s_{ovp}) to compare the performance of the worst-case provisioned and overprovisioned systems that we designed with the same cost and power budgets. In the subsections that follow, we use this ratio to determine situations where overprovisioning has a net benefit (when $s_{ovp} > 1$). We illustrate this with an example by choosing default input parameters based on real data for our model and designing a worst-case provisioned and an overprovisioned system. We then analyze the impact of each individual parameter (while keeping others constant) on the performance of the two systems. These default input parameters are specified in Table 4.7 and have been derived based on the node data from Section 4.3. The values for the node power for the high-end and the older-generation node include the CPU, memory and base power. We use application-specific scalability models; an overview of these models was provided in Sections 4.2 and 4.4.3.

Table 4.7: Example: Default Input Parameters

ID	**Value**
Effective cost ratio, r_c	1.7
Performance Parameter, r_p	27%
System Power Budget, P_{sys}	7000 W
Maximum Node Power, P_{n_max}	380 W
Minimum Node Power, P_{n_min}	180 W

Table 4.8: Example: Workload Scalability Model Parameters

Application	**Parameters** $\{m, c\}$
SPhot	$\{-1.114, 73.07\}$
SP-MZ	$\{-0.112, 7.00\}$
BT-MZ	$\{-0.069, 8.50\}$
LU-MZ	$\{-0.542, 25.93\}$

Table 4.9: Example: Intermediate and Output Values

Application	$\mathbf{n_{wc}}$	$\mathbf{n_{lim}}$	$\mathbf{c_{wc}}$	$\mathbf{n_{ovp}}$	$\mathbf{t_{ovp}}$ **(s)**	$\mathbf{t_{wc}}$ **(s)**	$\mathbf{s_{ovp}}$
SPhot	18	50	30.6	30	39.64	38.70	0.98
SP-MZ	18	50	30.6	30	3.51	3.61	**1.02**
BT-MZ	18	50	30.6	30	6.41	5.29	0.83
LU-MZ	18	50	30.6	30	9.66	11.80	**1.22**

Table 4.8 specifies the workload scalability model parameters for the four applications and Table 4.9 shows the intermediate values as well as the predicted output values for our example. Because we use a simple linear workload scalability model, we need to enforce a limit on the maximum number of nodes for the validity of this linear behavior. For our applications in this example, we assume that this limit is 48 nodes.

We present results for SPhot, SP-MZ, BT-MZ and LU-MZ because they all have distinct workload scalability models. As can be observed from Table 4.9, for workloads that scale well, such as the ones with characteristics similar to LU-MZ (refer to Figures 4.1 and 4.2), it is possible to achieve better performance with overprovisioning with the same cost budget (s_{ovp} of 1.22). Similarly, for workloads with characteristics similar to SP-MZ and SPhot, a break even point can be determined. On the other hand, for applications that do not scale well, such as BT-MZ, worst-case provisioning leads to better performance. This can be observed from the s_{ovp} value of 0.83 for BT-MZ.

We now present some detailed graphs to better understand the scenarios where overprovisioning leads to a net benefit and to understand the impact of the input parameters on s_{ovp}. For each graph, the y-axis is the derived ratio, s_{ovp}. The x-axis varies based on the input parameter under consideration. For each input parameter that is being varied on the x-axis, all other input parameters are held constant and have values from Table 4.7. For readability, the graphs are not centered at the origin, and the break even points have been marked by drawing a red line at $s_{ovp} = 1$. Anything above this line is a scenario where overprovisioning does better than worst-case provisioning.

We conduct this analysis to explore the various possible scenarios that may occur during the procurement of a real HPC system. For example, the effective cost ratio

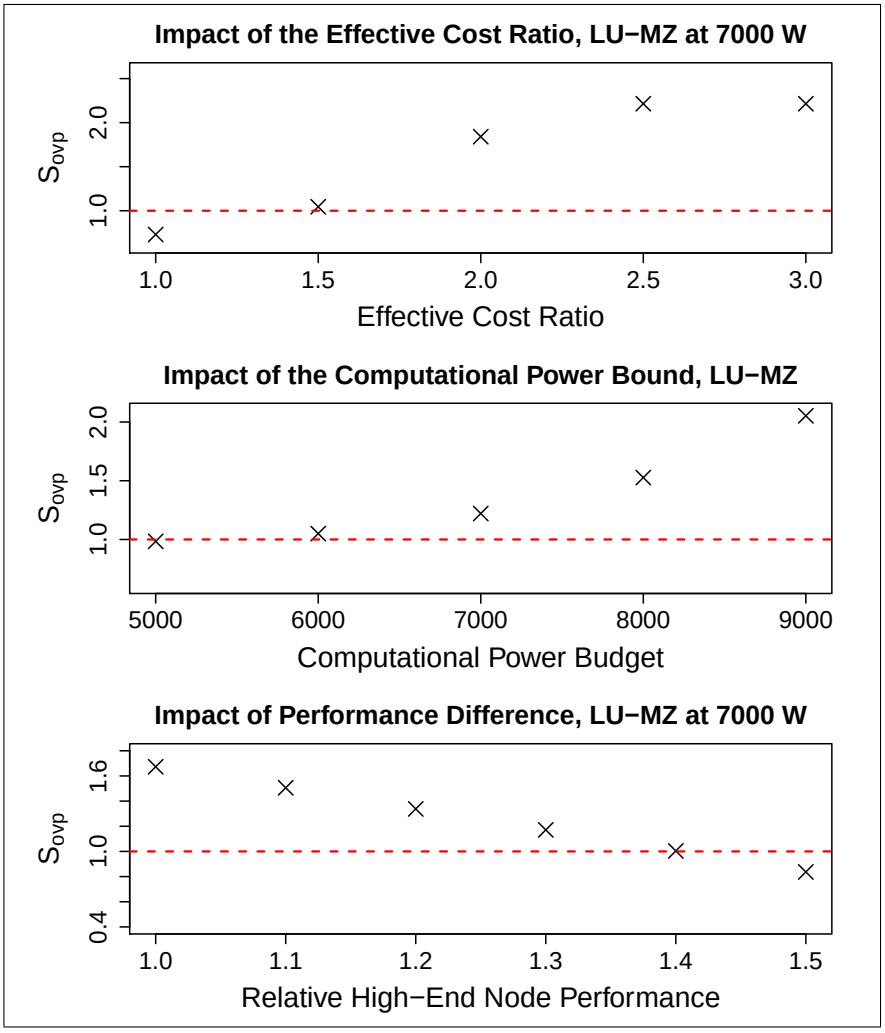


Figure 4.3: LU-MZ Analysis

may vary based on the negotiation with the vendor. Similarly, the performance across two nodes may differ based on which micro-architectures are being considered.

Figure 4.3 shows results for the LU-MZ application. In this figure, there are three sub-graphs. The first one depicts the impact of varying the effective cost ratio, r_c on s_{ovp}. As can be observed from the graph, for LU-MZ, overprovisioning leads to a net benefit when the effective cost ratio is low (for example, when r_c is 1.5). The effective cost ratio affects the degree of overprovisioning directly. When the effective cost ratio is high, it is possible to buy many more cheaper, older-generation nodes than when the effective cost ratio is low. A cost ratio of 1 indicates that the high-end node and the older-generation

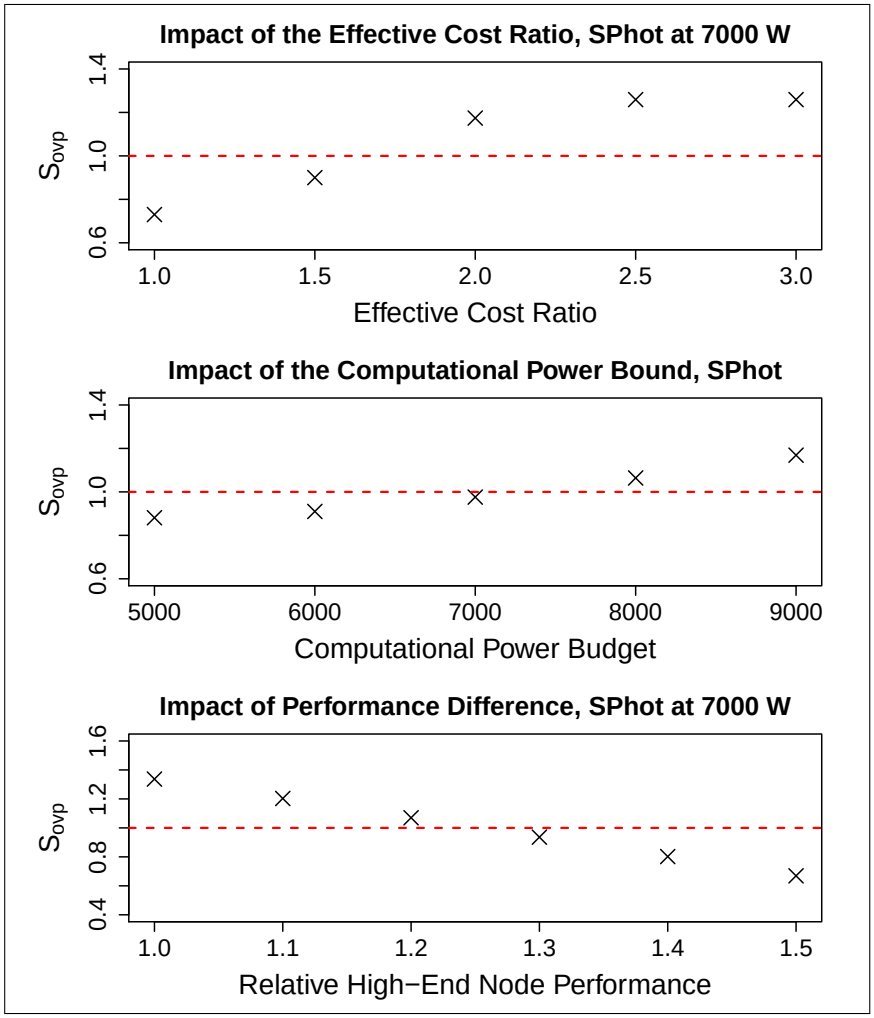


Figure 4.4: SPhot Analysis

node have the same price. This means that there is no overprovisioning, because both the worst-case provisioned system and the overprovisioned system will have the same node count. This, however, is not a realistic scenario. It is expected that the high-end node will be more expensive than the older-generation node. The higher the effective cost ratio, the easier it is to overprovision by a larger degree. In case of LU-MZ, we see a net benefit with overprovisioning even with relatively low cost ratios.

The second sub-graph shows the impact of varying the system power bound, P_{sys} on s_{ovp}. We observe that a higher system power bound results in almost super-linear improvement in s_{ovp} for LU-MZ. This can be attributed to the fact that a higher system

power bound has the potential to increase the degree of overprovisioning. With a higher degree of overprovisioning, it is possible for an application to scale to more nodes. However, this is dependent on the scalability characteristics of the workload under consideration. In case of LU-MZ, it has good scaling behavior, as we noted in Figure 4.2. As a result of this, s_{ovp} is impacted significantly when the system power bound is varied. This will not always be the case, though.

The third sub-graph in Figure 4.3 analyzes the impact of the performance parameter, r_p on s_{ovp}. This parameter is determined by benchmarking the high-end node and the older-generation node using a standard suite such as PassMark (single node tests). It indicates the performance gap between the high-end and the older-generation nodes as a percentage value. A high value of r_p indicates that the high-end node is significantly better than the older-generation node when it comes to single-node performance. When r_p is 0%, it means that both nodes have the same performance. A value of 0% for r_p is unrealistic though, because the high-end node will always perform better than the older-generation node. For LU-MZ, we observe that even when the high-end node is 40% faster, the overprovisioned system results in a net benefit.

Figures 4.4 (see previous page) and 4.5 show the results for SPhot and SP-MZ, respectively. As discussed previously, the scalability characteristics of the application affect s_{ovp} significantly. With SPhot as well as SP-MZ, the overprovisioned system results in a net benefit when the effective cost ratio of the high-end node to the older-generation node is around 2. Also, increasing the effective cost ratio further does not result in proportional improvements in s_{ovp}. One might expect that a higher cost ratio means that the degree of overprovisioning will be high, and that this will result in performance improvements in an overprovisioned system. However, this is not the case. This is because the degree of overprovisioning is also constrained by the power budget, which determines n_{lim}. As a result, while it might be possible to buy many more older-generation nodes because of a high effective cost ratio, it is not possible to do so without exceeding the system power budget. Also, workloads with scaling characteristics similar to SPhot and SP-MZ are also less sensitive to changes in the system power budget, unlike LU-MZ.

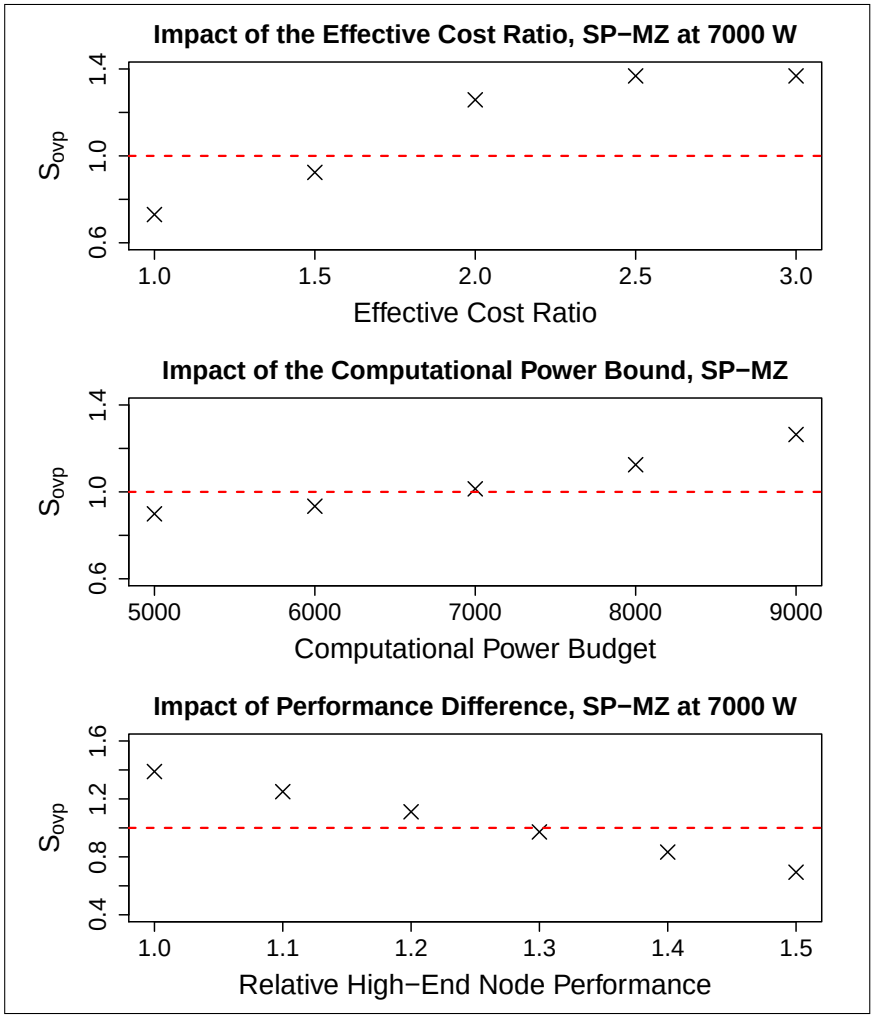


Figure 4.5: SP-MZ Analysis

The scaling characteristics for the applications depend on several factors, such as their per-node memory requirements and their communication traces. Depending on how the application scales and performs when we add more nodes and pick the best performing configuration (core count and power per node) at each node count, the benefits of overprovisioning vary. For example, with LU-MZ, we observe good scaling behavior. With SP-MZ and SPhot, this trend is less pronounced, and they have average scaling characteristics. As a result, the benefits of overprovisioning are limited when compared to LU-MZ.

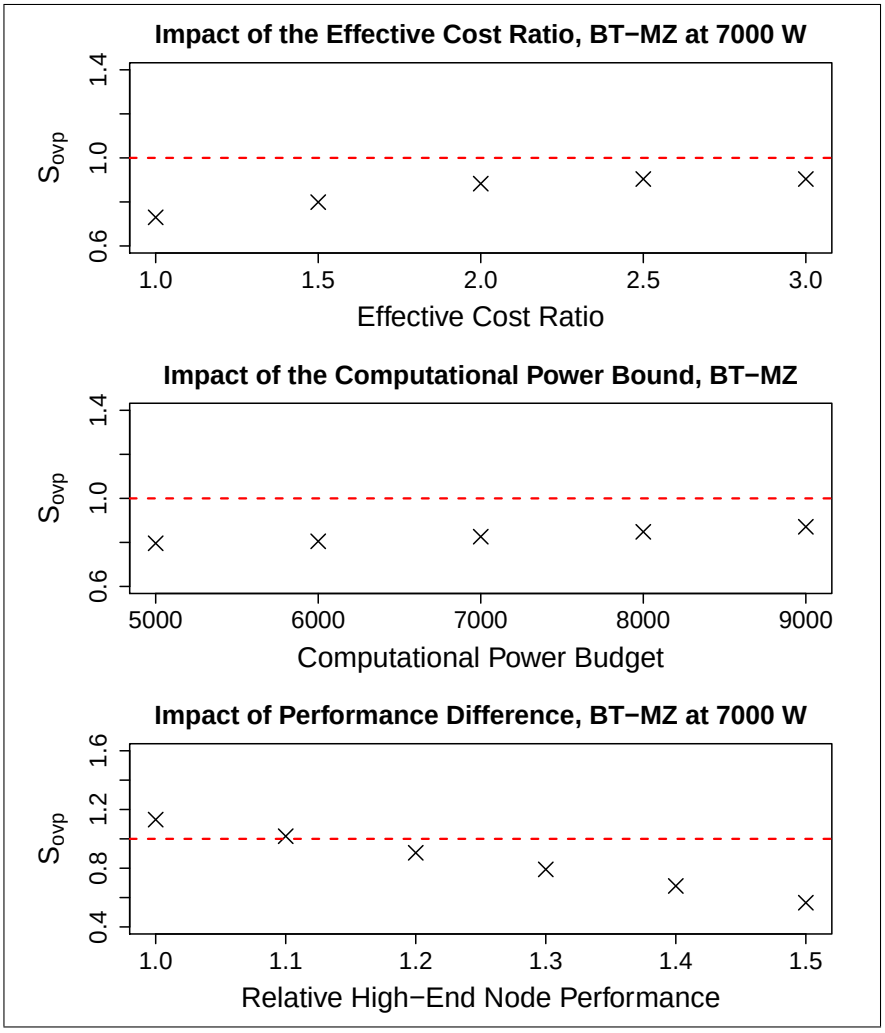


Figure 4.6: BT-MZ Analysis

Let us now analyze BT-MZ, which has poor scaling characteristics when compared to the other three applications (refer to Figures 4.1 and 4.2). As a result of this scaling behavior, for BT-MZ, the worst-case provisioned system almost always does better than the overprovisioned system, as can be observed from Figure 4.6. While increasing the effective cost ratio and the system power budget improves the performance of the overprovisioned system, these improvements are not sufficient to reach a break even point. The only scenario in which the overprovisioned system does better than the worst-case provisioned system, is when r_p is in the range of 10-20%. It is thus critical to consider the characteristics of the workload when designing overprovisioned systems.

4.5 Summary

Overall, we observe that there are several scenarios in which it is possible to reap the benefits of an overprovisioned system with the same cost budget. This indicates that overprovisioning has potential as we venture toward the power-limited exascale era. Overprovisioned systems can be cost-effective in some scenarios, and can improve utilization and lead to significant performance improvements. Our goal in this chapter was to develop a model that future supercomputing system designers can use to determine whether or not overprovisioning is a viable option for their facility. In this chapter, we limited ourselves to analyzing a single application on an overprovisioned system. Future work involves understanding the impact of this model in a more realistic scenario where several users submit jobs to a shared HPC system. In the next chapter, we introduce RMAP, a low-overhead, scalable resource manager targeted toward power-constrained, overprovisioned systems. Extending the model presented in this chapter by analyzing workload scalability characteristics and then analyzing it further with a system such as RMAP is part of our future work in this area.

CHAPTER 5

RESOURCE MANAGER FOR POWER

Supercomputers today are underutilized from the point of view of power. This results in wasted power and limited performance, and as we enter the exascale era, utilizing power effectively to accomplish more science and improve performance becomes a challenging and important problem. We presented the benefits of overprovisioning in Chapter 3. For overprovisioning to be feasible, system software needs to be made power-aware. In this chapter, we present the design and implementation of Resource MAnager for Power (RMAP), a low-overhead, scalable resource manager targeted at future overprovisioned, power-constrained systems.

We design three policies within *RMAP*: a baseline policy for safe execution under a power bound; a naive policy that uses overprovisioning; and an adaptive policy that is designed to improve application performance by using overprovisioning in a power-aware manner. The goal of the latter strategy is to provide faster job turnaround times as well as to increase overall system resource utilization.

We accomplish this by introducing *power-aware backfilling*, a simple, greedy algorithm that allows us to trade some performance benefits of overprovisioning for better power utilization and shorter job queuing times. We discuss the design and implementation of RMAP in the following sections.

5.1 Designing Power-Aware Schedulers

Power-aware schedulers are presented with several new design challenges. First, they must enforce job-level power bounds as they manage and allocate nodes in order to meet the global system level power constraint. Additionally, they need to optimize for overall system throughput as well as individual job performance under job-level power bounds, which means they must minimize the amount of unused (leftover) power.

Initial research in the area of power-aware resource management for overprovisioned systems deployed Integer Linear Programming (ILP) techniques to maximize throughput of data centers under a strict power budget (Sarood et al., 2014). While this is an interesting research approach, the proposed algorithm is not fair-share and is not sufficiently practical for deployment on a real HPC cluster. Each per-job scheduling decision involves solving an NP-hard ILP formulation, incurring a potentially high scheduling overhead and limiting scalability. One of the design and implementation goals for *RMAP* is thus of being a practical resource manager with a scheduling overhead (O(1)). ILP-based algorithms may additionally lead to low resource utilization as well as resource fragmentation, which are major concerns for high-end supercomputing centers (Feitelson, 1997; Frachtenberg and Feitelson, 2005; Feitelson and Rudolph, 1996; Feitelson et al., 2005). While allowing jobs to be *malleable* (changing node counts to grow/shrink at runtime) might help address some of these problems, very few scientific HPC applications are expected to support malleability due to the data migration, domain decomposition and scalability issues involved.

For simplicity, we assume that all jobs have equal priority, use MPI+OpenMP, and are moldable (not restrictive in terms of the number of nodes on which they can be executed). We also assume that the global power bound on the cluster is $P_{cluster}$, and that the cluster has $N_{cluster}$ nodes. We derive a power bound for each job fairly by allocating it a fraction of $P_{cluster}$ based on the fraction of $N_{cluster}$ that it requested (n_{req} being the number of requested nodes). Thus, $P_{job} = \frac{n_{req}}{N_{cluster}} \times P_{cluster}$. This allocation for P_{job} can be extended easily to a priority-based system by using weights (w_{prio}) for the power allocation. Thus, $P_{job} = w_{prio} \times \frac{n_{req}}{N_{cluster}} \times P_{cluster}$. For example, higher priority jobs could be allocated more power by using $w_{prio} > 1$, and lower priority jobs could be allocated using $w_{prio} < 1$. This work does not explore priorities further.

At any given point in time (say t), the available power in the cluster, P_{avail_t}, can be calculated by subtracting the total power consumption of running jobs from the cluster level power bound. P_{avail_t} is used to make power-aware scheduling decisions. Thus for r running jobs at time t, $P_{avail_t} = P_{cluster} - \sum_{j=1}^{r} P_{job\_j}$.

The performance of an individual job can be optimized by using overprovisioning with respect to the job-level power bound, P_{job}, as discussed in Chapter 3. However, in order to optimize overall system throughput and to minimize overall unused power in the system, a scheduler can make use of two possible compatible strategies: (1) dynamically redistribute the unused power to jobs that are currently executing, or (2) suboptimally schedule the next job with the unused (available) power and nodes.

Dynamically redistributing power to executing jobs to improve performance can be challenging because allocating more power per node may result in limited benefits (discussed later in detail in Section 5.4). In order to improve performance and to utilize power better, the system may have to change the number of nodes (or cores per node) at runtime. However, varying the node count at runtime (malleability) is not possible with current MPI implementations and resource managers (although the MPI standard provides the functionality). Additionally, dynamically changing the node counts of a job would incur data decomposition and migration overhead (Laguna et al., 2014).

We explore extensions of traditional backfilling for a power-aware scenario. Traditional backfilling attempts to utilize as many nodes as possible in the cluster by breaking the FCFS order. Similarly, our new greedy approach, *power-aware backfilling*, attempts to use as much global power as possible by scheduling a job with currently available power. Most cases involve sacrificing some performance benefits attained from overprovisioning. The key idea is to schedule a job with less power than was requested (derived using fair share) and schedule it with a suboptimal configuration, and to do so with execution time guarantees. Power-aware backfilling can adapt to extremely power-constrained scenarios and to scenarios with too much leftover power. Our approach builds on standard backfilling.

Figure 5.1 shows an example in which we assume that jobs `A` and `B` are currently waiting in the queue. Job `A` requested more power than is currently available in the system. Traditionally, Job `A` waits in the queue until enough power is available. Our approach schedules Job `A` immediately with *available* power. While we increase the execution time of Job `A`, our approach improves the overall turnaround time of Job `A` as well as the other jobs in the queue, and utilizes power better.

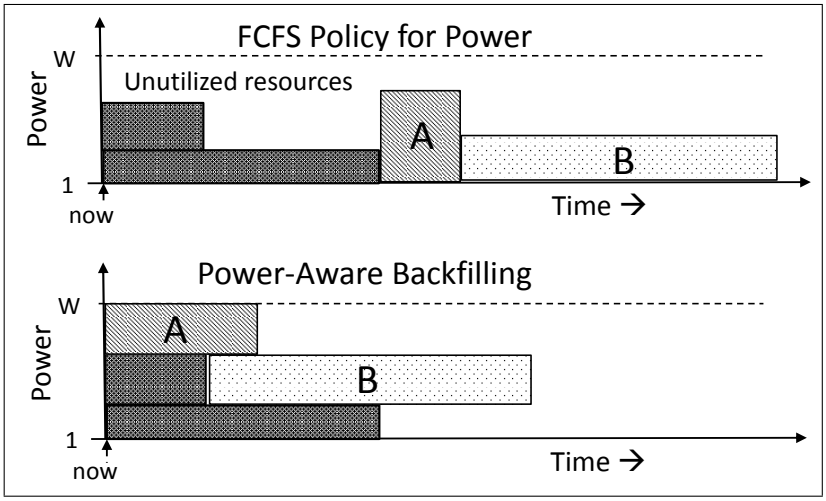


Figure 5.1: Advantage of Power-Aware Backfilling

The key idea is to use power-aware backfilling while adhering to the user-specified time deadline for the job. Overprovisioning under a job-level power bound will often exceed the user's performance expectations. However, allowing users to specify a maximum slowdown for their job and thus trade their job's execution time for a faster turnaround time is an added incentive.

We focus primarily on trading some of the benefits obtained from overprovisioning to utilize all available power to run jobs faster and to schedule more jobs. Keeping cluster resources utilized (both nodes and power) via backfilling leads to better average turnaround times for the jobs, which in turn increases throughput. We thus focus on minimizing the average per-job turnaround time in this work. The policy that we develop is called the *Adaptive* policy, and we discuss it in the next section.

5.2 Scheduling Policies

We now discuss the power-aware scheduling policies that we implemented in *RMAP*. Each of these policies needs to obtain job configuration information given a power bound. The details of how these configurations are determined are presented in Sections 5.3 and 5.4, which discuss the low level implementation details. Users specify nodes and time as input, along with an optional threshold value for the *Adaptive* policy.

Policy	**Input to Policy**	**Description**
Traditional	(n_{req}, t_{req})	Pick the *packed* configuration $(c = 16, p = max = 115W)$
Naive	(P_{job}, t_{req})	Pick the optimal configuration under the derived job power limit
Adaptive	$(P_{job}, t_{req}, thresh)$	Use power-aware backfilling to select a configuration

Table 5.1: RMAP Job Scheduling Policies

All policies require that the user provide a number of nodes (n_{req}) and maximum job time (t_{req}). We derive, P_{job}, which is the job-level power bound based on user input, as discussed in the previous subsection. This job-level power bound is an input to our scheduling policies (see Table 5.1). All three policies use basic node-level backfilling.

5.2.1 The *Traditional* Policy

In this policy, the user is allocated a configuration with their requested node count that uses all available cores on a node at maximum possible power. A job that requests large node counts may exceed the system's global power bound. In this case, the *Traditional* policy allocates as many nodes (with all cores on the node and maximum power per node) as it can to the job without exceeding the system-wide budget (an unfair job-level power allocation). Alternatively, we could reject the job due to power constraints.

More formally, let c_{max} be the maximum number of cores per socket, p_{max} the maximum package power per socket, $P_{(n \times c,p)}$ the total power consumed by the job in the $(n \times c, p)$ configuration, and $P_{cluster}$ the global power bound on the cluster. Then, for a job requesting n_{req} nodes for time t_{req}, the *Traditional* policy allocates the $(n_{req} \times c_{max}, p_{max})$ configuration if $P_{(n_{req} \times c_{max}, p_{max})} \leq P_{cluster}$.

Otherwise, it allocates the $(n_{max} \times c_{max}, p_{max})$ configuration to the job, where n_{max} is the maximum n such that $P_{(n \times c_{max}, p_{max})} \leq P_{cluster}$.

Here, the job must wait in the queue if sufficient resources (nodes and power) are unavailable when the scheduling decision is being made (if $P_{avail_t} < P_{job}$).

5.2.2 The *Naive* policy

In this policy, we overprovision with respect to the job-level power bound. Given the derived job-level power bound, $P_{job} \leq P_{cluster}$, and an estimated runtime, t_{req}, the *Naive* policy allocates the $(n \times c, p)$ configuration that leads to the best time t_{act} under that power bound. Thus, $t_{act} = min(T)$, where $T = \left\{t_{(n \times c,p)} : \ P_{(n \times c,p)} \leq P_{job}\right\}$.

If $t_{act} > t_{req}$, the system sets the deadline $t_{deadline}$ for the job to t_{act} instead of t_{req} during job launch, so that the job does not get killed prematurely. This scenario occurs if the user's performance estimates are inaccurate and cannot be met with the derived power bound, and the best performance level that the *Naive* policy can provide under the specified power bound P_{job} is worse than t_{req}. In the scenario that $t_{act} < t_{req}$, $t_{deadline}$ is not updated until job termination. *RMAP* will kill the job after $t_{deadline}$. The main purpose for t_{req} is to have a valid deadline in case the job fails or crashes. User studies suggest that t_{req} is often over-estimated (by up to 20%) (Tsafrir et al., 2007).

Again, the job may have to wait in the queue until enough power is available to schedule it.

5.2.3 The *Adaptive* policy

This policy's goal is to allow (1) users to receive better turnaround time for their jobs, and (2) the system to minimize the amount of unused power to achieve better average turnaround time for all jobs. Similar to the *Naive* policy, the inputs are a (derived) job-level power bound and duration. However, the *Adaptive* policy considers these values as suggested and uses power-aware backfilling. It also trades the raw execution time of the application as specified by the user for potentially shorter turnaround times. The user can specify an optional *threshold (th)*, which denotes the percentage slowdown in actual runtime that the job can tolerate. When *th* is not specified, we assume that it is zero (no slowdown). Algorithm 1 gives an overview of this policy.

The *Adaptive* policy uses the suggested job-level power bound to check if the requested amount of power is currently available. If so, it obtains the best configuration under this power bound (similar to the *Naive* policy). If not, it determines a

Algorithm 1
***Adaptive* Policy**
Inputs:
(1) Currently available power, P_{avail},
(2) Requested power and time, P_{job}, t_{req},
(3) Threshold value (optional), th, which is the percentage slowdown the job can tolerate,
(4) Information about job power profiles and configurations. More specifically, a way to determine, $P_{(n \times c,p)}$, the actual measured power for the configuration $(n \times c, p)$, and the time $t_{(n \times c,p)}$ it takes.
The threshold th is assumed to be zero percent if it has not been provided as an input.

if $P_{job} \leq P_{avail}$ **then**
    Use the *Naive* policy.
else
    Determine $t_{act} = min(T)$, where $T = \{t_{(n \times c,p)} : \ P_{(n \times c,p)} \leq P_{avail}\}$

    if $t_{act} <= (1 + th) \times t_{req}$ **then**
        Schedule the job (immediately) with the configuration $(n \times c, p)$ with time t_{act}
    else
        Job waits in queue, as it does not meet the slowdown requirements.
    end if
end if

suboptimal configuration based on currently available power and the threshold value. The advantage for the user is that the job wait time may be significantly reduced when compared to the other two policies. Similarly, for the administrators, the advantage is better resource utilization (in terms of nodes and overall power) and throughput.

More specifically, if $P_{avail_t} > P_{job}$, the *Adaptive* policy uses the same mechanism as the *Naive* policy. However, when $P_{avail_t} < P_{job}$, it determines $t_{act} = min(T)$, where $T = \{t_{(n \times c,p)} : \ P_{(n \times c,p)} \leq P_{avail_t}\}$, and schedules the job immediately with the $(n \times c, p)$ configuration with time t_{act} as long as $t_{act} <= (1 + th) \times t_{req}$. Thus, the job's wait time is reduced while meeting the performance requirement.

5.2.4 Example of Policies

As an example, consider SP-MZ from the NAS Multizone benchmark suite (Bailey, 2006). Some configurations for SP-MZ (Class C) are listed in Table 5.2.

ID	Configuration $(n \times c, p)$	Total Power (W)	Time (s)
C1	$(6 \times 16, max = 115)$	796.4	447.9
C2	$(8 \times 12, 65)$	783.8	415.3
C3	$(8 \times 10, 80)$	738.2	439.2

Table 5.2: List of Configurations for SP-MZ

We now discuss three scenarios in which 750 W of power and 10 nodes are available in the cluster, and job `A` that is currently executing terminates in 1000 seconds. Also, a user has requested 6 nodes for 450 seconds, and, the derived job-level power bound is 800 W.

Scenario 1, Traditional Policy

Here, *RMAP* allocates configuration `C1` to the job but it waits until job `A` terminates and enough power is available.

Scenario 2, Naive Policy

RMAP allocates configuration `C2` to the job but also waits until job `A` terminates and enough power is available.

Scenario 3, Adaptive Policy (threshold=0%)

A threshold value of 0% means that the user cannot compromise on performance. Under the *Adaptive* policy, *RMAP* checks if enough power (800 W) is available in the system. It then determines that `C3` does not violate the performance constraint (450 seconds), and job `A` can be launched immediately with the currently available power (750 W). We distinguish this case from Scenario 2, which will *always* pick `C2`.

Picking `C3` reduces the wait time of the job significantly (by 1000 seconds). Also, in scenarios 1 and 2, 750 W of power is wasted for 1000 seconds. In this scenario, power is utilized more efficiently and turnaround time for the job is reduced.

5.3 Implementation

We implemented *RMAP* within the widely-used, open source resource manager for HPC clusters, SLURM (Yoo et al., 2003). SLURM is used on several Top500 (Strohmaier et al., 2014) supercomputers. It provides a standard framework for launching, managing and monitoring jobs on parallel architectures. The `slurmctld` daemon runs on the head node of a cluster and manages resource allocation. Each compute node runs the `slurmd` daemon for launching tasks. `Slurmdbd`, which also runs on the head node, collects accounting information with the help of a MySQL interface to the `slurm_acct_db` database.

As described earlier, *RMAP* supports overprovisioning and implements three power-aware scheduling policies that adhere to a global, system-wide power budget. We refer to our extension of SLURM as P-SLURM. RMAP can similarly be implemented within other resource managers.

Our scheduling policies require the ability to produce execution times for a given configuration under a job-level power bound. Table 5.3 shows the information that P-SLURM requires. We refer to this as the `job_details_table`, and we added this table to the existing `slurm_acct_db`. Values for `exec_time` and `tot_pkg` can be measured or predicted.

We developed a model to predict the performance and total power consumed for application configurations in order to populate this table. The next section presents the details of this model. Furthermore, to understand and to analyze the benefits of having exact application knowledge, we also included another table within the SLURM database (with the same schema) that contains an exhaustive set of empirically measured values. For simplicity, we populated both tables in advance and the scheduler queried the database for information when making decisions, making the decision complexity O(1). The model can also be used to generate values dynamically without needing a database. However, this may incur scheduling overhead and call for advanced space-search algorithm implementations within the scheduler (such as hill climbing). We do not address this issue in this work.

Field	Description
id	Unique Index (Primary)
job_id	Application ID
nodes	Number of nodes
cores	Number of cores per node
pkg_cap	PKG Power Cap
exec_time	Execution Time
tot_pkg	Total PKG Power

Table 5.3: Schema for Job Details Table

5.4 Predicting Performance and Power

In this section, we discuss the models that *RMAP* deploys in its policies. The models predict execution time and total power consumed for a given configuration (number of nodes, number of cores per node, and power cap per socket). We first collect exhaustive power and performance information. We range the node counts from 8 to 64, core counts from 8 to 16, and power from 51 W to 115 W. The dataset we build contains 2840 data points, with 5 different power caps, 15 different node counts, 5 different core counts per node and 8 applications (355 per application). Before we discuss the model, we present the details of our dataset in the following subsection.

5.4.1 Dataset

In order to study HPC application power profiles, we selected eight strongly-scaled, load-balanced, hybrid MPI + OpenMP applications (described below) and gathered power and performance data for these at 64 nodes on the *Cab* cluster at LLNL. *Cab* is a 1200-node Intel Sandy Bridge cluster, with 2 sockets per node and 8 cores per socket. We measured per-socket power with Intel's *Running Average Power Limit* (RAPL) technology (Intel, 2011; Rountree et al., 2012). The maximum power available on each socket was 115 W. We only measured socket power in these experiments, as support to measure memory power was not available due to BIOS restrictions.

Similar to Chapter 3, we used eight applications in our study. Four of these are real HPC applications, which include SPhot (Lawrence Livermore National Laboratory,

2001) from the ASC Purple suite (Lawrence Livermore National Laboratory, 2002), and BT-MZ, SP-MZ and LU-MZ from the NAS suite (Bailey, 2006). SPhot is a 2D photon transport code that solves the Boltzmann transport equation. The NAS Multi-zone benchmarks are derived from Computational Fluid Dynamics (CFD) applications. BT-MZ is a the Block Tri-diagonalsolver, SP-MZ is the Scalar Penta-diagonal solver, and LU-MZ is the Lower-Upper Gauss Seidel Solver. We used Class D inputs for NAS, and for SPhot, the NRuns parameter was set to 16384.

We also used four synthetic benchmarks in our dataset to cover the extreme cases in the application space. These are (1) Scalable and CPU-bound (SC), (2) Not Scalable and CPU-bound (NSC), (3) Scalable and Memory-bound (SM), and (4) Not Scalable and Memory-bound (NSM). The CPU-bound benchmarks run a simple spin loop, and the memory-bound benchmarks perform a vector copy in reverse order. Scalability is controlled by adding `MPI_Alltoall` communication. We used MVAPICH2 version 1.7 and compiled all codes with the Intel compiler version 12.1.5. We used the scatter policy for OpenMP threads.

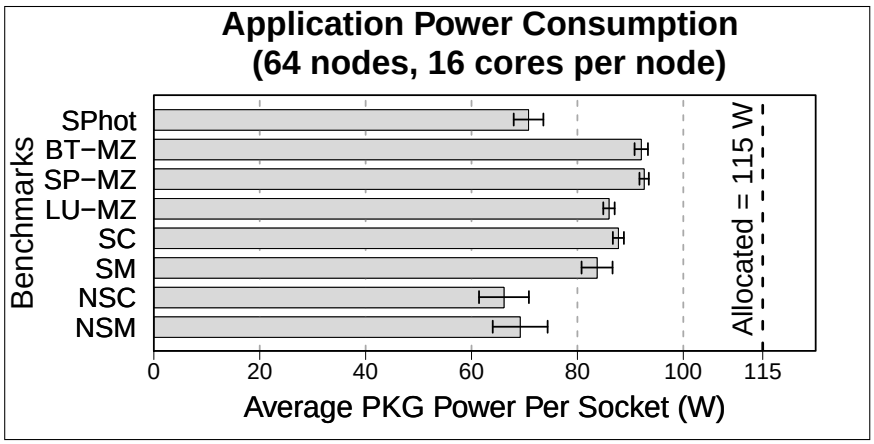


Figure 5.2: Application Power Consumption

Figure 5.2 shows data for application power consumption for the eight applications running at 64 nodes, 16 cores per node, and maximum power per node. Each bar represents the average power consumption per socket (averaged over 128 sockets on 64

nodes) for an application. The minimum and maximum power consumed per socket by the application are denoted by error bars. While all applications were allocated 115 W per socket, they only used between 66 W (NSC) to 93 W (SPMZ). On average, they only used 81 W or 71% of the allocated socket power.

The maximum global power bound for our cluster was $64 \times 2 \times 115\ W$, which is 14720 W. In order to analyze various degrees of overprovisioning, we chose five global power bounds for our study—6500 W, 8000 W, 10000 W, 12000 W and 14720 W. These were determined by the product of (1) the number of nodes and (2) the minimum and maximum package power caps possible per socket (51 W and 115 W).

With n_{max} being the maximum number of nodes that one can run at peak power without exceeding the power bound, the worst-case provisioned configuration is $(n_{max} \times 16, 115)$. We measured execution time and total power consumed for each of the benchmarks in the configuration space discussed earlier.

Figure 5.3 shows results of overprovisioning when compared to worst-case provisioning for the four HPC benchmarks at the five global power bounds discussed above. The x-axis is normalized to the worst-case provisioned power (14720 W). For this dataset, we saw a maximum improvement of 34% (11% on average) in performance compared to worst-case provisioning.

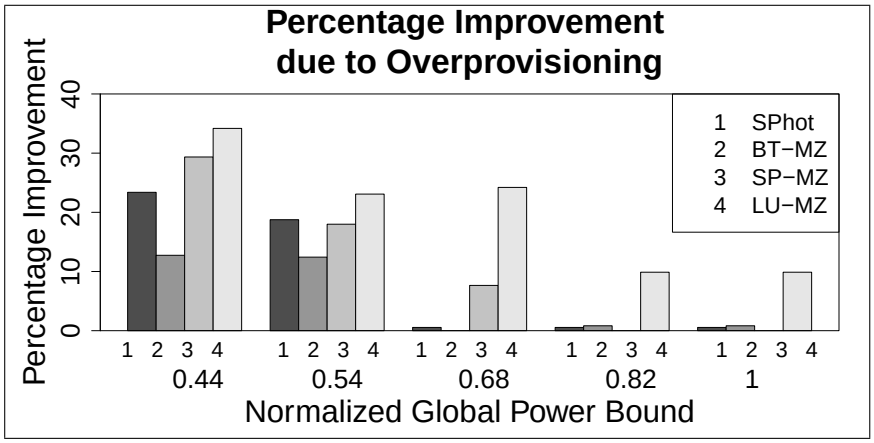


Figure 5.3: Performance with Overprovisioning

5.4.2 Model

We used 10% of this data for training and obtained application-specific linear regression parameters that allowed us to predict application execution time and total package power consumption at a given configuration. A logarithmic polynomial regression of degree two was used for this purpose.

Next, we validated our models with our previously measured data. When using only 10% of data for model training, the average error for execution time is below 10%, and the maximum error for the same is below 33%. Figure 5.4 shows the absolute (seconds) and relative (percentage) error quartiles for all benchmarks when predicting execution time at arbitrary configurations. For all benchmarks, the third quartile is under 13.2% with the median below 7.7%.

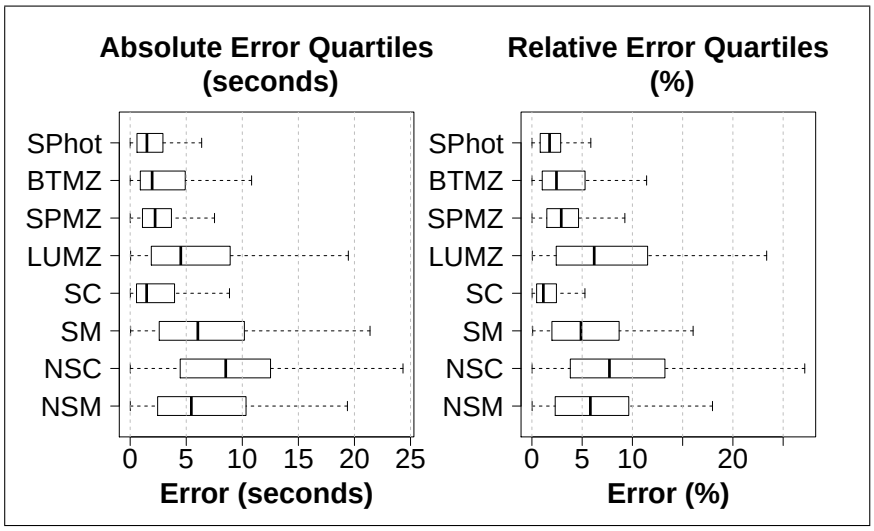


Figure 5.4: Error Quartiles of Regression Model

Figure 5.5 shows the detailed error histograms for both performance and power across the entire dataset (355 data points per application). The x-axes on both graphs represent the error bins, and the y-axes represent the frequencies. Regions with over-predicted and under-predicted values have been identified. It is important to note that if we over-predict the power consumed by a job, we may block the next job in the queue due to lack of enough power. On the other hand, under-predicting the power

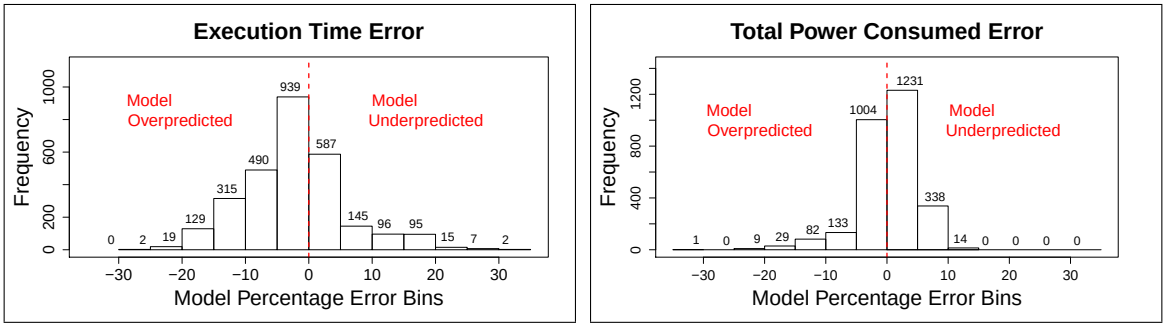


Figure 5.5: Range of Prediction Errors

may lead to a situation where we exceed the cluster-level power bound (worst-case scenario). In our model, for 96% of our data, the under-prediction for power was no more than 10%, and the worst case was under 15%. This issue can be addressed by giving *RMAP* a conservative cluster-level power bound that is 15% less than the actual power bound, or by relying on the fact that most supercomputing facilities are designed to tolerate these kind of surges (NPFA, 2014).

5.5 Experimental Details

In order to set up our simulation experiments for *RMAP*, we populate the `job_details_table` with application configuration information, as discussed in Section 5.3. In all our experiments, we consider the same architecture as *Cab*. We consider a homogeneous cluster with 64 nodes and global power bounds ranging from 6500 W to 14000 W, based on the product of the number of nodes and the minimum and maximum package power caps that can be applied to each socket (51 W and 115 W). Each node has two 8-core sockets.

We generate job traces from a random selection of our recorded configuration data as inputs for P-SLURM. Each trace has 30 jobs to ensure a reasonable simulation time. The total simulation time with all traces, power bounds, node counts and policies was about 3 days (approximately 30 minutes for each trace). We use a Poisson process to simulate job arrival (Feitelson and Jette, 1997; Feitelson, 2002). Job arrival rate is sparse on purpose (to make queue times short in general), so we can be conservative

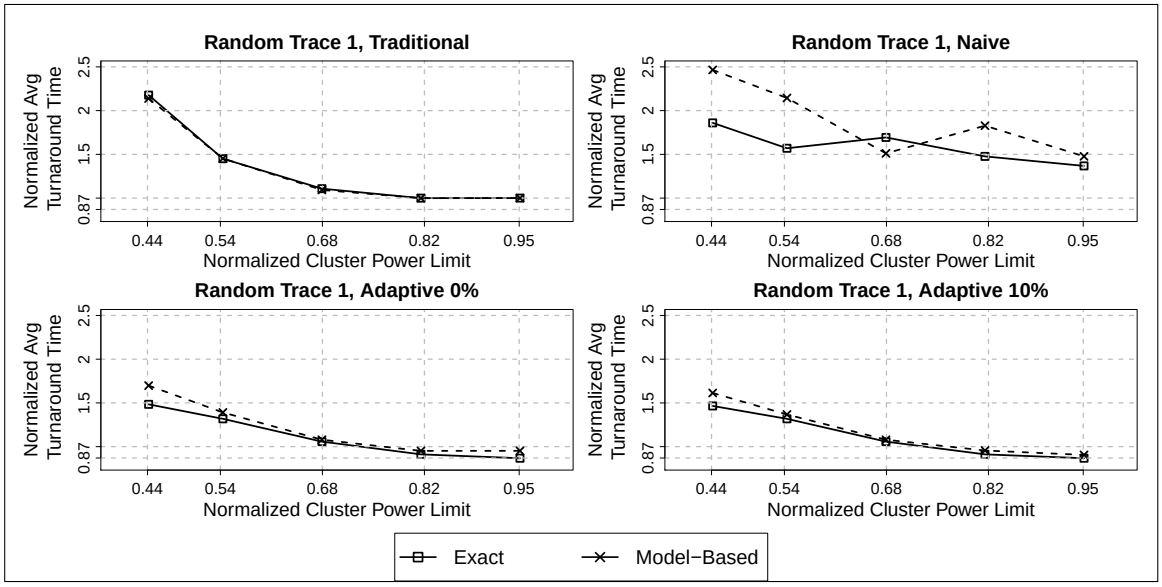


Figure 5.6: Model Results on the Random Trace

in the improvements that we report with the *Adaptive* policy. We select the following types of job traces to evaluate our scheduling policies.

- Traces with *small-sized* and *large-sized* jobs: To identify scenarios which may favor one power-aware scheduling policy over another, we create trace files with small-sized and large-sized jobs. Users could request up to 24 nodes in the former, and have to request at least 40 nodes for the latter.

- *Random* traces: For completeness, we also generate two random job traces. Users could request up to 64 nodes for these traces. We refer to these traces as *Random Trace 1* and *Random Trace 2*. The two traces differ in the job arrival pattern as well as job resource (node count) requests, thus exhibiting different characteristics. While both traces have the same number of jobs and used the same arrival rate parameter in the Poisson process, *Random Trace 1* has many jobs arrive early in the trace, whereas arrival times are more uniform in *Random Trace 2*.

5.6 Results

In this section, we discuss our results and evaluate our scheduling policies on a Sandy Bridge cluster (which was described in Section 5.4.1) that we simulated with P-SLURM. In our experiments, we assume that all jobs have equal priority. For fairness and ease of comparison, in each experiment we assume that all users can tolerate the same slowdown (threshold value for the *Adaptive* policy). For readability, we do *not* center our graphs at the origin.

All figures in this section compare the *Traditional* and the *Naive* policies to the *Adaptive* policy when the global, cluster-level power bound is varied across the cluster. The x-axis is the global power limit enforced on the cluster (6500 W–14000 W), normalized to the worst-case provisioned power (in this case, that equals 64 $\times$ 115 W $\times$ 2, which is 14720 W). The y-axis represents the average turnaround time for the queue, normalized to the average turnaround time of the *Traditional* policy at 14720 W (lower is better). The *Traditional* policy mimics worst-case provisioning, which unfairly allocates per-job power and always uses Turbo Boost, unlike the other two policies, which are fair-share and use power capping. Also, all three policies have O(1) decision complexity, so we do not compare their scheduling overheads.

We start by evaluating the model discussed in Section 5.4 when applied to *RMAP* and its policies. We then compare and analyze the three policies by applying them to different traces at several global power bounds. Finally, we analyze two traces in detail to explore how altruistic behavior on the part of the user can improve turnaround time, and how the *Adaptive* policy can improve system power utilization.

5.6.1 Model Evaluation Results

This section explores the impact of using our model for predicting application configuration performance and power. Figure 5.6 compares average turnaround time for *Random Trace 1* at 5 different global power caps. Configuration performance and total power consumed are predicted for each job in the trace. The former is used for determining execution time, and the latter is used to determine available power.

For the *Traditional* and *Adaptive* policies, our model is farily accurate (error is always under 10%; and is 4% on average across the two policies). We observe similar results with the other traces.

While performance prediction introduces error and affects overall turnaround times, the errors introduced by overprediction of the total power consumed by a configuration propagate and impact the turnaround time more. Scheduling and backfilling decisions can be significantly affected when they depend on available power. For example, at a lower cluster power bound, if we overpredict the power consumed by a small amount (even 3%), we might not be able to schedule the next job or backfill a job further down in the queue, resulting in added wait times for all queued jobs, particularly for the *Naive* policy at lower global power bounds.

In the subsections that follow, we conservatively establish the *minimum improvements* that the *Adaptive* policy can provide. For this purpose, we use oracular information for the *Traditional* and *Naive* policies, which are our baselines, and the model for the *Adaptive* policy.

5.6.2 Analyzing Scheduling Policies

In this subsection, we compare and analyze the power-aware scheduling policies within RMAP on the four different traces at various global power bounds.

Trace with Large-sized Jobs

Each job in this trace file requests at least 40 nodes. For all enforced global power bounds, the *Adaptive* policy leads to faster turnaround times than the *Traditional* and *Naive* policies, primarily because it fairly shares power and uses power-aware backfilling to decrease job wait times. Figure 5.7 shows that the *Adaptive* policy with a threshold of 0% improves the turnaround time by 21.7% when compared to the *Naive* policy and by 14.3% when compared to the *Traditional* policy on average (up to 47.3% and 24.6%, respectively). *Adaptive* policy with a threshold of 10% further improves the overall turnaround time by 16.1% on average when compared to the *Traditional* policy.

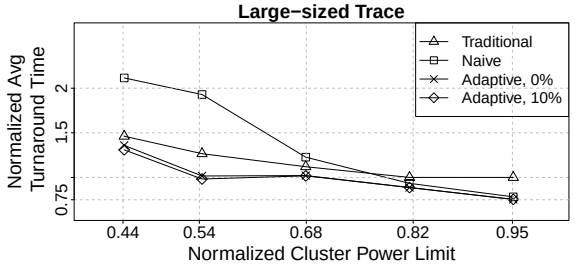


Figure 5.7: Results for Large-sized Jobs

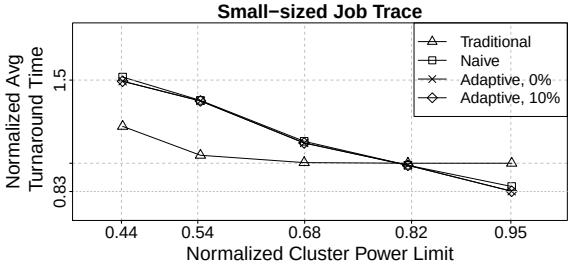


Figure 5.8: Results for Small-sized Jobs

At lower global power bounds, the *Traditional* policy serializes the jobs, leading to longer wait times and larger turnaround times. The *Naive* policy always allocates the optimal configuration under the user-specified power bound, and this may lead to longer wait times if the best configuration uses a large number of nodes.

Trace with Small-sized Jobs

Figure 5.8 depicts the results for small-sized jobs and compares all three policies. Each job in this trace file requests a maximum of 24 nodes. For small-sized jobs, the average turnaround time of the *Traditional* policy is significantly better than that of the *Adaptive* policy. This is because original wait times for small jobs are fairly short, leaving limited opportunity for the *Adaptive* policy to backfill for power. Additional improvements can be gained by running the jobs at a higher frequency in Turbo mode (3.0 GHz with Turbo on 16 cores, 2.6 GHz otherwise) due to smaller power requirements. When using power-aware policies, the small-sized jobs can be run on a small, but separate partition (such as *pSmall*) with the *Traditional* policy if needed, as discussed in Section 5.1. For the small-sized job trace, the *Adaptive* policy with a threshold of 0% leads to 16.6% worse average turnaround time (21.1% maximum degradation in average turnaround time) when compared to the *Traditional* policy. The *Adaptive* policy with a threshold of 10% depicts similar results. At the highest power bound though, the *Traditional* policy fails to adapt and the *Adaptive* policy improves the overall turnaround time by 17.1%.

Random Traces

Figure 5.6 (from the previous subsection) compares the three policies. For both the random traces, the *Adaptive* policy with a threshold of 0% does 18.52% better than the *Traditional* policy and 36.07% better than the *Naive* policy on average (up to 31.2% and 53.8%, respectively). In some scenarios, the policies lead to larger turnaround times at higher global power bounds. An example of this is the *Naive* policy at 10000 W (corresponding to value of 0.68, when normalized, in Figure 5.6). This happens because of two reasons. One, this policy strives to optimize individual job performance, so it sometimes chooses configurations with large node counts under the power bound for minor gains in performance (less than 1% improvement in execution time). This leads to other jobs in the queue incurring longer wait times and increased serialization of jobs. The other reason for this trend is aggressive and inefficient backfilling (picking the *first-fit* instead of the *best-fit*).

Figure 5.9 depicts the impact of varying threshold values on the *Adaptive* policy for the large-sized and the random traces and compares it with the *Naive* policy (which does not support thresholding). Threshold values that tolerate a slowdown of 0% to 30% in application performance are shown. For large jobs, thresholding helps the user improve the turnaround time for their job by greatly decreasing queue time. However, when there is not enough queue time to trade for, as in the case of extremely small-sized jobs, it is expected that adding a threshold will lead to larger turnaround times. The random traces shown here have a mix of small-sized and large-sized jobs. For all our traces, the *Adaptive* policies with thresholding up to 30% either improve the overall turnaround time (by up to 4%) or maintain the same turnaround time when compared to the *Adaptive* policy with a threshold of 0%. The unbounded *Adaptive* policy, which assumes that the job can be slowed down indefinitely, is also shown for comparison, and this leads to worse turnaround times.

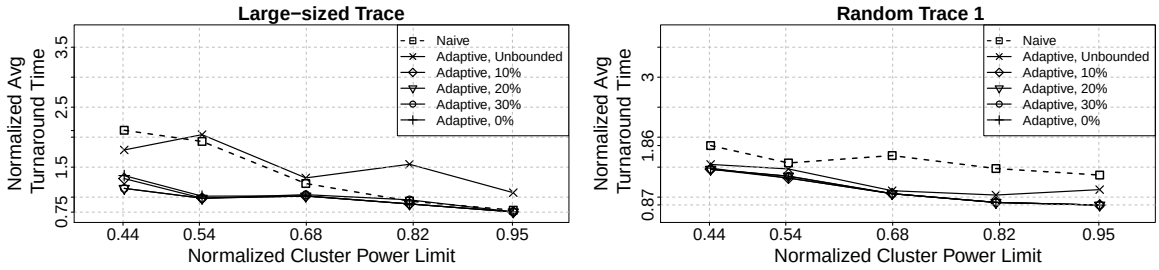


Figure 5.9: The *Adaptive* Policy with Varying Thresholds

For the large trace, *Adaptive* with a 0% threshold does 33.8% better on average than the unbounded *Adaptive*. Slowing down by 10% to 30% improves the average turnaround time by 2.1% on average (up to 4%) when compared to *Adaptive* with 0% thresholding. For the other three traces, the improvement obtained by slowing down the jobs is under 2.6% on average when compared to *Adaptive* with a threshold of 0%, and this depends on the power bound as well as the job mix. It is important to note that each data point in these graphs represents the average across all jobs in the trace at a particular global power bound. We analyze per-job performance for a single trace at a fixed global power bound in the next subsection.

5.6.3 Analyzing Altruistic User Behavior

We now present detailed results on the large-sized job trace in a power-constrained scenario, where only 6500 W of cluster-level power is available (50% of worst-case provisioning). We pick this scenario because most important jobs in a high-end cluster typically have medium-to-large node requirements. Recall that each job is requesting at least 40 nodes, so all these jobs are allocated the entire 6500 W ($P_{cluster}$) with the *Traditional* policy, leading to unfair power allocation; the scheduler runs out of power even when enough nodes are available.

Figure 5.10 shows individual job turnaround time for the *Traditional* policy, and for the *Adaptive* policy with 0% and 20% thresholding. Note that with 20% thresholding, users are being altruistic. The absolute values of average turnaround times for the job trace for all the policies are shown in Table 5.4. We limit the graph to the main policies.

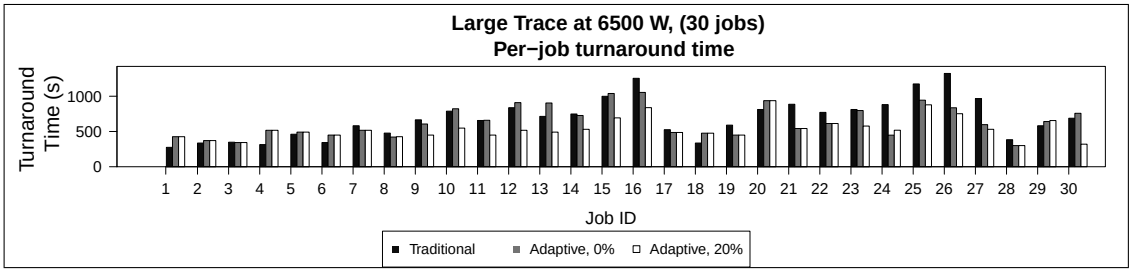


Figure 5.10: Benefits for Altruistic User Behavior

Policy	**Average Turnaround Time (s)**
Traditional	684
Naive	990
Adaptive, 0%	636
Adaptive, 10%	613
Adaptive, 20%	536
Adaptive, 30%	536

Table 5.4: Average Turnaround Times

Allocating power fairly with the *Adaptive* policy with a threshold of 0% can lead to better turnaround times for most users (17 out of 30), even when they choose not to be altruistic. The average turnaround time improved by 7.1% for the job queue when compared to the *Traditional* policy in this case. For the *Adaptive* policy with 10% and 20% thresholding, 18 and 22 jobs resulted in better turnaround times respectively, improving the average turnaround time of the job queue by 10.5% and 21.1% when compared to the *Traditional* policy. This demonstrates the benefits of altruistic behavior.

Altruistic users also get better turnaround times when compared to the *Adaptive* policy with a threshold of 0%. For example, when the threshold was set to 20%, 13 users got better turnaround times (up to 57.7% better, for job 30; and 12.8% on average) than what they did with a threshold of 0%. 14 users had the same turnaround time, and for 3 users, the turnaround times increased slightly (by less than 1.9%). The average turnaround time for the queue improved by 21.1%, as discussed previously. These benefits come from power-aware backfilling as well as hardware overprovisioning.

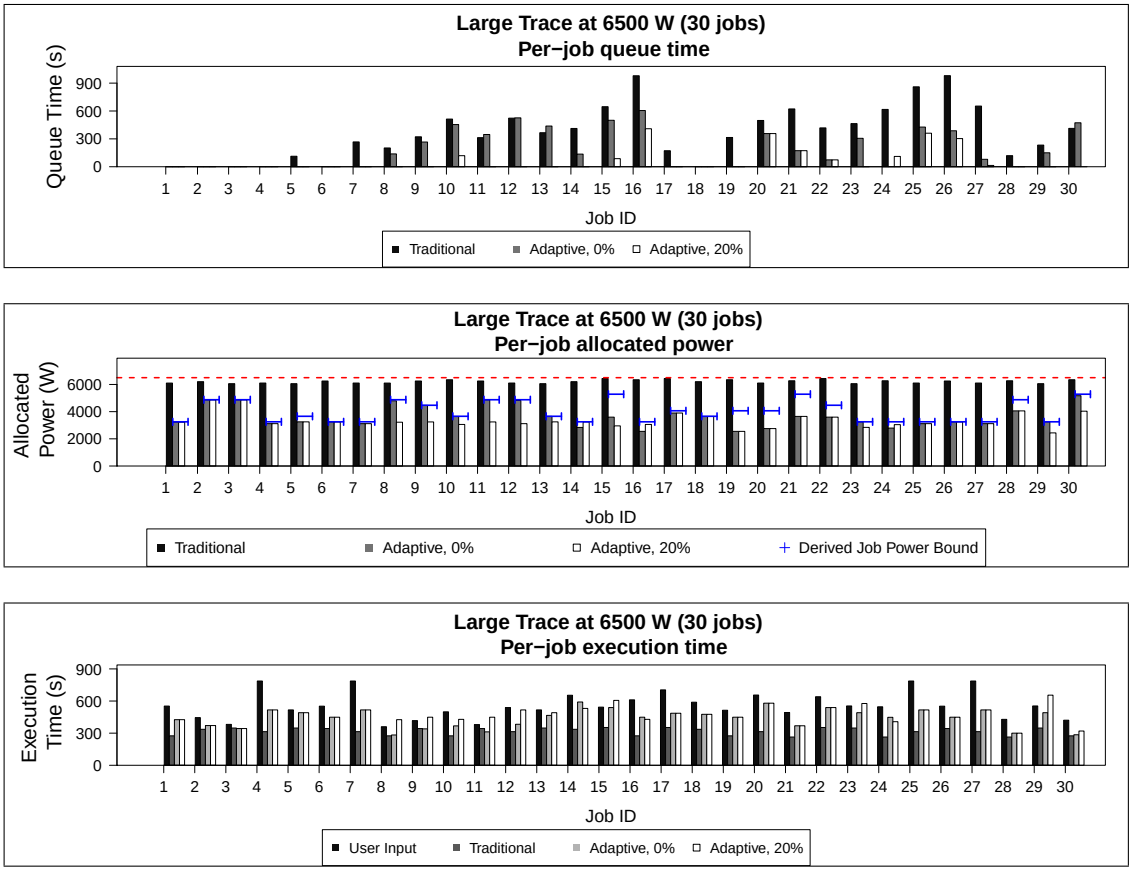


Figure 5.11: Detailed Results for the Large Trace at 6500 W

Figure 5.11 shows the breakdown of queue time, execution time, and allocated power for the jobs. The fair-share derived job-level power bound for the *Traditional* policy has also been shown. In some cases, such as for the first 5 jobs in the queue, the turnaround times with the *Adaptive* policy increased when compared to the *Traditional* policy. There are several reasons for this increase. One, all the jobs were allocated significantly more power with the *Traditional* policy (because there was no fair-share derived power bound, resulting in allocating the entire power budget to most jobs) and executed with Turbo Boost enabled (as no power capping was enforced), resulting in better execution times compared to the other policies. Also, depending on when a job arrived, it may have had zero wait time with the *Traditional* policy. In such a case, when the execution time increases, the turnaround time increases as well, because there

is no wait time to trade. Despite these issues, the *Adaptive* policy with 0% thresholding improved the turnaround times for 17 out of 30 jobs, which shows the benefits of fair sharing. For this example, the utilization of system power by both the *Traditional* and *Adaptive* policies was fairly high and there was not much leftover power, mostly because this was a tight global power bound (50% of peak) and the jobs were large-sized.

5.6.4 Power Utilization

We now analyze *Random Trace 2* in detail in a scenario at 14000 W, when the global power bound is 95% of peak power. We show that the *Adaptive* policy, even with a 0% threshold, improves system power utilization. Again, we take the conservative view by looking at a sparse job arrival rate in our dynamic job queue (short queue times in general), so we can test the limits of our *Adaptive* policy. With a sparse arrival rate, we expect significant amount of wasted power in this scenario. Figure 5.12 shows a relative timeline for the random trace with current system power data being reported when each job is scheduled (30 data points) for both the *Traditional* policy and the *Adaptive* policy with 0% thresholding. The job arrivals are marked at the bottom with points (blue). The dotted (red) line represents the *Adaptive* policy with 0% thresholding. As can be observed from the graph, the *Adaptive* policy allocates more power (resulting in better utilization), even when there are limited jobs to backfill. The exceptions occur when the *Adaptive* policy runs out of jobs to schedule, while the *Traditional* policy is still scheduling pending jobs (higher wait times).

This can be verified from Figure 5.13, which shows the corresponding per-job queue times, allocated power, and turnaround times for the *Traditional* policy, and for the *Adaptive* policy with 0% and 20% thresholding. The derived fair-share, job-level power bounds have been shown as well, which apply only to the *Adaptive* policy. The *Adaptive* policy cannot exceed the per-job power bound. The *Traditional* policy has job-level power allocation of 5% more power than that of the *Adaptive* policy in this scenario, as we are looking at 95% of peak power as the cluster power bound (14000 W).

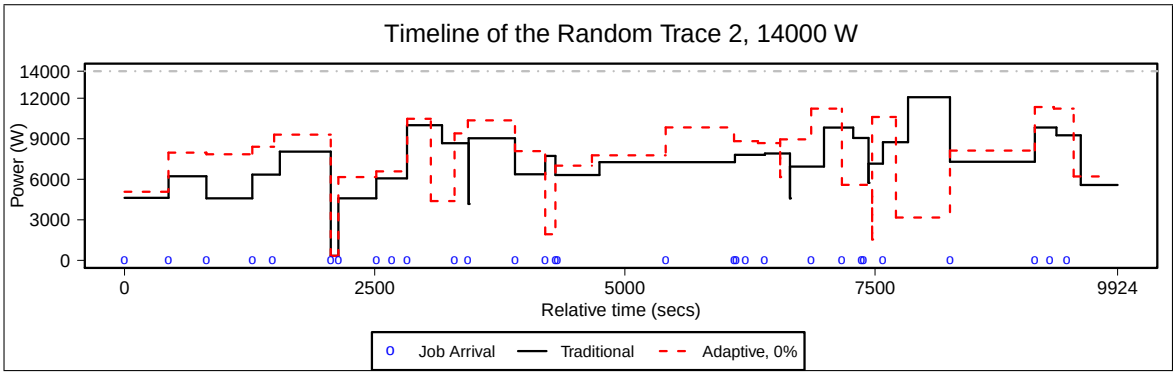


Figure 5.12: Timeline of Scheduling decisions

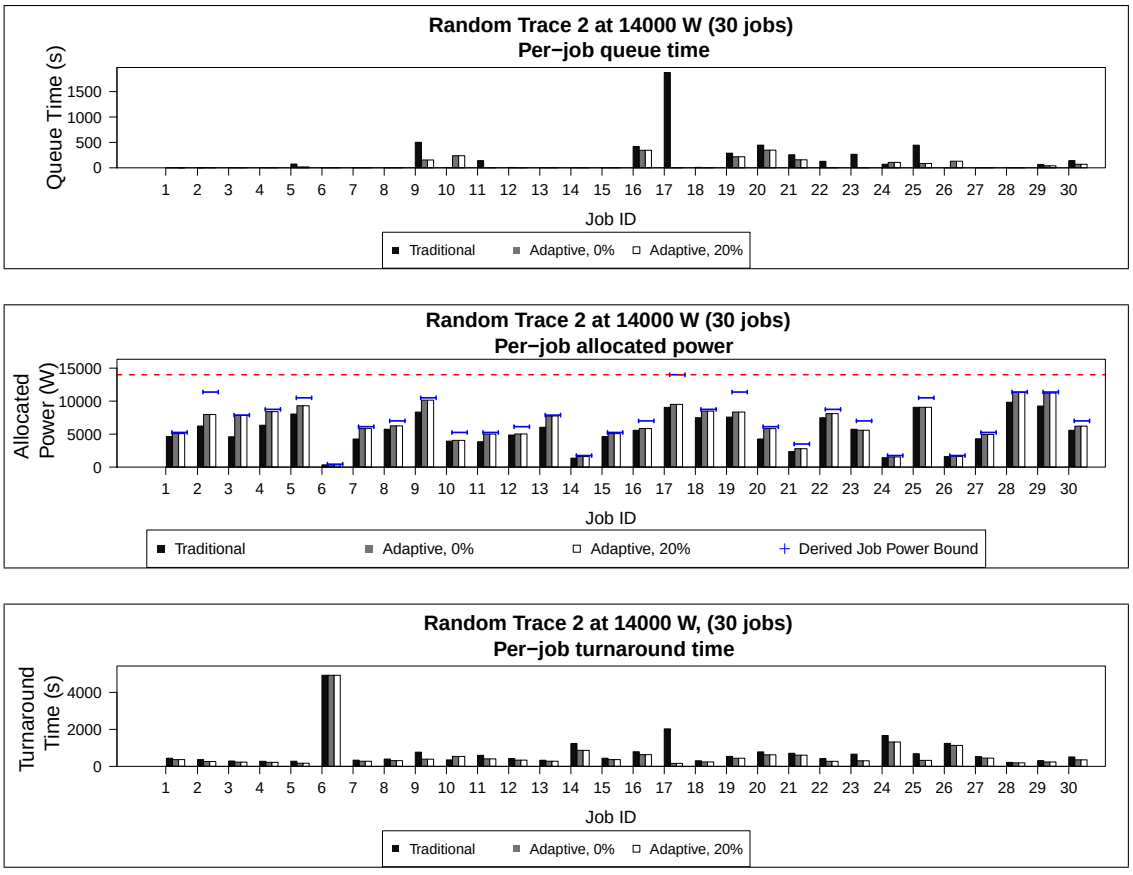


Figure 5.13: Utilizing Power Efficiently

For this trace, 14 of the 30 jobs did not have to wait in the queue at all (even with the *Traditional* policy). Even when there was no wait time to trade, the *Adaptive* policy improved the turnaround time for 28 of these 30 jobs (except Jobs 6 and 10). It tried to utilize all the power without exceeding the job-level power bound to improve application performance. The per-job power utilization improved by 17% on average with the *Adaptive* policy when compare to the *Traditional* policy. In some cases, such as for Job 3, the power utilization improved by 70%. The average improvement in turnaround time was 13%, and for 8 jobs in the trace this was by more than 2x. The *Traditional* policy fails to utilize the power well, and leads to larger turnaround times.

Another observation here is that at higher global power bounds, as in this example, benefits of the *Adaptive* policy with thresholding (see *Adaptive*, 20% for example) are limited. This is expected as the system is not significantly constrained on power anymore.

5.7 Summary

Our results have yielded three interesting lessons for power-aware scheduling. First, our encouraging results with the *Adaptive* policy show that jobs can significantly shorten their turnaround time with power-aware backfilling and hardware overprovisioning in a power-constrained system. In addition, by being altruistic, most users will benefit further in terms of turnaround time.

Second, naive overprovisioning, as implemented by the *Naive* policy, can lead to significantly *worse* turnaround times than the non fair-share policy (*Traditional*) for some job traces. An example of this was shown in the random job trace in Figure 5.6. Careful thought must be put into power-aware scheduling, or average turnaround time may actually *increase*.

Third, the node count requests made by jobs determine the best scheduling policy. The *Adaptive* policy is aimed at the most important jobs in a high-end cluster, which are those jobs that request the more resources. If most jobs are small, a simpler scheme such as the *Traditional* policy is often superior.

CHAPTER 6

RELATED WORK

This chapter presents existing work related to the research conducted in this dissertation. We broadly classify the related work in two categories: (1) energy, power and performance research, and (2) power-aware scheduling research.

6.1 Energy, Power and Performance Research

The focus of this dissertation is to optimize application performance as well as system throughput under a power constraint. We were the first to analyze application configurations and study hardware overprovisioning in the context of HPC systems. Before we discuss related research in power-constrained HPC, we present details about research in energy-efficient computing in HPC. This is mostly because there is a large body of work in energy-efficient HPC that is relevant and has inspired the general field of power-constrained supercomputing.

6.1.1 Energy-Efficient HPC

About a decade ago, one of the main areas in HPC research was energy efficiency, where the goal was to save energy given some allowable performance degradation. While this incurred a performance penalty, it was justified in terms of the cost reduction of energy utility bills. Researchers focused on using techniques such as dynamic voltage and frequency scaling, which enabled them to analyze the tradeoffs between energy and performance. Some notable work in this area included using DVFS-based algorithms such as thrifty barriers (Li et al., 2004), Jitter (Kappiah et al., 2005) and CPUMiser (Ge et al., 2007) to trade some performance for lower energy (Cameron et al., 2005; Hsu and Feng, 2005). Analytical models to predict and understand energy consumption in the context of scalability were also subsequently explored (Li and Martìnez, 2006). Springer

et al. (Springer et al., 2006) studied supercomputing under an energy bound. However, their study was limited to eight single-core nodes and thereby avoided any scaling issues due to limited on-node memory bandwidth. Additionally, static scheduling techniques with linear programming to find near-optimal energy savings without performance degradation (Rountree et al., 2007) were studied. Rountree et al. (Rountree et al., 2009) also developed the Adagio runtime system, which accomplished the goal of achieving energy savings without impacting performance at runtime. They applied DVFS on application ranks that were not on the critical path to obtain significant energy savings. Similar research on understanding the tradeoffs between energy savings and performance loss has been conducted in the real-time community. Using DVFS to save energy without incurring a performance loss for real-time systems has been studied (Ishihara and Yasuura, 1998; Saputra et al., 2002; Swaminathan and Chakrabarty, 2000, 2001). However, most of this research was limited to a single processor. Several other real-time approaches have also analyzed energy saving techniques (Zhang et al., 2002; Mochocki et al., 2002, 2005; Zhu et al., 2003; Moncusí et al., 2003).

6.1.2 Power-Constrained HPC

IBM Roadrunner, which was hosted at Los Alamos National Laboratory, was the first supercomputer to enter the petascale era in 2008. It had 130,464 cores, required 2.5 MW, and delivered about 444 megaflops per watt. By 2013, IBM Roadrunner was retired despite being the twenty-second fastest supercomputer in the world at the time (Sin, 2013). The key reason for this was its unreasonable power consumption, low energy efficiency and soaring utility bills. The Oakleaf-FX supercomputer in Tokyo, which was the twenty-first fastest at the time (one rank above IBM Roadrunner), was a 1.1 MW machine delivering 866 megaflops per watt. Moreover, Titan supercomputer, which was the world's fastest at the time, delivered 2143 megaflops per watt. As supercomputing researchers started pushing toward systems that could deliver 100 petaflops, power started becoming an expensive resource. In 2008, the U.S. Department of Energy released the Exascale Report, which identified power as one of the most critical challenges on the path to exascale, and set an ambitious target of achieving an

exaflop under 20 MW (Bergman et al., 2008; Sarkar et al., 2009; Ashby et al., 2010; InsideHPC, 2010).

In order to meet the DOE target of 20 MW, research and innovation needs to happen in all areas—hardware, system software, and applications. As a result, the focus of the community has now shifted toward improving performance under a power constraint, and specifically toward designing more efficient processor architectures. For example, manycore architectures such as Xeon Phis and hybrid architectures such as APUs are now in production (Intel, 2012; Li et al., 2014). There is additional research in designing power efficient interconnection networks (Saravanan et al., 2013; Miwa et al., 2014) and memory technologies (Hu et al., 2011). At the processor level, additional research in techniques such as Near-Threshold Voltage computing as well as power gating is being conducted (Kaul et al., 2012; Karnik et al., 2013), and design architectures such as the Runnamede system are being proposed (Carter et al., 2013). Accurate and efficient techniques are being developed for power measurement and control across different architectures (Intel, 2011; David et al., 2010; Wallace et al., 2013a,b; Yoshii et al., 2012; DeBonis et al., 2012; Laros et al., 2013). These include RAPL, EMON and PI, which were discussed in detail in Chapter 2.

Little systems and software research exists at the HPC system design level, however. Understanding performance under a power bound is a relatively new area in supercomputing research. Rountree et al. (Rountree et al., 2012) were the first to explore the idea using a hardware-enforced power-bounds in the HPC environment and to propose RAPL as an alternative to DVFS. They studied performance with and without power capping and analyzed power as well as performance variation across processors. Additionally, they demonstrated that variation in processor power translates to variation in processor performance under a power bound, which is an important concern for future power-limited systems.

Curtis-Maury studied application characteristics and configurations and introduced Dynamic Concurrency Throttling (DCT). DCT is a mechanism that controls the number of active threads in parallel regions of OpenMP applications to optimize for power and performance (Curtis-Maury et al., 2008a,b, 2006). Li et al. (Li et al.,

2012) further extended this work to hybrid programming models. While these teams looked at configurations for general power savings, they did not consider and study the impact of hardware-enforced or system-level power bounds on configurations. Femal and Freeh (Femal and Freeh, 2005) were the first to explore safe overprovisioning in power-limited scenarios. However, they developed network-centric models and focused on sets of CGI programs executed by web servers in data centers.

To the best of our knowledge, our work is the first to bring a combination of ideas from existing work together and to apply it to the supercomputing domain. We are the first to explore application configurations under several system-level power bounds, and the first to explore the benefits of hardware overprovisioned systems in HPC. Our results have opened a new direction for research. Sarood et al. (Sarood et al., 2013, 2014; Sarood, 2013) have studied overprovisioned systems from the perspective of malleable programming languages such as Charm++ and have extended our work to develop models for analyzing configuration spaces. Additionally, Marathe et al. (Marathe et al., 2015) have explored runtime systems based on this dissertation research. More power models are being actively developed (Storlie et al., 2014).

6.2 Power-Aware Scheduling Research in HPC

Backfilling is the current state-of-the-art for scheduling in HPC systems and has been studied widely in the literature (Feitelson et al., 2004; Davis and Burns, 2009; Shmueli and Feitelson, 2003; Jackson et al., 2001; Lifka, 1995; Mu'alem and Feitelson, 2001; Skovira et al., 1996; Srinivasan et al., 2002; Tsafrir et al., 2007; Feitelson et al., 1997; Feitelson and Rudolph, 1995). These studies have examined the advantages and limitations of various backfilling algorithms (conservative versus easy backfilling, lookahead-based backfilling, and selective reservation strategies).

Additionally, there has been research in the area of gang scheduling and coordinated scheduling, where the nodes are not a dedicated resource (Feitelson and Jette, 1997; Setia et al., 1999; Batat and Feitelson, 2000; Andrea Apraci-Dusseau, 1998). Coordinated scheduling improves system utilization in some cases, but is not considered

to be a feasible option in supercomputing due to the high context-switching and paging costs involved.

Early research in the domain of power-aware and energy-efficient resource managers for clusters involved identifying periods of low activity and powering down nodes when the workload could be served by fewer nodes from the cluster (Lawson and Smirni, 2005; Pinheiro et al., 2001). The disadvantage of such schemes was that bringing nodes back up had a significant overhead. DVFS-based algorithms were then applied in the scheduling domain to avoid this cost (Fan et al., 2007; Elnozahy et al., 2003; Mudge, 2001; Etinski et al., 2010a,b, 2011, 2012). Fan et al. (Fan et al., 2007) looked at power provisioning strategies in data centers and proposed a DVFS-based algorithm to reduce energy consumption for server systems. They analyzed power usage characteristics of large collections of servers and proposed a DVFS-based algorithm to reduce energy consumption when system utilization is low. Additionally, Zhou et al. (Zhou et al., 2014) explored knapsack-based scheduling algorithms with a focus on saving energy on BG/Q architectures.

While most of this work identified opportunities for reducing energy consumption, Etinski et al. (Etinski et al., 2010a,b, 2011, 2012) were the first to look at bounded slowdown of individual jobs and job scheduling under a power budget in the HPC domain. They proposed three policies over a span of three papers: Utilization-driven Power-Aware Scheduling (UPAS), Power-Budget Guided (PB-guided), and a Linear Programming based policy called `MaxJobPerf`. The UPAS policy assigns a CPU frequency to the job prior to its launch depending on current system utilization. The PB-guided policy predicts job-level power consumption and satisfies a power constraint while meeting a specified bounded slowdown condition. The `MaxJobPerf` policy extends the PB-guided policy to use a reservation condition that examines requested nodes and job *wait time limits*. Their `MaxJobPerf` policy does better than their PB-based policy due to decreased wait times. While their work is relevant, they did not optimize for performance under a power budget, instead, they looked at bounded slowdown. Also, they did not analyze application characteristics and study configurations. The technique used was DVFS, which is limited in comparison to power capping. Zhang et al. (Zhang

et al., 2014) further improved the work by Zhou et al. (Zhou et al., 2014) by using power capping and using leftover power to bring up more nodes when possible. They also explored policies similar to Etinski et al. (Etinski et al., 2010a,b, 2011, 2012) but used power capping instead. Similar to other existing research, they did not analyze application configurations. Ellsworth et al. (Ellsworth et al., 2015) have looked at dynamic reallocation of power to jobs with power capping, but have not focused on application configurations.

Recently, SLURM developers have looked at adding support for energy and power accounting (Georgiou et al., 2014). However, this work does not discuss any new scheduling policies. Bodas et al. (Bodas et al., 2014) explored a policy with dynamic power monitoring to schedule more jobs with stranded power. This work, however, has several limitations—the job queue is static and comprises three jobs, application performance is not clearly quantified, and overall job turnaround times are not discussed.

The research closest to ours which explored resource management on overprovisioned systems under a power constraint was conducted by Sarood et al. (Sarood, 2013; Sarood et al., 2014). They designed an Integer Linear Programming based technique to maximize throughput of data centers under a strict power budget, with the key goal of maximizing power-aware speedup of strongly-scaled applications. The proposed algorithm, however, is not fair-share and is not sufficiently practical for deployment on a real HPC cluster. This is because an NP-hard ILP formulation needs to be solved before every scheduling decision, for every job. This incurs a potentially quite high scheduling overhead and limits scalability greatly. Additionally, it is well known that ILP-based algorithms may lead to low resource (power and nodes) utilization as well as resource fragmentation (Feitelson, 1997; Frachtenberg and Feitelson, 2005; Feitelson and Rudolph, 1996; Feitelson et al., 2005). Their work attempts to address the issue of fragmentation by allowing jobs to be *malleable* (change node counts to grow/shrink at runtime), however, less than 1% of scientific HPC applications are expected to support malleability.

The ideas proposed in this dissertation for power-aware resource management are significantly different from any existing research in this area. First, we introduce power-aware backfilling and show that it can improve system turnaround times as well as power utilization when used in conjunction with hardware overprovisioning. Our proposed policies within RMAP are low-overhead (O(1) scheduling decision), which makes our approach scalable from the point of view of a real HPC production system. We also derive job-level power bounds in a fair manner, preventing starvation. Additionally, we show that users can benefit even more in terms of turnaround times by being altruistic. We analyze the benefits of trading execution time improvements obtained from overprovisioning for shorter queue wait times, and study the impact of altruistic user behavior on system throughput.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter we summarize the contributions of this dissertation and we present some open problems and ideas for future research. The focus of this dissertation was to improve application as well as system performance in power constrained supercomputing. We introduced and explored the potential of *hardware overprovisioning* in power-constrained supercomputing. Hardware overprovisioning is a novel approach for designing future systems and for improving application execution times under a power bound. We then analyzed the economic aspects of overprovisioned systems, and finally we designed and implemented system software in the form of a batch scheduler, RMAP, targeted at future overprovisioned systems. We summarize these contributions below.

7.1 Hardware Overprovisioning Summary

The central focus of this dissertation was to make the case for hardware overprovisioning in power-constrained supercomputing. As we venture into the exascale era, power is becoming an expensive and limiting component in supercomputing design. This research was motivated from the observation that current systems are worst-case power provisioned and lead to both under-utilization of procured power and limited performance. To successfully and efficiently utilize the available power, it is necessary to adapt to the workload characteristics. This calls for a reconfigurable approach to supercomputing design, and overprovisioning hardware with respect to power gives us the ability to do so.

The key idea is to buy more capacity than can be fully powered up under the specified site-level power budget, and to tune the system to respond to the needs of the applications. Such a system can provide limited power to a large number of nodes, peak

power to a smaller number of nodes, or use an alternative allocation in between for the same power constraint. In this work, we define a configuration as a combination of three values—number of nodes, number of cores per node, and power per node. The holy grail of an overprovisioned system is picking the best application-specific configuration under a power bound. For example, if an application is highly scalable, it might benefit from running more nodes at lower power per node. Or, if an application is memory intensive, it might be beneficial to run fewer cores per node.

Our results indicate that choosing the right configuration on an overprovisioned system can improve performance (execution time) under a power constraint by 62% (or 2.63x) when compared to the worst-case provisioned configuration, which always allocates as many nodes as possible with all the cores on a node at peak power to the application. The average performance improvement was 32% (1.8x). Overall, the benefits of overprovisioning were promising.

However, these benefits come with the investment cost associated with buying more capacity. To understand this further, we analyzed the economic viability of hardware overprovisioning and developed a model to determine the scenarios which make overprovisioning more practical and feasible. In Chapter 4 we showed how it is possible to reap the benefits of overprovisioning with the same cost budget as that of a worst-case provisioned system. Supercomputing system designers can use our model to determine whether or not overprovisioning is beneficial for their facility.

7.2 Power-Aware Resource Management Summary

In this dissertation, we first focused on understanding the costs and benefits of overprovisioning for a single application on a dedicated cluster in detail. For overprovisioning to be practical, it was important for us to look at scenarios where multiple users and applications were sharing an HPC system. We thus designed and implemented RMAP, which is a low-overhead, scalable resource manager targeted at future overprovisioned and power-constrained systems. Within RMAP, we designed a novel policy, which we referred to as the *Adaptive* policy. The goals for the

Adaptive policy were to improve application performance as well as to minimize wasted power (that is, increase the system power utilization), and to do so in a fair-share, scalable manner. We accomplished this by using a greedy technique called *power-aware backfilling*, which let us trade some of the performance benefits of overprovisioning for shorter queue wait times. Within RMAP, we also implemented two baseline policies, the *Traditional* policy and the *Naive* policy. The *Adaptive* policy resulted in up to 31% (19% on average) and 54% (36% on average) faster turnaround times for jobs, along with improving system power utilization. We also analyzed altruistic behavior of users with the *Adaptive* policy, where users were allowed to specify a slowdown that they can tolerate for their job. We showed that altruism can benefit both the user and the system—by providing faster turnaround times as well as better system throughput and better resource utilization.

7.3 Future Work

This dissertation opens up several avenues for power-aware research as we head toward exascale supercomputing. Some of these research directions are described below.

7.3.1 Impact of Manufacturing Variability

Over the past few decades, the semiconductor industry improved processor performance by doubling the transistor density per unit area on a chip every two years, following Moore's Law. Initially, the power usage of transistors decreased proportionally as they were made smaller (Dennard's scaling law), making Moore's Law successful. However, in the past few years, the industry has hit a wall as shrinking a transistor beyond a certain limit leads to increased static power loss, making it impossible to improve processor performance without increasing its power consumption. Shrinking transistors beyond a certain limit makes the lithography step in their fabrication challenging, as distortions can now occur in transistor channel lengths and film thickness. These distortions cause variations in CPU frequency, power and thermal design points (Harriott, 2001; Tschanz et al., 2002; Borkar et al., 2003; Esmaeilzadeh

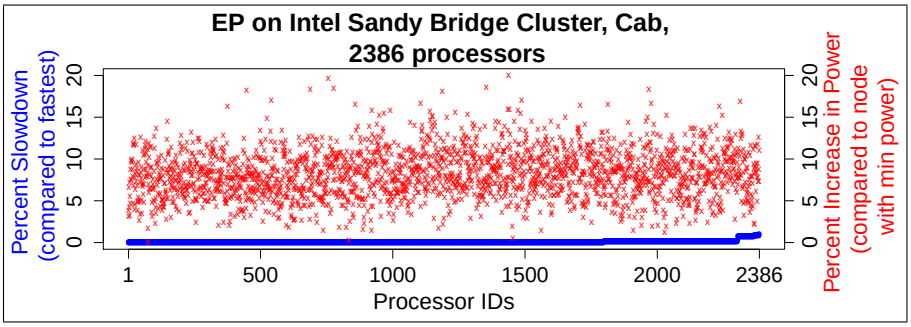


Figure 7.1: Performance and Power on Cab, LLNL

et al., 2013, 2011) for chips with the exact same micro-architecture. We refer to these variations in CPU power and performance owing to the fabrication process as *manufacturing variability*.

Several vendors employ *frequency binning* to address these variations—processors with the same performance characteristics are placed in the same bin (typically, HPC systems obtain all their processors from the same bin). As a result, current HPC systems are typically homogeneous with respect to performance, but experience *power inhomogeneity* (Rountree et al., 2012). Despite advances in fabrication technologies, it is expected that manufacturing variability will worsen in the future (Jilla, 2013; Samaan, 2004; Liang and Brooks, 2006). Similar variation is observed in DRAM manufacturing.

We conducted an initial study on power inhomogeneity on large-scale systems with different underlying micro-architectures. For our analysis, we first use the Cab system at LLNL, which is a 1,200 node, Intel Sandy Bridge system. Each node has 2 sockets with 8 cores per socket and 32 GB of memory. RAPL is used for power measurement. We use Class D of the NAS Embarrassingly Parallel (EP) benchmark to conduct single socket runs. We choose EP because it is a simple CPU-bound benchmark with no communication or synchronization overheads and exhibits no per-run noise. Also, most of its working set fits in cache, which allows us to analyze CPU power and performance in isolation. Our results are shown in Figure 7.1. The x-axis represents the sockets. The y-axes indicate the CPU power and performance data. The performance data in

each case is represented as a percentage slowdown compared to the fastest socket in the system, and the power data is reported as a percentage increase when compared to the most power-efficient socket. Additionally, processors are sorted by performance characteristics. No power caps have been enforced on the Cab system, and Turbo Boost (Intel, 2008) has been enabled. DRAM power readings were unavailable on Cab due to BIOS restrictions. We thus focus only on CPU power.

For the Cab system, we observe a maximum variation in power of 23%, and observe almost no performance variation. This is expected when processors belong to the same frequency bin. We ran similar experiments on two other systems with different underlying micro-architectures. These were the BlueGene/Q Vulcan system at LLNL and the AMD Teller system at SNL. The underlying architectures were IBM Power PC2 and AMD Piledriver with 16 and 4 compute cores, respectively. BG/Q EMON and PowerInsight were used to measure power on the Vulcan and Teller systems, respectively. On the Vulcan system, we observe a maximum variation in power of 11% across 1536 processors and do not observe any performance variation. On the Teller system, we obtain data for 64 processors and observe both power and performance variation. The maximum power variation is 21%, and the maximum performance variation is 17%. We believe this could be because of a different binning strategy that we are unaware of. Our key observation from these experiments is that manufacturing variability affects CPU power consumption significantly across different computing micro-architectures and does not necessarily correlate with performance, making it extremely difficult to make application-level power and performance predictions.

Power inhomogeneity translates to *performance inhomogeneity* under a power constraint. This is shown in Figure 7.2. When we enforce a socket level power cap of 65 W with RAPL on the Cab cluster, we observe a 10% variation in single-socket EP performance. Power capping was unavailable on Vulcan and Teller, so we could not conduct similar experiments there. This performance variation is expected to be much worse when we have multi-node applications with different memory characteristics executing on this cluster. A recent study has shown that performance among application ranks can vary by up to 64% on a 1920 Ivy-Bridge system when running the popular

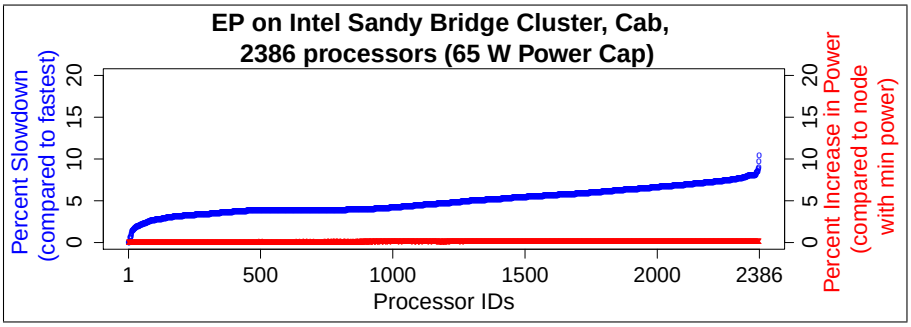


Figure 7.2: Variation in Performance Under a Power Constraint on Cab

matrix multiplication kernel, DGEMM (Inadomi et al., 2015). This raises several new concerns; for example, a perfectly load balanced application will now experience load imbalance, and application performance will depend significantly on the physical processors allocated to it during scheduling.

As we design future systems, especially overprovisioned systems, we will need to accommodate for these performance and power inhomogeneities and develop variation-aware power allocation algorithms to improve the critical path of applications. Also, mapping application ranks to physical nodes in order to improve overall performance will be important—this includes adjusting for manufacturing variability, network congestion and I/O bandwidth requirements when scheduling jobs.

7.3.2 Understanding Impact of Temperature on Power, Performance and Resilience

Another important area for future research is the impact of temperature on application power consumption, performance and resilience. This is especially important because the cost of cooling a HPC system is significant, and previous studies have shown that cooling energy can be as high as 50% (Sarood and Kale, 2011). There are two aspects to temperature—the ambient air temperature in a machine room where an HPC system resides, and the actual socket temperatures under varying workloads. Figure 7.3 shows the results of varying the ambient air temperature from 27 C to 95 C when running High-Performance Linpack on a single Ivy Bridge 24-core node. The x-axis shows the

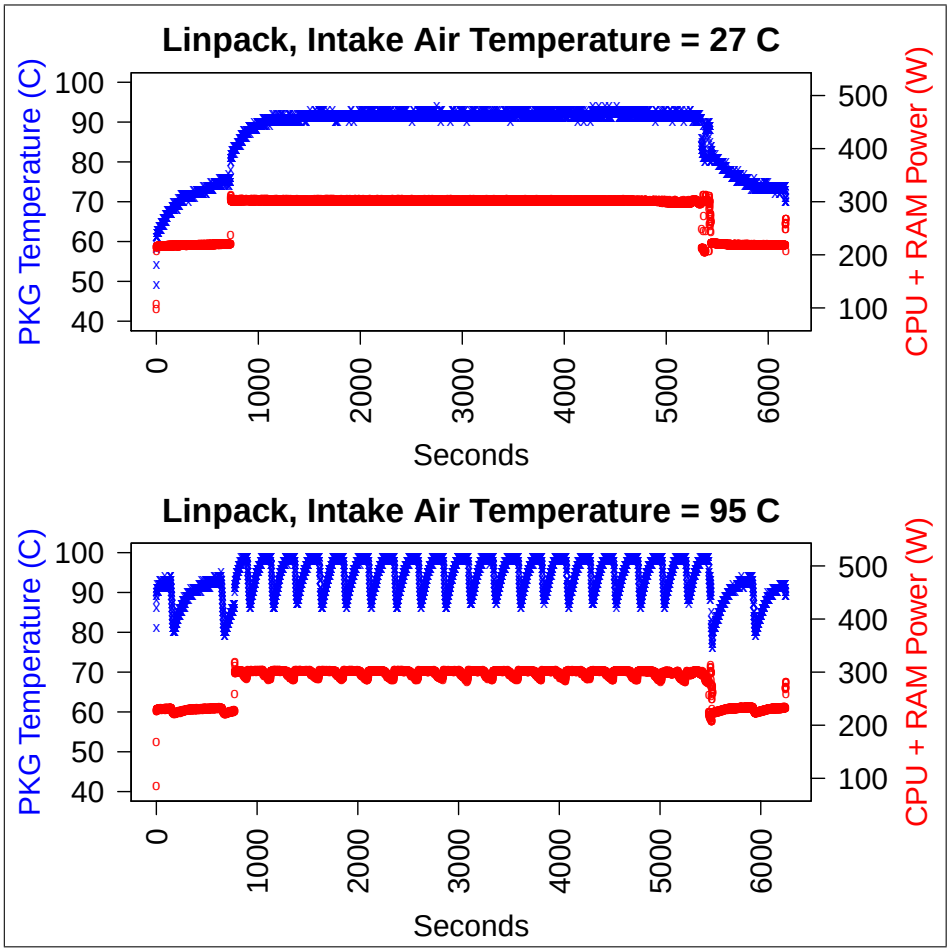


Figure 7.3: Impact on Ambient Air Temperature on High-Performance Linpack

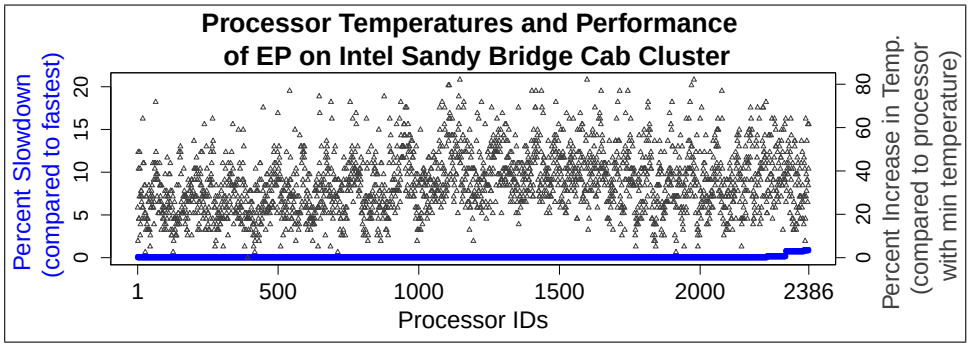


Figure 7.4: Performance and Temperature on Cab

progress of the application in seconds, and the y-axes depict the package temperature (blue) and node power consumption (CPU and DRAM, red) measured via RAPL. The ambient air temperature was controlled by pointing a space heater directly at the motherboard. The results seen here are interesting—varying the ambient temperature from 27 C to 95 C did not affect application power or performance significantly. The socket and core temperatures were affected by the ambient temperature. Every time the socket temperature crossed a certain threshold, the CPU fan on the motherboard went up to the highest speed possible so as to cool the chip down. This throttling also affected CPU power in a minor way, and the periodic CPU as well as fan throttling can be clearly observed from the graph. This result can directly impact the cost of the cooling system, and it is important to analyze the impact of such extreme variation on system reliability and component resilience.

Figure 7.4 depicts socket-level temperature data collected on the Cab system as described in Section 7.3.1. The y-axes here denote performance (blue) and socket temperature (gray), and the x-axes are the 2,386 sockets on Cab. The data has been sorted by performance. Similar to the previous set of experiments, we executed single-socket EP. We observed over 80% variation in package temperatures across the Cab cluster, and this did not seem to affect performance. It is unclear as to what the source of this variation is, and what its impact is. Addressing these two aforementioned issues about temperature is part of our future work.

7.3.3 Dynamic Reconfiguration in Overprovisioning

The research presented in this dissertation relies on choosing an application-specific configuration under a power constraint *statically*. That is, the assigned configuration does not change over the course of the application's execution. Most high-performance computing production codes, however, exhibit strong phase behavior during their execution. For example, most applications have initialization, computation and I/O or checkpointing phases, each of which have different power requirements. As part of future research, we plan on analyzing these application phases and redistributing power using *dynamic* reconfiguration with overprovisioning. This includes accommodating for phase

behavior within an application as well as synchronizing similar phases across currently running applications in order to improve throughput even further. Applications can thus be more cooperative and can donate/borrow power to/from other applications as well as use techniques such as dynamic concurrency throttling. This will require us to be able to explore the configuration space with low-overhead at runtime, and dynamically re-allocate configuration options (specifically cores per node and power) based on the characteristics of the application's phase. Dynamically changing node counts is supported by limited programming models such as Charm++, as discussed in Chapter 5.

7.3.4 Self-tuning Resource Managers

One of the key observations from prior research with RMAP is that the effectiveness of scheduling policies is workload dependent. Some policies work well for long-duration, large jobs. Others work well for short-duration, small jobs. This effect is more prominent in power-constrained scenarios. For example, as shown in Chapter 5, the Traditional policy works well for job queues that have jobs with small node and power requests. Similarly, the Adaptive policy with altruistic users and thresholding works well for queues with large resource requests. Currently, most administrators pick a single scheduling policy within the resource manager and apply this policy to the entire system at all times. A *self-tuning* resource manager has the ability to adapt to varying workloads by dynamically switching the scheduling policy based on some criteria (such as job size requirements, utilization thresholds, and varying system-level power bounds). We plan on addressing this problem by gathering relevant data and developing a model to determine which policies work best in which scenarios and then extending RMAP to adapt to these scenarios.

REFERENCES

AMD. AMD BIOS and Kernel Developers Guide for AMD Family 15h Models 30h-3Fh Processors, 2015. `http://support.amd.com/TechDocs/49125_15h_Models_30h-3Fh_BKDG.pdf`.

Andrea Apraci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *PhD Dissertation, University of California-Berkeley*, 1998.

Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, Tony Mezzacappa, Parviz Moin, Mike Norman, Robert Rosner, Vivek Sarkar, Andrew Siegel, Fred Streitz, Andy White, and Margaret Wright. The Opportunities and Challenges of Exascale Computing, 2010.

David H. Bailey. NASA Advanced Supercomputing Division, NAS Parallel Benchmark Suite v3.3, 2006.

Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A Regression-based Approach to Scalability Prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 368–377, 2008.

Anat Batat and Dror Feitelson. Gang Scheduling with Memory Considerations. In *International Symposium on Parallel and Distributed Processing Symposium*, pages 109–114, 2000.

Natalie Bates, Girish Ghatikar, Ghaleb Abdulla, Gregory A. Koenig, Sridutt Bhalachandra, Mehdi Sheikhalishahi, Tapasya Patki, Barry Rountree, and Stephen W. Poole. Electrical Grid and Supercomputing Centers: An Investigative Analysis of Emerging Opportunities and Challenges. *Informatik Spektrum*, 38(2): 111–127, 2015.

Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems, 2008.

Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. Simple Power-aware Scheduler to Limit Power Consumption by HPC System Within a Budget. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 21–30. IEEE Press, 2014.

Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter Variations and Impact on Circuits and Microarchitecture. In *Proceedings of the 40th annual Design Automation Conference*, pages 338–342, June 2003.

Kirk W. Cameron, Xizhou Feng, and Rong Ge. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *Supercomputing*, Seattle, Washington, November 2005.

Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua B. Fryman, Ivan Ganev, Roger A. Golliver, Rob C. Knauerhase, Richard Lethin, Benoît Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, pages 198–209, 2013.

Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-based Prediction. In *International Conference on Supercomputing*, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8.

Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1396–1410, October 2008a. ISSN 1045-9219.

Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores. In *International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2008b. ACM.

Howard David, Eugene Gorbatov, Ulf Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, pages 189–194, 2010.

Robert Davis and Alan Burns. A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems. In *Technical Report YCS-2009-443, Department of Computer Science, University of York*, 2009.

David DeBonis, James H Laros, and Kevin Pedretti. Qualification for PowerInsight Accuracy of Power Measurements. *Sandia National Laboratories Report*, 2012.

Department of Energy. The Magellan Report on Cloud Computing for Science, 2011.

Department of Energy. Preliminary Conceptual Design for an Exascale Computing Initiative, November 2014.

EE HPC WG. Power Measurement Methodology , 2014. `http://www.green500.org/sites/default/files/eehpcwg/EEHPCWG_PowerMeasurementMethodology.pdf`.

Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. POW: System-wide Dynamic Reallocation of Limited Power in HPC. In *High Performance Parallel and Distributed Computing (HPDC)*, June 2015.

Elmootazbellah Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-Efficient Server Clusters. In *Power-Aware Computer Systems*, volume 2325 of *Lecture Notes in Computer Science*, pages 179–197. Springer Berlin Heidelberg, 2003.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, 2011.

Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power Challenges May End the Multicore Era. *Commun. ACM*, 56 (2):93–102, February 2013.

Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Utilization Driven Power-aware Parallel Job Scheduling. *Computer Science - R&D*, 25(3-4):207–216, 2010a.

Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Optimizing Job Performance Under a Given Power Constraint in HPC Centers. In *Green Computing Conference*, pages 257–267, 2010b.

Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Linear Programming Based Parallel Job Scheduling for Power Constrained Systems. In *International Conference on High Performance Computing and Simulation*, pages 72–80, 2011.

Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Parallel Job Scheduling for Power Constrained HPC Systems. *Parallel Computing*, 38(12): 615–630, December 2012.

Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andr Barroso. Power Provisioning for a Warehouse-sized Computer. In *The 34th ACM International Symposium on Computer Architecture*, 2007.

Dror Feitelson. Job Scheduling in Multiprogrammed Parallel Systems, 1997. `http://www.cs.huji.ac.il/~feit/papers/SchedSurvey97TR.pdf`.

Dror Feitelson. Workload Modeling for Performance Evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 114–141, London, United Kingdom, 2002. Springer-Verlag.

Dror Feitelson and Morris Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer-Verlag LNCS, 1997.

Dror Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. *Job Scheduling Strategies for Parallel Processing*, pages 1–18, 1995.

Dror Feitelson and Larry Rudolph. Towards Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 1–26, London, UK, UK, 1996. Springer-Verlag.

Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, pages 1–34, 1997.

Dror Feitelson, Uwe Schwiegelshohn, and Larry Rudolph. Parallel Job Scheduling - A Status Report. In *In Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004.

Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel Job Scheduling: A Status Report. In *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2005.

Mark E. Femal and Vincent W. Freeh. Safe Overprovisioning: Using Power Limits to Increase Aggregate Throughput. In *International Conference on Power-Aware Computer Systems*, Dec 2005.

Eitan Frachtenberg and Dror Feitelson. Pitfalls in Parallel Job Scheduling Evaluation. In *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'05, pages 257–282, Berlin, Heidelberg, 2005. Springer-Verlag.

Hormozd Gahvari. *Improving the Performance and Scalability of Algebraic Multigrid Solvers Through Applied Performance Modeling*. PhD thesis, University of Illinois, Urbana-Champaign, 2014. `https://www.ideals.illinois.edu/handle/2142/50516`.

Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

Rong Ge, Xizhou Feng, Wu chun Feng, and Kirk W. Cameron. CPU Miser: A Performance-Directed, Run-Time System for Power-aware Clusters. In *International Conference on Parallel Processing*, Xi'An, China, 2007.

Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy Accounting and Control with SLURM Resource and Job Management System. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 96–118. Springer Berlin Heidelberg, 2014.

Lloyd R Harriott. Limits of lithography. *Proceedings of the IEEE*, 89(3):366–374, 2001.

Matti Heikkurinen, Sandra Cohen, Fotis Karagiannis, Kashif Iqbal, Sergio Andreozzi, and Michele Michelotto. Answering the Cost Assessment Scaling Challenge: Modelling the Annual Cost of European Computing Services for Research. *Journal of Grid Computing*, 13(1), 2015.

Chung-Hsing Hsu and Wu-Chun Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Supercomputing*, November 2005.

Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and EH-M Sha. Towards Energy Efficient Hybrid On-chip Scratch Pad Memory with Non-volatile Memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, and Martin Schulz. Analyzing and Mitigating the Impact of Manufacturing Variability in Power-Constrained Supercomputing. *Supercomputing (to appear)*, November 2015.

InsideHPC. Power Consumption is the Exascale Gorilla in the Room, 2010.

Intel. Intel Turbo Boost Technology 2.0, 2008. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`.

Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide, 2011.

Intel. Intel Many Integrated Core Architecture, 2012. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

Tohru Ishihara and Hiroto Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *International Symposium on Low power Electronics and Design*, pages 197–202, 1998.

David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–102. Springer Berlin Heidelberg, 2001.

Sudhakar Jilla. Minimizing The Effects of Manufacturing Variation During Physcial Layout. *Chip Design Magazine*, 2013. http://chipdesignmag.com/display.php?articleId=2437.

Nandani Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *Supercomputing*, November 2005.

Tanay Karnik, Mondira Pant, and Shekhar Borkar. Power Management and Delivery for High-Performance Microprocessors. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, 2013.

Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold Voltage (NTV) Design: Opportunities and Challenges. In *The 49th Annual Design Automation Conference 2012, DAC '12*, June 2012.

R. Kent Koeninger. The Ultra-Scalable HPTC Lustre Filesystem. *Cluster World*, 2003.

Jonathan Koomey, Brill Kenneth, Pitt Turner, John Stanley, and Bruce Taylor. A Simple Model for Determining True Total Cost of Ownership for Data Centers. *Uptime Institute White Paper, Version 2*, 2007.

Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating User-Level Fault Tolerance for MPI Applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

James H. Laros, Phil Pokorny, and David Debonis. PowerInsight - A commodity power measurement capability. In *IGCC'13*, pages 1–6, 2013.

Lawrence Livermore National Laboratory. SPhot–Monte Carlo Transport Code, 2001. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/sphot/.

Lawrence Livermore National Laboratory. The ASCI Purple Benchmark Codes, 2002. `http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html`.

Barry Lawson and Evgenia Smirni. Power-aware Resource Allocation in High-end Systems via Online Simulation. In *International onference on Supercomputing*, pages 229–238, June 2005.

Bo Li, Hung-Ching Chang, Shuaiwen Song, Chun-Yi Su, Timmy Meyer, John Mooring, and Kirk W Cameron. The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1448–1456. IEEE, 2014.

Dong Li, Bronis R. de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk Cameron. Strategies for Energy Efficient Resource Management of Hybrid Programming Models. *IEEE Transaction on Parallel and Distributed Systems*, 2012.

Jian Li and José F. Martìnez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *12th International Symposium on High-Performance Computer Architecture*, Austin, Texas, February 2006.

Jian Li, José F. Martínez, and Michael C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *10th International Symposium on High Performance Computer Architecture*, Madrid, Spain, February 2004.

Xiaoyao Liang and David Brooks. Mitigating the Impact of Process Variations on Processor Register Files and Execution Units. In *International Symposium on Microarchitecture*, pages 504–514, December 2006.

David Lifka. The ANL/IBM SP Scheduling System. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg, 1995.

Aniruddha Marathe, Peter Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. A Run-time System for Power-constrained HPC Applications. In *In International Supercomputing Conference (ISC)*, July 2015.

Sparsh Mittal. A Survey of Architectural Techniques for DRAM Power Management. *International Journal of High Performance Systems Architecture*, 4(2), 2012.

Shinobu Miwa, Sho Aita, and Hiroshi Nakamura. Performance Estimation of High Performance Computing Systems with Energy Efficient Ethernet Technology. *Computer Science - Research and Development*, 29:161–169, August 2014.

Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. A Realistic Variable Voltage Scheduling Model for Real-time Applications. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, 2002.

Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. Practical On-line DVS Scheduling for Fixed-Priority Real-Time Systems. In *11th IEEE Real Time and Embedded Technology and Applications Symposium*, 2005.

M. Angels Moncusí, Alex Arenas, and Jesus Labarta. Energy Aware EDF Scheduling in Distributed Hard Real Time Systems. In *Real-Time Systems Symposium*, December 2003.

Ahuva W. Mu'alem and Dror Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

Trevor Mudge. Power: A First-class Architectural Design Constraint. *IEEE Computer*, 34(4):52–58, 2001.

NPFA. National Electric Code, 2014. `http://www.nfpa.org/codes-and-standards/document-information-pages?mode=code&code=70`.

Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *International Conference on Supercomputing*, June 2013.

Tapasya Patki, Anjana Sasidharan, Matthias Maiterth, David Lowenthal, Barry Rountree, Martin Schulz, and Bronis de Supinski. Practical Resource Management in Power-Constrained, High Performance Computing. In *High Performance Parallel and Distributed Computing (HPDC)*, June 2015.

Antoine Petitet, Clint Whaley, Jack Dongarra, and Andy Cleary. High Performance Linpack, 2004. `http://www.netlib.org/benchmark/hpl/`.

Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Workshop on Compilers and Operating Systems for Low Power*, 2001.

Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2), 2012.

Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis de Supinski, and Martin Schulz. Bounding Energy Consumption in Large-Scale MPI Programs. In *Supercomputing*, November 2007.

Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent Freeh, and Tyler Bletch. Adagio: Making DVS Practical for Complex HPC Applications. In *International Conference on Supercomputing*, June 2009.

Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, and Martin Schulz. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *IPDPS Workshops (HPPAC)*, pages 947–953. IEEE Computer Society, 2012.

Samie B. Samaan. The Impact of Device Parameter Variations on the Frequency and Performance of VLSI Chips. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 343–346, Nov 2004.

H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, J.S. Hu, C.-H. Hsu, and U. Kremer. Energy-Conscious Compilation Based on Voltage Scaling. In *Joint Conference on Languages, Compilers and Tools for Embedded Systems*, 2002.

Karthikeyan P. Saravanan, Paul M. Carpenter, and Alex Ramirez. Power/Performance Evaluation of Energy Efficient Ethernet (EEE) for High Performance Computing. In *International Symposium on Performance Analysis of Systems and Software*, pages 205–214, April 2013.

Vivek Sarkar, William Harrod, and Allan Snavely. Software Challenges in Extreme Scale Systems. In *Journal of Physics, Conference Series 012045*, 2009.

Osman Sarood. *Optimizing Performance Under Thermal and Power Constraints for HPC Data Centers*. PhD thesis, University of Illinois, Urbana-Champaign, December 2013.

Osman Sarood and Laxmikant V. Kale. A 'Cool' Load Balancer for Parallel Applications. In *Supercomputing*, November 2011.

Osman Sarood, Akhil Langer, Laxmikant V. Kale, Barry Rountree, and Bronis R. de Supinski. Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems. In *IEEE International Conference on Cluster Computing*, pages 1–8, Sept 2013.

Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kale. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *Supercomputing*, November 2014.

Sanjeev Setia, Mark S. Squillante, and Vijay K. Naik. The Impact of Job Memory Requirements on Gang-scheduling Performance. *SIGMETRICS Perform. Eval. Rev.*, 26(4):30–39, March 1999.

Edi Shmueli and Dror Feitelson. Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 228–251. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20405-3.

Kathleen Shoga, Barry Rountree, and Martin Schulz. Whitelisting MSRs with msr-safe. *Third Workshop on Extreme-Scale Programming Tools, held with SC 14*, November 2014.

Gloria Sin. IBMs Roadrunner Supercomputer Hits Early Retirement due to Power Usage, 2013. `http://www.digitaltrends.com/computing/ibms-roadrunner-supercomputer-hits-early-retirement/`.

Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY LoadLeveler API Project. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer Berlin Heidelberg, 1996.

Rob Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In *ACM Symposium on Principles and Practice of Parallel Programming*, March 2006.

Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective Reservation Strategies for Backfill Job Scheduling. In *Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 55–71. Springer-Verlag, 2002.

Curtis Storlie, Joe Sexton, Scott Pakin, Michael Lang, Brian Reich, and William Rust. Modeling and Predicting Power Consumption of High Performance Computing Jobs. *arXiv preprint arXiv:1412.5247*, 2014.

Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 Supercomputer Sites, November 2014.

Vishnu Swaminathan and Krishnendu Chakrabarty. Investigating the Effect of Voltage-Switching on Low-Energy Task Scheduling in Hard Real-Time Systems. In *Asia South Pacific Design Automation Conference*, January 2001. `http://www.ee.duke.edu/~krish/pubs.html`.

Vishnu Swaminathan and Krshnendu Chakrabarty. Real-Time Task Scheduling for Energy-Aware Embedded Systems. In *IEEE Real-Time Systems Symposium*, November 2000. `http://www.ee.duke.edu/~krish/pubs.html`.

Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. To Move or Not to Move: The Economics of Cloud Computing. In *USENIX Conference on Hot Topics in Cloud Computing*, 2011.

Dan Tsafrir, Yoav Etsion, and Dror Feitelson. Backfilling using System-generated Predictions Rather than User Runtime Estimates. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):789–803, 2007.

James W. Tschanz, James T. Kao, Siva G. Narendra, Raj Nair, Dmitri A. Antoniadis, Anantha P. Chandrakasan, and Vivek De. Adaptive Body Bias for Reducing Impacts of Die-to-die and Within-die Parameter Variations on Microprocessor Frequency and Leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396–1402, Nov 2002.

Edward Walker. The Real Cost of a CPU Hour. *IEEE Computer*, 2009.

Sean Wallace, Venkatram Vishwanath, Susan Coghlan, Zhiling Lan, and Michael E Papka. Measuring power consumption on IBM Blue Gene/Q. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 853–859. IEEE, 2013a.

Sean Wallace, Venkatram Vishwanath, Susan Coghlan, John Tramm, Zhiling Lan, and ME Papkay. Application Power Profiling on IBM Blue Gene/Q. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013b.

Andy Yoo, Morris Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, 2003.

Kazutomo Yoshii, Kamil Iskra, Rinku Gupta, Pete Beckman, Venkatram Vishwanath, Chenjie Yu, and Susan Coghlan. Evaluating power-monitoring capabilities on IBM Blue Gene/P and Blue Gene/Q. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 36–44. IEEE, 2012.

Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Task Scheduling Voltage Selection for Energy Minimization. In *Proceedings of the 39th annual Design Automation Conference*, 2002.

Ziming Zhang, Michael Lang, Scott Pakin, and Song Fu. Trapped Capacity: Scheduling under a Power Cap to Maximize Machine-room Throughput. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 41–50. IEEE Press, 2014.

Zhou Zhou, Zhiling Lan, Wei Tang, and Narayan Desai. Reducing Energy Costs for IBM Blue Gene/P via Power-Aware Job Scheduling. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 96–115. Springer Berlin Heidelberg, 2014.

Dakai Zhu, Rami Melhem, and Bruce R. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2003.