SYSTEMWIDE POWER MANAGEMENT TARGETING EARLY HARDWARE OVERPROVISIONED HIGH PERFORMANCE COMPUTERS

by

DANIEL ELLSWORTH

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2017

DISSERTATION APPROVAL PAGE

Student: Daniel Ellsworth

Title: Systemwide Power Management Targeting Early Hardware Overprovisioned High Performance Computers

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Allen Malony	Chair
Martin Schulz	Core Member
Henry Childs	Core Member
Reza Rejaie	Core Member
Douglas Toomey	Institutional Representative

and

Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2017

DISSERTATION ABSTRACT

Daniel Ellsworth

Doctor of Philosophy

Department of Computer and Information Science

June 2017

Title: Systemwide Power Management Targeting Early Hardware Overprovisioned High Performance Computers

High performance computing (HPC) systems are an important enabling tool for modern scientific discovery. These large scale computing systems have, since the 1990s, been increasing built as clusters of commodity computers. The operational energy needs of these clusters has lead the HPC community to focus on energy efficient hardware and programming practices. One of the major side effects of introducing energy efficient hardware is variability in power consumption between components within the cluster. In practice, power variability at scale has resulted in poor power utilization and challenges for energy providers contracted to provide the needed power. Hardware overprovisioned HPC systems have been proposed to improve power utilization however production deployment of such a system would compound the challenge for energy providers.

This dissertation presents foundational work on HPC power scheduling, a technique that reduces the risks associated with operating hardware overprovisioned HPC systems. Power scheduling is formalized using the power scheduling invariant. Generalized application behavior, for applications running under a power cap, are experimentally studied. Study insights are used to develop a power scheduler

and a power capping cluster simulator. Comparative behavior of different power scheduling strategies as also examined.

Utilizing the power scheduling invariant, the safety of any power scheduler for deployment can be proven through analyzing scheduler's algorithm and mechanism. A general trend exists in power capped application performance that can be related to application progress, the underlying physics of the hardware, and expected runtime dilation. PowSched provides a proof by construction that power scheduling can be done safely and effectively without application specific models using a simple feedback mechanism. Experimentally, PowSched was shown to produce a 14% improvement in throughput compared to a fair distribution of power between cluster components. PowSim provides a proof by construction that the generalized effects on runtime can be efficiently simulated at scale, providing critical simulation infrastructure for researchers exploring power scheduling at scale. Using simulation, power scheduling strategies are studied and dynamic power scheduling appears to out perform static and reservation based techniques.

This dissertation includes previously published and unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Daniel Ellsworth

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2017, University of Oregon
Bachelor of Science, Computer Science, 2004, University of Oregon

AREAS OF SPECIAL INTEREST:

High Performance Computing
Programming Language Acquisition

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, University of Oregon, 2011-2017

Intern, Lawerence Livermore National Laboratory, 2014-2017

Software Developer, Electrical Geodesics Incorporated, 2013-2014

Owner, Octahedral Softworks, 2005-2011

Security Software Architect, Guident, 2010-2011

Trainer & Developer, Nortel Government Solutions, 2009-2010

Director of Application Architecture, Croix Connect, 2009

GRANTS, AWARDS AND HONORS:

Don Hubbard Scholarship Award, Computer and Information Science Department, University of Oregon, 2016

Best Graduate Teaching Fellow, Computer and Information Science Department, University of Oregon, 2013

PUBLICATIONS:

Ellsworth, D., Patki, T., Schulz, M., Rountree, B., & Malony, A. (2016, November). A Unified Platform for Exploring Power Management Strategies. *In Proceedings of the 4th International Workshop on Energy Efficient Supercomputing* (pp. 24-30). IEEE Press.

Ellsworth, D. (2016). *Dynamic Power Management For Hardware Over-provisioned Systems* (No. LLNL-CONF-705518). Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

Ellsworth, D., Patki, T., Perarnau, S., Seo, S., Amer, A., Zounmevo, J., ... & Schulz, M. (2016, May). Systemwide Power Management with Argo. *In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International* (pp. 1118-1121). IEEE.

Ellsworth, D. (2016). Topics Toward Automated Multiobjective HPC System Management.

Ellsworth, D. A., Malony, A. D., Rountree, B., & Schulz, M. (2015, November). Dynamic Power Sharing for Higher Job Throughput. *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (p. 80). ACM.

Ellsworth, D. A., Malony, A. D., Rountree, B., & Schulz, M. (2015, June). Pow: System-wide Dynamic Reallocation of Limited Power in HPC. *In Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (pp. 145-148). ACM.

Ellsworth, D. (2013). Improving Dynamic Invariant Saliency with Static Dataflow Analysis.

ACKNOWLEDGEMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

This work was supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

This chapter contains ideas and themes that have been previously published in D. Ellsworth (2016); D. Ellsworth, Patki, Perarnau, et al. (2016); D. Ellsworth, Patki, Schulz, Rountree, and Malony (2016); D. A. Ellsworth et al. (2015a); D. A. Ellsworth, Malony, Rountree, and Schulz (2015b). The narrative as presented in this chapter is my original work derived from discussions with the co-authors of the previously published works regarding the motivation for the power capping when the work was initiated.

High performance computing (HPC) platforms are a critical enabling technology for modern science and engineering. Both of these disciplines make extensive use of mathematical models to analyze observations and make predictions. Executing the models is computationally intense due to the mathematics and amount of data used to represent a physical system under study. In the case of simulations, these computations must be done repeatedly to accumulate effects as the system understudy evolves through simulated time. The computational power provided by HPC platforms allow cost prohibitive, impractical, or technologically impossible hypothises to be explored via simulation.

Early HPC platforms like the Cray-1, capable of 160 million (10^6) floating point operations per second (FLOPS)[1](*Cray-1*, n.d.), were engineering marvels. Efficent techniques for getting higher performance from shared memory computing

[1]Computational power, in the context of HPC, is measured in floating point operations per second. A floating point number is a rounded representation of real number. An operation is something like adding or multiplying. A person capable of multiplying two floating point numbers per second would be a computer operating at 1 FLOPS.

were pioneered using early HPC machines. Vector processing and other technologies common in modern consumer devices were pioneered in the HPC setting. The market for such powerful shared memory systems was very small since few organizations can afford the expense of these machines. Scaling (i.e., increasing the computational power) for early HPC systems required new engineering effort and completely replacing the machine.

Starting in the late 90s, the majority of HPC machine designs moved to clusters. Rather than purchase an extremely expensive single computer, a cluster is made by connecting many individual computers. Cluster computing allows system owners to take advantage of the economies of scale when pricing and repairing a machine. The shift brought new challenges as coordination across nodes became a key concern (e.g., interconnection technologies, distributed programming abstractions, etc.). As the community has improved techniques to coordinate across cluster nodes, HPC scaling has become a problem primarily of identifying how many nodes an organization can afford to include in the cluster. The top cluster based HPC system in 2016, Sunway TaihuLight, is theoretically able to complete 125 quintillion (10^{15}) FLOPS and contains over 10 million processor cores(*November 2016 — TOP500 Supercomputer Sites*, n.d.).

Around 2010, the US Department of Energy (DOE) became interested in the procurement of an HPC system capable of roughly 1 sextillion (10^{18}) FLOPS, an exascale system. Table 1, common in presentations and reports produced by the HPC community, compares 2010 petascale system attributes to the expected first generation exascale system attributes. Of particular concern for horizontal scaling was the power requirement, as most system attributes would increase by two or three orders of magnitude, but power was only to increase by a single order

Attribute	2010	2018	Increase
System Peak	2 PF	1 EF	$\mathcal{O}(1000)$
Power	6 MW	20 MW	$\mathcal{O}(10)$
Memory	0.3 PB	32-64 PB	$\mathcal{O}(100)$
Node Performance	125 GF	1-10 TF	$\mathcal{O}(10) - \mathcal{O}(100)$
Node Memory BW	25 GB/s	.4-4 TB/s	$\mathcal{O}(100)$
Node Concurrency	12	1-10k	$\mathcal{O}(100) - \mathcal{O}(1000)$
Total Concurrency	225 000	10^9	$\mathcal{O}(10000)$
Total Node Interconnect BW	1.5 GB/s	200 GB/s	$\mathcal{O}(100)$
MTTI	days	1 day	$-\mathcal{O}(10)$

Table 1. 2010 estimate of first generation exascale system properties compared to 2010 petascale systems. (GF: giga (10^9) flops; TF: tera (10^12) flops; PF: peta (10^15) flops: EF: exa (10^18) flops; BW: bandwidth; MW: megawatt; GB: giga byte; TB: tera byte; PB: peta byte; MTTI: mean time to interruption)

of magnitude[2]. The consensus from the community was that the horizontal scaling approaches used for petascale systems would not be able to achieve the DOE objectives and energy efficiency emerged as a top problem for the HPC community.

Many of the techniques for increasing computational energy efficiency make power consumption variable. Different operations the computer can perform come at different energy costs based on the complexity of the operation and the speed at which the operation is performed. Energy efficiency is usually achieved by only performing the needed operations at the minimum speed required to avoid delays. Different programs and phases within the same program will use differing mixes

[2]The 20 megawatt power consumption target for first generation exascale systems is likely motivated by cost. HPC systems are extremely expensive assets for the owning organization. Procurement and installation of hardware are multimillion dollar capital investments. Ongoing operational expenses from HPC systems are also significant. US wholesale power rates in 2017 are roughly estimated at one million dollars per megawatt of power purchased. The top US computer in 2016 was Titan(*November 2016 — TOP500 Supercomputer Sites*, n.d.). Titan had an estimated power cost of around 8 million dollars per year and a theoretical peak performance of roughly 27 petaflops (10^{15} FLOPS). Observed performance on Titan was around 17 petaflops. Purely horizontal scaling of Titan to support 1 exaflop at theoretical peak would result in roughly 30 million dollars per year in power costs.

of operations, resulting in different energy needs over time. The power variability introduced by energy efficiency will be discussed more in Chapters II and III.

Traditionally, power for HPC systems is purchased and power distribution infrastructure built based on the theoretical maximum power consumption of the system, also known as worst case power provisioning. Figure 1 shows the power consumption of a relatively energy efficient HPC system, Vulcan[3]. Vulcan can consume roughly 2.4 megawatts of power at peak, resulting in roughly 2.4 megawatts of power being procured for the system. Vulcan is observed to consume almost 2.4 megawatts on occasion, however, usually consumes about 1.5 megawatts. This means that roughly 30% of the procured power goes unused during much of Vulcan's lifetime. Scaling up to a 20 megawatt exascale system, 6 megawatts would be expected to go unused most of the time. Since the energy contracts in some centers are based on a committed rate of power consumption, 30% of the fiscal power budget may be spent on energy that is never used.

Hardware overprovisioning has been recently proposed (Rountree, Ahn, de Supinski, Lowenthal, & Schulz, 2012) and aims to convert the power savings from energy efficient computation into useful computation. A hardware overprovioned HPC system will contain more hardware than can be run at peak consumption and yet will be controlled to stay within the procured power. In the case of Vulcan that would mean the ability to add 30% more nodes, making approximately 10 thousand more cores available for users. The additional nodes can then be used to run larger scale computations or run additional computations concurrently. Alternatively, the owning organization could leave Vulcan with the same node count but reduce

[3]Vulcan is an IBM BG/Q system hosted at Lawerence Livermore National Laboratory. Vulcan was initially deployed in 2012 and was ranked 21 on the Top500 list in 2016(*Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect — TOP500 Supercomputer Sites*, n.d.).

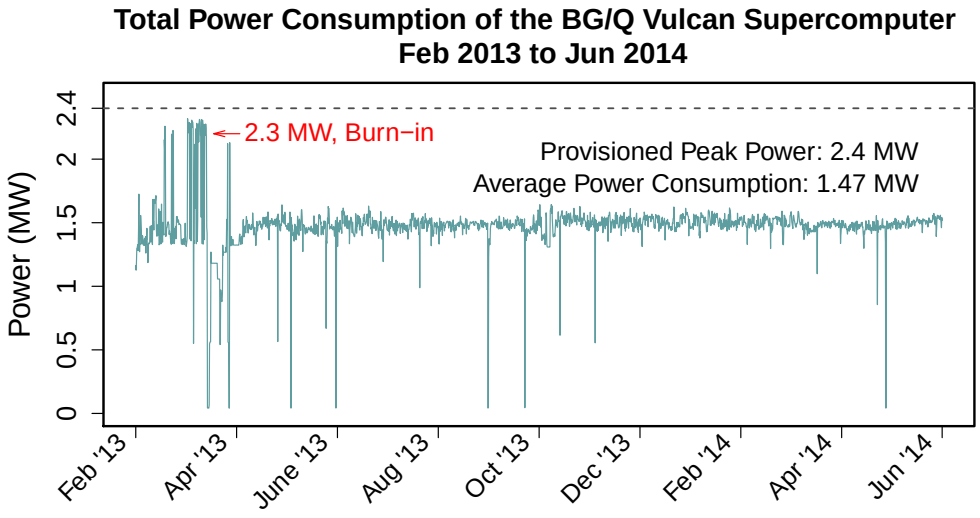


Figure 1. A plot of the power consumption of Vulcan over approximately 16 months. Reproduced from Patki et al. (2015).

the power procurement to something closer to 1.5 megawatts[4]. Applying either of these changes would result in a system that could significantly exceed the procured power, introducing risk to the owning organization.

A new and significant resource management challenge is introduced by hardware overprovisioning. Power generation and consumption must remain in balance across the larger power grid to avoid surges and brownouts that may damage the power infrastructure. Consuming significantly more energy than contracted for may cause the energy provider's generation capacity to be exceeded or have severe financial penalties. Supposing the energy provider can generate sufficient power, the distribution infrastructure may not be able to support the higher rate and will be damaged. Simply adding additional hardware without changing the power infrastructure is therefore unsafe. To safely deploy hardware

[4]At an estimated 1 million dollars per megawatt, this would save nearly 1 million dollars per year on power that is never used.

overprovisioned HPC systems, a mechanism to contain HPC system power consumption is required.

This dissertation presents research on power scheduling, a mechanism to contain systemwide power consumption, done to support first generation hardware overprovisioned HPC systems. Energy and power in HPC have only recently become topics of interest to the research community and power control mechanisms like RAPL[5] have only recently become available for experimentation. The broader research question underlying this work is: how can power utilization be increased in large scale computing systems when there exists a fixed upper limit on available power. Contributions made in this dissertation may be relevant outside of the HPC context (e.g., datacenters). An exploration of these other contexts is out of scope for this work.

Chapter II discusses related work and introduces the power scheduling invariant to partion solutions by the provability of power schedule enforcement. Chapter III contributes research filling a foundational gap in the literature regarding HPC application power consumption behavior. Chapter IV contributes a novel power scheduler, experimentally shown to have good performance, that requires no apriori information regarding applications running on the cluster. Chapter V contributes the only simulator currently in the HPC community that is able to simulate the general effects of arbitrary power bounds on application progress at scale without extensive trace based modeling. Chapter VI contributes

[5]Running average power limit (RAPL) is an Intel technology that allows software to set a target energy consumption, over a sliding window, that is enforced by hardware. Similar technology, with different names, are available from other vendors. Experiments reported in this dissertation were conducted only on Intel systems due to experimental platform availability but the results are expected to be portable to other architectures. Confirmation of result portability is out of scope for this dissertation.

	1	2	3	4	5	6	7
D. A. Ellsworth et al. (2015a)	X	X		X			X
D. A. Ellsworth et al. (2015b)	X	X	X	X	X		X
D. Ellsworth (2016)	X	X					X
D. Ellsworth, Patki, Perarnau, et al. (2016)	X	X		X			X
D. Ellsworth, Patki, Schulz, et al. (2016)	X	X		X		X	X

Table 2. Mapping between prior work and dissertation chapters containing shared material.

an approach to comparing power scheduling solutions to potentially be deployed on a hardware overprovisioned HPC system.

Previously Published Work

Much of the core material presented in this dissertation has appeared in other venues, however, the presentation has been altered for use in the context of this dissertation. Table 2 summarizes the published works and chapters in which previously published material appears.

CHAPTER II

BACKGROUND

This chapter contains ideas and themes that have been previously published in D. Ellsworth (2016); D. Ellsworth, Patki, Perarnau, et al. (2016); D. Ellsworth, Patki, Schulz, et al. (2016); D. A. Ellsworth et al. (2015a, 2015b). The narrative as presented in this chapter is my original work summarizing existing research in the community. The power scheduling invariant is my original work, however the language used to express the invariant has been tuned through conversations with my co-authors in the previously published work.

Hardware overprovisioned systems have only recently been proposed in the HPC literature. In "Beyond DVFS: A First Look at Performance Under a Hardware-Enforced Power Bound", Rountree et al. (2012) observe that modern energy efficient processors are overprovisioned with respect to power. These processors have more circuitry than can be run at the highest clock frequency while remaining within the processor's thermal design power (TDP). Owners of these processors are buying the flexibility to achieve higher concurrency through activating more gates at lower frequency or fewer gates at a higher frequency. Energy efficient performance is achieved, in part, through more effective utilization of power in an overprovisioned processor. Rountree et al. (2012) suggest that a similar approach may be applicable to HPC system design.

Classically, HPC nodes and power are provisioned with the expectation that all nodes will operate at maximum power consumption at all times. In practice, nodes rarely consume the maximum amount of power resulting in unutilized, yet available power. A hardware overprovisioned system provides more nodes than

the provisioned power can support at maximum consumption. The *extra* nodes in the cluster are available to perform useful computation when sufficient unutilized power is available. System owners of hardware overprovisioned systems have the flexibility to run all nodes at low power or a subset of nodes at high power based on the needs of the running workloads. The hardware overprovisioned cluster, even if built from homogenous nodes, is a dynamically reconfigurable and heterogenous execution environment in the presence of variable power allocations. Both work and power must be scheduled across a hardware overprovisioned cluster to maximize system energy efficiency and performance.

Existing work on energy efficiency and power scheduling for HPC systems is briefly surveyed in this chapter. First work on energy efficiency in the HPC community will be briefly presented and categorized as using power reduction or runtime reduction as the primary mechanism to achieve effeciency. Energy efficiency work focuses on reducing the amount of energy expended to complete a given computation and is necessary to utilize hardware overprovisioned systems. Next, work on HPC power scheduling is presented and categorized based on enforcement and when power allocation decisions are made. Power scheduling is the major focus of this thesis and existing work will be organized based on how the power bounding is enforced and when power allocations are adjusted.

Energy Efficiency

Power and energy are strongly related concepts that are often used interchangeably in normal conversation, leading to confusion when these concepts must be discussed separately[1]. The standard unit of power, P, is the Watt and the standard unit of energy, E, is the Joule. Power is the rate at which energy is

[1]For example, the residential "power bill" should be referred to as an energy bill since the utility is charging based on energy used rather than power. 1 kilowatt hour, kWh, is 3600 joules.

transfered, leading to the following standard formulas for conversion:

$$P = \frac{E}{t} \tag{2.1}$$

$$E = Pt \tag{2.2}$$

Equation 2.1 states the power, P, in watts is equal to the energy, E, in joules divided by the time, t, in seconds. Similarly, equation 2.2 states that the energy, E, is equal to the power, P, times the duration, t. The values of power and energy are frequently related to one another by these formulas, which assume constant power over the interval.

Energy efficient computation techniques are important supporting technology for hardware overprovisioned systems since these provide opportunities to shift power between jobs. The objective of energy efficient computation techniques is to reduce the total energy required to complete a specific computation. There are two, nonexclusive, basic ways to accomplish energy reduction:

Power reduction Reduce the needed power without significantly increasing the time.

Runtime reduction Reduce the computation runtime without significantly increasing the needed power.

In some cases a significant power reduction and extended runtime may result in higher energy efficiency, however, extending runtimes are highly undesirable in HPC. Table 3 summarizes the works to be cited in this section.

Using either power reduction or runtime reduction, the energy per computation is reduced. Energy efficient techniques that make use of dynamic power gating (DPG) or dynamic voltage and frequency scaling (DVFS) are

Research	Reduces	Mechanism
Bambagini, Bertogna, Marinoni, and Buttazzo (2013)	Power	DPG, DVFS
Rountree et al. (2009)	Power	DVFS
Tiwari, Laurenzano, Peraza, Carrington, and Snavely (2012)	Power	DVFS
Marathe et al. (2015)	Power	DVFS, RAPL
Patki, Lowenthal, Rountree, Schulz, and de Supinski (2013)	Power	Indirect, RAPL
Zhang and Hoffmann (2016)	Power	Indirect, RAPL
Hoffmann, Maggio, Santambrogio, Leva, and Agarwal (2013)	Power	Indirect
Enos et al. (2010)	Runtime	GPGPU
Betkaoui, Thomas, and Luk (2010)	Runtime	FPGA

Table 3. Table mapping energy reduction research to the primary knob used (power or runtime reduction) and mechanism used to actuate that knob.

generally making use of the first strategy. Due to some properties of the computation, the processor can be slowed (i.e., clock frequency reduced) or halted, reducing power, without impacting perceived performance. Energy efficient techniques that make use of general purpose graphics processing unit (GPGPU) computing are generally making use of the second strategy. GPGPUs tend to require more energy per unit time than the processor however an overall energy savings is realized due to much shorter runtimes when the parallelism can be effectively utilized, see Figure 2.

The HPC community frequently uses floating point operations per second per watt, FLOPS/W, as the metric of choice when discussing both HPC power and energy. The metric represents the energy per floating point operation[2]. Supposing the number of floating point operations was known in advance, a total energy cost

[2]Since both FLOPS and watts are rates, the time cancels out making the metric floating point operations per joule, FLOP/J. Like a power bill's use of kWh, leaving the watts in the FLOPS/W measure is misleading since it does not represent a rate.

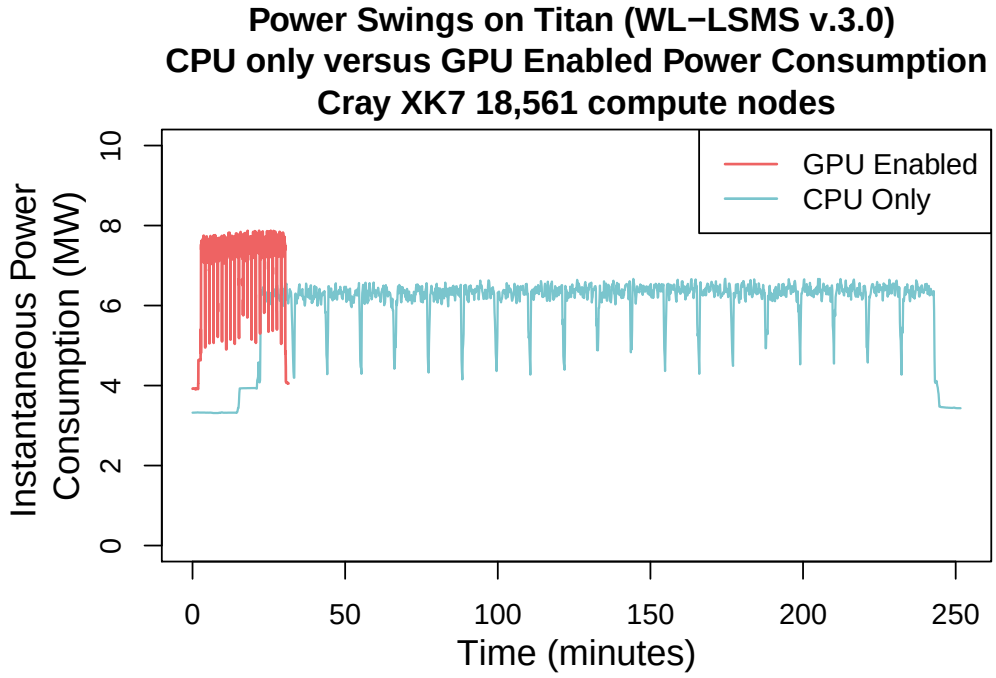


Figure 2. The same computation with and without GPU support. The GPU execution is significantly more energy efficient even though more power is required. Reproduced from Patki et al. (2016).

for a computation could be computed. However, the runtime and power would still be unrecoverable. Knowing the number of parallel execution units does not resolve the problem of recovering runtime since any computation that is not embarrassingly parallel will involve some runtime in which some execution units are unused. Given the loss of information regarding the computation, FLOPS/W is not useful for comparing application energy performance and is better applied only to comparing hardware floating point energy costs.

Power Reduction. As a by-product of their operation, processors convert electrical energy into thermal energy. A major contributor to energy consumption and heat generation within a computer is transistor switching. Each transistor converts a small amount of energy to heat when changing state and that amount depends, in part, on the voltage and switching frequency. Additionally, some energy is lost due to leak current. A model to describe the power loss due to switching is $W = \eta CV^2 f$(De Vogeleer, Memmi, Jouvelot, & Coelho, 2014); the watts W lost are directly related to the square of the voltage V and the frequency f of switching. Since voltage and frequency must be increased together, resulting in a nonlinear relationship between watts and instruction execution speed dominated by the V^2 term. Power lost to leak current is also related to the voltage.

Two major, processor internal, approaches to power management are power gating and DVFS. Power gating turns off circuits not currently being used within the processor. For the effected circuits, power gating avoids both leak and switching losses. DVFS changes the voltage and frequency of operation within the processor. For the effected circuits, DVFS reduces the leak and switching losses.

Three mechanisms exposed by modern processors for power control are dynamic power gating (DPG), dynamic voltage and frequency scaling (DVFS),

and capping[3](e.g., Intel's Running Average Power Limit (RAPL)). DPG is used to signal that the processor should switch to an ultra low power mode, turning off most of the processor's circuitry. Reasoning about the performance impact of DPG is straight forward since the computation is effectively halted on the processor. DVFS is used to signal that the processor should change to a specific clock frequency and voltage. Reasoning about the performance impact of DVFS is more complex since runtime may or may not increase[4]. RAPL is an Intel feature used to signal that the processor should respect a power cap, which the processor does primarily through DVFS but may involve other internal mechanisms. Reasoning about the performance impact of a RAPL power cap is non-trivial since the processor frequency is variable based on the concurrent work executing within the processor.

Power reduction techniques are the most common energy efficiency techniques in the HPC literature. The power reduction work is organized in this section based on the primary mechanism used to achieve power reduction.

DPG Impact. DPG is a useful technique for embedded realtime workloads, but is less useful for HPC workloads. The embedded realtime workload is typified by a time driven sense-compute-act cycle[5]. Bambagini et al. (2013) present an effective scheduler for realtime systems with limited preemption using both DPG and DVFS that increases battery life. The effectiveness of the technique

[3]Different vendors use different names for their power capping technology. Intel's Running Average Power Limit (RAPL) is the only mechanism we have seen used in the power scheduling literature.

[4]Reducing frequency will proportionally extend hardware instruction execution time. The complexity in estimating effect comes from the interaction with the memory subsystem. If the rate of progress was bound by memory latency, reducing the clock speed will have limited effect on runtime.

[5]Periodically, the device reads a sensor value, performs a computation based on the sensed value, and performs some action.

relies on the compute workload being predictably periodic with regular not-later-than deadlines to complete the computation. While a specific HPC job may internally have periodic behavior during execution, HPC jobs are typically a single computation with all input data available at launch. Effectively halting a processor participating in a job is risky since work allocated to the processor may be on the critical path and halting will extend the critical path. No HPC energy efficiency work using DPG was encountered while working on this dissertation.

DVFS Impact. The majority of work on energy efficiency in HPC is based on DVFS, possibly due to the similarity to the established technique of load balancing. A common technique for improving application performance in HPC applications is to use a technique called *load balancing*[6] to improve the time alignment of work completion by each computing unit. DVFS makes reasonably predictable changes to the runtime of compute intensive workloads on a node, allowing load balancing via adjusting relative node computation speed. We highlight two works here.

In "Adagio: Making DVS Practical for Complex HPC Applications", Rountree et al. (2009) look at reducing energy consumption of HPC applications while maintaining runtime performance using DVFS. The authors use the insight that many nodes in an HPC application spend some amount of time waiting, causing these nodes idle while other nodes complete computations. As the application executes, Adagio records the time spent between collective communication and the time spent waiting within the collective. On future iterations, Adagio adjusts the clock frequency during the computation portion of

[6]The load (i.e. amount of work) is the same (i.e. balanced) across all workers. Load imbalance leads to longer runtime since heavily loaded workers will take longer than lightly loaded workers and all workers must complete before the next iteration of the computation can begin.

execution to reduce the time spent waiting within the collective. Adagio runtime and energy savings are favorable across a wider range of applications than the comparison algorithms Rountree et al. tried.

In "Green Queue: Customized Large-scale Clock Frequency Scaling" Tiwari et al. (2012) also investigate energy saving for HPC applications using DVFS. The approach used by Green Queue is considerably more invasive than Adagio, involving static analysis and traced executions prior to the execution compared to the baseline. While the reported Green Queue results are impresive, with an average 17% energy savings, the results are somewhat out of context since the energy and time cost of executions needed to generate models are not considered. For applications that have consistent power behavior and must be run repeatedly, this energy cost for model building can be amortized though.

RAPL Impact. The runtime and energy performance impact of RAPL is still an area of active research in the HPC community, which complicates the application of RAPL to energy efficiency. One of the interesting and confounding properties of RAPL is that small manufacturing differences are exposed as noticeable performance differences when a power cap is applied (Inadomi et al., 2015; Rountree et al., 2012). Minute variations in the power cost for each individual transistor on a processor, summed over billions of transistors operating at gigahertz frequencies, can result in substantial variations in power consumption. Most HPC RAPL research focuses on execution in power limited contexts, which is a component of the power scheduling problem to be discussed in the next section. However energy efficiency gains can be inferred from this work.

In "A Run-Time System for Power-Constrained HPC Applications", Marathe et al. (2015) present *Conductor*. *Conductor* shifts power between

nodes within a job, decreasing the aggregate power consumption of the job. The technique is actually hybrid, DVFS is used for load balance and estimates on the expected power impact is used to set the updated RAPL power cap. A way to understand *Conductor* in the context of energy savings is that *Conductor* provides the energy benefits of DVFS techniques and provides an enforced upper bound on job power consumption.

In "Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing", Patki et al. (2013) sweep node count, core count, and processor power cap. From the results presented, the energy cost of configurations can be computed. In many cases, the most energy efficient configuration does not use the maximum number of nodes and cores at the highest power setting. Follow-on work by Patki et al. (2015), discussed in the Power Scheduling section of this chapter, used energy efficiency gains from selecting energy optimal configurations to support power-aware job scheduling.

Indirect Mechanisms. Power consumption is a side effect of the active computations. In the case where direct processor power control mechanisms are not exposed, changes to the instruction stream or degree of parallelism can be used used to effect power consumption. Memory, network, fans, and other subsystems effect overall system power consumption as well and these subsystems rarely expose mechanisms for direct power control. Some work exists that addresses how to control power through only indirect means.

Hoffmann (2013) and Hoffmann et al. (2013) approach power reduction as a multiobjective optimization problem. Rather than looking at a bound as a hard limit, these works consider a bound as an optimization target when adjusting other properties of the environment. Environmental controls used in the work

extend beyond direct power control, actions like changing the DVFS settings, to indirect controls, like changing the work distribution across cores. Hoffmann (2013) demonstrates an ability to reduce energy cost without performance impact for applications where iterations faster than a realtime interval produce no benefit. Hoffmann et al. (2013) demonstrates a online tuning system, PTRADE, that can learn environmental settings to improve power efficiency during runtime.

The challenge with application of indirect mechanisms is the latency required to converge to a target power cap. Zhang and Hoffmann (2016) use RAPL in addition to indirect controls to achieve faster convergence to a power target than indirect control alone provides. In this dissertation, only the processor contribution to system power is considered due to the lack of a RAPL like mechanism for most other subsystems[7]. The timeliness guarantees required to respond to power allocation changes and maintain bounded consumption for safe operation of a hardware overprovisioned HPC system cannot currently be supported through indirect control mechanisms.

Runtime Reduction. Performance optimization, defined as minimizing job runtime, has been the classic measure for goodness in HPC systems and applications. Given the numerical intensity and duration of HPC jobs, even a small percentage gain in performance can result in significantly shorter time to solution. Compiler optimizations to maximally utilize processor, memory, and IO subsystems to reduce runtimes have been traditional areas of HPC. Accelerator technologies, like GPGPUs and vector units, have also been traditionally studied with respect to their impact on job runtimes. While significant work exists on reducing computational runtime for HPC applications and increasing hardware

[7]RAPL also supports power capping memory, however this feature is often disabled in the BIOS.

energy efficiency, there is little work that discusses computational energy costs directly when runtime is reduced. Two papers are mentioned here.

In "Quantifying the Impact of GPUs on Performance and Energy Efficiency in HPC Clusters"(Enos et al., 2010), the energy cost of CPU only computation is compared to a computation using GPU acceleration. Four HPC codes are run with and without GPU acceleration on the test system. The authors conclude, for their hardware, a speedup of greater than 3x is required for the GPU accelerated code to be more energy efficient than a CPU-only code.

In "Comparing Performance and Energy Efficiency of FPGAs and GPUs for High Productivity Computing"(Betkaoui et al., 2010), the energy costs of CPU-only, FPGA, and GPU execution is compared. Four applications are run on the CPU, FPGA, and GPU. Comparison of the FPGA and GPU are of primary interest to the authors since the CPU, while having the lowest power, has the highest energy cost for large inputs. The memory access pattern of the workload was shown to have a major impact on the comparative efficiency of FPGA and GPU acceleration. For applications with streaming and sequential memory access, the GPU provided better performance. For applications with more random memory accessed, the FPGA solution provided better performance.

Reflection on Energy Efficiency. Energy efficiency is an important property for HPC computations and systems, but energy efficiency does not help in bounding power consumption. Power reduction strategies for energy efficiency reduce the power cost of a computation and enable the *extra* hardware within a hardware overprovisioned HPC system to be used. Runtime reduction strategies for energy efficiency may increase the power cost of a computation, but provide a net benefit if sufficient power can be made available for the computation. Which

strategy provides the best performance is dependent on the specific hardware and computation being optimized with respect to energy. Since improved energy efficiency may involve either increasing or decreasing power, work on energy efficiency alone is insufficient to safely support a hardware overprovisioned, with respect to power, HPC system.

Energy efficient computations are necessary however for realizing performance improvements from a hardware overprovisioned system. Job scheduling on HPC systems typically results in several applications, each application having an application specific power consumption, running concurrently on different partitions of the cluster. The presence of some applications using power reduction strategies, explicitly or due to hardware energy optimizations, and other applications using runtime reduction strategies provide excellent opportunities to load balance power rather than work across the cluster. An effective power scheduler will leverage these power differences across jobs to allocate power to nodes, allowing more work to complete per unit time, than would be possible if all node received an equal amount of power.

Power Scheduling

Power scheduling allows the gains from energy efficiency to be converted into improved computational performance for the same system power bound. A system power bound may originate from an administrative concern (e.g. budgeting) or physical concern (e.g. electrical distribution infrastructure). In either case, power scheduling utilizes *power shifting*[8] to redistribute the available power across cluster nodes. Rather than attempt to reduce the total energy cost of the system, power

[8]Power is conceptually *shifted* from one set of nodes to another set of nodes for some duration. Power shifting is also used internally by modern processors to improve performance with in the TDP (Felter, Rajamani, Keller, & Rusu, 2005).

scheduling work aims at maximizing power utilization within some upper power bound.

Two important attributes for partitioning existing power scheduling work are, limit enforcement and schedule time. These attributes will be discussed prior to a brief review of several power scheduling solutions in the literature. Following the review will be brief commentary an the performance comparability issues with the current literature.

Global Power Limit Enforcement. The environment an overprovisioned HPC system is deployed in, will determine how rigorously a power scheduler must maintain the system wide power bound. Exceeding the power bound for bounds originating from physical concerns may damage the computing or power infrastructure. In such environments, the power limit must be provably enforced for safe operation. In many cases power bounds are likely to originate due to administrative concerns (e.g. power is cheaper at different times of day.). For administratively power bound systems, failure to enforce the system power bound from time to time may be safe, but costly and therefore undesirable.

The power scheduling invariant can be used to reason about a power scheduler's ability to enforce the system wide power bound. Formally, the power scheduling invariant can be written as:

$$\forall t, L \geq \sum_{i=1}^{n} a_i^t \geq \sum_{i=1}^{n} c_i^t \tag{2.3}$$

Stated in english, at all times the system wide power bound must be greater than or equal to the power allocated to components within the system and each component must consume no more than the component's current allocation. Table 4 defines the variables used in the formalization.

Symbol	Meaning
L	Global power limit
t	Time interval
n	Total number of components
a_i^t	Power allocated to component i at interval t
c_i^t	Power consumed by component i at interval t

Table 4. Table defining the symbols used in the power scheduling invariant (Equation 2.3).

An HPC power scheduler can be decomposed into algorithm and mechanism. The algorithm determines when and how power is allocated (i.e. shifted) across nodes in the cluster and is modelled in the power scheduling invariant by:

$$\forall t, L \geq \sum_{i=1}^{n} a_i^t \tag{2.4}$$

Failure to provably meet the condition on the algorithm indicates that a power scheduler may, at some point during operation, produce allocations that will exceed the system wide power limit.

The mechanism determines how the component level power cap is maintained and is modelled in the power scheduling invariant by:

$$\forall t, a_i^t \geq c_i^t \tag{2.5}$$

Failure to provably meet the condition by the mechanism indicates that a component, at some point during operation, may disobey the power scheduler and exceed the allocation.

Hard Enforcement. Hard power limit enforcement matches the case where the power scheduling invariant is provably satisfied by the power scheduler. A scheduler capable of hard enforcement must be able to guarantee that the power bound is never exceeded by the cluster. Providing hard enforcement requires some hardware support due to the fine granularity of the timescale at which a power

bound must be maintained. In cases where physical limitations motivate the system wide power bound, operation of the cluster without a scheduler capable of hard enforcement may result in physical damage to the cluster.

Soft Enforcement. Soft power limit enforcement matches the case where algorithm or mechanism cannot be proven to satisfy the power scheduling invariant. There is an argument that power schedulers providing only soft enforcement should be rejected as unfit for purpose. A power scheduler has a requirement to allocate power such that a system wide power bound is maintained and, therefore, a power scheduler providing soft enforcement does not meet the functional requirements. On the other hand, systems without hardware support for direct power management can never be proven to meet the power scheduling invariant's requirements for mechanism. Research on power schedulers providing soft enforcement are interesting due to their ability to provide reasonable power bound enforcement without hardware support.

Another case for schedulers only providing soft enforcement can be made based on electricity provider contracts. Abrupt changes in power consumption create challenges for an electricity provider since power consumption and generation must remain balanced across the larger power grid. Some providers require large power customers to estimate their power consumption in advance and penalize customers for significantly going over or under the estimate. Another consideration is that the wholesale cost of power in many markets depends on the time of day the power is being used. A financially advantageous power schedule for an organization may involve operating the cluster with an artificially low power bound during times when power costs are higher. The infrastructure in these cases is capable

of supporting the full load of the HPC cluster but there are some benefits to attempting to maintain bounded consumption even if the attempt fails.

Schedule Time. When power schedules are computed and applied can also be a useful way to partition the space of power schedulers. The time and frequency of schedule application can effect the observed level of power utilization across the HPC system. One way to think about why schedule time should be expected to impact power utilization is to consider that each time a schedule is applied, there is an opportunity to adapt the power schedule to the current system conditions. Due to the entrance, execution behavior, and exit of jobs from the HPC system, the power needs of a cluster tend to be dynamic and evolve over time. These major schedule times are present in the literature: static, reservation, and dynamic.

Static Techniques. Static techniques set the power cap once at system install time and never change the cap there after. Trivially, the static algorithm can be shown to maintain the power scheduling invariant since there are no time varying allocations to consider. As long as the sum of the component power caps is less than the total system power bound at system start time the invariant will be satisfied.

The simplicity of reasoning about static power scheduling techniques makes them a common baseline for comparison. Current HPC systems use a degenerate form of this technique, where the component level power cap is the TDP of the component. Many works on hardware overprovisioned power scheduling will compare to a system where the power per component is statically set to the average

power for the system. One can imagine a cluster in which different nodes have different statically assigned power caps[9].

Reservation Techniques. Reservation techniques set component power caps at job start time to keep the system within the total power budget. Each time a job is scheduled, the power scheduler is responsible for setting an appropriate power cap on the nodes associated with the entering job. In relation to the power scheduling invariant reservation based techniques group components by job and make job level power allocations, which are then distributed across the components assigned to the job. The invariant is satisfied if the sum of concurrent job power allocations is less than or equal to the system power bound. Most HPC power scheduling work to date uses this technique.

Generally, reservation techniques merge the job and power scheduling activities. A motivation for approaching job and power scheduling together is the ability to preserve job runtime. Since an insufficient power allocation will increase the runtime of individual jobs, the integrated job and power scheduler should only concurrently run jobs for which there is sufficient power available. One of the challenges in the implementation of reservation techniques is power consumption modeling for the jobs to be executed on the system since the component level power caps must be set prior to job launch.

Dynamic Techniques. Dynamic techniques adjust component power caps during job runtime such that the total power consumed remains beneath the system wide power limit. Proof of the power scheduling invariant must be done at the component level for systems using dynamic techniques since there may not be

[9]Job scheduling in such a cluster might resemble job scheduling in a cluster with both fat and thin nodes. Users would select a partition to run their job in based in anticipated power consumption.

Year	Scheduler	Enforcement	Schedule Time
NA	Naive	Hard	Static
2014	PARM	Soft	Reservation/Dynamic
2015	SLURM	Soft	Reservation
2015	RMAP	Hard	Reservation
2015	PowSched	Hard	Dynamic
2016	PMJPC	Soft	Reservation
2016	Shifter	Hard	Phase/Dynamic
2016	DAPM	Hard	Dynamic

Table 5. Table relating work to enforcement and scheduling strategy

groupings in space (i.e. nodes) or time to use to simplify the analysis. HPC power scheduling work using dynamic techniques are currently limited.

The highest gains in system-wide power utilization are expected using fully dynamic power scheduling techniques. In static and reservation techniques, a job using less than the power allocated to its components has no mechanism to yield the power to other jobs that may be able to use the power, resulting in unutilized power. Power utilization improvements with dynamic techniques are possible since power can be shifted between jobs based on the current phase of the concurrently executing work. The gains in power utilization may cause application runtime determinism to be lost since power shifted away from a component may not be available when the component returns to a high power consumption phase.

Literature. There are many power schedulers in the literature, eight are discussed in this section. Two common baselines are used for comparison with hardware overprovisioned power schedulers, both of which are covered by what will be referred to as the naive power scheduling strategy. A matrix, comparing the schedulers based on enforcement and schedule time is in Table 5.

Naive. The naive power scheduling strategy is the only static technique considered in this thesis. Using this strategy, power is scheduled once at the time

the machine is installed an never changed there after. Using technology such as RAPL the naive strategy can provide hard enforcement of the node power caps. Existing HPC systems can be considered to use this strategy since each node receives a power allocation matching the node TDP. When using the naive scheduling strategy in a hardware overprovisioned system, the system power bound is typically divided evenly across all nodes in the HPC cluster.

Naive power scheduling, while capable of enforcing a system power bound, tends to produce suboptimal performance. A major limitation of the naive strategy is the lack of adaptation to the power consumption behavior of different nodes running different applications. For a given power cap, some application runtimes will be strongly penalized while other applications will not see an impact to runtime (D. A. Ellsworth et al., 2015b)[10]. The basic performance problem with the naive strategy is poor power utilization for applications that consume less than the average power and poor runtime for applications that would consume more than the average power allocation.

PARM. In "Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget", Sarood, Langer, Gupta, and Kale (2014) present a power scheduler named PARM. PARM expects power capping to be available on the hardware and prefers jobs that support both *moldability* and *malleability*[11]. When a job is queued or a job terminates, PARM attempts to find and optimal job and power schedule to maximize job throughput. Hardware

[10]The relationship between power cap, consumption, and runtime is interesting and will be covered more in chapter III

[11]A *moldable* job allows the resource scheduler to launch the job on more or less nodes than were requested at job queue time. A malleable allows the compute resource allocation (e.g. number of nodes) to be changed dynamically at runtime

power capping and job malleability enable PARM to dynamically apply the new configurations.

The theoretical foundation formulates the problem is an integer linear programming (ILP) problem taking into account estimated power for all possible malleable configurations of work to be scheduled. ILP problems are NP-hard in the general case, which makes computing the optimal solution potentially intractable on the timelines required for online job scheduling. Sarood et al. (2014) introduce a power aware speedup metric to reduce the number of variables to be considered by the ILP solver, to improving scheduling time. Quantization of the power settings per node are also used to reduce solution search space.

Like other reservation strategies, PARM requires power estimates to be available apriori to make good scheduling decisions. The time and power costs required to generate sufficiently good power estimates were not discussed in "Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget". Empirical evaluation of PARM was conducted using Charm++ and the baseline for performance comparison was a classically power provisioned cluster (i.e. a cluster where all nodes are allocated TDP). Additionally, simulation using a trace from a production system was used to evaluate PARM's expected performance. Results indicate a significant improvement in average job turnaround time from the baseline. A concern from the work is the time required to compute an updated schedule, the authors note 15 seconds were needed to compute a schedule from a queue of 200 jobs.

RMAP. In "Practical Resource Management in Power-Constrained, High Performance Computing", Patki et al. (2015) present RMAP, a power-aware backfilling scheduler. RMAP expects jobs to be moldable and to have a database

of job configurations that provides node count, power cap, and runtime for the job. When scheduling work, RMAP selects the most runtime efficient configuration for a job that fits within the available power. The job level power bound is enforced through setting a RAPL power cap on the components at job launch time.

Evaluation of RMAP is done exclusively using simulation. The exhaustive database of job configurations and runtime make the simulation results valid even though the simulator has no mechanism to compute runtime increases due to power capping[12]. The cost in runtime and power to generate the configuration database is not discussed and is expected to be substantial since the database must contain an exhaustive search of node counts, core counts, and component power caps. Supposing such a database exists, RMAP can deliver improved job turn around time and enforce a system power bound.

SLURM. In "Adaptive Resource and Job Management for Limited Power Consumption", Georgiou, Glesser, and Trystram (2015) discuss extensions to SLURM that makes power a schedulable resource. A user is responsible for providing a power cap with their job submission. The job scheduler uses this information to generate job schedules that provide the requested power for all concurrently executing jobs. Enforcement is handled through selecting a DVFS setting for the node processors that guarantees the power cap cannot be exceeded. In addition to changing DVFS settings, SLURM is able to power on/off nodes to change the system power consumption. This scheduler is classified as providing soft enforcement since the DVFS settings are software controlled based on a model of the relation between DVFS setting and maximum power consumption, which is not guaranteed to be correct.

[12]Runtime effects of different power caps can be retrieved from the configuration database avoiding the need to compute what the performance would be at a specific power cap.

Evaluation is done primarily in simulation and involves replay of a job trace. Use of DVFS as the primary power management mechanism simplifies simulation since increasing runtime proportionally to the reduction in clock frequency is a reasonable approximation. Comparisons are made between energy, job launches, and CPU time during a 5 hour interval. Results show the scheduler successfully reduces energy consumption over the evaluation interval at different power caps but the impact on job throughput is unclear.

The solution presented by Georgiou et al. (2015) is unique in the current HPC literature since thier solution also involves powering nodes off and on to avoid the idle power costs associated with nodes that are on, but not actively participating in a job. Most other work on power scheduling has ignored idle node power costs. Idle node power consumption can have a significant impact on a hardware overprovisioned system (Sakamoto et al., 2017) and will hopefully became a more common consideration in work going forward. Idle node power will make an appearance in Chapter VI as the effect on system performance can be significant when comparing techniques.

PMJPC. In "Predictive Modeling for Job Power Consumption in HPC Systems", Borghesi, Bartolini, Lombardi, Milano, and Benini (2016) discuss power scheduling using only the job scheduler. By only scheduling concurrent jobs that, in aggregate, consume less than the system power limit there is no need for active power capping. With no active power enforcement mechanism, the ability of the scheduler to maintain the system wide power cap depends completely on the quality of the job power estimates. Generating high quality estimates therefore is the major focus of the work and makes use of machine learning techniques applied to the job log and system power consumption history.

Evaluation is done using production traces from the Eurora Supercomputer. Unlike most DOE HPC systems, the Eurora Supercomputer allows multiple jobs to be concurrently scheduled on the same node. Thus, the processor level power measurements used to generate estimates may have consumption contributions from several different jobs. Borghesi et al. (2016) report an ability to predict power consumption with a mean error of under 5%, after excluding outliers. While the estimation quality achieved is impressive, especially given the coresidency of jobs, the size of the training set and weakness of enforcement would be challenges to adoption as a practical power scheduler.

PowSched. In "POW: System-wide Dynamic Reallocation of Limited Power in HPC", D. A. Ellsworth et al. (2015a) introduce PowSched, which will be covered in detail in Chapter IV. D. A. Ellsworth et al. (2015a) suggest that acceptable power scheduling performance can be achieved via a simple online feedback mechanism that operates without job awareness or power consumption history. At the highest conceptual level, PowSched simply gives higher allocations to components consuming power near the current allocation and gives lower allocations to components consuming significantly less power than the current allocation. RAPL is used to provide hard power limit enforcement and limits are changed at arbitrary times by the power scheduler.

Scaling and performance for PowSched is studied in "Dynamic Power Sharing for Higher Job Throughput" (D. A. Ellsworth et al., 2015b). D. A. Ellsworth et al. (2015a) select the naive strategy as the baseline for comparison in their work and use a mix of three common HPC benchmark applications (AMP, CoMD, and LULESH). In the best empirical case, PowSched achieves a 14% improvement over the baseline and at worst produces results within

system jitter from the naive strategy. Plots of power consumption versus allocation indicate that PowSched can significantly improve power utilization. Simulation studies are also conducted to investigate in what cases PowSched can be expected to out perform the naive strategy. Due to the limited data required and simplicity of the power scheduling algorithm, PowSched scales well to hundreds of thousands of components.

PowSched is unique in the literature for several reasons. At the time of PowSched's introduction, there were very few dynamic power schedulers and very few power schedulers with hard enforcement. PowSched does not require any job modeling or apriori estimates to effectively schedule power. In fact, PowSched is completely unaware of the mapping between jobs and hardware, hence power scheduling decisions are made exclusively through viewing the HPC cluster as flat pool of compute components. PowSched is also completely unaware of the system job scheduler, which is different from most other power scheduling work where the power scheduler is deeply integrate with the job scheduler. Follow-on work has shown that PowSched can be trivially adapted to support hierarchical power scheduling (D. Ellsworth, Patki, Perarnau, et al., 2016).

Shifter. In "I/O Aware Power Shifting", Savoie et al. (2016) present Shifter, a power scheduler that is able to effectively shift power between compute units in I/O versus compute phases. Savoie et al. (2016) note that many HPC applications alternate between compute intensive and IO intensive phases. Shifter assumes that the frequency and duration of an application's IO phases are know apriori and Shifter is able to use this knowledge to perturb application execution in an attempt to better align IO and compute phases. During compute intensive phases processor power consumption is significantly higher than during IO intensive

phases. Phase boundaries are detected in Shifter by instrumentation of the MPI library and are used to signal that the power schedule should be adjusted. Processor power caps are set across the participating nodes, using RAPL, as IO boundaries are detected.

Savoie et al. (2016) evaluate Shifter empirically with three HPC codes (LAMMPS, ParaDiS, and Cactus) via simulation. The baseline configuration for comparison is a system using naive power scheduling with a 60W per processor power cap. In all reported cases, Shifter is able to improve performance over the baseline.

DAPM. In "Demand-Aware Power Management for Power-Constrained HPC Systems", Cao, He, and Kondo (2016) propose a demand aware power scheduler. CPU performance monitoring counters are used to monitor and estimate an application's power need. A machine learning approach is used to learn a processor variation aware model for processor power settings. Component power caps are set dynamically during runtime using RAPL.

Evaluation is done using FIFO and backfilling job scheduler policies on a real HPC cluster. The baseline for comparison is a non-capped cluster with the same power bound. Three other naive strategies are also used for comparison. Several NAS parallel benchmarks are used for the experimental workload and performance evaluation is reported in terms of power utilization and job completion rate. The adaptive strategy studied outperforms the naive strategies.

Discussion. One of the most discouraging gaps in the current state of the literature for hardware overprovisioned systems and, more broadly, power scheduling research is the lack of good comparison studies. In the cited works different workloads and systems have been used to evaluate each system. Sakamoto

et al. (2017) have shown that hardware overprovided HPC performance is impacted by the number of nodes in the cluster, system power bound, and the compute intensity of the jobs to be executed. Without executing comparable work on comparable clusters, the relative practical performance of the schedulers is incomparable from the publication text. Work done using simulation is similarly challenged in the power scheduling space as no standardized simulator exists that supports the needs of the community. Each cited study using simulation uses a different simulator, which will encode different assumptions about the behavior of applications and hardware when power capped. Chapter VI discusses proposals for how the community can fill these gaps as well as preliminary results using PowSim. Work towards unified experimental platforms is anticipated as future work and beyond the scope of this dissertation.

Schedulers supporting hard enforcement will be necessary for practical deployment of hardware overprovisioned HPC systems at a large scale and in contexts where exceeding the system wide power bound can cause physical harm. Hard enforcement will require hardware support to satisfy the power scheduling invariant since hardware support is the only way to guarantee that component level power caps will be rigorously maintained. Without hard enforcement, power procurement and infrastructure must still support the case where all components operate at peak consumption since application behavior may diverge from the power scheduler's expectations.

PowSched, part of the work contributed in this thesis, is the only dynamic power scheduler in the literature that does not require a model of job level power consumption for operation. In Chapter IV, PowSched will be discussed in detail.

Chapter Summary

In this chapter we referenced Rountree et al. (2012), the initial work proposing hardware overprovisioned HPC systems. A hardware overprovisioned cluster has more nodes that can be powered at TDP and, hence needs external mechanisms, to remain with a physical or administrative system wide power bound. Hardware overprovisioning relies on energy efficiency computing to allow at least some of the HPC nodes to operate beneath the TDP for much of the time. Power schedulers are able to convert the energy savings, which appear as a power savings, to additional cluster efficiency by supporting the scheduling of additional work on the *extra* cluster nodes. Schedulers proving hard enforcement (i.e., can be proven to satisfy the power scheduling invariant) are needed when physical damage can occur from exceeding the system wide power bound since schedulers using soft enforcement cannot guarantee an upper bound on component power consumption.

CHAPTER III

POWER CAPPING RUNTIME EFFECT

This chapter contains material that has been previously published in D. A. Ellsworth et al. (2015b). The PowMon monitor, experimental setup, and analysis of results are my original work. Co-authors assisted in the language used to present this material in the previously published work.

PowSched (Chapter IV) and PowSim (Chapter V) both rely on an understanding the effects of RAPL power capping on application runtimes. When work on this disseration began in 2014, HPC systems exposing RAPL were extremely limited in the community and the majority of research was focused on the energy efficiency problem. Non-hardware based approaches to HPC energy efficiency primarily use DVFS to create better time alignment between bulk synchronous phases. As a result, very little work was available in the community discussing observations when running applications under RAPL power caps. Of the limited HPC work found on RAPL power caps at the time, results were primarily energy focused and provided little insight into when or why a power cap would impact application runtimes.

This chapter will present a power monitor constructed to support investigating the effect of power capping on application execution. The power monitor will be used to perform experiments and gather data and runtime effects of power bounding on HPC benchmarks. Following presentation of these results, there will be a discussion to connect low level physical details of processors with the high level expression of algorithms, explaining why the observations are expected.

Power Monitoring

Understanding processor power consumption in relation to a power cap over time will require observing these values at runtime[1]. When and how often to make observations as well as the impact of the observation activity on the observed system are challenges that any monitor must address. Several tools exist for collecting observations, but were were not a good fit for this work for reasons to be discussed later in this section. PowMon, the power monitor initially developed to support this work, is now distributed and maintained by libmsr project[2].

An ideal power monitor for this work has several desirable characteristics. The power monitor should not require physical changes to nodes in the cluster. The power monitor should operate at the granularity of the processor package since power control will also occur at that granularity. The power monitor should not require instrumentation of or interfere with the work being observed. The power monitor should make observations periodically since power behaviors are not well enough understood to identify other triggers. The power monitor should support fine temporal resolution. PowMon will satisfy these characteristics.

Missing Capabilities. At the time of this work, there were three common approaches for power observation. Performance measurement tools were just beginning to have support for reading power. System administration tools on some systems captured power and energy readings. Hardware monitors also appeared frequently in the literature, however these require physical modifications. None of these solutions to the measurement problem were a good fit for this work's power monitoring needs.

---

[1] If power consumption varies over time, then observations must be made periodically to capture when these changes in consumption happen and the magnitude of the change.

[2] `https://github.com/LLNL/libmsr`

Performance measurement tools like PAPI and TAU have the ability to collect power measurements. The challenge with using traditional performance tools for power is a mismatch in granularity of measurement. Instrumentation occurs logically, often at the level of threads, and focuses on the specific application. Tools like PAPI go to great lengths to mask out the contribution of computation outside of the target application to the performance counters being read. Power, on the other hand, is shared by all CPUs in the processor package. Thread level metrics are the wrong granularity for measurement of uncore counters. Further, the same thread may execute on different processors at different times, adding to the complexity of mapping power measurements in multi-socket nodes. Given the need to understand whole system power performance rather than individual application performance and the mismatch in granularities, existing performance measurement tools were not suitable for the needed observations.

Systems like the IBM BlueGene/Q provide several advanced system monitoring features, including power monitoring. These operational power monitoring solutions are targeted at the needs of system administrators to understand the behavior of the computer system. Power measurement intervals are too long (i.e. in the range of seconds) for good correlation with the activity on the host. Access to the power measurements also tends to be difficult due to organizational security policies. Using existing system level monitoring was also not viable for the gathering the needed observations.

Owing in part to the newness of power as a concern, few computer components had power monitoring capabilities. Many early works looking at power and energy required researchers to physically instrument their hardware with power monitors. Adding physical instrumentation supports the ability to monitor several

different components and monitor without adding overhead to the monitored system. Making the hardware modifications, however are time consuming and correlating timestamps between the system and monitors adds complexity. With the desire to use the monitoring technology to make measurements at scale, physical instrumentation was also not a viable approach for generating observations.

PowMon Design. The basic functional requirements of a periodic monitor sound simple: At regular time intervals, make an observation and capture the value to storage. Several practical challenges complicate monitor design. Monitors, like PowMon, share hardware with observed system, creating conflicts and resource contention between the monitor and workload. Naively, an author of a monitor might elect to sleep for the scheduling interval between completing the write of the last observation and generating the next observation. Implementing the interval naively will result in long and nonuniform intervals since the amount of wall clock time to complete the observation and write change based on clock frequency and level of contention for processor cycles.

PowMon does not require any modification to existing software to provide monitoring services for distributed applications and does not coordinate across nodes at runtime. Monitored programs are started using PowMon as a wrapper. During startup, PowMon instances on the same node use shared memory to elect a single PowMon process to make measurements and then wait for the wrapped process to terminate. Only the elected PowMon process runs the measurement thread and the elected process remains running until all other PowMon processes on the node have exited. The pseudocode for PowMon is given in Algorithm 1.

As a post-processing step, the data in the measurement files can be correlated by timestamp and source node for reporting on observed power performance.

Algorithm 1 PowMon logic in psuedocode

```
procedure MAIN
    Stopping flag is set to false
    Detect other instances on host
    if first instance then
        Start measurement thread
    end if
    Fork and start the wrapped process
    waitpid on forked process
    if first instance then
        Use semaphore to wait on other instances
        Set stopping flag to true
        Write final measurement and summary data
    else
        Use semaphore to indicate done
    end if
    Halt
end procedure
procedure MEASUREMENT THREAD
    while stopping flag is false do
        Read MSRs
        Write MSRs to disk
        Sleep until next interval
    end while
end procedure
```

Observation Mechanism. Observation mechanisms must be available on the test platform. The target experimental platform used Intel processors, so mechanism selection was limited to available Intel technologies. Specifically, RAPL was used for this dissertation work. Modern Intel processors provide model specific registers (MSRs) to expose several CPU and processor features. Among the many features provided by modern Intel processors are the abilities to set processor power caps and read processor energy consumption via MSRs. The MSR used to

count energy consumption is updated roughly every millisecond, providing energy information with a very high temporal resolution. As a built-in hardware capability, the Intel MSR mechanism is an ideal choice for PowMon.

Generally, access to processor MSRs requires privileged system access. For experimentation on a production cluster, gaining privileged access is problematic due to potential security implications. Lawrence Livermore National Laboratory (LLNL) has produced two software components that are able to address the operational security issue and provide safe access to MSRs for researchers. Msr-safe[3] and libmsr provide sufficient security controls to enable the power specific MSRs to be exposed to researchers on a production cluster via regular user accounts. Rather than access the MSRs directly, PowMon accesses the relevant MSRs via libmsr and msr-safe.

Observation Timing. Due care is required in implementation of observation timing for a monitor using periodic sampling. A simple monitor may make an observation and then sleep for the full sampling interval, Δt, before making the next observation. Such systems produce unreliable results since the computation takes some time, ϵ, making the actual interval $\epsilon + \Delta t$[4]. Since conversion accuracy between energy and power depends on the accuracy of the time measurement the presence of the additional ϵ time is undesirable.

PowMon makes use of the linux real time clock library to access a real time clock with subsecond timing and *select()* to sleep for the appropriate interval. When the monitor starts, the current clock value, T_0, is captured and

[3]https://github.com/LLNL/msr-safe

[4]Each round the measurement computation takes ϵ seconds to make and is followed by Δt seconds of sleep before the next round starts. The total time for the round (e.g., time between starting the first and second rounds) is, therefore, $\epsilon + \Delta t$ seconds.

used as the basis for all future monitor measurements. With an interval, Δt, the kth measurement will occur at $T_0 + k\Delta t$. At the end of an observation, the current high resolution real time is retrieved and the time remaining before the next measurement is computed. The measurement thread then uses *select()* to sleep until the next observation, resulting in near uniform sampling intervals. A maximum temporal error between any two measurements on the same host is $2\Delta t$, a property provided by computing the sampling times from a fixed time in the past.

Another highly complex issue when handling time is correlating the time at which events happen across nodes within a cluster. Each node in cluster has an independant clock that is subject to jitter and timing anomalies. Using very fine grain realtime clock timestamps to report events across nodes can lead to poor conclusions since clock drift may cause events to become out of order when only sorted by local timestamps. Before using PowMon for monitoring a distributed appliction, the clock drift on the experimental system is investigated using simple MPI collectives to read all of the clocks. Clock drift between nodes on the experimental system was no more than a few milliseconds, which is much shorter than the 100 millisecond sampling interval used in the experiments. For the temporal granularity of this work, the effect of real time clock drift is considered to be negligible.

Observation Storage. For monitor observations to be useful, the observations must be captured for analysis. PowMon makes use of a standard buffered file stream to write out the collected measurements. The operating system controls when results are actually spooled to disk, avoiding performance impacts from actively flushing measurements to disk. The filesystem to which writes occur could potentially conflict with the workload being observed, since the network and

IO devices might be shared. Users can avoid much of this contention, if needed, by using a local filesystem to store the file containing the measurements.

Startup and Shutdown. Many performance tools provide their service by injecting instrumentation code into monitored code at compile or link time. An advantage of execution via instrumentation is the ability to make use of application startup and other facilities (e.g. MPI communicator) to bootstrap the monitoring software as part of the monitored application. PowMon's ultimate target use is monitoring support for power scheduling research, which targets the power behavior of the node rather than specific jobs that might run on the node[5]. Implementation of PowMon through instrumentation of specific applications is not a good fit for the monitoring need. Instrumentation inside of an application would also hide power behaviors during the part of the application startup and shutdown before and after the monitor became active.

Rather than instrument a single application or run stand alone, PowMon operates as a wrapper around some other application. PowMon, on startup, begins a thread to gather measurements at the sampling interval and forks a child process for the target application. After forking the child process, the main PowMon thread waits for the child to complete before stopping the monitoring thread and halting the monitor[6]. The only challenge when starting PowMon comes from the system MPI launcher (e.g., mpirun or srun) launching multiple instances on the same host, which is often a desirable property for the wrapped target application. A shared

[5]Over the lifetime of a node many different jobs will run on the node. Additionally the node itself has a power consumption behavior even when no job is present. The idea of *idle power*, capturing the contribution of nodes in the cluster not currently executing a job, will appear in Chapter VI.

[6]To collect a trace for a single application, the application is wrapped by PowMon. To collect a trace of several different applications, PowMon wraps an application that sleeps while the applications to be traced are run.

memory segment and semaphore are used to elect a single PowMon instance per host to record measurements and signal when all of the target processes on the host have completed execution. Using PowMon is trivial for most jobs on commodity linux HPC clusters[7] since the wrapper is completely transparent to the wrapped application, job scheduler, and MPI library.

PowMon Performance Impact. The performance impact of PowMon on monitored applications is negligible. Table 6 shows runtime for three HPC benchmarks run with and without PowMon on the cab cluster at LLNL. Times are reported in seconds, using the measurement mechanism provided by the benchmark, and PowMon used the arbitrarily selected default sampling interval of 100 milliseconds. Observed runtime impacts of PowMon on the monitored applications are well within the system jitter, being well under one percent in most cases. The largest observed impact was a speedup of 1.65%, which is unexpected if system jitter is ignored since PowMon should be contending for resources. In the analysis, impacts of less than 1 percent are assumed to be likely due to jitter.

Experimental Data

An initial understanding of application power behavior over time was developed by running HPC benchmarks under various power capping configurations. Experiments were conducted on the Cab cluster at LLNL. The plots in this section are primarily *smear plots*[8], produced by plotting the raw PowMon measurements from participating processors on top of one another.

Dynamic Response. In the dynamic response experiments, the objective was to verify that the RAPL power capping mechanism had an immediate

[7]Linux fork and realtime clock support must exist in the compute node linux OS.

[8]Differences in consumption between sockets and differences in measurement times appear in the plots as vertical and horizontal smearing, respectively.

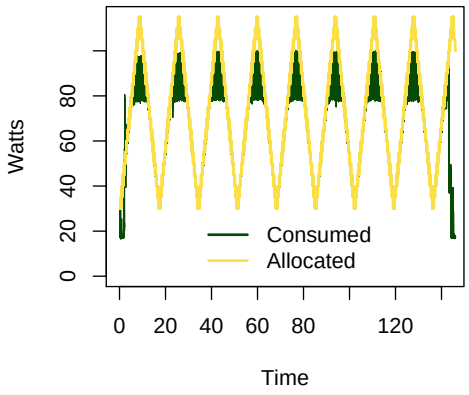
App	Nodes	Unwrapped	Wrapped	Speedup %
LU	16	119.77	119.84	-0.06
LU	4	112.39	112.92	-0.47
LU	2	118.17	118.69	-0.44
CoMD	16	107.1491	105.3836	1.65
CoMD	8	109.3181	109.2498	0.06
CoMD	4	92.4329	91.9755	0.49
CoMD	2	125.2268	125.7022	-0.38
AMG	16	102.573688	103.323772	-0.73
AMG	8	88.667316	88.173036	0.56
AMG	4	76.821048	76.763169	0.08
AMG	2	65.079914	65.436914	-0.55

Table 6. Runtimes for three HPC benchmarks at differing node counts with and without PowMon (Wrapped and Unwrapped, respectively). Observed performance is well within system jitter indicating that PowMon overheads are negligible.

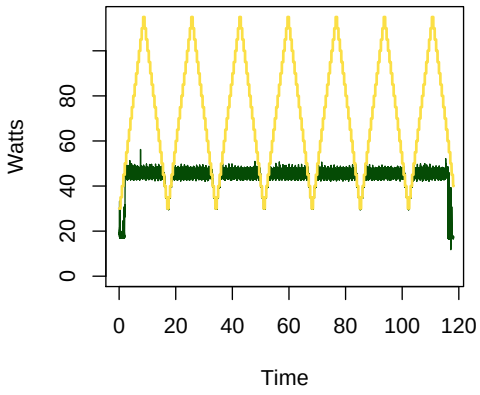
effect on the observed power consumption for the effected processors. To conduct this experiment, a modified version of PowMon was used to create power allocations that produced a saw tooth pattern when plotted against time. The PowMon measurement interval was held at 100 milliseconds and the power cap for each processor was adjusted by 5 watts every 5 measurement intervals. Power cap values range between 30 watts, the lowest power cap that our experiments indicated RAPL could maintain on the hardware, and 115 watts, the processor TDP. The traces for LULESH, CoMD, and AMG are shown in Figure 3. Power consumption remains under the allocation, as expected when the power cap is changed dynamically at runtime[9], an important property for a power control mechanism to be used in a power scheduler that is supposed to be capable of hard enforcement.

Characteristic Power Consumption. Plotting applications on effectively uncapped processors (i.e. the power cap is set to the TDP) provides

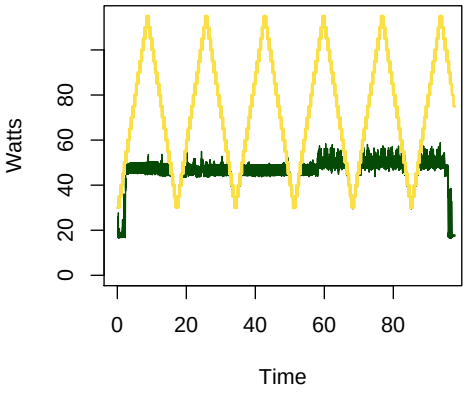
[9]The large spike in 3a is due to the sampling interval, 100 ms, being smaller than the RAPL window of 1000 ms.



(a) LULESH 30-115 Watts



(b) CoMD 30-115 Watts



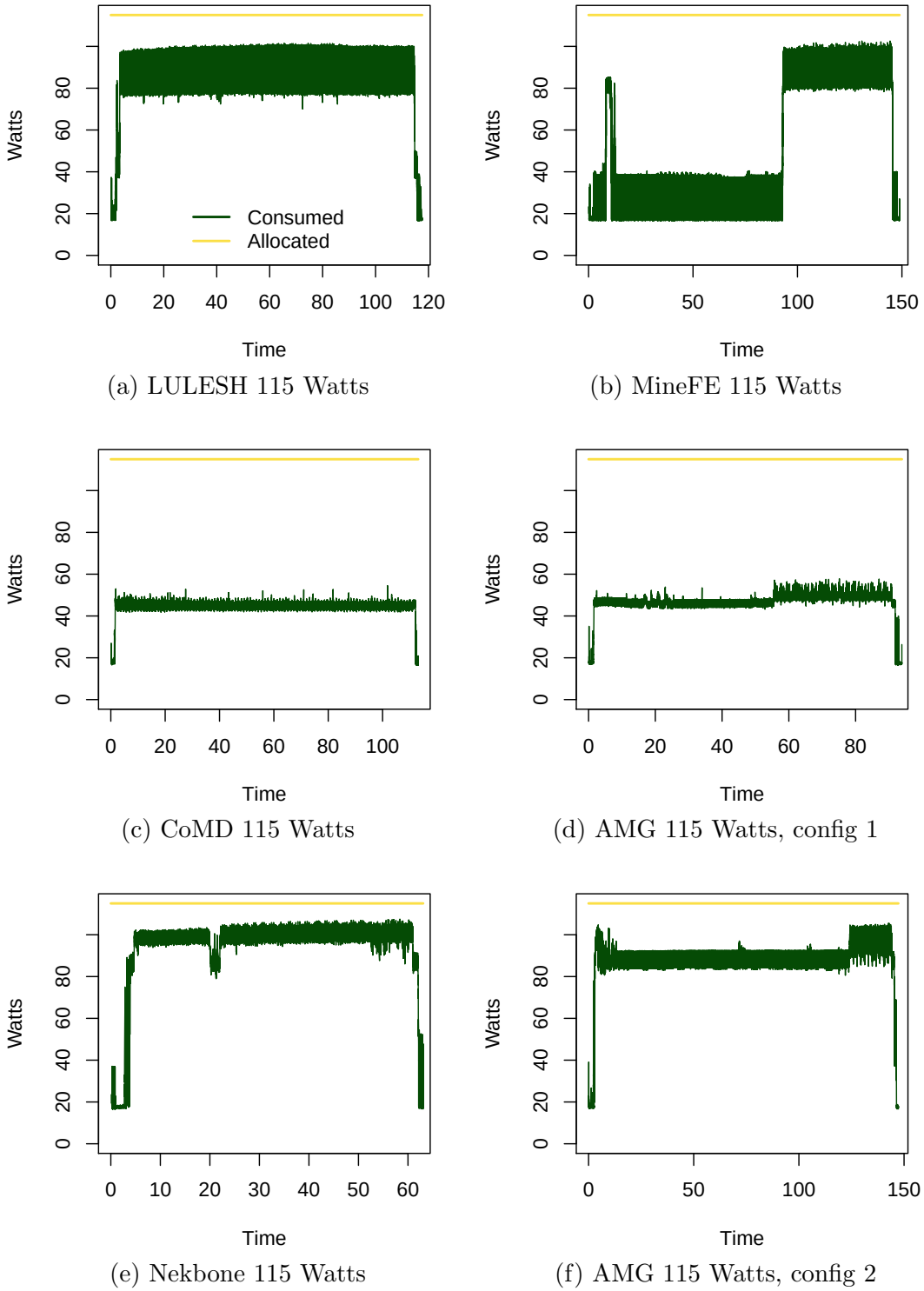
(c) AMG 30-115 Watts

Figure 3. Consumption when power allocation is varied during execution. Sampling at 100 ms intervals with a 1000 ms RAPL window.

a plot that is stable for a given deterministic benchmark and input. Power consumption behavior, on an uncapped processor, is referred to as the characteristic power consumption of the application. Figure 4 shows the plots for a configuration of LULESH, AMG, CoMD, Nekbone, and MiniFE. Due to the configuration parameters selected, the AMG (Figure 4d) and CoMD (Figure 4c) power consumption is lower than expected for most HPC applications. Of the benchmark configurations plotted, LULESH has consistently high power consumption, consuming 80 to 100 watts per socket during execution. The MiniFE plot is perhaps the most interesting since the plot shows multiple application phases that have different levels of power consumption. AMG also demonstrates a phased power consumption behavior, through the difference between high and low phases is less pronounced.

Decreasing Bounds. In the decreasing bounds experiments, the objective was to observe the relationship between application runtime and power consumption under progressively lower power caps. The first goal from these experiments was to understand in what cases a RAPL power cap would impact application performance. RAPL's behavior was somewhat mysterious when compared with the dominate DVFS techniques due to RAPL power caps only impacting runtime in some cases. A secondary goal from these experiments was to develop a characterization for the amount of runtime impact that an application could expect to experience. The runtime impact of a power cap, extending the total time to complete an execution, will be referred to as *runtime dilation*.

Figure 5 plots benchmark runtimes for different power caps. While some benchmarks produce smoother curves than others, the smoothness of the curves are



(a) LULESH 115 Watts

(b) MineFE 115 Watts

(c) CoMD 115 Watts

(d) AMG 115 Watts, config 1

(e) Nekbone 115 Watts

(f) AMG 115 Watts, config 2

Figure 4. Consumption when power allocation matches the processor TDP. Sampling at 100 ms intervals with a 1000 ms RAPL window.

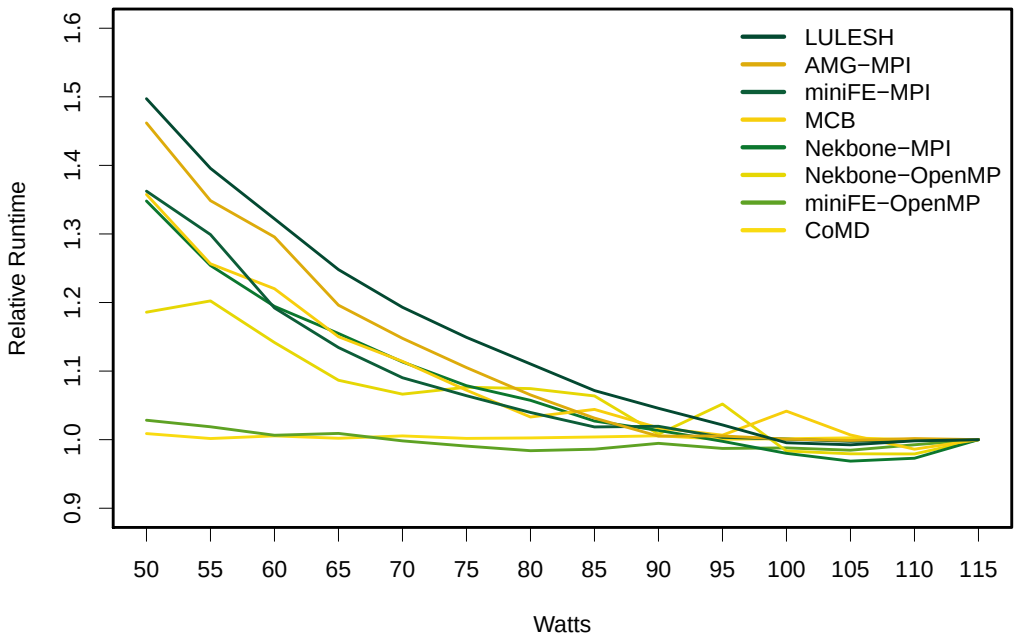
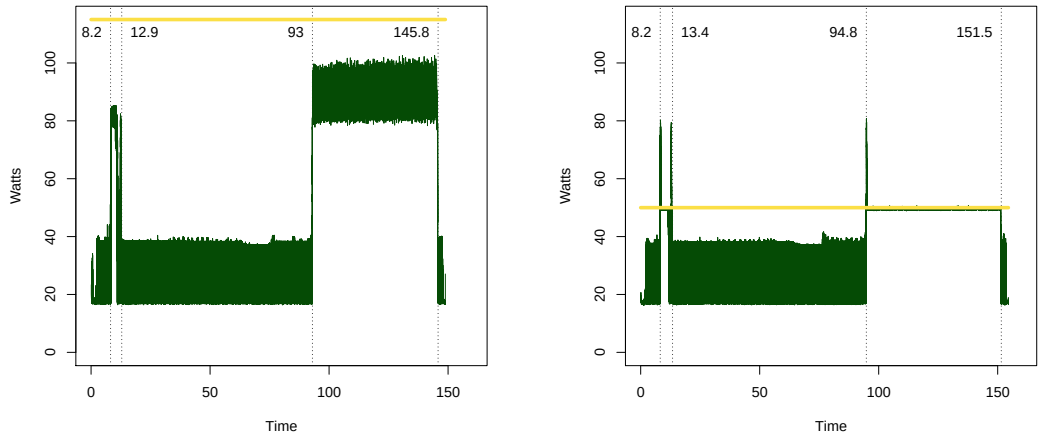


Figure 5. Runtime effect of decreasing power bounds for 8 benchmarks and parameters.

not studied in this work[10]. However a clear trend can be seen in Figure 5, a linear reduction in power does not produce linear runtime dilation. When the power cap is sufficiently low, runtime dilation appears to follow a polynomial curve as the bound is further reduced.

Table 7 looks at the energy used and runtime for the AMG (Config 2), LULESH, MiniFE, and Nekbone benchmarks. The characteristic power consumption for these benchmarks can be seen in Figure 4. An interesting insight from the table is that runtime dilation occurs before the power cap is reduced beneath the average power. MiniFE and Nekbone are interesting since average power continues to be beneath the power cap even as the runtime dilation

[10]The non-smooth curves are speculated to occur due to non-determinism in the benchmark that leads to different power consumption on different runs. Further investigation was not conducted since individual application performance and characteristics are out of scope for the work, instead this work focuses on generalized application performance trends relevant to the power scheduling problem.



(a) MineFE 115 Watts (b) MineFE 50 Watts

Figure 6. Consumption when power allocation matches the processor TDP. Sampling at 100 ms intervals with a 1000 ms RAPL window.

increases. To understand application runtime behavior when power is capped, it is necessary to look at the data as a time series rather than an aggregate energy or average power value.

Figure 6 shows smear plots for MiniFE capped at 50 watts and effectively uncapped, the plot also labels the time phases are entered and exited. Computing the duration of the application phases, the runtime increases are localized to particular application phases. Specifically, phases for which the characteristic power consumption exceeds the power cap, runtime dilation is observed. Phases for which the characteristic power consumption is less than the power cap do not experience runtime dilation. Fukazawa et al. (2014) report a similar finding in "Power Consumption Evaluation of an MHD Simulation with CPU Power Capping" when studying a single HPC code. Average power is, hence, not a good estimate for the power needed by HPC applications with phased behavior.

App	Cap	Used kJ	Runtime	Avg W	Relative Time
amg	115	154.77	146.36	66.09	1.00
	110	154.51	146.09	66.10	1.00
	100	154.56	146.25	66.05	1.00
	90	152.49	147.01	64.83	1.00
	80	195.22	155.80	78.31	1.06
	70	184.55	167.87	68.71	1.15
	60	179.16	189.49	59.09	1.29
	50	168.88	213.80	49.37	1.46
lu	115	163.79	116.47	87.89	1.00
	110	163.58	116.12	88.05	1.00
	100	164.18	117.00	87.71	1.00
	90	165.30	122.35	84.44	1.05
	80	160.93	129.91	77.42	1.12
	70	152.69	139.59	68.37	1.20
	60	145.70	154.68	58.87	1.33
	50	138.28	175.17	49.34	1.50
minife	115	117.84	147.53	49.92	1.00
	110	117.42	147.03	49.91	1.00
	100	118.55	149.32	49.62	1.01
	90	117.65	148.52	49.51	1.01
	80	110.94	146.92	47.19	1.00
	70	103.51	149.06	43.40	1.01
	60	95.72	150.28	39.81	1.02
	50	88.75	153.56	36.12	1.04
nek	115	94.29	62.96	93.60	1.00
	110	93.86	61.26	95.77	0.97
	100	81.35	61.71	82.39	0.98
	90	65.58	63.82	64.23	1.01
	80	61.23	66.57	57.48	1.06
	70	56.69	70.10	50.54	1.11
	60	52.52	75.20	43.65	1.19
	50	50.05	84.88	36.86	1.35

Table 7. Runtime and energy for AMG (configuration 2), LULESH, MiniFE, and Nekbone under varying power caps.

Summary of Experiments. Several useful observations can be made from the data presented in this section regarding the behavior of applications running on processors supporting power capping. Applications have a characteristic power consumption associated with their execution that is application specific. Power caps above the characteristic power consumption have no observable effect on runtime or power consumption. Power caps beneath the characteristic power consumption will extend application runtime. Additionally, runtime impacts are only observed during application phases where the unbounded characteristic power consumption would exceed the current power cap. Finally, the runtime impact of a linearly decreasing power cap is not linear. Linearly decreasing the power cap appears to lead to a polynomial increase in runtime that is directly related to the difference between the characteristic power consumption and the current cap.

While less exciting from a research perspective, the dynamic response experiment verifies that RAPL power caps can be adjusted without application coordination and the correct bounding effect will occur. This result is of practical importance since a power scheduler claiming hard enforcement must use a power control mechanism capable of enforcing component level power caps. RAPL has been shown to be such a mechanism.

Connecting Programs and Power

The results in the previous section show that a relationship must exist between an executing program text and the power consumption of the hardware. Considerations, such as computation correctness and time, to solution are generally more important than how a computation is physically performed, resulting in programmers having a better understanding of the mapping from code to performance that the mapping from code to power or energy. Modern processors

are complex and the role of optimizing compilers make a detailed mapping from program text to power intractable. Numerous simplifying assumptions will be used in the discussion to keep the model easy to follow. Instead of making a detailed mapping, this section develops an intuition to account for the generally observed trends from the previous section. These intuitions will be needed to understand the content of Chapters IV and V.

Reasoning about the power consumption of a hardware component will starts understanding the processor as a collection of transistors. A transistor can be thought of as a switch, turning on and off current in a circuit. Changing the state (i.e., activation) of an individual transistor requires a small amount of power, commonly referred to as the switching power. A small amount of power is also lost due to leak current by transistors. The specific amount of power lost due to leak current and switching power is related to the voltage applied, higher voltages resulting in higher loses. Since modern processors are composed of billions of transistors that change state up to billions of times per second, the aggregate amount of power consumed can be substantial.

Transistors are grouped together to form logic gates. Each logic gate implements one boolean operation (e.g., *and*, *or*, *not*, etc.). More complex operations can be built by composing several logic gates into a logic unit. Logic gates can also be used to route inputs to logic units and outputs from logic units within the processor.

Each CPU core will be implemented using some number of logic units. One of the units in the core will be responsible for *decoding* the operation to perform. Conceptually, CPUs are generally though of as *decoding* and executing

one instruction every clock tick[11]. The program is experienced by the CPU as a stream of hardware instructions that must be decoded and, based on the specific instruction, executed using different logic units. Note that different logic units involve differing numbers of transistors and, therefore, are expected to consume different amounts of power. In a multi-core computer, several CPU cores will be on a single processor and may share some logic units (e.g., memory controller, or cache).

Computation physically occurs as the sequence of transistor activations over time[12]. One can consider the algorithm to define all possible instruction sequences for a computation, but a single executing computation will only ever use one of the possible sequences defined by the algorithm. If the number of instructions needed for a computation could be determined in advance, program execution could be measured by the number of instructions executed versus the total number needed to complete the execution. Since instructions ultimately map to transistor activations, program execution could also be measured by the number of transistor activations that have occurred versus the total number needed to complete the computation. Additionally, there must be a time ordering to which transistors may activate due to dependencies between instructions in the instruction stream.

During all phases of execution, applications on processors without a power cap, will use as much power as the application can induce on the processor. The low power phases represent periods where the instruction stream, due to instruction dependencies or the logic units involved, make use of relatively few concurrent

[11]The reality is much more complex with hyperthreading, pipelining, out-of-order execution, and micro-instructions all being done by the processor.

[12]A tempting alternative would be to claim that the computation is represented by the algorithm. The algorithm does represent the computation to be done in the general sense however the computation itself requires executing the algorithm.

transistor activations per unit time. The high power phases represent periods where the instruction stream is able to make use of significantly more concurrent transistor activations per unit time. Looking at the results of power capping, the runtime dilates since the processor internally uses DVFS as a primary mechanism to maintain the processor power limit. The specific amount of dilation is hard to compute directly since computing the needed slowdown would involve detailed knowledge of the concurrent transistor activations to be controlled. Though a specific dilation is hard to compute, the general trend observed is that a polynomial increase in runtime is observed for a linear decrease in power.

Chapter Summary

In this chapter power monitoring, observed impacts of power limits, and proposed an intuition to understand the observations were discussed. PowMon was needed to support this work since hardware based approaches to power monitoring would not scale and existing tools monitored at the incorrect granularity. Using PowMon a study was conducted on the power behavior of several HPC benchmarks to provide insight into the behavior of power capped execution that was missing from the HPC literature. Finally, an intuition connecting the electrical properties of the hardware to an algorithm being executed was given and will be needed to support Chapters IV and V.

CHAPTER IV

DYNAMIC POWER SCHEDULING

This chapter contains ideas and themes that have been previously published in D. Ellsworth, Patki, Perarnau, et al. (2016); D. Ellsworth, Patki, Schulz, et al. (2016); D. A. Ellsworth et al. (2015a, 2015b). PowSched and its analysis are my original work. My co-authors assisted in developing the language and narrative to describe PowSched and its analysis for the previously published work. Allen Malony produce Figure 7 as part of the narrative development for D. A. Ellsworth et al. (2015a).

Chapter II introduced hardware overprovisioned systems as a technique to better utilize power; a technique that relies on energy efficiency and power scheduling. Additionally, Chapter II presents the power scheduling invariant, which formalizes what a power scheduler must do to guarantee system wide power consumption remains within a given bound. Chapter III presented the technique used for power measurement and discussed the power behavior observed in HPC applications. In this chapter a power scheduler, PowSched, suitable for hardware overprovisioned HPC systems and based on the ideas in the preceding chapters will be discussed.

Design Discussion

The envisioned deployment context motivating PowSched is an HPC system where exceeding the systemwide power bound would cause physical harm to the computer or data center[1]. While individual application runtime performance is

[1]PowSched does not consider power limitations at finer granularity that the HPC system (e.g. rack level power distribution limitations). Physical power distribution within a system is expected

still of concern in for the owning organization, the cost of a system outage is high enough to admit power scheduling solutions that significantly degrade application runtimes or abruptly terminate jobs running on the system. Having an application finish late, or never finish, is clearly preferable to the computer being unavailable for hours or weeks due to repairs.

The importance of maintaining the system power limit led to an early design decision to separate the power and job scheduling activities in the HPC system. Co-mingling the job and power scheduling activities creates a multi-objective optimization problem, increasing the complexity of constructing and reasoning about the scheduler. Instead, the PowSched system model assumes that the power and job schedulers are completely disjoint system components. For any job schedule the job scheduler selects, the power scheduler must be able generate a power schedule that keeps the system within the systemwide power bound. From the enforcement perspective separating power and job scheduling does not change the power scheduling problem. There is an expectation that having the power scheduler and job scheduler collaborate, rather than operating completely independently, could result in better overall system throughput[2]. Even if the job and power scheduling activities were merged into the same software, the power scheduling logic would still need to protect the system from bad job scheduling decisions that would cause an uncapped system to exceed the power bound. Effectively, the job and power scheduler must separate but interacting subsystems even when merged into a

to also expected to be a challenge but this finer power control problem is out of scope for the current work.

[2]Performance optimizations involving collaboration between the power scheduler and job scheduler are not studied in this work.

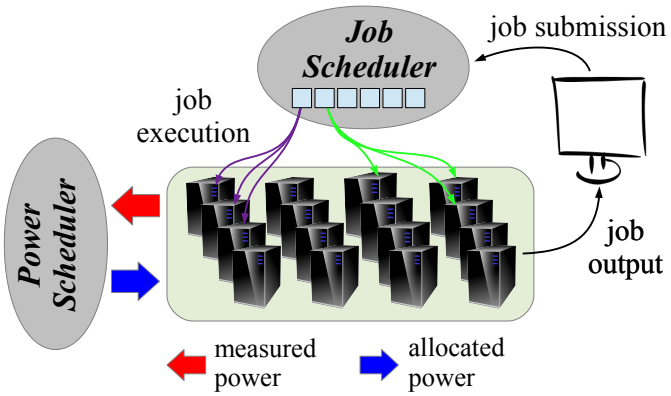


Figure 7. Model relating the cluster, job scheduler, and power scheduler. Graphic created by Allen Malony and reproduced from D. A. Ellsworth et al. (2015a).

single program to provide hard enforcement. Figure 7 shows a highlevel diagram of the PowSched HPC system model.

In competing power scheduling work, the job scheduler plays an active role in power scheduling activity. When scheduling a job in these solutions, a power estimate is used to reserve sufficient power for the job to complete without performance impact. The system never becomes *overloaded* with respect to power because the job scheduler would never knowingly start work when insufficient power exists to service the work. When power and job scheduling are decoupled, the power scheduler must be able to response sensibly to overload since the job scheduler is unaware of any power constraints.

A significant portion of the work on energy efficiency and power scheduling involves traced based models of the individual applications to be run. For a production power scheduler, needing application trace data to make power scheduling decisions can be problematic. The first run of any application cannot have a valid trace based model since there has been no prior execution to observe and build the model from. Even if the application has been run previously, the behavior for a given input may diverge from the previously observed executions.

While a future work extending PowSched may be able to use application models to good effect when available, the current work makes power scheduling for first application runs a first class use case and eschews application specific models.

Temporal resolution of measurements and responses can be a significant problem in a distributed system. For a 3 gigahertz processor, light in a vacuum can only travel roughly 10cm per clock cycle. The transmission latency alone, to move data from side of the HPC system to the other, is significant in comparison to the rate of computation within node processors. Direct control of components by the systemwide power scheduler at realtime operating frequencies is practically impossible due to physics. Instead, the systemwide power scheduler must make decisions at a relatively low frequency and leave high frequency adjustments to the individual hardware components. PowSched assumes a hardware technology, like RAPL, is available on each component to keep power component consumption beneath the last received allocation.

Changing power allocations must be done carefully to avoid exceeding the power bound due to latencies. Realtime synchronization is hard to achieve between nodes in an HPC cluster and the latency between the systemwide power scheduler and the components may differ by component. Naively setting new allocations as they are computed or in a single pass may cause the systemwide power limit to be exceeded. Power allocations, therefore, should occur in two phases. The first phase should send the new allocations to all components receiving a smaller allocation. After all acknowledgements from the first phase have been received, the second phase can then send the new allocations to all components receiving a larger allocation. Two phase allocation will be discussed more when proving PowSched provides hard enforcement.

Scheduling Heuristic. The vast majority of power schedulers in the literature use trace based application specific models to generate power allocations. Application specific models enable the power scheduler to predict the amount of power to be consumed and produce schedules that make that power available. As discussed in Chapters II and III, an insufficient power allocation increases application runtime and the power required varies by application. Unfortunately, generation of good application model requires tracing several executions of the application.

Rather than identify application specific behaviors, PowSched makes scheduling decisions based on a heuristic informed by the generalized effect of power caps on program execution (discussed in Chapter III). The following four observations have particular relevance for the design of PowSched:

1. Each application execution has a characteristic power consumption.
2. Power consumption usually remain stable during application phases.
3. Allocating more power than the characteristic consumption does not improve performance.
4. Allocating less power than the characteristic consumption reduces performance nonlinearly.

PowSched can reduce the power allocated to components observed to consume less power than the component's current allocation. Application phases are expected to have relatively stable power consumption, which makes the last observed power consumption a good estimate of the power consumption expected for the rest of the application phase. Reducing the power allocation to be near the observed consumption is not expected to reduce application performance since

the characteristic consumption during the current application phase is roughly the observed consumption.

PowSched should increase the power allocated to components where the characteristic power consumption would be greater than the current allocation. Without an application model, PowSched cannot directly estimate wether a component could use more power or not. During application phases where the characteristic power consumption would be greater than the current power allocation, the hardware power controller will use up to the current allocation. When confronted with consumption near the allocation, PowSched will assume that the component would use more power if additional power was allocated.

Ideally, the HPC system will never be overloaded and all applications will be powered at their characteristic consumption. Due to the lack of coordination between the power and job schedulers, PowSched must be prepared to handle the system being overloaded. PowSched attempts to resolve overload by converging towards a fair power allocation (i.e., one in which all components are allocated the same amount of power). Using this heuristic, in the worst case, PowSched is expected to converge to the power allocations that a naive static systemwide power scheduler would produce.

The heuristics used by PowSched operates on the hardware components and is application agnostic. It is sufficient to know that an application specific characteristic power consumption exists to derive the heuristic for increasing and decreasing allocations, knowledge of the specific characteristic power consumption for running applications is not necessary. Using these heuristics, PowSched is able to operate on the hardware components without any details regarding the work executing on them.

Algorithm. The highlevel algorithm used by PowSched is simple and has three conceptual phases. In Phase 1, the current power consumption is gathered by the power scheduler. The scheduler operates at a coarse temporal resolution with respect to the computation and power consumption within phases is expected to be fairly consistent, making small differences in the collection time between nodes acceptable for making scheduling decisions. In Phase 2, the current component power readings are compared against the last allocations and the new power schedule (i.e., component level allocations) is generated. Only the components receiving smaller allocations are sent new allocations during phase 2. Phase 3 begins when the components receiving smaller allocations in Phase 2 have acknowledged that the new lower allocations have been applied. All components receiving larger allocations are then sent the new allocations during Phase 3. Following Phase 3, the power scheduler waits for the reset of the scheduling interval. Algorithm 2 provides pseudocode for the algorithm backing PowSched.

Each component subject to power scheduling is represented in PowSched as two numbers; a consumption number collected from the component to the scheduler and an allocation number sent from the scheduler to the component. Phase 1 is responsible for collecting and updating the consumption numbers for each component in the scheduler (see Algorithm 3). Phase 2 will generate the new allocations for all components receiving smaller allocations based on the previously discussed heuristics (see Algorithm 4). When no components can have their allocation reduced in phase 2, power is taken from all components consuming more than the average power. Phase 3 redistributes the power reclaimed in Phase 2

Algorithm 2 PowSched logic in pseudocode

```
q ← target w_i
C stores {c_0, ··· , c_{n−1}}
A stores {a_0, ··· , a_{n−1}}
M stores {m_0, ··· , m_{n−1}}
numdown ← count of nodes yielding power
interval ← scheduling interval
reclaimfactor ← power to reserve when stealing

procedure MAIN
    while True do
        getReadings                                  ▷ Phase 1
        allocDown                                    ▷ Phase 2
        allocUp                                      ▷ Phase 3
        sleep rest of interval
    end while
end procedure
```

to all components that are consuming near the component's current allocation (see Algorithm 5).

Algorithm 3 PowSched phase to gather recent consumption

```
procedure getReadings
    for all sockets do
        Update c_i with the current reading
    end for
end procedure
```

A common question when discussing PowSched is the use of direct feedback rather than proportional integral derivative (PID) controller logic when determining allocations[3]. If PID logic was used, the PID control would need to be tuned to be conservative to avoid exceeding the systemwide power cap, which would extend the number of iterations required for convergence to a new good allocation. Unfortunately, the systemwide power scheduler operates at a glacial

[3]Responding directly has the potential for *flapping* when a phase length is approximately the same as the scheduling interval.

Algorithm 4 PowSched phase to reduce power alocations

```
procedure allocDown
    numdown ← 0
    for all sockets do
        if c_i < a_i − q then
            Update a_i to max{c_i + q, A_min}
            numdown ← numdown + 1
            Update m_i to False
        else
            Update m_i to True
        end if
    end for
    if numdown= 0 and Σ a_i + n ≥ L then
        for all sockets do
            if a_i > L/n then
                a_i ← a_i − (a_i − L/n) × (1−reclaimfactor)
                m_i ← True
            end if
        end for
    end if
    for all sockets do
        Set the socket to limit a_i
    end for
end procedure
```

Algorithm 5 PowSched phase to increase power alocations

```
procedure allocUp
    u ← (L−Σ a_i) / (n−numdown)
    for all sockets do
        if m_i then
            a_i ← min{a_i + u, A_max}
        end if
    end for
    for all sockets do
        Set the socket to limit a_i
    end for
end procedure
```

speed compared to the individual components due to the communication latencies. Additionally, a power target is only valid until the end of an application phase so long convergence time may also cause the majority of a phase to run with a suboptimal power configuration. Using a mechanism like RAPL, PowSched does not need to attempt to converge particular components to power targets. Instead, PowSched is able to directly set a component's power target and allow the component to determine how to best achieve the target power.

Hard Enforcement. Power schedulers providing hard enforcement must be able to prove satisfaction of the power scheduling invariant (Equation 2.5). As discussed in Chapter II, the power scheduling invariant has two parts that must be addressed. First, the power scheduling mechanism must guarantee that the power actually consumed by a component never exceeds the power allocation. Second, the power scheduling algorithm must provably guarantee that the aggregate power allocations never exceed the systemwide power limit. By the manufacturer documentation and experiments in Chapter III, PowSched claims that the RAPL mechanism provides the needed guarantee for each component. The PowSched algorithm, however, must still be proven to provide the needed algorithmic guarantee across components.

Proof by induction is used to show the PowSched algorithm satisfies the algorithm component of the power scheduling invariant. Description of the power scheduling invariant can be found in Chapter II and the symbols used are defined in Table 4. First, the base case, a safe initial configuration, will be presented and shown to satisfy the invariant.

A fair assumption for the algorithm is that the initial power allocations will satisfy the power scheduling invariant. One simple way to enforce this would be for

the power allocation to be set evenly across all components when the system starts. If the systemwide power limit is L and there are n components, then the initial per component power allocation would be $\frac{n}{L}$. Trivially, this initial condition satisfies the requirement that $L \geq \sum a_i^t$.

In phase 1, no power allocations are made. Trivially, phase 1 does not impact the relationship between the systemwide power limit and the aggregate power allocations.

In phase 2, strictly smaller power allocations are made. For all components, the previous allocation, a_i^t, will be less than or equal to the upcoming allocation, a_i^{t+1}. Since $L \geq \sum a_i^t$ and $\forall i, a_i^t \geq a_i^{t+1}$ it must be the case that $L \geq \sum a_i^{t+1}$. Additionally, at the end of Phase 2, there exists some strictly positive amount of *surplus* power, S, such that $L \geq S + \sum a_i^{t+1}$ and $S = \sum a_i^t - a_i^{t+1}$. Clearly, Phase 2 preserves the power scheduling invariant.

In Phase 3, strictly larger power allocations are made. The amount of power distributed in Phase 3 is bounded by the amount of surplus power reclaimed in Phase 2. Surplus power is evenly divided between the components receiving a larger power allocation. If k components are receiving larger allocations in Phase 3, each component will receive an additional allocation of $\frac{S}{k}$. For each of the k components receiving a larger allocation, the allocation, a_i^{t+2}, at the end of Phase 3 will be $a_i^{t+1} + \frac{S}{k}$. The total power allocated at the end of Phase 3 will therefore be $k\frac{S}{k} + \sum a_i^{t+1}$, which matches the satisfying $L \geq S + \sum a_i^{t+1}$ allocation at the end of Phase 2.

At the beginning and ending of each phase, the PowSched algorithm maintains the power scheduling invariant. Since Phase 1 follows Phase 3, the power scheduling invariant is maintained throughout scheduler execution.

A natural question at this point would be why transmitting the allocations computed in Phase 2 and Phase 3 must be done separately. From an algorithmic proof perspective, the separation into two distinct transmission phases seems unnecessary. One could logically compute the down allocations (phase 2) and the up allocations (phase 3) and transmit all of the allocations at the same time. Due to real world latencies and uneven delays between nodes sending all allocations at once is unsafe. Practically, the separate phases are critical to providing a guarantee that the systemwide power bound is never exceeded.

To highlight the danger in transmitting up and down allocations at the same time, consider the case of a system with a power limit of 3 and two components A and B with allocations a_A and a_B respectively. At time t, $a_A^t = 2$ and $a_B^t = 1$. The scheduler computes the allocations for time $t+1$ as $a_A^{t+1} = 1$ and $a_B^{t+1} = 2$ and sends the new allocations at the same time. Some unpredictable delay will be experienced as the allocations travel from the scheduler to the components. If component B receives and applies the allocations for time $t + 1$ before component A, then there will be an interval in which the aggregate allocated power will be $a_A^t + a_B^{t+1} > L$.

For this reason, PowSched implementations must receive acknowledgement of down allocations (Phase 2) before sending up allocations (Phase 3). Unacknowledged up allocations are safe since components using the previous allocation can only consume less power than the new allocation would permit.

Implementations

The PowSched algorithm has been implemented several times using different supporting technologies. PowSched performance has been primarily evaluated using an MPI based implementation. The PowSched algorithm has also been

implemented using SOS[4], Glasgow Cache (Cache)[5], and on BEACON[6]. Only the MPI and Glasgow Cache implementations will be discussed in this chapter.

MPI. The MPI implementation of PowSched is a single MPI application and directly follows Algorithm 2. PowSched is started as a separate MPI job, with one process per node. On startup, PowSched sets all component power caps to a fair share allocation. Readings are collected using *MPI_Gather* and allocations are distributed using *MPI_Scatter*. The timing mechanism used to maintain the monitoring interval in PowMon (see Chapter III) is also used by PowSched to maintain a uniform scheduling interval. The MPI implementation does not have a mechanism for shutdown and relies on the system job scheduler to kill the PowSched processes when the run is complete.

Glasgow Cache. To investigate if tight coupling between phases is required, PowSched was implemented using a publish/subscribe platform with loosely coupled applications. The Glasgow Cache implementation of PowSched is in C and deviates from Algorithm 2 by decoupling power measurement, scheduling, and allocation. Phase 1 is done by the power monitor in the Glasgow Cache implementation and one monitor is started per node participating in power scheduling. The generation of power allocations is done by the power scheduler, which is started on a single node. Allocations are applied by the power actuator, which is implemented, for simplicity, using MPI rather than Glasgow Cache to signal the down allocations have been successfully applied. Coordination between

[4]Scalable Observation System, SOS, is an ongoing project at the University of Oregon to build a streaming introspection platform for HPC systems. More information on SOS can be found in Wood et al. (2016).

[5]Cache is a high performance publish/subscribe middleware system. More information on Glasgow Cache can be found in Sventek and Koliousis (2012).

[6]BEACON is the publish/subscribe layer in the Argo ExaOS/R project. More information on the Argo ExaOS/R projectcan be found in *Backplane* — (n.d.).

the three applications is done using only the publish/subscribe mechanism exposed by Glasgow Cache.

Communication in the Glasgow Cache implementation uses the following pattern: At regular and unsynchronized intervals, the power monitor on each node publishes the current energy reading via the Glasgow Cache infrastructure. At the scheduling interval, the power scheduler retrieves the latest readings from the infrastructure for all of the monitors and publishes new allocations to the infrastructure for each node. Once the complete schedule has been published, the power scheduler publishes a message indicating the schedule is complete. The lead power actuator subscribes to the completion message, which is received with the new allocations for all of the nodes. *MPI_Scatter* is then used by the lead power actuator to set the down and up power allocations safely on all of the nodes[7].

Results

A series of experiments with the MPI PowSched implementation were conducted on the Cab system at LLNL[8]. Results from these experiments are presented in this section first. Next, results from a scaling study PowSched's computation and communication costs, performed on the vulcan system an LLNL, are presented. Vulcan does not support libMSR, so the PowSched implementation stubs out reading with a random number generator and does not actual set power caps, however new allocations are computed and transmitted based on the readings gathered. This section concludes with results using the Glasgow Cache PowSched

[7]Implementing the synchronization needed could be done using Cache but was not for simplicity. Actuators would subscribe to a *set_allocation* event and would publish an *allocation_set* event after making the change. Cache automata would use the *allocation_set* events to determine that the down allocations completed before publishing *set_allocation* events for the up allocations.

[8]Cab is a commodity Linux cluster. For more information see `https://computation.llnl.gov/computers/cab`.

App	Nodes	App Only	+POWmon	@115W	@dyn	≈ Overhead
LU	16	119.77	119.84	120.99	121.25	0.01
LU	4	112.39	112.92	112.05	113.30	0.00
CoMD	16	107.1491	105.3836	107.3378	107.0001	0.00
CoMD	8	109.3181	109.2498	109.9474	110.1558	0.01
CoMD	4	92.4329	91.9755	92.2450	92.7113	0.00
AMG	16	102.573688	103.323772	103.71112	103.71112	0.00
AMG	8	88.667316	88.173036	89.631203	90.110953	0.01
AMG	4	76.821048	76.763169	77.002957	76.873345	0.00

Table 8. Runtimes reported by the benchmarks in seconds. PowSched @115W run forces PowSched to assign 115W per socket over the lifetime of the job. PowSched @dyn allows PowSched to dynamically adjust the per socket allocation with a global bound permitting 115W per socket.

implementation to show that tight coupling of the power reading and setting mechanisms may not be required.

Overhead. PowSched shares resources with the applications running on the cluster and may contend with the applications running on the controlled nodes, reducing application performance. A first experiment for PowSched is to measure the impact of running PowSched and PowMon versus running an application alone on the test machine. Table 8 presents the runtimes, as reported by three CORAL benchmarks, for invocations on 2, 4, 8, and 16 nodes with different modes of monitoring and scheduling enabled. Runtimes were perturbed by less than 0.1% compared to application running alone. PowMon and PowSched together appear to interfere negligibly with applications.

MPI Experimental Results. The first set of the experiments used 128 Cab nodes. Logically, the 128 nodes can be thought of as being partitioned into 8 enclaves, each containing 16 nodes (32 sockets with 8 cores each). During each experiment, all enclaves will run simultaneously and each enclave will run a workload of two benchmark apps in sequence, with a 10 second sleep between

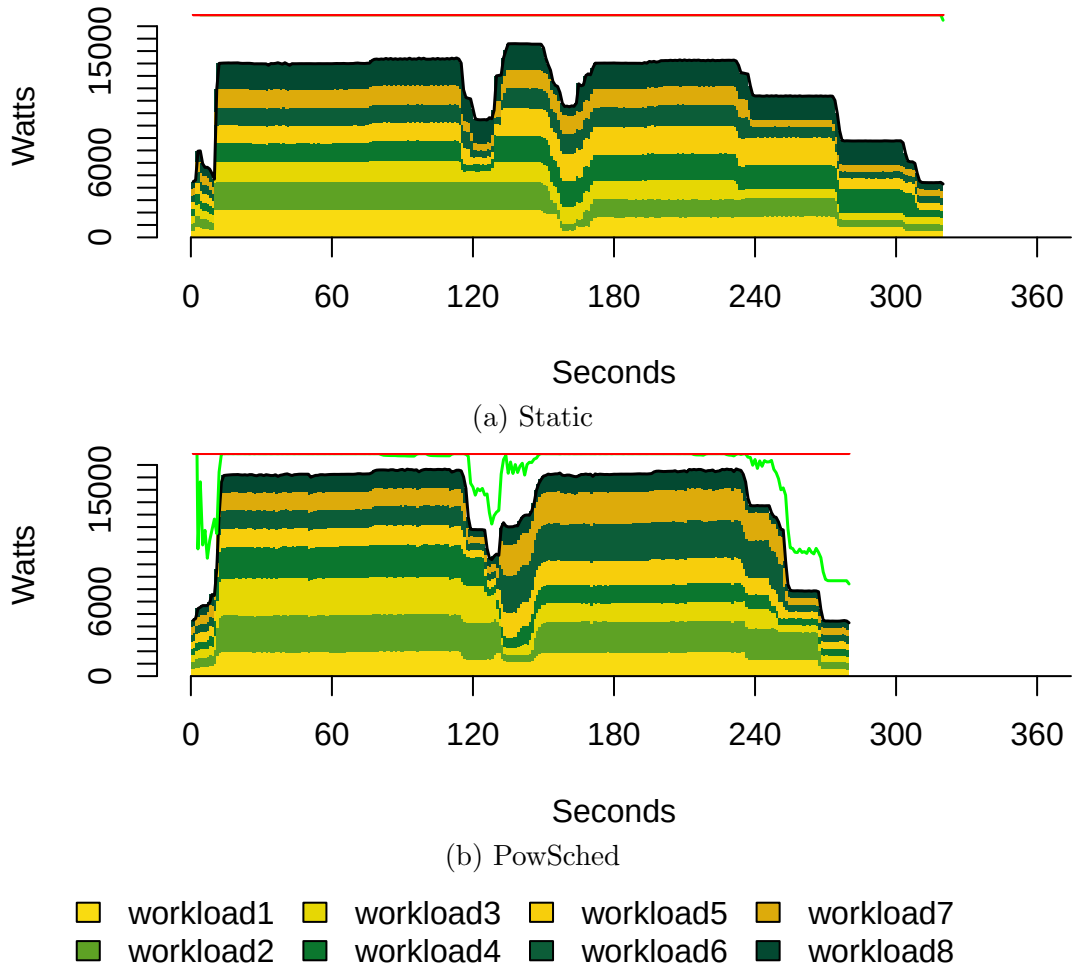
benchmark apps. Workloads of this form are chosen to ensure a window of unevenness in the maximum power consumption, per node, during the experiment run. The sleep also simulates the window of time expected between completion of one job and the system job scheduler starting another job on the nodes. Workloads with fixed node counts per workload are used rather than individual jobs due to complexities of running and tracking concurrent subjobs in existing job schedulers.

Figures 8, 9 and Table 9 use workloads with three application benchmarks (AMG, LULESH, and CoMD). For experiment control, each workload was run with each socket receiving the maximum power allocation, 115 watts. 115 watts is expected to result in the shortest possible runtime. Experiments where also run where each socket received a specific power allocation (90 watts, 70 watts, and 50 watts), simulating the naive static power scheduling strategy. The static runs provide a baseline for comparison between PowSched and a system where each active socket is given an equal static allocation based on the global power available (e.g., a system has a global power bound of 17,920 watts and 256 active sockets, resulting in an average allocation of 70 watts per socket). The runs with a fixed per socket allocation will be referred to as *static*. The experimental runs using PowSched will be referred to as *dynamic* and rely on the same total system power bound as the corresponding static run.

Figure 8 shows total power allocation across the 8 jobs for a 50 watt average bound using static and dynamic scheduling. Table 9 shows the runtime impact of PowSched over 10 runs at each bound with outliers removed. These results use a scheduler interval and RAPL window of 1 second. The time to complete all workloads with PowSched, when power is constrained, is better than the static schedule. We also note from Table 9 that PowSched clearly is not attempting

Experiment	Runtime	dev	Imp.	kj Alloc	dev	kj Used	dev
115W static	278.26	9.57		8192	282	4008	98
115W dynamic	276.24	4.84	0.7%	5475	53	3977	37
90W static	284.63	3.20		6572	73	3985	30
90W dynamic	277.13	5.04	2.6%	5339	66	3980	47
70W static	323.83	4.90		5829	87	3904	34
70W dynamic	278.02	4.97	14.1%	4638	69	3985	38
50W static	401.76	5.47		5178	73	3938	38
50W dynamic	371.92	13.23	8.7%	4562	124	4016	79

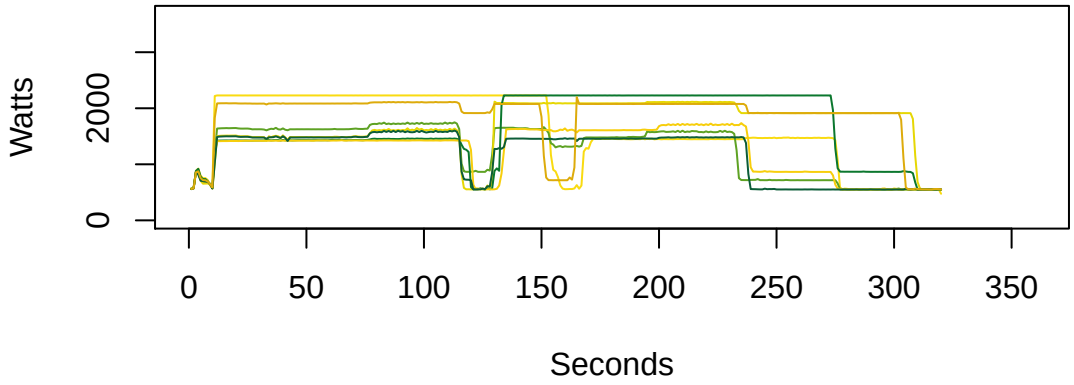
Table 9. 128 nodes, 16 nodes workloads per workload, 10 runs, same workload for all runs reported with improvement percent and energy.



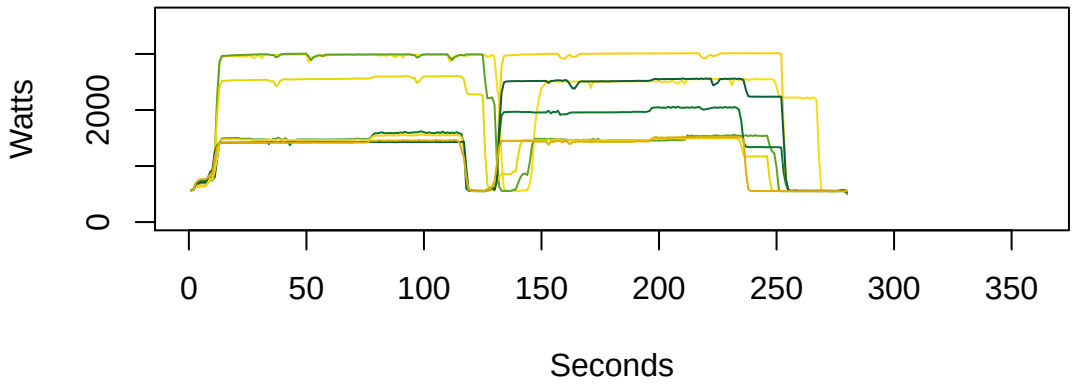
(a) Static

(b) PowSched

Figure 8. Workload consumption and global bound for a 128 node cluster using an average of 70 watts per socket.



(a) Static



(b) PowSched

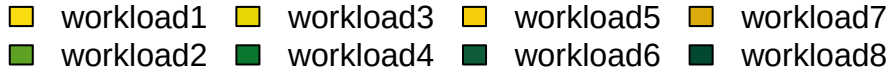


Figure 9. Workload consumption over time for a 128 node cluster using an average of 70 watts per socket.

Benchmark	Domain	Processes
LULESH	Shock Hydro	27
miniFE	Finite Element	8
miniFE	Finite Element	64
AMG	Linear Solver	32
AMG	Linear Solver	64
MCB	Monte Carlo	32
CoMD	Molecular Dynamics	32
Nekbone	Science App	8

Table 10. Benchmarks used for 8 node workloads in the 128 and 256 node experiments.

energy optimization. In all cases, roughly 4 megajoules are used to complete the workloads. The primary effect of PowSched is on the runtime relative to static required to complete all workloads.

Figure 9 shows per enclave allocation and consumption for the corresponding 50 watt runs, comparing static and dynamic. What is interesting to see is the dynamic spreading of power to workload applications that can use it, some of which end up consuming significantly above the 1,600 watts per enclave (50 watts per socket) constraint used by the static allocation.

Unallocated power is present as a side-effect of the greedy reclamation strategy and can be seen in Figure 8 as the space between the total allocated power and the global limit. There is no unallocated power in the static strategy since the full power limit is allocated across all sockets at all times. Several co-located clusters sharing a power infrastructure could potentially make use of unallocated power by shifting the power from one cluster to another. In such a scenario, the global system power limit is also a dynamic policy-driven value for each HPC cluster. Similar to shifting budget across clusters, a hierarchy of power schedulers in a single cluster might be used to achieve extreme scales.

Window Size Sensitivity. The results in Table 9 use a scheduler interval and RAPL window of 1 second. To investigate sensitivity to these settings, the workloads were re-run 10 times at each bound with scheduling intervals of 250, 500, and 2000 milliseconds and RAPL windows of 1 millisecond and 1 second. The small number of phases and mostly constant power consumption of the benchmark applications provide little opportunity to gain performance improvement using faster power scheduling response. The results obtained for varying scheduling interval and RAPL window size were inconclusive when considering the level of system jitter. Further investigation with different benchmarks may show sensitivity that the study conducted in this work did not find.

Scaling Experiment. Few large HPC platforms exist for experimenting with dynamic hardware enforced power bounding. However, there is a desire to understand the scaling performance to determine if PowSched would still be suitable for an extreme scale system. For this purpose, a modified version of PowSched was deployed on the Vulcan IBM BG/Q platform at LLNL[9] and the time PowSched spent in communication and computation was measured. Since the BG/Q platform does not support RAPL, we used random numbers for consumptions read. Use of random numbers should not disrupt the results, since the per node time to read from or write to the RAPL registers should remain constant[10]. The performance of PowSched at scale will be dominated by the time taken to communicate the per socket readings or the time taken to perform computation over the socket readings.

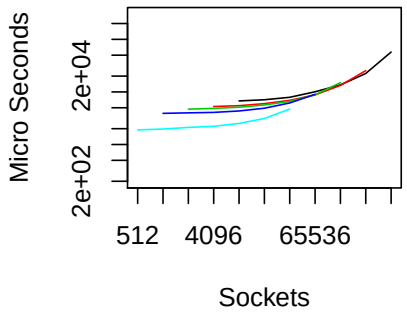
[9] Additional information on Vulcan can be found at `http://computation.llnl.gov/computers/vulcan`.

[10] Measurements we made on Cab show an average of 44 and 15 microseconds are required to read and set the RAPL registers, respectively, via libmsr.

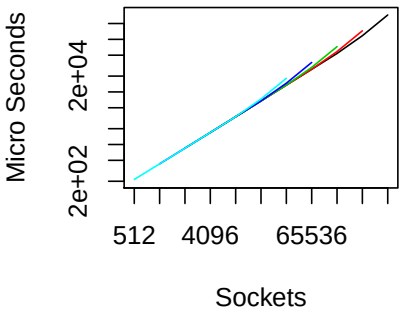
Each scheduler process launched represents a simulated node with 2 sockets and for each run we use 1, 2, 4, 8, 16, 32, or 64 processes per physical BG/Q node. BG/Q node counts from 1 to 8k are used to sweep the space from a single simulated node to 500k simulated nodes. Linear scaling for computation and slowly growing communication cost were observed (Figure 10). Linear scaling for computation is expected due to the linear scans conducted by the dynamic scheduler each interval. BG/Q's optimized network for low-latency and high-bandwidth MPI collectives results in slow all-gather communication time growth. Figure 11 shows the cross over region between computation and communication being the dominant time cost. Even at the largest number of simulated nodes, 512k simulated nodes, scheduler communication and computation complete in under 400ms. Depending on system scale and network performance, dynamic centralized power scheduling may be viable.

Glasgow Cache Experimental Results. The PowSched implementation using Glasgow Cache was run on cab to compare the performance against static power scheduling at node counts of 128 and 256 nodes. Due to limited machine time, each configuration was run only once. Table 11 presents the results obtained running the decoupled PowSched implementation using the same work loads from Table 9. Table 12 represents results comparing a more varied workload using 128 and 256 nodes. Table 10 summarizes the configuration of workload applications for the more varied 128 and 256 node experiments.

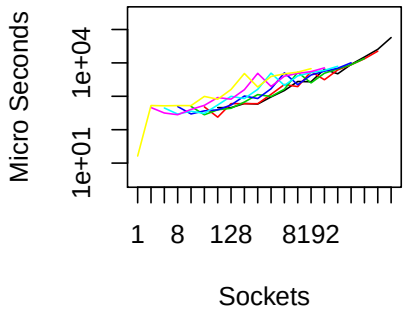
From the data provided in Table 12, decoupled PowSched appears to have poor performance when power is plentiful. The greedy reclamation of power combined with the lack of measurement synchronization are likely contributors to the performance penalty. Lack of measurement synchronization may cause sockets



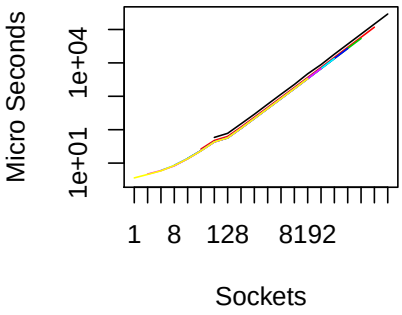
(a) Physical Communication



(b) Physical Computation



(c) Simulated Communication



(d) Simulated Computation

Figure 10. Lines represent the performance for physical node counts used from 8k to 512 nodes in (a) and (b). Lines represent simulated node count per physical node in (c) and (d).

Experiment	Runtime	kj Alloc	kj Used
115W static	278.26	8191.85	4007.80
115W dynamic	281.18	5662.18	3930.92
90W static	284.63	6571.76	3984.68
90W dynamic	284.49	5547.65	3961.87
70W static	323.83	5829.02	3904.29
70W dynamic	288.58	4830.906	3972.80
50W static	401.76	5178.29	3937.65
50W dynamic	381.33	4736.71	3822.73

Table 11. Comparison of 1 decoupled run with averaged runs using static.

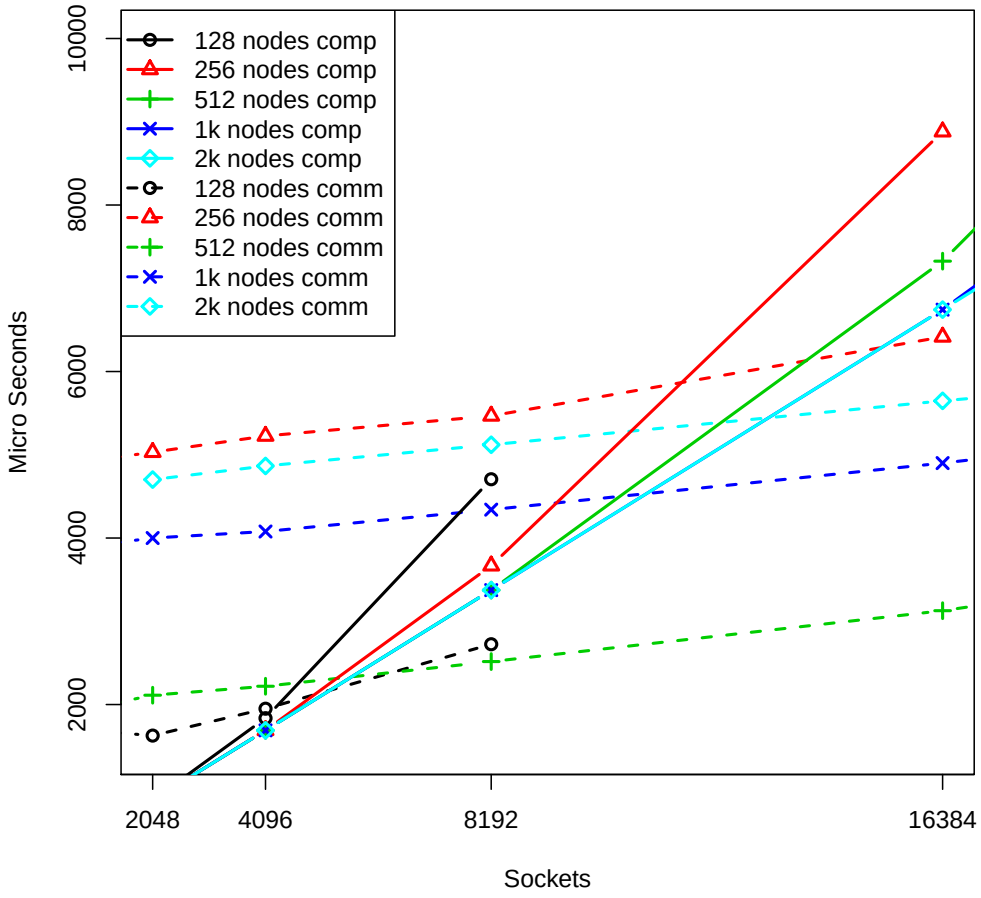


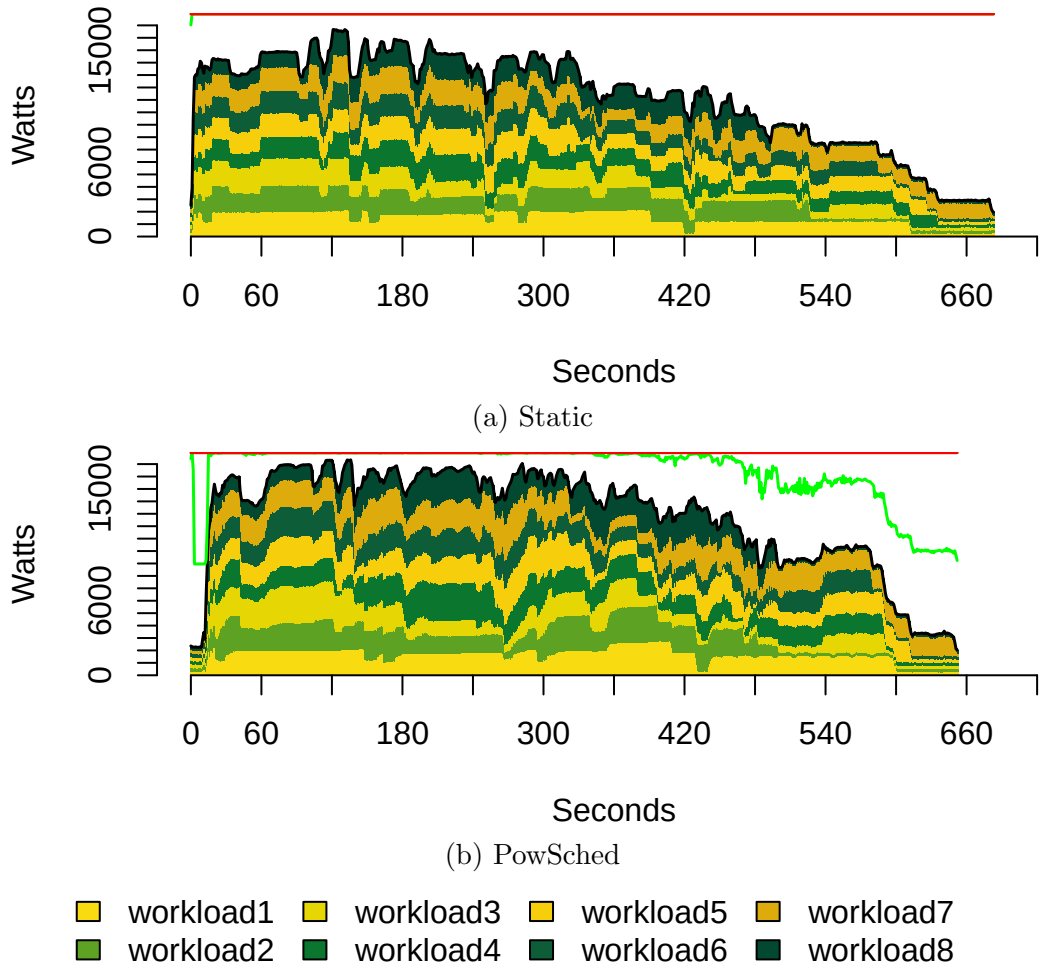
Figure 11. Crossover between computation and communication based on physical count.

	128 nodes			256 nodes		
Experiment	Runtime	kj Alloc	kj Used	Runtime	kj Alloc	kj Used
115W static	635.28	18735.93	8423.71	638.21	38025.55	16742.65
115W dynamic	651.29	12168.17	8529.40	650.98	24311.96	17001.13
90W static	631.92	14572.81	8217.15	631.31	29175.82	16266.65
90W dynamic	645.16	11947.70	8515.70	654.49	24048.24	17101.51
70W static	686.06	12312.24	7629.02	681.91	24828.42	15168.44
70W dynamic	655.40	10825.60	8390.14	654.24	21433.79	16525.66
50W static	832.01	10656.16	7525.93	849.91	21801.45	14956.87
50W dynamic	737.56	9250.22	7573.55	759.09	18854.32	15095.90

Table 12. Comparison of static and decoupled with more varied workloads at 128 and 256 nodes.

working on the same application to provide measurements on different sides of a phase boundary for a scheduler interval. In the case of an application entering a high power phase, sockets reporting before the phase change will receive less power than the sockets reporting after the phase change. The slower progress of the sockets with low power allocations may reduce progress and power consumption on the high power sockets, since the higher power sockets would need to wait. When power is less plentiful, PowSched is able to increase overall performance sufficiently to over come the penalty of asynchronous operation (i.e., the performance penalty from static capping exceeds the performance penalties of the uncoordinated operation).

Figure 12 show that the workloads have low power consumption and a long tail. A significant amount of the processor time for most processors is spent with idle consumption waiting on the longer applications to finish. Additionally, the majority of the workload configurations have low power consumption, reducing the opportunity for performance increases through power shifting. Better experiments would require adding a job scheduler to keep the nodes busy. However,



(a) Static

(b) PowSched

Figure 12. Job consumption and global bound for a 128 node cluster using an average of 70 watts per socket using the decoupled scheduler.

orchestrating such an experiment is difficult since the production job scheduler and experiment job scheduler would conflict.

Chapter Summary

In this chapter, PowSched was presented. PowSched is a power scheduler providing hard enforcement without application specific models or job scheduler integration. Power scheduling decisions are made per component based on a simple heuristic, provide more power to components consuming more power and less power to components consuming less power. When power is limited, PowSched

is able to improve overall system performance, measured as the time to complete all queued work. The MPI implementation shows that PowSched can produce power schedules of roughly the same quality as an uncapped system when power is plentiful. PowSched remains unique in the power scheduling literature due to the lack of job scheduler integration and being application agnostic.

The studies of PowSched presented in this chapter were all conducted experimentally using systems hosted and LLNL. Experimental work suffers from jitter due to non-determinism within the system. The effect of jitter can be mitigated by analyzing several runs of the same experimental configuration. However, this involves additional machine time. HPC machine time is generally limited and the requirements for power scheduling research (a non-trivial size machine exposing power control to researchers) further reduces the available resources for such experiments. In the next chapter, a simulator suitable for power scheduling research is presented.

CHAPTER V

SIMULATING POWER CAPPING

This chapter contains ideas and themes that have been previously published in D. A. Ellsworth et al. (2015b). Additionally, this chapter contains unpublished work under submission to Cluster 2017. The PowSim motivation, model, implementation, and analysis are my original work. Co-authors on the publications assisted with language and narrative to present these contributions in the previously published work.

One of the common challenges in HPC research is the cost of HPC machine time. Power scheduling research is doubly challenged since experiments require a machine of nontrivial scale that also exposes potentially dangerous MSRs to research code. Simulation is a common strategy to address lack of hardware availability and support experiments at reduced cost. Prior to this work, the HPC community did not have a simulator capable of simulating the effect of power capping at scale.

Several systems exist for simulating hardware near the gate level, allowing high resolution simulated power consumption to be captured. Hardware approaches using GPGPU (Chatterjee, DeOrio, & Bertacco, 2009), HPC clusters (Gonsiorowski, Carothers, & Tropper, 2012), and FPGAs (D. Kim et al., 2016) can greatly accelerate simulation performance over gate level techniques run on CPUs. However, even with hardware acceleration, a single relatively simple processors can only be simulated at clock speeds of a few kHz. To simulate an HPC cluster, hundreds to thousands of processors must be simulated. Functional architecture simulators, such as Simics (Magnusson et al., 2002), are faster than

gate level simulation, but are still too computationally intense for our purposes. Due to the computational costs, existing low level simulation techniques are not able to provide the needed performance for exploring power scheduling at scale.

The resource scheduling community has several existing simulators. While SLURM has a simulation capability that allows a small number of nodes to act like a much larger cluster, this is unsuitable since changing the power cap of a virtual node will effect all work mapped to the same physical node. BSC (Lucero, 2011) and CSCS (Trofinoff, 2015) have implemented simulators on top of SLURM. However, neither are capable of changing simulated application runtime after the job is launched. While BatSim (Dutot, Mercier, Poquet, & Richard, 2016) provides a model for adjusting runtime, BatSim only models DVFS, bounding clock speed, rather than power capping, bounding power. A given DVFS setting will result in predictable linear computation slowdown, for compute bound code, while a given RAPL limit may or may not slow a computation. Other existing simulation frameworks have similar challenges regarding changing runtime in response to power cap changes made at runtime or lack an energy model. Due to the lack of a model capturing RAPL effects or other simulator model limitations, existing simulators for the resource scheduling community are not able to provide the needed functionality for exploring power scheduling based on the RAPL mechanism at scale.

In the remainder of this chapter PowSim is described and evaluated. A model for the effect of RAPL power caps on an application and process, based on the observations in Chapter III, is presented. Then implementation details of a simulator using the model are given. Finally, results of simulator validation and a small simulation study are shown before the chapter concludes.

Modeling Power Capping

In this section, we consider the observations motivating the *PowSim* simulation model and how these are encoded. The objective of PowSim is to observe how changing power caps on arbitrary processors at arbitrary times impacts job throughput on a large HPC cluster. First, a generalization of the interplay between programs, processors, and power is presented. Next, the mathematical model used by the simulation to represent this interplay is presented. The section concludes with a description of how the model scales to support simulation of an HPC cluster.

Model Intuitions. Real HPC applications are generally characterized as having alternating compute and IO phases, which would respectively have high and low processor power consumption(Fukazawa et al., 2014). Figure 6 shows the behavior of MiniFE, an application displaying high and low power consumption phases, under high and low power caps. Assuming applications have constant power consumption when studying HPC power scheduling performance is a poor assumption.

An application specific range of processor power caps exist for which the application runtime is identical to an uncapped processor. Figure 5 plots the normalized runtime of several HPC benchmarks under progressively lower power caps, showing runtime dilation starts at different power caps for different applications. Plots were also shown in Chapter III highlighting that even when power caps are high, the maximum power consumption remains bounded by the characteristic power consumption. Additionally, Figure 6 indicates that applications with phased behaviors only experience runtime dilation during the phase that would consume more power than the current cap. The key observation

is that an application's instruction stream, on a given processor, is able to induce some maximum power consumption. All power caps greater than or equal to the characteristic power consumption result in the same application runtime, within system jitter.

From these observations, the performance of a power capped application will always be limited by the instruction stream, *program bound*, or the power cap, *processor bound*. When *program bound*, the dependencies between instructions in the program's instruction stream on the processor produce an upper bound on power consumption and rate of instruction eviction. *Program bound* execution yields the characteristic power consumption and runtime of the program will be considered the *normal time*. When *processor bound*, the processor likely used DVFS to reduce the clock frequency or DPG to halt the processor so power consumption remains within bound, changing the rate of instruction eviction per wall clock unit time. *Processor bound* execution experiences *time dilation*, that is, the wall clock time for the program to make the same amount of progress increases.

An application can be modeled as a function from instantaneous normal time to instantaneous power consumption. The program starts execution at time 0 and completes execution at the last instantaneous time for which the power consumption is non-zero. In program bound regions, application progress proceeds at the same rate as simulated wall clock time. Processor bound regions of execution can be trivially found by identifying where the function value exceeds the processor cap. The challenge for a simulator is to estimate the runtime dilation experienced during the periods of processor bound execution.

Figure 5 shows the runtime of several benchmarks under progressively lower power caps, normalized against the uncapped runtime. Runtime dilation starts

at a different power cap for each of the plotted benchmarks. However, all of the benchmarks exhibit a polynomial shaped curve. A linear reduction in the power cap, beneath the characteristic consumption, results in a polynomial increase in runtime. The key observation is that during processor bound phases, the difference between the uncapped power consumption and current power cap is related to the amount of runtime dilation.

Power measures the rate at which work is done and energy is the standard unit of work. Physically, the work of computation within the processor is done using transistor state changes and power is often modeled with the following formula $P = ACV^2f + VI_{\text{leak}}$ (N. S. Kim et al., 2003). The formula states that the power consumed is related to the active transistors (A), the total capacitance (C), the input voltage (V), the activation frequency (f), and the leak current (I_{leak}). Terms C and I_{leak} are physical properties of the processor. A is determined by the executing instruction stream[1]. The remaining terms V and f may be controlled by the processor, but a relationship between the values of V and f must be maintained for stable operation. A 10% reduction in supply voltage results in a greater than 10% reduction in frequency (N. S. Kim et al., 2003).

Mapping an application to hardware instructions to circuit activation is extremely complex on modern architectures and would require at least gate level simulation, which will not scale for simulation at the scale of an HPC cluster. Rather than work directly with hardware instructions, circuitry details, and electrical energy, *instruction work* is introduced as a simplifying abstraction. Over an interval, a processor is able to service some maximum amount of

[1]One can imagine processor architectures where the processor dynamically selects different implementations for the same hardware instruction, which would allow the processor some control over the number of active transistors per unit time required to complete a particular instruction.

instruction work, which represents some number of processor transistor activations corresponding to hardware instructions and energy consumption over the interval. Given an interval in program normal time, a certain amount of *instruction work* is induced by an executing program, abstractly representing the active circuits and corresponding energy consumption over the interval.

Processor power consumption is expected to fall within a fixed range of values. The upper bound on processor power consumption is assumed to be the TDP. TDP is given in the processor specification and is usually used as the maximum power consumption for the processor when designing a system. There is also a lower bound on processor power consumption that occurs due to leak current and other background activity, including the operating system. The *idle power*, power consumed for background activity, is assumed to not do useful computation work for an application. Observed power consumption for a processor running a computation is expected to fall between the *idle power* and TDP.

The following core ideas shape how runtime dilation is estimated:

1. The amount of dilation is related to the difference between what the uncapped power consumption would be and the current power cap

2. TDP occurs when the maximum amount of instruction work possible is being done by an uncapped processor

3. Idle power occurs when no application work is done by the processor

As this subsection concludes, note that we have sketched a relationship from program logic through hardware instructions to physical activity within a processor. Power consumption and runtime have a strong relationship with program text, compiler output, and the hardware used to execute the program.

I_proc	Socket instruction work possible
I_prog	Program instruction work
E_s	Socket energy allocated
E_p	Program energy consumed
$W_{(t)}$	Program power consumption at time t
s	Interval start time wallclock time
e	Interval end time wallclock time
s'	Interval start time in normal time
e'	Interval end in normal time
S_{cap}	Socket power bound
S_{idle}	"Idle" socket power consumption
S_{TDP}	Maximum socket power consumption
A_{min}	Minimum power allocation
A_{max}	Maximum power allocation

Table 13. Symbols use in the simulation model

Real life consumption values observed for a specific application code may differ based on the compiler used due to differences in the binary produced. Even for the same binary, consumption observed on one hardware platform may not be portable to other hardware platforms due to differing instruction implementations within the processors. It is important to note that the general nonlinear relationship between power capping and application runtime is still expected to be preserved across hardware since it follows from the electrical properties of transistors, although the specific shape of the curve will likely depend on numerous details of the actual fabrication process. Tight alignment of the simulator to actual applications is unnecessary for exploring the general behavior of power scheduling.

Model Formalization. In this subsection the formal description of the power simulator model is given. The model describes both an application execution and a processor. Using instruction work, the model is able to estimate program progress during processor bound execution and produce reasonable runtime dilation. Table 13 summarizes the symbols used for the simulator model.

A processor's power consumption ranges between a minimum value, S_{idle}, and maximum value S_{TDP}. S_{idle} represents the power consumption of the processor when no application work is being done, the case when no program is running or a running program is blocked. S_{TDP} represents the power consumption of the processor when the most application work is being done. The power cap, S_{cap}, or power allocation of a processor may be set to a value in a range specified by the manufacturer, A_{min} to A_{max}. The range of valid allocations should relate to the power characteristics of the processor, $S_{\text{idle}} < A_{\text{min}}$ and $S_{\text{TDP}} = A_{\text{max}}$. For an uncapped processor, the power cap can be considered to be S_{TDP}.

A program running on a processor changes the power consumption of a processor and will be modeled as a function, $W_{(t)}$, that maps from normal time, t, to the observed power consumption on an uncapped processor. A program starts at time $t_s = 0$ and ends at time t_e in normal time. The values of $W_{(t)}$ may range from S_{idle} to S_{TDP}. Program progress can be measured as the percentage of normal runtime elapsed. For a wall clock time intervals in which $W_{(t)} < S_{\text{cap}}$, the wall clock time and program time progress at the same rate. Computing program progress when $W_{(t)} > S_{\text{cap}}$ is less straight forward.

Instruction work is used in the model to address challenge of computing progress when $W_{(t)} > S_{\text{cap}}$. Recall that instruction work does not directly represent a count of transistors or a count of hardware instructions, even though the metric tries to abstractly encapsulate both ideas. The rate of energy consumption is related to the rate of instruction work done which should allow convertion from the modeled watt values to a measure of instruction work. Over a wall clock time interval from s to e, a power capped processor will be able to complete a finite amount of instruction work, I_{proc}. Over a normal time interval from s' to e', a

program will be able to induce a finite amount of instruction work, I_{prog}, on an uncapped processor. It must be the case that $I_{prog} \leq I_{proc}$ since the processor cannot do more work than the program can induce. Additionally, $e - s \geq e' - s'$ since a normal time interval may induce more work than the walk clock interval will support, leading to some instruction work being deferred to the next interval. Computing time dilated program progress involves finding e' such that $I_{proc} = I_{prog}$ for the wall clock interval s to e.

Equation 5.2 is models the work that could be done by a processor over a wall clock interval and Equation 5.1 is models the work induced by a program over a normal time interval. Instantaneous instruction work values are normalized to a range between 0 and 1. Observed power, S_{cap} or $W_{(t)}$ for processor and program respectively, less than S_{idle} are nonsensical since consumption must be greater than idle power for any program progress to occur. The maximum progress should occur when the program's power consumption is equal to the processor's TDP. Doubling power does not double the rate at which instruction work is done, even for programs able to fully utilize the additional power. For simplicity, a square root is used to get the desired behavior. Changing the formula adjusts the shape of the runtime dilation under progressively lower caps.

$$I_{proc} = \int_{s}^{e} \sqrt{\frac{S_{\text{cap}} - S_{\text{idle}}}{S_{\text{TDP}} - S_{\text{idle}}}} \mathrm{d}t \tag{5.1}$$

$$I_{prog} = \int_{s'}^{e'} \sqrt{\frac{W_{(t)} - S_{\text{idle}}}{S_{\text{TDP}} - S_{\text{idle}}}} \mathrm{d}t \tag{5.2}$$

Cluster Scale. A cluster can be abstractly modeled as a collection of processors that programs run on. Clusters are partitioned into nodes and each node has some number of processors. A job is a unit of work within the cluster and, for simplicity, maps to a program to be run across some number of nodes. The job

scheduler assigns a job to some number of nodes, which starts the program on all of nodes associated with the job. Other resource schedulers, such as power schedulers, may periodically make adjustments to the resource configuration of the nodes in the cluster.

Cluster scale simulation follows directly from the description of clusters and jobs. A number of processors are simulated corresponding to the size of the cluster and, for convenience, are grouped into nodes. A job is defined by the associated program function, $W_{(t)}$. When a job is assigned to a set of nodes, the program is simulated as running in parallel across the processors associated with those nodes. Processors in the cluster not running any job consume energy at the idle rate, S_{idle}.

A simulated running program has a program function, $W_{(t)}$, and a progress counter to indicate the normal time the program is currently at. Each step of the simulation, the progress counter of each program on each processor is updated to reflect the progress made during the simulation step. When the progress counter reaches the normal time runtime of the program, the program has completed execution and the processor can return to the idle state.

Algorithm 6 outlines the core simulation loop. The simulation step size impacts the accuracy of the simulation since the instruction work integral hides when during the interval consumption might be high or low. Using the simulation loop outlined, a job will make uniform progress across processors if the power is set uniformly across the processors. The lack of a model for communication between processors in the core simulation loop is a limitation of the current solution and future work can address this limitation by introducing, at a minimum, a model for barriers. Until a communication model is integrated, power schedulers must set all processors within a job to the same cap so that progress is uniform. Adding

Algorithm 6 Core simulation algorithm

for all wall clock time intervals s to e **do**
  Run the job scheduler to place jobs
  Run the power scheduler to set caps
  for all processors **do**
    if No program is assigned to the processor **then**
      Consume $\int_s^e S_{\text{idle}}$ joules
      Advance program progress $e - s$
    else
      if Program Bound **then**
        Consume $\int_{s'}^{e'} W_{(t)}$ joules
        Advance program progress $e - s$
      else
        Solve $\int_s^e \sqrt{\frac{S_{\text{cap}} - S_{\text{idle}}}{S_{\text{TDP}} - S_{\text{idle}}}} \mathrm{d}t = \int_{s'}^{e'} \sqrt{\frac{W_{(t)} - S_{\text{idle}}}{S_{\text{TDP}} - S_{\text{idle}}}} \mathrm{d}t$ for e'
        Consume $\int_s^e S_{\text{cap}}$ joules
        Advance program progress $e' - s'$
      end if
    end if
  end for
end for

a communication model would also support modeling a job as several programs, potentially with different consumption behaviors, and allow for simulating jobs with load imbalance.

Implementation

The design of a simulator using the model in the previous section is described here. Implementation is done in `python` and uses an object oriented approach. Code for the simulator is contained primarily in three files, which loosely align with physical, scheduling, and program concerns. An experiment using the simulator is expressed as a python program that configures a cluster, some schedulers, and some number of jobs to simulate before calling the *run_sim* function. The *run_sim* function will step through the simulation timewise, giving all

registered schedulers a chance to interact with the cluster at each time step, until both the registered job scheduler and cluster indicate all work is done.

Machine - sim.py. The machine is implemented with objects corresponding to the physical model of the machine. A *Cluster* instance has many *Node* instances and each *Node* instance has one or more *Socket* instances. A *Cluster* can be told to launch a *Job* instance using the *launch_job* method. The *launch_job* call is expected to be made from a *Scheduler* instance. If sufficient nodes are available to launch the job, the *Cluster* will call the *distribute* method in the *Job* with the set of *Node*s that should execute the job. At the time *distribute* is called, the *Job* is responsible for providing each *Socket* instance with the *Program* instance to run.

Job objects bridge between the logical program to be run and the hardware jobs are run on. A *Job* is able distribute work across an allocation of nodes and can determine if the job has completed execution on the assigned nodes. *Job* classes have been implemented for four basic power consumption behaviors. *SawJob* and *StepJob* have regular periodic behavior which is interesting but likely do not align closely with real application traces. *StaticJob* and *PrePostJob* are more representative of HPC benchmork behaviors, where an application has a low power preamble, followed by a high power compute phase, and ending with a low power IO phase. *Job*s only distribute *Program* instances to *Socket*s, the consumption functions are implemented in the *Program* class hierarchy.

Scheduler - schedulers.py. Scheduler-like logical functionality is implemented using *Scheduler* instances. At each time step, the simulation will give the registered schedulers a chance to act by calling the *schedule* method with

the length of the wall clock step interval. The three basic scheduler functions considered are job scheduling, monitoring, and power scheduling.

The current implementation has only a basic first-come first-served (FCFS) job scheduler, *FIFOScheduler*. Jobs are added using the *addJob* method, which wraps a *Job* instance and some other data in a *Task* instance. When the job scheduler's *schedule* method is called, the scheduler attempts to place additional work on the cluster until no more work can be scheduled. Having a clean abstraction for job schedulers enables more advanced schedulers to be easily added to the simulator later.

Monitoring type schedulers gather and output during simulation experiments. By default, the simulator does not provide output regarding the simulation state. The *PowMon* monitor provides the same information in the same format as the PowMon executable discussed in chapter III, allowing the analysis to be done uniformly between real and simulated experiments. Each time the *schedule* method is called the current energy counter and power cap for each simulated processor are spooled to disk. For large scale simulations, this results in too many open file handles, so *BulkPOWmon* was introduced to write all data to a single file.

Power schedulers also have a clean abstraction, enabling easy introduction and experimentation with additional power scheduling strategies. *PowerScheduler* subclasses each implement a single power scheduling strategy. A *PowerScheduler* *schedule* call provides an opportunity for the power scheduler to adjust the power setting of all *Socket* instances in the cluster. Small scale results using *StaticPowerScheduler* and *DynamicPowerScheduler* appear in D. A. Ellsworth et al. (2015b).

Program - program.py. *Program* instances encapsulate a program's progress and the effects of time dilation during processor bound execution. The *advance_with_bound* method determines if the execution is program or processor bound for the current simulation step and updates the program progress counter for the step. *IdleProgram* is executed on sockets when no other program is running and all other programs inherit from *PartedProgram*. The *IdleProgram*'s consumption function is constant, $W_{(t)} = S_{\text{idle}}$, making evaluation of the instruction work and energy integrals trivial. The *PartedProgram* instances support discontinuous power consumption, such as the phases observed in power traces on real hardware. In these cases, the continuous regions can be integrated separately.

Four basic *Program* implementations have been written so far. These *Program*s implement correspond to the basic *Program*s. *StaticProgram* implements a program with constant power consumptions. *PrePostProgram* implements a program with startup, computation, and shutdown phases. *SawProgram* implements a program with sawtooth shaped power consumption. *StepProgram* implements a program with squarewave shaped power consumption. Internally, each program is a sequence of one or more continuous *Part*s.

A *Part* encapsulates the continuous function for a phase of application execution and how to compute the instruction work and energy values over that function. Most of the time *ConstantPart* is used since it provides a constant power consumption over the interval, matching with the step shape changes in consumption generally observed in HPC benchmarks. *UpPart* and *DownPart* model linear changes in power consumption and are used to implement sawtooth

shaped power consumption[2]. Any continuous function could be implemented as a *Part* and used to support implementation of a new type of *PartedProgram*.

Simulator Behavior

The development of the model and the corresponding implementation should result in simulated executions with properties similar to real application execution. Overfitting to a real hardware platform or application is undesirable due to the interplay of the myriad low level details of program execution. Generalized behavior is sufficient since simulation is intended to be used for exploration of general power scheduler properties. Additionally, the simulator is expected to operate at scale so the scaling behavior of the simulator should also be explored.

Validation. Initial validation of the simulator involves producing curves of a similar shape to the real runtimes of applications as the power cap is steadily decreased. Additionally, applications with high and low phase behavior should only see runtime increases for the phases where consumption would be over the power cap.

Figure 13 shows the normalized runtimes for small mix of jobs under progressively lower power caps. The general shape and runtime trends from the simulator are consistent with experimental data and can be compared with Figure 5 and Table 14 provides information on the configuration of each simulated job.

Figure 14 shows the power traces for runs of the same simulated application under different power caps. The figure is annotated with time and show dilation and non-dilation in phases similar to a phased application trace (see Figure 6). As the simulator aims to match generalized behavior for comparing different strategies

[2]The Nekbone characteristic consumption (figure 4e) appears to have a slight incline. While uncommon, consumption ramp up and ramp down should probably be considered in addition to stepwise changes.

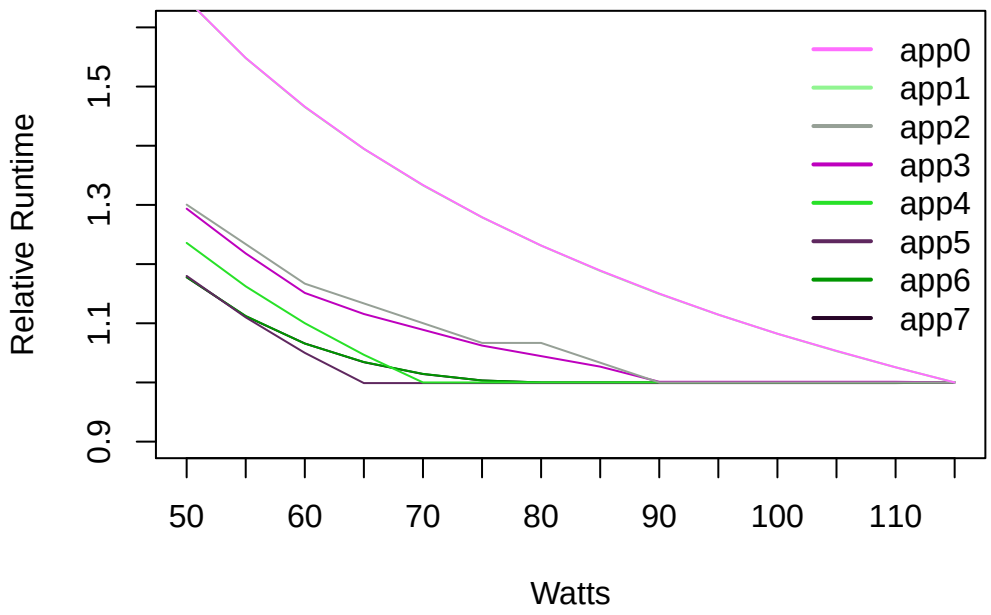


Figure 13. Runtime effect of decreasing processor power caps for a collection of simulated programs.

Job	Type	Min(W)	Max(W)	Runtime	Period
app0	Static	115	115	15 min	
app1	Static	115	115	30 min	
app2	Step	60	90	10 min	15 sec
app3	Step	60	90	15 min	5 min
app4	Static	70	70	15 min	
app5	Static	65	65	15 min	
app6	Saw	50	80	15 min	225 sec
app7	Saw	50	80	20 min	5 min

Table 14. Description of the simulated jobs plotted in Fig 13. Runtime in this table refers to the job runtime in normal time.

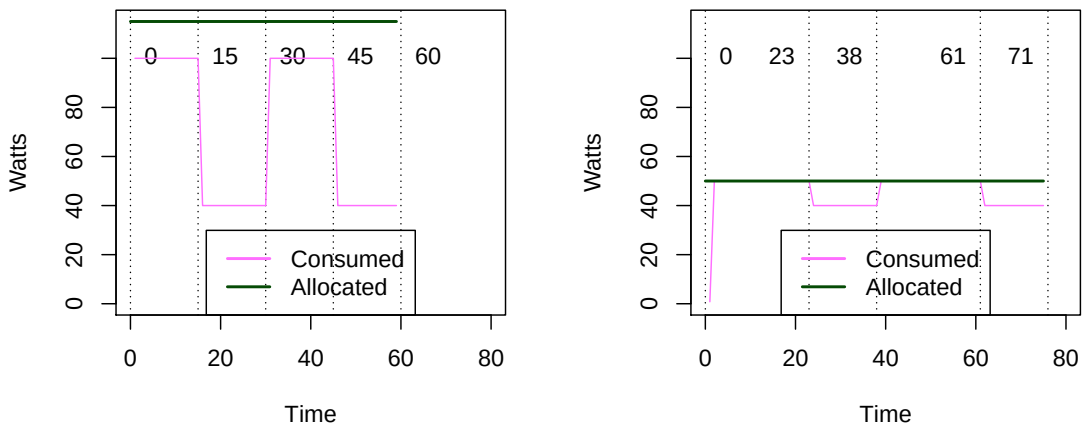


Figure 14. Power consumption for a simulated application with high and low power consumption phases (unbounded (left) and bounded(right)).

rather than tightly fit a real system, this limited validation against generally observed trends is sufficient.

Scaling Performance. To study the performance of the simulator, the simulator is run in several configurations and the wall clock runtime is recorded. Cluster size and simulated steps are individually expected to have linear impacts on simulation runtime. Job type may have an effect on simulator performance since some programs may be easier to compute than others. The power scheduler used may also impact runtime performance.

All experiments in this section are run on a 4.0 GHz Intel i7-4770 with 32 GB of RAM. The simulated cluster has a power cap of 80 watts per socket. Python, due to the global interpreter lock (GIL), is effectively single threaded and only able to effectively use one core per experiment. To better utilize the system, and reduce the time required to generate the complete results, the python multiprocess module is use to run one experiment per core (8 experiments run concurrently). The effects of memory contention are not expected to change the asymptotic behavior of the simulation and are not studied in this paper.

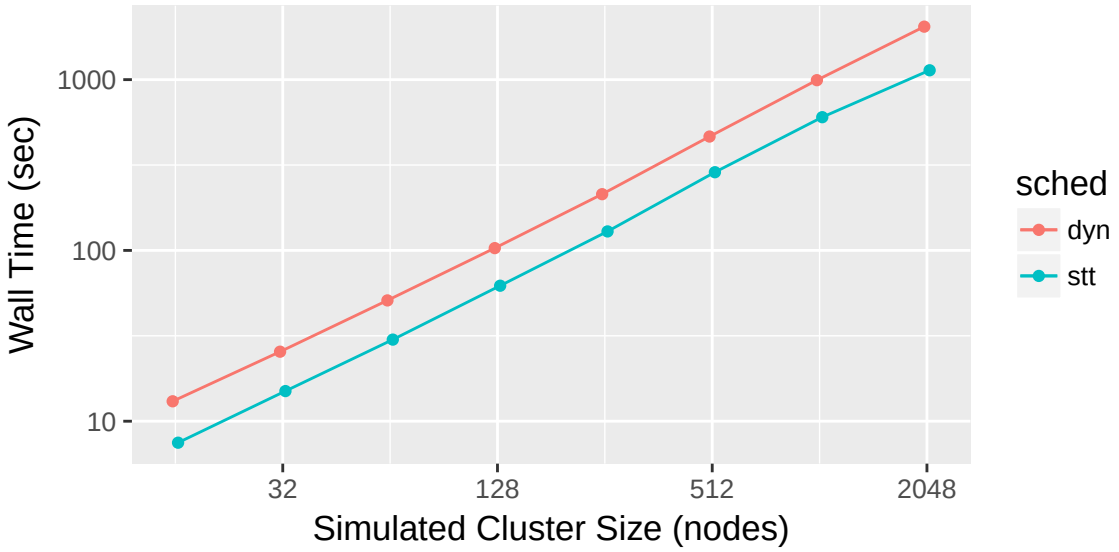


Figure 15. Wall clock time to simulate 12 hours of runtime with cluster node counts ranging from 16 to 2048 nodes. Static (stt) and Dynamic (dyn) schedulers are compared.

The first scaling study investigates the performance as the number of nodes are increased in the simulated cluster. Node counts range from 16 to 2048 and runtimes are plotted in Figure 15 on a log-log scale. All jobs are of the step job type and simulation runs for 12 simulated hours. Each experiment is run 10 times and the average across all 10 runs is plotted for both the static and dynamic scheduler algorithms. As expected, the size of the simulated cluster has a linear impact on the wall clock runtime of the simulation.

The second scaling study investigates the performance as the number of simulated time steps are increased. Simulated durations range from 1 hour to 128 hours and wall clock runtimes are plotted in Figure 16. All jobs are of the step job type and run on a simulated 1024 node cluster. Each experiment is run 10 times and the average across all 10 runs is plotted for both the static and dynamic

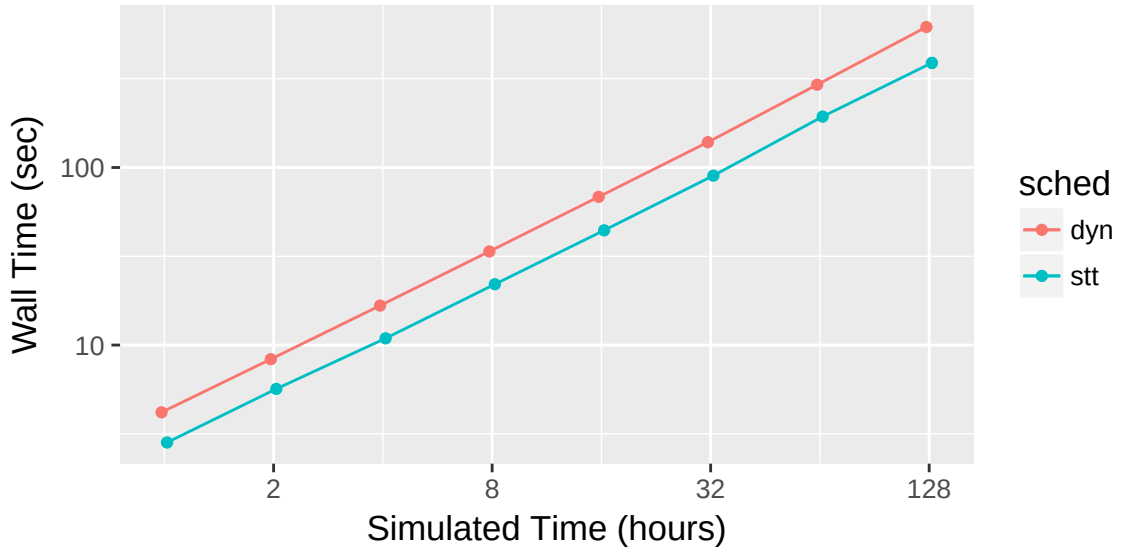


Figure 16. Wall clock time to simulate a cluster with 1024 nodes from 1 hour to 64 hours of simulated time. Static (stt) and Dynamic (dyn) schedulers are compared.

scheduler algorithms. As expected, the simulated duration has a linear impact on the wall clock runtime of the simulation.

To investigate the impact of job type, the cluster size is fixed and job type is varied. Static, Step, and Saw job types are compared using the static and dynamic schedulers. Static jobs were run with a consumption of 100 watts, *ostatic*, and 50 watts, *ustatic*. The Step job has a period of 60 simulated seconds and ranges from 50 watts to 90 watts. The Saw job has a period of 60 simulated seconds and ranges from 50 watts to 90 watts. Each experiment is run 10 times and the average across all 10 runs is plotted. Figure 17 plots the results for the static scheduler and indicates that job type has a roughly constant cost on simulation runtime. The dynamic scheduler produces a similar plot.

In all of the experiments, the static scheduler operates faster than the dynamic scheduler, which is expected. The static scheduler does no work at each scheduling interval since the power caps are set only once during static scheduler

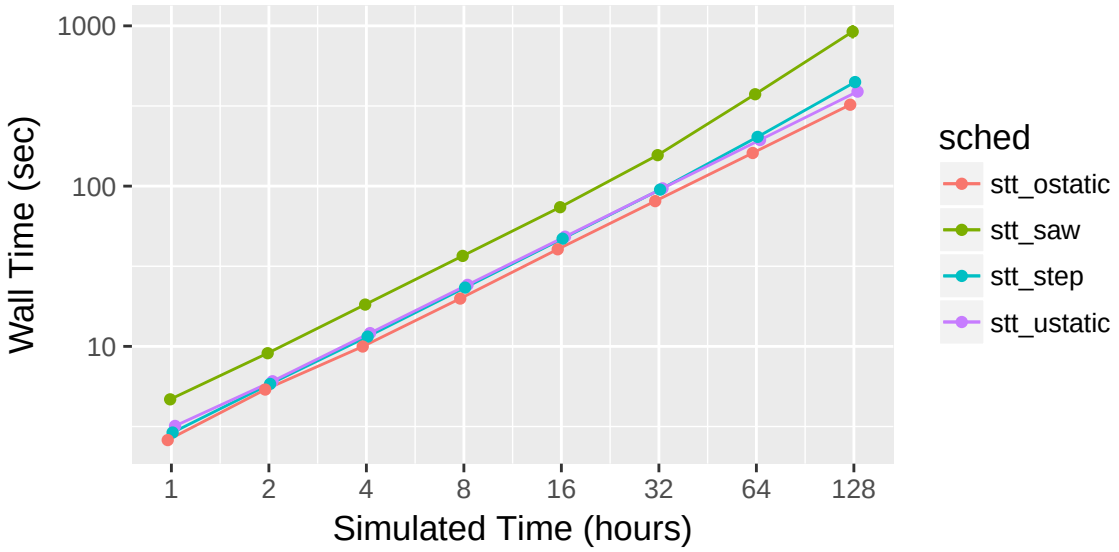


Figure 17. Wall clock time to simulate a cluster with 1024 nodes for 1 to 64 hours of simulated time. Different job types are compared.

initialization. The dynamic scheduler must process the cluster state, generate a schedule, and apply new caps to the cluster at each scheduling interval. Dynamic scheduler work is linear with the number of nodes and must be done each simulated time step. The performance difference between the dynamic and static schedulers is accounted for by the additional work that the dynamic scheduler must complete.

Simulated PowSched

Having verified that PowSim produces the expected trends in runtime dilation and is suitable for scale, the simulator is next used to explore PowSched. Exploration using simulation avoids the challenges of getting HPC machine time, system jitter, and introduction of a research job scheduler. The simulator should be able to reproduce the performance behavior similar to the experimental results in Table 9. An experiment is also conducted to highlight the relationship between concurrently running work needed to achieve performance improvements using PowSched, the lack of jitter will make power shifting easier to observe.

Parameter	Min	Max
Job Nodes	8	$\frac{\text{clustersize}}{4}$
Job Runtime (seconds)	10	6000
Phase 1 Watts	85	115
Phase 2 Watts	30	115
Phase Period	30	600

Table 15. Simulation parameters

Nodes	Bound	Static	Dynamic	%
1k	115W	40567	40581	-0.0
	70W	44541	43287	2.8
	50W	53249	54955	-3.2
8k	115W	43801	43825	-0.0
	70W	51652	51081	1.1
	50W	63085	65545	-3.9
16k	115W	44414	44429	-0.0
	70W	52656	51873	1.5
	50W	64097	66054	-3.1

Table 16. Simulated runtime with random workloads.

Cluster Simulation. Using PowSim we simulate the use of PowSched on clusters of 1k, 8k, and 16k nodes. For each node count a random mix of 100 jobs is generated, the same mix is used for each run at that particular node count. Each job is one of the three simulated functions with a random runtime, period, and consumption. Table 15 shows the parameter ranges used. All jobs in the run are queued in the job scheduler before the first time step executes. Table 16 shows the time taken to complete all queued jobs. Simulation results are consistent with the experimental results on random workloads – little effect at 115 watts, improvement around 70 watts, and reduced performance at 50 watts when power is overly constrained.

Anticipating Performance. While an experimental result indicating performance improvements occur when using PowSched are encouraging a more

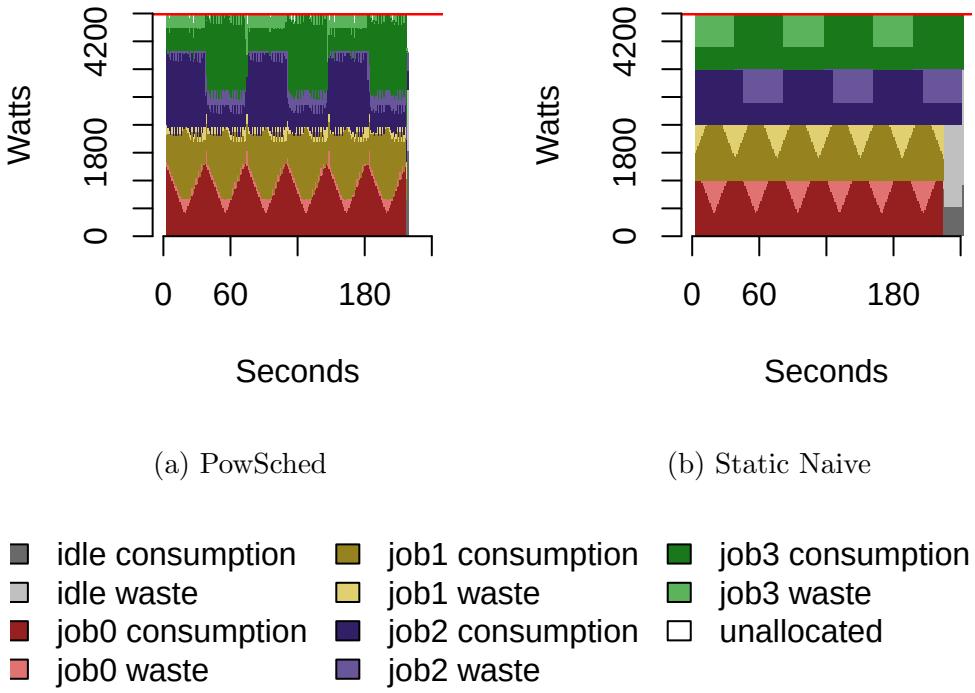
useful outcome would be understanding the circumstances that would result in performance improvement. A set of experiments, leveraging the lack of jitter, highlight the behavior of a system running PowSched with different concurrent workloads. The power needs of concurrently executing work and the system power limit, unsurprisingly, impact the system performance.

In these experiments, 4 concurrent jobs are executed using the naive static and PowSched power schedulers. For all simulated jobs power consumption ranges from 30 to 100 watts. Step and saw functions are used to represent the simulated jobs. The system power limit provides an average power per socket of 75 watts.

Best Case. Figure 18 shows the ideal case for PowSched. In this configuration the power consumption of the applications is time aligned such that the aggregate power consumption is less than the system limit. An increase in consumption by one job is offset in the same time step by a decrease in consumption of another job, maximizing the opportunity for power shifting. Performance using PowSched with the power cap is extremely similar to the uncapped execution.

Worst Case. Figure 19 shows the worst case for PowSched. In this configuration the power consumption of the applications is time aligned such that the aggregate power consumption changes by the same amount in the same direction at each time step. Power shifting is not possible since all jobs become power bound at the same time and release power at the same time. Performance is the same between naive static and PowSched power schedulers.

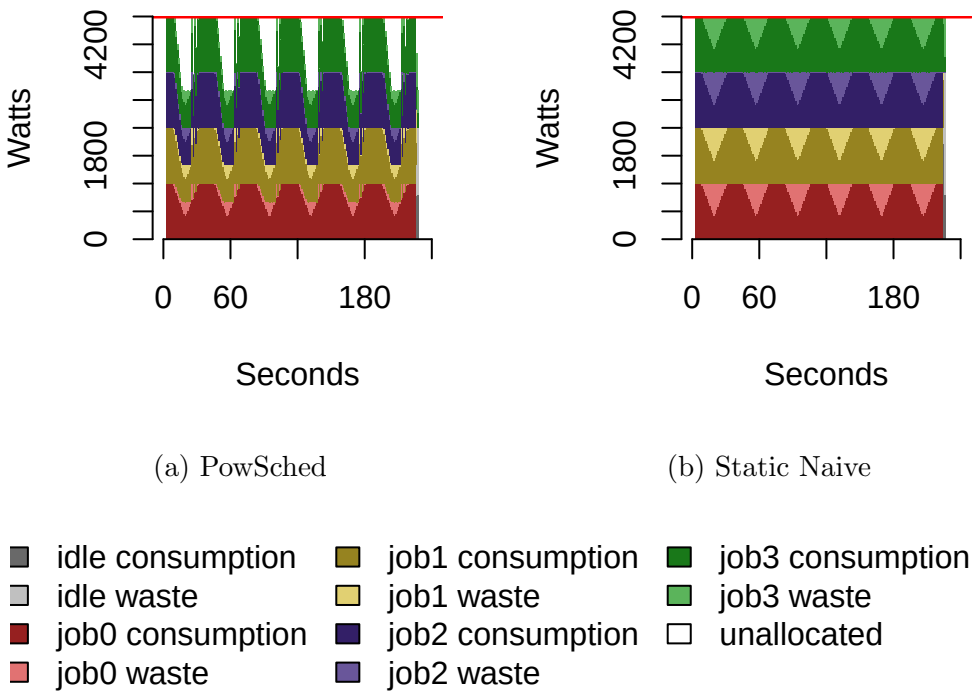
Middle Case. The previous cases are extremely unlikely to occur on real systems. Constructing real applications to have perfectly time synchronized power consumption behavior would be extremely difficult. A more likely case to occur



(a) PowSched

(b) Static Naive

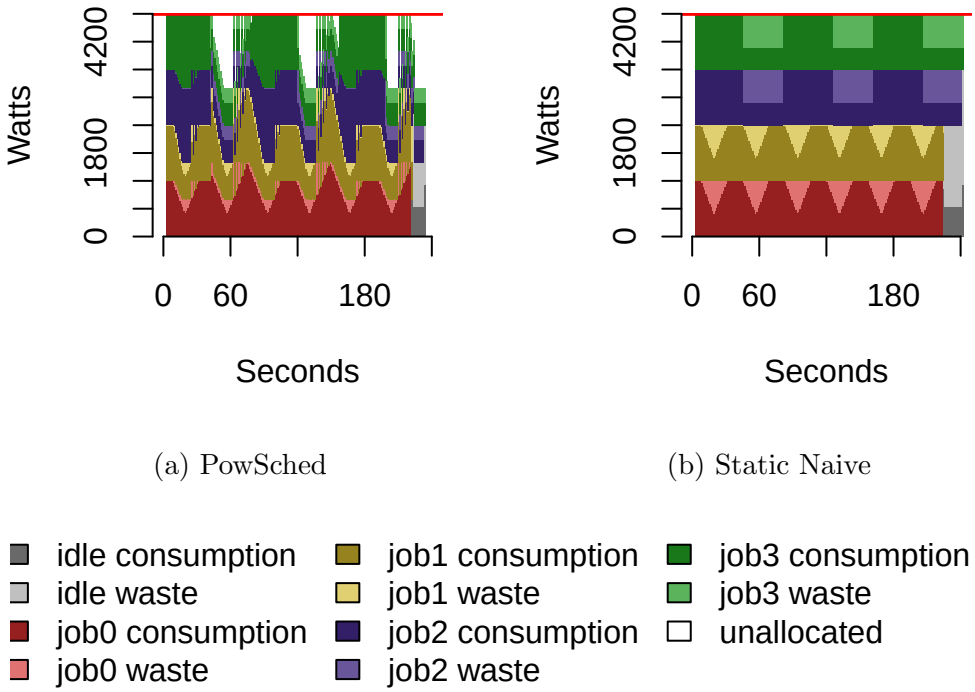
Figure 18. Optimally time aligned consumption across jobs.



(a) PowSched

(b) Static Naive

Figure 19. Worst case time aligned consumption across jobs.



(a) PowSched

(b) Static Naive

Figure 20. Bad time alignment with different rates of change across jobs.

in a real system would be several applications increasing and decreasing power consumption, at different rates, near the same time. Figure 20 attempts to show this case by mixing step and saw functions. PowSched is able to take advantage of power shifting to reduce runtime when compared with naive static scheduling.

Chapter Summary

A novel simulator to support power scheduling research, PowSim, was presented in this chapter. PowSim is the only simulator currently able to simulate the runtime dilation that occurs when power caps are changed during application execution and scale to thousands of simulated nodes an modest hardware. The simulation model used by PowSim connects program progress and the physical construction and attributes of modern processors. Simulators from the job scheduling community, like the one useds by Savoie et al. (2016) and Patki et al.

(2015), require extensive modeling of real application executions and do not track computation progress. Simulators from the hardware architecture community, like D. Kim et al. (2016) and Gonsiorowski et al. (2012), cannot scale due to the computational requirements of simulated instruction execution. In addition to enabling power scheduler research at scale, PowSim's deterministic behavior and analytic model simplify exploration of why a power scheduling strategy produces a particular result.

CHAPTER VI

COMPARING SCHEDULING APPROACHES

This chapter contains ideas and themes that have been previously published in D. Ellsworth, Patki, Schulz, et al. (2016). Additionally, this chapter contains unpublished work under submission to Cluster 2017. Comparison experiments and analysis are my orignal work. Co-authors on the publications assisted with language and narrative to present these contributions in the previously published work.

A direct comparison of power scheduling solutions available in the literature is currently not possible, but is sorely needed to make practical system implementation decisions. Each project has developed their own experimental harness and has conduct their experiments on different platforms with different measurement approaches and capabilities. Different researchers have also selected different benchmarks and run them in different configurations. Evaluation of PowSched using simulation in Chapter V highlights that details of the workload and system impact the performance of a particular power scheduling strategy. Even for a single researcher working on a single system, system upgrades and other platform changes that occur between experimental runs create challenges and impact the quality of comparative observations[1]. Going forward, power scheduling researchers should try to converge on a single experimental platfrom and single

[1]Several times during this research msrsafe, libmsr, and OS upgrades required changes be made to codes when rerunning experiments on the same machine. System configuration changes can also impact the background activity and behavior of an HPC system. In general these changes are expected to have little effect but call into question the accuracy of direct comparisons of experiments by even the same research group separated in time.

simulation platform so that experiments can be easily conducted comparing solutions from different researchers.

Experimental Comparison

As mentioned in Chapter II, groups producing experimental results on power scheduling are doing so using independently developed research platforms. The lack of a unified experimental platform creates two practical barriers for production deployment. First, research platforms do not have sufficient feature richness or security for use on production systems. Due to the lack of such features, any research solution requires costly reimplementation to productize. Second, research platforms are inconsistent in configuration and requirements. For organizations wishing to evaluate power scheduling solutions via comparison, the inability to use the same infrastructure for job submission, scheduling, execution, and measurement creates barriers for idenpendent testing.

To address these challenges, the power scheduling community should move to a standard implementation platform. In "A Unified Platform for Exploring Power Management Strategies", D. Ellsworth, Patki, Schulz, et al. (2016) argue for SLURM[2] (Jette, Yoo, & Grondona, 2002) to be the base platform for experimental power scheduling implementations. Sakamoto et al. (2017) also base their work on SLURM and introduce a power abstraction layer to make power schedulers easier to implement. Unification of experimental solutions, where independently proposed solutions are delivered as SLURM plugins, will effectively remove the reimplementation barrier for solution adoption. Additionally, the plugin mechanism standardizes the implementation of advanced scheduling features (e.g., backfilling) and supports evaluation of proposed solutions in the presence of advanced features

[2]SLURM is a production resource manager with a flexible plugin architecture used on many of the top HPC systems.

that would be too costly for reimplementation in research code. Finally, using a single research management platform allows the evaluation harness to be cleanly decoupled from the power control strategy which helps to eliminate evaluation bias due to differences in measurement locations and approach.

While standardization of the platform will resolve a set of problems for comparison across power scheduling solutions, some significant challenges will remain. The foremost challenge will be in finding HPC systems of nontrivial size that support deployment of an alternative resource manager. It is hoped that by basing the research platform on a trusted production scheduler there will be less opposition from systems administrators regarding deployment. Going forward, if hardware overprovisioning is widely adopted, platforms for experimentation may actually become more difficult since a buggy research plugin may not provide the hard enforcement required for a deployment on a hardware overprovisioned HPC system. In the event that the systems allowing experimentation become plentiful, getting sufficient machine time is likely to always be challenging due to competing scientific work using the HPC cluster.

Experimental comparison work using a unified experimental platform is ongoing work. Getting mindshare around the need for standardizing the experimental platform is hindered by the extra implementation work required. A secondary challenge is that the number of systems supporting replacement of the system resource manager is extremely limited. It is hoped that the interface suggested by Sakamoto et al. (2017) will be adopted by other researchers and that more HPC systems will support deployment of research SLURM modules in the future.

Simulation Comparison

As mentioned in Chapter II, groups producing simulation based results tend to use their own simulation platforms. The incomparability of these simulation results is much higher than the incomparability of the experimental results. In addition to differences in system and work attributes, the simulator adds additional assumptions regarding behaviors of both the system and the workload that may not be grounded in real behaviors. Development of PowSim, the subject of Chapter V, was necessary since the existing simulators in the community do not have a model for runtime dilation that respects characteristic power consumption. Unification of simulation platform, or at least the interface between the platform and scheduling algorithm, would allow for better comparison studies in the community.

Even without a unified simulation environment, there are advantages to comparison via simulation rather than experimentation. The foremost advantage is the ability to run simulation on arbitrary hardware, reducing cost and increasing the number of experiments that can be done. Reimplementation effort of another researcher's algorithm is also likely to be simplified by the simulator since several real life opperational details are abstracted away by the simulation framework. Simulators are also convenient for providing better explanations for why different approaches provide different results since many of the nondeterministic and chaotic performance effects that occur in real executions can be better controlled in simulation.

Comparison using a single simulation platform is ongoing work. Ideally the interface between scheduling algorithms and simulators would be standardized allowing for direct reuse. Since direct reuse is not currently an option and most simulators do not model progress and time dilation, reimplementation of

various algorithms for PowSim is required. Initial comparative simulation work is comparing gross strategies, with more specific scheduler implementations being reserved for future work. Additionally, advanced job scheduling features (e.g., backfilling) are out of scope for the work presented here. Existing simulation results will be discussed in the remainder of this chapter.

Simulation Study

Comparison by schedule time seems like a natural axis for studying general scheduler behavior. The naive, reservation, range, and dynamic schedulers capture the schedule time classes found in the power scheduling literature (discussed in chapter II). *Naive* is the power scheduling strategy in which the component power allocations are set to the average of the system power limit at install time and never changed. *Reservation* is the power scheduling strategy in which jobs are submitted with a power estimate, the job scheduler only starts work that will not exceed the system power limit, and the power allocations are made only at job launch time. *Range* is the power scheduling strategy in which jobs are submitted with upper and lower bound power estimates, the job scheduler only starts work where all concurrent work can be serviced at the work's low estimate, and the power allocations are changed when any new job launches. *Dynamic* is the power scheduling strategy where allocations may be changed at arbitrary times, PowSched is used.

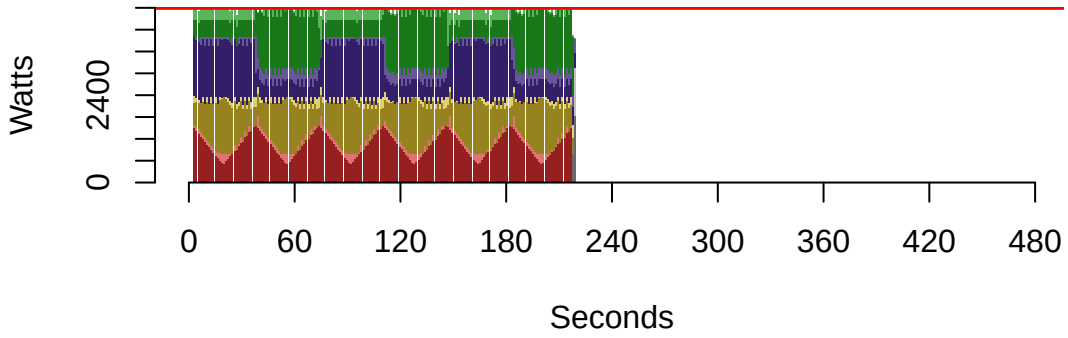
Comparison by job awareness (i.e., how much information the power scheduler has about the job) would also be an interesting axis for studying generalized scheduler behavior. There are several job attributes that might be of interest for job and power scheduling, however the correct scheduling decision depends on organization specific policies, making generalized scheduler

implementation and experiments difficult. A job aware variant of the *dynamic* scheduler has been implemented that gives preference to the highest priority job when allocating power, *priority*. Exploration of comparative scheduler behavior along the job awareness axis is left for future work.

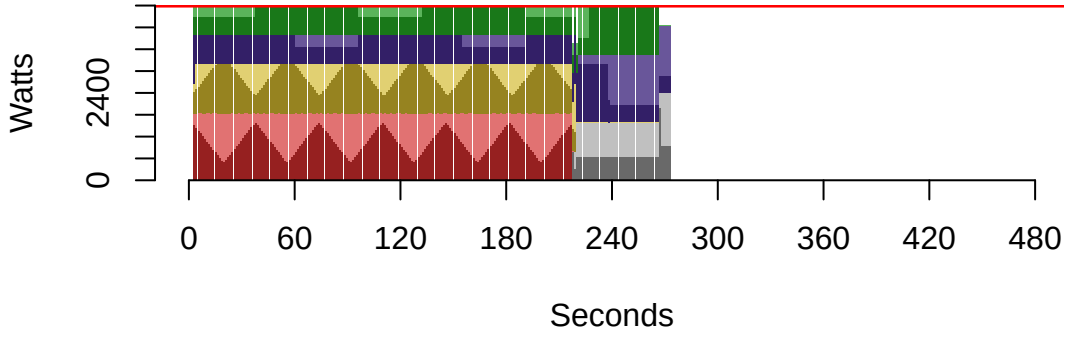
Base Behavior. A short simulation experiment, suitable for plotting consumption and allocation, is shown in figure 21 to build an intuition regarding each simulation strategy. The job mix from figure 18 is run with the reservation and priority schedulers in addition to the dynamic and naive. As expected, dynamic shows the best performance on the job mix best suited for PowSched. The priority scheduler successfully finishes the highest priority job first, however perturbation of the other jobs leads the priority order to be violated (figure 21b). Reservation scheduling had the worst performance, effectively doubling the time required to complete all work in the queue (figure 21d).

Given the amount of work in the community using reservation like approaches to power scheduling, the comparative badness of the reservation strategy was surprising. Even though the jobs individually have the shortest runtime using the reservation strategy twice as much time is required to complete all of the queued work. Idle node power and unutilized but reserved power limit the reservation strategy to scheduling only two of the jobs concurrently; if another job was placed the total power reservations plus the idle power for unused nodes would exceed the system limit. Evaluating power scheduler performance based on individual job runtimes is likely a poor choice since the actual time to solution involves the time a job remains in queue.

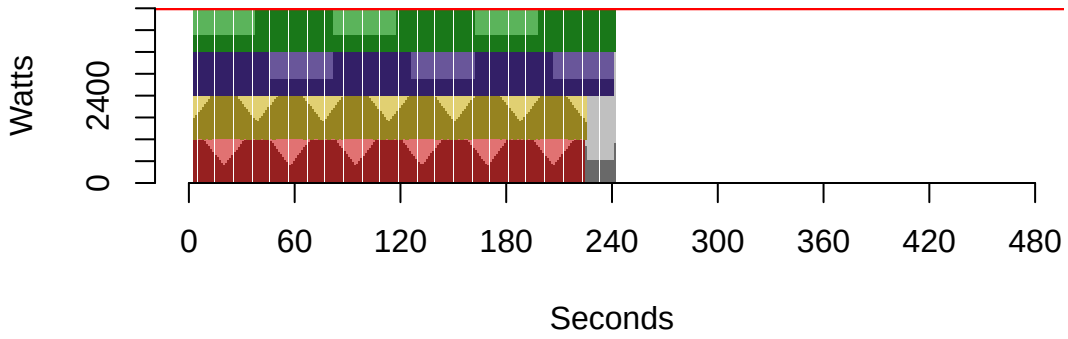
The poor performance of the reservation based job scheduler could also be a side effect of the size of the simulated cluster and the consumption characteristic



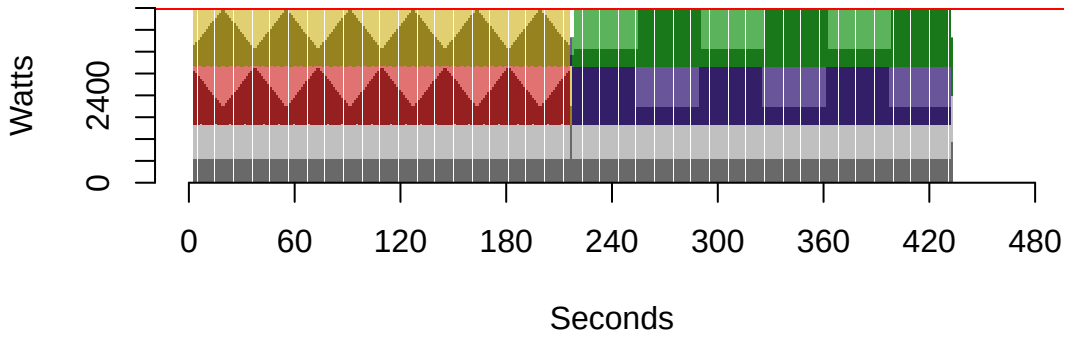
(a) Dynamic



(b) Priority



(c) Naive



(d) Reservation

Figure 21. Scheduler comparison using optimally time aligned consumption across jobs for PowSched.

of the jobs. An ideal case for a reservation based power scheduler should be a workload where jobs have constant power consumption that matches the estimated power consumption. A queue of waiting work, having varied node counts, may also help to enable additional jobs to run.

Random Queues. A larger simulation study is conducted to test the predictions made in the previous section. The simulated cluster has 128 nodes with a systemwide power limit of 20480 watts (i.e., 80 watts per socket). Naive, reservation, range, and dynamic power schedulers are run with a first come first served (FCFS) job scheduling discipline. Three job queues, each with 200 jobs, are used; a job queue containing jobs with constant power consumption, a job queue containing jobs with startup and shutdown consumption differing from the main computation power consumption, and a job queue containing jobs with high consumption and low consumption phases. On a worst case provisioned cluster, each of the simulated jobs would take between 5 and 60 minutes to run. Each job uses a power of two number of nodes, ranging from 1 to 128 nodes of the simulated cluster. Additionally, since reservation and range performance are impacted by the estimates provided at job submission time, the experiments are run with estimation errors between 0 and 15 percent. Sweeping this space results in 48 experiment runs, tables 17-19 show the raw results.

Unexpectedly, the job type and error appears to have little effect on the relative ordering of the power scheduling strategies. In order from most performant to least performant: dynamic, naive, range, and reservation. Dynamic an naive strategies do not make use of the job power estimates and the runtime as the amount of estimated error changes remains the same. Range and reservation schedulers are impacted by error, however the direction of impact is unclear.

Strategy	Job Type	Error ± %	Runtime	Slowdown %
dynamic	const	0	58623	0.00
static	const	0	60121	0.03
range	const	0	65917	0.12
reservation	const	0	65917	0.12
dynamic	const	5	58623	0.00
static	const	5	60121	0.03
range	const	5	65669	0.12
reservation	const	5	67354	0.15
dynamic	const	10	58623	0.00
static	const	10	60121	0.03
range	const	10	64617	0.10
reservation	const	10	70533	0.20
dynamic	const	15	58623	0.00
static	const	15	60121	0.03
range	const	15	63688	0.09
reservation	const	15	71468	0.22

Table 17. Runtimes of the schedulers ordered by error and run duration for constant type work.

Strategy	Job Type	Error ± %	Runtime	Slowdown %
dynamic	prepost	0	58838	0.00
static	prepost	0	59495	0.01
range	prepost	0	72283	0.23
reservation	prepost	0	72283	0.23
dynamic	prepost	5	58838	0.00
static	prepost	5	59495	0.01
range	prepost	5	68904	0.17
reservation	prepost	5	71267	0.21
dynamic	prepost	10	58838	0.00
static	prepost	10	59495	0.01
range	prepost	10	67839	0.15
reservation	prepost	10	73772	0.25
dynamic	prepost	15	58838	0.00
static	prepost	15	59495	0.01
range	prepost	15	67091	0.14
reservation	prepost	15	74622	0.27

Table 18. Runtimes of the schedulers ordered error and run duration for prepost type work.

Strategy	Job Type	Error ± %	Runtime	Slowdown %
dynamic	step	0	47382	0.00
static	step	0	49545	0.05
range	step	0	50157	0.06
reservation	step	0	69639	0.47
dynamic	step	5	47382	0.00
static	step	5	49545	0.05
range	step	5	50750	0.07
reservation	step	5	70701	0.49
dynamic	step	10	47382	0.00
static	step	10	49545	0.05
range	step	10	51460	0.09
reservation	step	10	71593	0.51
dynamic	step	15	47382	0.00
static	step	15	49545	0.05
range	step	15	51869	0.09
reservation	step	15	71370	0.51

Table 19. Runtimes of the schedulers ordered by error and run duration for step type work.

The expected ideal case for the reservation scheduler should occur when the power consumption is constant and the estimate has no error. In these cases, all of the reserved power will be utilized and no extra power will be allocated to the job. Unfortunately, the simulation data indicates that reservation based power scheduling has relatively poor performance even when used in what should be the best configuration.

Poor performance is likely due to the fixation the reservation and range based approaches place on individual job performance. Reservation and range strategies only allow a job to start if the estimated power is available, in all other cases the job start is delayed. Non-reservation strategies allow a job to be started when insufficient power is available, allowing the job to start but extending the job duration. Table 20 shows simulation results for the reservation scheduler run with constant work an fixed estimate error. Negative error (i.e., underestimation)

Strategy	Job Type	Error %	Runtime
Reservation	const	-15	42940
Reservation	const	-10	46708
Reservation	const	-5	47016
Reservation	const	0	65917
Reservation	const	5	73703
Reservation	const	10	79798
Reservation	const	15	84425

Table 20. Runtimes of the reservation scheduler with constant type work for fixed estimate errors.

appears to improve reservation scheduling, likely by allowing jobs to run that would have been held back if the estimates were more accurate. The turnaround time penalty of waiting for sufficient power to be available appears to exceed the penalty of runtime dilation.

Chapter Summary

One of the major gaps in the existing power scheduling research is the lack of comparability. Power capping has complex effects that have broad and complicated interactions, making the diversity of hardware and workloads problematic for comparison using just the literature. Direct experimental comparison of strategies by individual researchers are complicated by machine and machine time availability as well as configuration of each of the solutions. For experimental work, the community should move towards a standardized platform to simplify configuration. Direct comparison of strategies using simulation is less complex but currently requires reimplementation.

A major area of concern for the power scheduling community moving forward should be the impact of idle power. Much of the existing power scheduling work uses the job scheduler to reserve power at job launch, which makes figure 21d unsettling. Generally production HPC codes are talked about in terms of IO and

compute phases which have different power characteristics, so reservation may be the wrong general approach since reservation provides no opportunity for power shifting across jobs. Even if it is the case that HPC applications generally use constant power, like the LULESH benchmark, reservation based scheduling does not appear perform well when compared to other power scheduling strategies.

CHAPTER VII

CONCLUSION

This chapter contains ideas and themes that have been previously published in D. Ellsworth (2016); D. Ellsworth, Patki, Perarnau, et al. (2016); D. Ellsworth, Patki, Schulz, et al. (2016); D. A. Ellsworth et al. (2015a, 2015b). The narrative as presented in this chapter is my original work derived from discussions with the co-authors of the previously published work.

Power scheduling for hardware overprovisioned HPC systems is a new area of research. The technique is suggested by Rountree et al. (2012), and was originally motivated by the energy concerns for building an exascale system with a maximum power consumption of 20 MW. Even though this goal has changed, energy efficiency is still crucial for providing a high degree of computational performance within a bounded power budget. However, energy efficiency alone is insufficient. Power scheduling is an additional technology needed to turn energy savings into additional performance. In some cases, hard enforcement of the power limit is needed and there are a limited number of power schedulers that provide this guarantee. Much work remains for future HPC researchers looking at power scheduling.

Power control using hardware limiters, like Intel RAPL, has a somewhat complicated relationship with application runtime. Techniques like DVFS and DPG have effects that are relatively simple to estimate. Hardware power limiters are more complex since reducing the power cap may or may not impact runtime. Observable runtime effects of RAPL power capping depend on the specific power cap, specific application and phase, as well as the specific hardware. There are

general trends in the performance of power capping; a power cap above the characteristic power will not impart runtime while a power cap beneath the characteristic consumption will result in runtime dilation.

PowSched uses the relationship observed between power consumption and allocation to optimize power utilization. The basic algorithm gives less power to components consuming less than their current allocation and more power to components consuming near their current allocation. PowSched has been observed to improve HPC job throughput without the need for history, application specific models, or even awareness of a mapping between work and hardware.

One of the challenges to power scheduling research is the availability of systems and machine time for experimentation. Simulation can be used to address the availability challenge, however simulation technology must provide a way to simulate the effects of power capping on execution time. PowSim uses the relation between characteristic consumption and power cap to provide a novel power scheduling simulator.

Finally, the challenges of power scheduling strategy comparison is briefly discussed. Suggestions for the community are made, including unification for the experimental and simulation platforms used by researchers in the community. Preliminary simulation based power scheduler comparison results show that dynamic power scheduling out performs other power scheduling strategies in all cases. Ongoing work aims to provide better understandings of comparative power scheduler behavior so that appropriate power schedulers can be selected for a given HPC system and workload.

The newness of hardware overprovisioned HPC systems has presented the community with several open questions regard how to best manage power.

While this dissertation does not provide definitive answers, several important contributions are made regarding how these questions should be approached and understood. Researchers should be looking at power as a time series rather than a constant attribute of HPC applications (Chapter III). Application modeling should not be assumed to be required for power scheduling (Chapter IV). Simulation models for power scheduling research should take into account the phased behavior of applications and the nonlinear behavior of hardware power capping (Chapter V). Researchers should be including meaningful comparisons between techniques and provide explanation for when and why one solution out performs another (Chapter VI). Now is an exciting time as the research community works toward development and adoption of systemwide power management for first generation hardware overprovisioned high performance computers.

CHAPTER VIII

FUTURE

At the time this dissertation work was conducted, power scheduling was a new area of research for the HPC community. Due to the lack of preexisting research, the solutions developed in this disseration may better be understood as a baseline for future work than the optimal solution. Many of the system behaviors under power scheduling and techniques for power scheduling have yet to be studied in depth. Additionally, the standard measures for comparative evaluation of power schedulers have yet to be converged on by the HPC community. In this chapter some near term work to support power scheduling is suggested based on the experience of developing this thesis.

Power Capped Application Behavior

Very little work has been done to understand the general behavior of applications under RAPL style power caps. Fukazawa et al. (2014), looking at an MHD code under several power caps, was the only work encountered when preparing this dissertation The study associated with this dissertation was done with a limited interest and background in the applications being power capped. Poor benchmark configurations likely explain the poor power consumption of the AMG and CoMD executions shown in figure 4. Additionally, the study was conducted using only benchmarks which are known to have a different behavior than full scientific applications. A well respected study is needed in the HPC literature to understand how, when, and why power caping changes application performance.

Simulation Enrichment

PowSim, presented in Chapter V, simulates the generalized effect of power capping but many features still need to be added to investigate interesting work configurations. A communications model and an ability to replay real power consumption traces are the two most requested features.

The lack of a communications model in PowSim creates severe limitations on the workloads that can be explored using PowSim. Without a communication model, all jobs executions are as if the job was embarrassingly parallel (i.e., there is no way to encode a dependency between the computation on components). Care was taken in the design of the simulated power schedulers to result in the same power cap across all components participating in a job since power imbalance would result in incorrect simulated termination times. More interesting simulated jobs would support load imbalance, which is expected to result in power consumption imbalance as well. Minimally, a network model allowing job wide barriers to be represented would be required to simulate the synchronization between the high and low load workers. Scientific workflows would also be interesting to observe in simulation but would need a network model to express the synchronization points between workflow components.

Trace replay would be a useful feature for stronger verification of the PowSim model and model tuning. As the first simulator able to model the generalized behavior of power capped applications at a cluster scale, the desired model fit was modest and done visually. A common criticism of the work on PowSim is that the applications used for verification are synthetic and a stronger validation would involve replaying experimental traces and getting matched results. The program models used are very clean since no program progress measures

were captured from benchmark runs and simulating coordinated computation progress between components is impossible without a communication model. The processor model selected for the simulator was the simpest function that produced the expected shape for runtime dilation rather than tightly fitting the observed performance of a real processor. Stronger simulator work for power scheduling will likely require generation of a tool to convert an observed characteristic power trace into a program function and generation of an autotuner to modify the processor function to better fit the observations.

Utility of Application Awareness

PowSched, as presented in Chapter IV is completely unaware of the applications and jobs being run on the HPC cluster. The decision to completely ignore applications was largely due to implementation simplicity and the lack of research results on power capped application behavior. There is reason to believe that adding some degree of application awareness to a dynamic power scheduler like PowSched may improve power scheduler performance.

One of the unexplored concerns with PowSched is the impact of busy waiting on power scheduling performance. Busy waiting is a common technique used to increase responsiveness when one thread must wait for data in another thread or from an IO device. During a busy wait, many instructions are executed that do not aid in computation progress which may cause a component's power consumption to be artificially high. Adding information from the application or network to identify busy waiting might allow PowSched to make better power scheduling decisions.

Fairness is a common concern in the resource scheduling community that was ignored during the development of PowSched. Performance improvement when

using a power scheduler that shifts power is due in large part to the unfairness of the power allocations (i.e., large performance improvements depend on some components receiving much less power than others). A fairness constraint of interest for a power scheduler might be fairness in runtime dilation. To enforce fair runtime dilation a power scheduler would need to estimate the expected effect of a power cap on specific running jobs.

Good scaling performance for PowSched is due in part to the small amount of data needed to support the scheduling algorithm. Adding application awareness will likely increase the computational cost of making the power scheduling decision and communication cost since additional data will need to be transfered to the power scheduler. An interesting question to consider for practical power scheduler deployment is how to balance the cost of the application awareness with the scaling and performance impact to the power scheduler.

Degree of Overprovisioning

The relationship between cluster size, systemwide power cap, and workload consumption needs additional study. Reducing the cluster size while maintaining the same systemwide power cap reduces the degree of hardware overprovisioning, making the theoretical peak power consumption closer to the systemwide power limit. Intuitively an HPC system running primarily low power jobs should be operated with a higher degree of overprovisioning than an HPC system running primarily high power jobs. What specifically the degree of overprovisioning should be or how to estimate the overprovisioning is an open question. Results from researching the degree of overprovisioning will be needed for system owners to make good purchasing and operational decisions.

In Chapter IV the 70 watt case performed well however that result may not be portable to workloads composed of different applications and inputs. The most performant setting is likely due to the differences in consumption between jobs as well as the aggregate characteristic power averaging to roughly 70 watts per socket. Had different benchmark applications and configurations been used, the most performant power limit would likely have been different. Power infrastructure is part of the upfront capital cost associated with the computer and decisions need to be made at that time regarding cabling. An open question is how to tune the systemwide power limit prior to deployment of an HPC system.

In Chapter VI one of the suggested reasons for the poor performance of reservation based scheduling was the contribution of idle power to the power budget. By adjusting the cluster size or the systemwide power limit the power utilization of reservation based scheduling approaches could be tuned. Work is needed to understand the impact of dynamically scaling the cluster, via powering on and off nodes, with respect to power scheduler performance.

Comparison Studies

Some basic comparison of high level power scheduling strategies was conducted in Chapter VI using simulation. Limitations of the simulator, job scheduler, and simulated workloads limit the applicability of the results. More in depth comparison studies are needed for the HPC power scheduling community to better understand relative performance across power scheduler solutions.

Using more robust simulators, additional simulation studies should be conducted. Rather than looking at general classes of power scheduling strategy, these studies should implement specific power schedulers proposed in the literature and compare behaviors. For these studies workload characteristics should better

model real applications (e.g., load imbalances). Communication delays for the power schedulers should also be introduced to better understand differences in relative latency sensitivity.

Experimental comparison studies are also sorely needed in the community. Each power scheduler in the existing literature, that includes experimental results from real hardware, uses different and likely non-comparable workloads for evaluation. Identification of the best power scheduler for adoption in an environment cannot be determined from the existing literature. To address this challenge, a valuable research output would be a study comparing several different power schedulers on the same hardware with the same workload.

REFERENCES CITED

Backplane —. (n.d.). Retrieved 2017-05-04, from `http://www.argo-osr.org/overview/backplane/`

Bambagini, M., Bertogna, M., Marinoni, M., & Buttazzo, G. (2013). An energy-aware algorithm exploiting limited preemptive scheduling under fixed priorities. In *Industrial embedded systems (sies), 2013 8th ieee international symposium on* (pp. 3–12).

Betkaoui, B., Thomas, D. B., & Luk, W. (2010). Comparing performance and energy efficiency of fpgas and gpus for high productivity computing. In *Field-programmable technology (fpt), 2010 international conference on* (pp. 94–101).

Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., & Benini, L. (2016). Predictive modeling for job power consumption in hpc systems. In *International conference on high performance computing* (pp. 181–199).

Cao, T., He, Y., & Kondo, M. (2016). Demand-aware power management for power-constrained hpc systems. In *Cluster, cloud and grid computing (ccgrid), 2016 16th ieee/acm international symposium on* (pp. 21–31).

Chatterjee, D., DeOrio, A., & Bertacco, V. (2009). Gcs: high-performance gate-level simulation with gp-gpus. In *Proceedings of the conference on design, automation and test in europe* (pp. 1332–1337).

Cray-1. (n.d.). Retrieved 2017-05-04, from `https://en.wikipedia.org/wiki/Cray-1`

De Vogeleer, K., Memmi, G., Jouvelot, P., & Coelho, F. (2014). The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel processing and applied mathematics* (pp. 793–803). Springer.

Dutot, P.-F., Mercier, M., Poquet, M., & Richard, O. (2016). Batsim: a realistic language-independent resources and jobs management systems simulator. In *20th workshop on job scheduling strategies for parallel processing.*

Ellsworth, D. (2016). *Topics toward automated multiobjective hpc system management.* Available at http://www.cs.uoregon.edu/Reports/ORAL-201603-Ellsworth.pdf (2017/04/12). Eugene OR: University of Oregon, Computer and Information Sciences Department. (Oral Comprehensive Exam)

Ellsworth, D., Patki, T., Perarnau, S., Seo, S., Amer, A., Zounmevo, J., ... others (2016). Systemwide power management with argo. In *Parallel and distributed processing symposium workshops, 2016 ieee international* (pp. 1118–1121).

Ellsworth, D., Patki, T., Schulz, M., Rountree, B., & Malony, A. (2016). A unified platform for exploring power management strategies. In *Proceedings of the 4th international workshop on energy efficient supercomputing* (pp. 24–30).

Ellsworth, D. A., Malony, A. D., Rountree, B., & Schulz, M. (2015a). pow: System-wide dynamic reallocation of limited power in hpc. In *Proceedings of the 24th international symposium on high-performance parallel and distributed computing.* New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2749246.2749277` doi: 10.1145/2749246.2749277

Ellsworth, D. A., Malony, A. D., Rountree, B., & Schulz, M. (2015b). dynamic power sharing for higher job throughput. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (p. 80).

Enos, J., Steffen, C., Fullop, J., Showerman, M., Shi, G., Esler, K., ... Phillips, J. C. (2010). Quantifying the impact of gpus on performance and energy efficiency in hpc clusters. In *Green computing conference, 2010 international* (pp. 317–324).

Felter, W., Rajamani, K., Keller, T., & Rusu, C. (2005). A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 19th annual international conference on supercomputing* (pp. 293–302). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1088149.1088188` doi: 10.1145/1088149.1088188

Fukazawa, K., Ueda, M., Aoyagi, M., Tsuhata, T., Yoshida, K., Uehara, A., ... Inoue, K. (2014). Power consumption evaluation of an mhd simulation with cpu power capping. In *Cluster, cloud and grid computing (ccgrid), 2014 14th ieee/acm international symposium on* (pp. 612–617).

Georgiou, Y., Glesser, D., & Trystram, D. (2015). Adaptive resource and job management for limited power consumption. In *Parallel and distributed processing symposium workshop (ipdpsw), 2015 ieee international* (pp. 863–870).

Gonsiorowski, E., Carothers, C., & Tropper, C. (2012). Modeling large scale circuits using massively parallel discrete-event simulation. In *2012 ieee 20th international symposium on modeling, analysis and simulation of computer and telecommunication systems* (pp. 127–133).

Hoffmann, H. (2013). Racing and pacing to idle: an evaluation of heuristics for energy-aware resource allocation. In *Proceedings of the workshop on power-aware computing and systems* (p. 13).

Hoffmann, H., Maggio, M., Santambrogio, M. D., Leva, A., & Agarwal, A. (2013). A generalized software framework for accurate and efficient management of performance goals. In *Embedded software (emsoft), 2013 proceedings of the international conference on* (pp. 1–10).

Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., ... others (2015). Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (p. 78).

Jette, M. A., Yoo, A. B., & Grondona, M. (2002). SLURM: Simple Linux Utility for Resource Management. In *In lecture notes in computer science: Proceedings of job scheduling strategies for parallel processing (jsspp) 2003* (pp. 44–60). Springer-Verlag.

Kim, D., Izraelevitz, A., Celio, C., Kim, H., Zimmer, B., Lee, Y., ... Asanović, K. (2016). Strober: fast and accurate sample-based energy simulation for arbitrary rtl. In *Proceedings of the 43rd international symposium on computer architecture* (pp. 128–139).

Kim, N. S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J. S., ... Narayanan, V. (2003). Leakage current: Moore's law meets static power. *computer*, *36*(12), 68–75.

Lucero, A. (2011). Simulation of batch scheduling using real production-ready software tools. *Proceedings of the 5th IBERGRID*.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., ... Werner, B. (2002). Simics: A full system simulation platform. *Computer*, *35*(2), 50–58.

Marathe, A., Bailey, P. E., Lowenthal, D. K., Rountree, B., Schulz, M., & de Supinski, B. R. (2015). A run-time system for power-constrained hpc applications. In *International conference on high performance computing* (pp. 394–408).

November 2016 — top500 supercomputer sites. (n.d.). Retrieved 2017-05-04, from `https://www.top500.org/lists/2016/11/`

Patki, T., Bates, N., Ghatikar, G., Clausen, A., Klingert, S., Abdulla, G., & Sheikhalishahi, M. (2016). Supercomputing centers and electricity service providers: a geographically distributed perspective on demand management in europe and the united states. In *International conference on high performance computing* (pp. 243–260).

Patki, T., Lowenthal, D. K., Rountree, B., Schulz, M., & de Supinski, B. R. (2013). Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international acm conference on international conference on supercomputing* (pp. 173–182).

Patki, T., Lowenthal, D. K., Sasidharan, A., Maiterth, M., Rountree, B. L., Schulz, M., & de Supinski, B. R. (2015). Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th international symposium on high-performance parallel and distributed computing* (pp. 121–132). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2749246.2749262` doi: 10.1145/2749246.2749262

Rountree, B., Ahn, D. H., de Supinski, B. R., Lowenthal, D. K., & Schulz, M. (2012). Beyond DVFS: A First Look at Performance Under a Hardware-Enforced Power Bound. In *Parallel and distributed processing symposium workshops & phd forum (ipdpsw), 2012 ieee 26th international* (pp. 947–953).

Rountree, B., Lownenthal, D. K., De Supinski, B. R., Schulz, M., Freeh, V. W., & Bletsch, T. (2009). Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on supercomputing* (pp. 460–469).

Sakamoto, R., Cao, T., Knoda, M., Inoue, K., Ueda, M., Patki, T., ... Schulz, M. (2017). Production hardware overprovisioning: Real-world performance optimization using an extensible power-aware resource management framework. In *Proceedings of the 31st internation parallel and distributed processing symposium.*

Sarood, O., Langer, A., Gupta, A., & Kale, L. (2014). Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Proceedings of the international conference for high performance computing, networking, storage and analysis* (pp. 807–818).

Savoie, L., Lowenthal, D. K., d. Supinski, B. R., Islam, T., Mohror, K., Rountree, B., & Schulz, M. (2016, May). I/O Aware Power Shifting. In *2016 ieee international parallel and distributed processing symposium (ipdps)* (p. 740-749). doi: 10.1109/IPDPS.2016.15

Sventek, J., & Koliousis, A. (2012). Unification of publish/subscribe systems and stream databases: the impact on complex event processing. In *Proceedings of the 13th international middleware conference* (pp. 292–311).

Tiwari, A., Laurenzano, M., Peraza, J., Carrington, L., & Snavely, A. (2012). Green queue: Customized large-scale clock frequency scaling. In *Cloud and green computing (cgc), 2012 second international conference on* (pp. 260–267).

Trofinoff, S. (2015). Using and Modifying the BSC Slurm Workload Simulator. *Slurm User Group Meeting*.

Vulcan - bluegene/q, power bqc 16c 1.600ghz, custom interconnect — top500 supercomputer sites. (n.d.). Retrieved 2017-05-04, from `https://www.top500.org/system/177732`

Wood, C., Sane, S., Ellsworth, D., Gimenez, A., Huck, K., Gamblin, T., & Malony, A. (2016). A scalable observation system for introspection and in situ analytics. In *Proceedings of the 5th workshop on extreme-scale programming tools* (pp. 42–49).

Zhang, H., & Hoffmann, H. (2016). Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *ACM SIGPLAN Notices*, *51*(4), 545–559.