SANDIA REPORT

SAND2017-10191
Unlimited Release
Printed August, 2017

FY17 CSSE L2 Milestone Report: Analyzing Power Usage Characteristics of Workloads Running on Trinity

Ryan E. Grant, James H. Laros III, Michael Levenhagen, Stephen L. Olivier, Kevin Pedretti, Lee Ward, Andrew J. Younge

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2017-10191
Unlimited Release
Printed August, 2017

FY17 CSSE L2 Milestone Report: Analyzing Power Usage Characteristics of Workloads Running on Trinity

Ryan E. Grant, James H. Laros III, Michael Levenhagen,
Stephen L. Olivier, Kevin Pedretti, Lee Ward, Andrew J. Younge

Abstract

This report summarizes the work performed as part of a FY17 CSSE L2 milestone to investigate the power usage behavior of ASC workloads running on the ATS-1 Trinity platform. Techniques were developed to instrument application code regions of interest using the Power API together with the Kokkos profiling interface and Caliper annotation library. Experiments were performed to understand the power usage behavior of mini-applications and the SNL/ATDM SPARC application running on ATS-1 Trinity Haswell and Knights Landing compute nodes. A taxonomy of power measurement approaches was identified and presented, providing a guide for application developers to follow. Controlled scaling study experiments were performed on up to 2048 nodes of Trinity along with smaller scale experiments on Trinity testbed systems. Additionally, power and energy system monitoring information from Trinity was collected and archived for post analysis of "in-the-wild" workloads. Results were analyzed to assess the sensitivity of the workloads to ATS-1 compute node type (Haswell vs. Knights Landing), CPU frequency control, node-level power capping control, OpenMP configuration, Knights Landing on-package memory configuration, and algorithm/solver configuration. Overall, this milestone lays groundwork for addressing the long-term goal of determining how to best use and operate future ASC platforms to achieve the greatest benefit subject to a constrained power budget.

Acknowledgment

Numerous people have helped us throughout the course of this work. In particular, we would like to thank: Simon Hammond for his many technical discussions with us and for providing early access versions of MiniMD, LULESH, and MiniMD mini-apps instrumented with the Kokkos profiling interfaces. Christian Trott for helping us understand how to best configure our mini-app experiments and interpret results. Andrew Bradley for helping us understand how to configure and measure the performance of the `SPARC` application and for providing early access versions of `SPARC` with new optimized Trilinos solvers. Micah Howard for helping us get up and running with `SPARC` and for suggesting appropriate input problems to evaluate. Amanda Bonnie, Jim Brandt, Adam DeConinck, Ann Gentile, Jason Repik, and Kevin Stroup for designing and setting up a method to retain long-term power and energy system monitoring information for Trinity and its related testbed systems. Jason Repik, Steven Martin, Matthew Kappel, and the rest of the Cray Advanced Power Managment Non-recurring Engineering (NRE) team for their numerous technical discussions with us and for developing and deploying robust implementations of the PowerAPI for Trinity. Courtenay Vaughan for helping us to configure and interpret our mini-app and application power profiling experiments.

Contents

List of Figures

List of Tables

Executive Summary

The overall goal of this work was to utilize the Advanced Power Management (APM) capabilities of the ATS-1 Trinity platform to understand the power usage behavior of ASC workloads running on Trinity and gain insight into the potential for utilizing power management techniques on future ASC platforms.

Milestone Description

As written in the ASC Implementation Plan, the milestone description is as follows:

> In anticipation of practical power consumption limits, the ASC program requires guidance for power management of future platforms and applications. The Trinity program's Advanced Power Management (APM) Non-recurring Engineering (NRE) project is delivering integrated power monitoring and control capabilities in the Trinity platform, building on prior work developing the Power API. This milestone will utilize these capabilities to collect information on the power usage characteristics of the ASC production workload running on Trinity. Methods will be developed to assist with understanding and applied to assess the potential impact of power-constraints in future ASC platforms. This milestone will lay groundwork for addressing the long-term goal of determining how to best use and operate future ASC platforms to achieve the greatest benefit subject to a constrained power budget.

Impact Statement

This work has provided insight into the power usage characteristics of ASC workloads running on ATS-1 Trinity hardware. Energy efficiency was examined from the perspective of compute node architecture, run configuration (e.g., OpenMP layout, on-package memory configuration), algorithm, software power management control, and scale. Each of these dimensions were quantified and found to be important contributors to overall energy efficiency improvements. For example, the SNL/ATDM SPARC application was found to perform 19% better and use 40% less energy to solution when running on Trinity's Knights Landing compute nodes compared to running on Haswell nodes. This work indicates significant potential for managing power as a resource in future ASC platforms. None of the workloads evaluated, even highly-tuned applications such as SPARC, utilized more than 90% of their allocated

power budget, with most using 60-85%. Trinity's node-level power capping mechanism was evaluated and found to function effectively so long as the cap level was set at or above the application's natural power usage. Power capping could therefore be a useful tool for reclaiming unused power headroom—the difference between wallplate power and an application's actual usage—to power additional compute nodes in future platforms. For example, a Trinity-like system with a 20 MW power budget and wallplate-rated 400 W compute nodes could potentially power an additional 25% more nodes by setting a 320 W power cap per node with minimal performance impact for many ASC workloads.

Summary of Work Done

Experiments were performed to understand the power usage behavior of mini-applications and the SNL/ATDM SPARC application running on ATS-1 Trinity Haswell and Knights Landing compute nodes. Techniques were developed to instrument application code regions of interest using the Power API (Publication 1) together with the Kokkos profiling interface and Caliper annotation library. A taxonomy of power measurement approaches was identified and presented, providing a guide for application developers to follow (Publication 2). Controlled scaling study experiments were performed on up to 2048 nodes of Trinity along with smaller scale experiments on Trinity testbed systems. Additionally, power and energy system monitoring information from Trinity was collected and archived for post analysis of "in-the-wild" workloads (Publication 3). Results were analyzed to assess the sensitivity of the workloads to ATS-1 compute node type (Haswell vs. Knights Landing), CPU frequency control, node-level power capping control, OpenMP configuration, Knights Landing on-package memory configuration, and algorithm/solver configuration (Publication 4, this document).

Publications:

1. R. Grant, M. Levenhagen, S. Olivier, D. DeBonis, K. Pedretti, J. Laros, "Standardizing Power Monitoring and Control at Exascale," Journal Article, IEEE Computer, Vol. 49, No. 10, pp. 38–46, October 2016.

2. R. Grant, J. Laros, M. Levenhagen, S. Olivier, K. Pedretti, L. Ward, A. Younge, "Evaluating Energy and Power Profiling Techniques for HPC Workloads," Conference Paper, International Green and Sustainable Computing Conference, October 2017.

3. A. DeConinck, H. Nam, D. Morton, A. Bonnie, C. Lueninghoener, J. Brandt, A. Gentile, K. Pedretti, A. Agelastos, C. Vaughan, S. Hammond, B. Allan, M. Davis, J. Repik, "Runtime collection and analysis of system metrics for production monitoring of Trinity Phase II," Conference Paper, Cray Users Group, May 2017.

4. R. Grant, J. Laros, M. Levenhagen, S. Olivier, K. Pedretti, L. Ward, A. Younge, "FY17 CSSE L2 Milestone Report: Analyzing Power Usage Characteristics of Workloads Running on Trinity," Sandia Technical Report, September 2017.

Path Forward

As mentioned above, the successful completion of this effort has given Sandia increased understanding of the power measurement and control capabilities of Trinity and has characterized the power usage behavior of ASC workloads running on Trinity. This insight will be used to help specify power-related requirements of future platforms. The results of this milestone also suggest significant potential for applying dynamic power management techniques to the SPARC application and a next step is to determine how well this works in practice on Trinity. Finally, the methods and tools developed by this milestone will be used to analyze additional ASC workloads.

Nomenclature

DOE United States Department of Energy

NNSA National Nuclear Security Administration, a semi-autonomous agency within DOE

ASC Advanced Simulation and Computing, a program of NNSA

CSSE Computational Systems & Software Engineering, a subprogram within ASC

ATDM Advanced Technology Development and Mitigation, a subprogram within ASC

FY17 Fiscal Year 2017, October 1, 2016 to September 30, 2017

L2 Milestone Level 2 milestone, a significant deliverable requiring formal review

SNL Sandia National Laboratories

LANL Los Alamos National Laboratory

NRE Non Recurring Engineering

HPC High Performance Computing

HBM High Bandwidth Memory, used generically in this document, not the JEDEC standard

Chapter 1

Introduction

This report summarizes the results of an FY17 ASC CSSE L2 milestone to analyze the power usage characteristics of workloads running on Trinity, a 42.3 PetaFLOP NNSA supercomputer based on the Cray XC40 architecture. A description of this milestone is as follows:

> In anticipation of practical power consumption limits, the ASC program requires guidance for power management of future platforms and applications. The Trinity program's Advanced Power Management (APM) Non-recurring Engineering (NRE) project is delivering integrated power monitoring and control capabilities in the Trinity platform, building on prior work developing the Power API. This milestone will utilize these capabilities to collect information on the power usage characteristics of the ASC production workload running on Trinity. Methods will be developed to assist with understanding and applied to assess the potential impact of power-constraints in future ASC platforms. This milestone will lay groundwork for addressing the long-term goal of determining how to best use and operate future ASC platforms to achieve the greatest benefit subject to a constrained power budget.

The goals of this L2 build upon functionality that has been in development at Sandia National Laboratories for several years. The capabilities of Trinity to gather power measurements and the work done on the Power API were critical to the success of this L2. Understanding the history of the build up of capabilities that has enabled the work done in this L2 is important to understand the context in which the work was performed.

The basis for work to understand the power/energy characteristics of large compute systems began many years ago at Sandia in understanding that we needed to include power as a first-class consideration in every aspect of High Performance Computing (HPC) as it was predicted to become a major system constraint in the near future. With further investigation into this areas, the lack of standard interfaces for power measurement and control became more evident. Sandia National Laboratories (Sandia) began investigating how to address this gap in 2012 by evaluating use cases revealed by early research in this area. The result of this effort was a document[45] that outlined the scope and interfaces that a power application programming interface should address if it were to meet the demanding

needs of HPC. Immediately following this effort, in January 2014, a team at Sandia formally began creating the *High Performance Computing - Power Application Programming Interface* specification[46] (Power API).

The Power API targets a broad range of interfaces ranging from low level capabilities exposed by technology providers to higher level interfaces that address use cases involving end users, applications and work-load managers, for example. Six months after focusing primarily on the core interfaces of the specification an early draft was vetted by a range of technology providers (Adaptive Computing, Cray, AMD, Penguin Computing, Intel, and IBM), laboratory (National Renewable Energy Laboratory, Oak Ridge National Laboratory) and university (University of New Mexico) representatives. The technology providers were specifically targeted since the success of any proposed standard depends on it being implemented. However, community involvement is just as critical to drive the development of the specification in an unbiased manner and ensure that it remains vendor-neutral. One of the primary goals of the Power API is to present a set of portable interfaces, shielding the end user, no matter what role they serve, from vendor specific implementation details.

During the same time that Sandia was preparing to begin development of the Power API, the Alliance for Computing at Extreme Scale (ACES), a collaboration between Sandia and Los Alamos National Laboratory (Los Alamos), was preparing to release a Request for Proposal (RFP) for Trinity, the DOE's National Nuclear Security Administrations (NNSA) first Advanced Technology System (ATS-1). An important aspect of this new effort by the DOE is the investment in advanced technologies in the form of non-recurring engineering (NRE). A portion of funding for each platform in the ATS line is invested in advanced technologies selected for their potential impact to the DOE/NNSA mission. For the Trinity (ATS-1) procurement, Burst-Buffer and Power were selected as the two focuses of NRE investment. This report uses the work developed in collaboration with Cray on the Trinity Advanced Power Management (APM) NRE program.

Measuring and understanding the power profiles of codes of interest is the first major goal of this L2. We have used a variety of codes and benchmarks to obtain an understanding of codes in general as well as a detailed study of SPARC to understand real application power profiles on traditional and many-core architectures. In obtaining these profiles we have developed a taxonomy for understanding the methods of gathering power profiles and have developed best practices for power measurements based on the required level of detail. We describe this methodology in Section 3.

This report will begin by providing background material on power measurements in Section 2 including examples of the application of measurements to system optimizations. Section 2 will also introduce the Power API including examples of its use and implementation details. Section 3 provides a taxonomy for measurement techniques as well as best practices for using these techniques to obtain the desired data outputs. Next, we discuss the Trinity power management infrastructure in detail in Section 4. Section 5 illustrates power profiles for mini-applications of interest and Section 7 details results from testing using SPARC. We conclude with discussion on the lessons learned from this L2 in Section 8 and final conclusions in Section 9.

Chapter 2

Background – Power Measurement Techniques

The fundamental concepts and mechanisms behind power measurement and control are numerous and elaborate. This is furthermore the case when considering such energy considerations within the context of extreme-scale HPC resources. As such, this section walks through the fundamentals of advanced power management with a particular focus on some of the latest supercomputing resources today.

Power Sampling

Power measurement of system components is a topic that has been studied for many years. Accurate power measurements are essential when energy budgets are regulated and finite, such as in mobile devices. Energy usage is important in real-time applications to identify opportunities to reduce energy usage while satisfying required deadlines. Commercial services such as cloud providers focus on overall system efficiency for cost optimization. Power usage is also of interest to sites hosting HPC platforms, as such sites must meet statutory regulations governing energy efficiency and seek to minimize electricity costs. However, US DOE HPC facilities are often more concerned with limitations that constrain the amount of power that can be provided to a given platform.

Approaches to power measurement are varied, and provide different types and rates of data. Out-of-band measurement is the easiest approach to understand. It uses equipment external to the compute resource to measure power consumption without perturbing computational performance. Classic examples of out-of-band measurement include devices like WattsUp! [18], WattProf [66], PowerInsight [44], and integrated devices like IBM's power measurement capabilities [8]. Out-of-band measurements avoid perturbing the ongoing computation, but they may not provide easily accessible information to the running processes. Since these devices are necessarily not part of the CPU, they must be interrogated over external device buses for data instead. Historically, simple out-of-band measurement techniques have had relatively low sampling rates, however new integrated designs have greatly improved sampling rate.

In-band measurement uses device-level integrated measurement capabilities, such as Intel's Running Average Power Limit (RAPL), or AMD's Advanced Power Management (APM). They provide real or estimated measurements of energy consumption through device-level interfaces. RAPL and APM use CPU counters to express energy usage, and can provide separate core, package and DRAM measurements. In-band measurements require active participation of the compute cores in a system to gather data on a regular basis. Therefore, point-in-time samples require frequent intervention to record the values in counters on the device. This corresponds to a read of a CPU register on both Intel's and AMD's solutions. If only total energy consumption of the whole application is needed, rather than point-in-time samples, then in-band measurement can have very little impact on computational performance. Point in time samples require several reads per second if the device's maximum sampling rate is used.

In-band measurement through CPU counters can often provide both CPU and memory subsystem energy measurement [4, 14]. This is not always possible with out-of-band measurement, depending on where the external measurement hardware is placed and its capabilities. Out-of-band measurement can capture whole node energy profiles more easily, while this is generally not possible for in-band measurement that relies on CPU counters. Whole node energy can be useful when other components such as network or motherboard chip set consume a large amount of the power budget for a node.

Application instrumentation and profiling can take multiple forms. Timestamping is a common practice amongst application developers to understand the performance characteristics of their code. Other more in depth profiling techniques and tools such as Intel's Vtune [67] or Cray's CrayPat [39] allow deeper inspection into program behavior through call-graph traces and CPU performance monitoring counter data. The Power API [26] can provide a portable solution to application level power measurement when application region hints are integrated with power measurement through the framework.

The concept of investigating energy and power consumption of large-scale HPC resources is not a new one. In the literature, most related work has addressed real power measurement using solely out-of-band [73, 15, 56, 57, 28] or in-band [69, 29, 77, 25] techniques. Some work has also sought to validate in-band measurements using out-of-band measurements at the same time to determine the overall accuracy of the measurements or integrated power model [14]. Previous work has used application instrumentation with power measurements to estimate energy usage to within 10% of actual consumption [37]. Several works have used estimated power values to determine the system energy consumption [9, 76, 24]. Other work uses power measurements to illustrate methods for operating power constrained systems [60, 23]. Work has also been done specifically on power consumption on very large systems with custom power measurement frameworks [43, 47]. Several power estimation frameworks/simulators exist as well such as WATTCH [11], and SST [34]. However, these simulators rely on estimations of energy consumption and therefore have high margins of error, particularly for architectures that are not the explicit target of the simulation. Older simulators were based on power profiles from DEC Alpha CPUs which may no longer be accurate for modern architectures. APIs have addressed the topic of gathering power and energy data

from systems including the Power API [41], Redfish [32], CapMC [53] and AMESTER [40]. Many vendors also have proprietary interfaces to specific hardware. CapMC and AMESTER are similar to these as they are only for Cray and IBM systems respectively, but they do work with a range of systems as opposed to a specific measurement hardware device. Cray's CAPMC allows for power monitoring and control capabilities on Cray systems, it is a RESTful interface that uses JSON for issuing and interpreting commands [51, 30, 52]. Redfish allows for collection of a variety of metrics on system performance on large parallel machines, however it is not specific to power measurement. Redfish uses a JSON interface that can be used for management of generic cloud infrastructures and provides basic support for managing power reserouces in a cloud environment. The Power API is an HPC specific power measurement and control API, and can be used by higher level APIs like Redfish or can serve as a portable interface to lower level APIs like CapMC and AMESTER, as well as lowest level readings like RAPL counters. Other APIs have been developed to gather power/energy data from custom measurement hardware. Most of these APIs are device specific, such as Powermon [7] and Powerpack [22]. Also, more recent out-of-band measurement devices have had specific targeted APIs, like PowerInsight [44] and WattProf [66]. Out-of-band measurement devices have provided fine-grained measurement and control via their device APIs, and have allowed remote measurement to occur, such as the PowerInsight [44] specific piapi. Such APIs allow for collection from their respective devices in an efficient manner, and as such are used in the Power API implementation through device-specific plugins. In addition to external or dedicated monitoring devices, some hardware has built in power/energy monitoring and management functionality. Measurement directly from Running Average Power Limit (RAPL) control mechanisms on Intel processors have been introduced through MSR-safe [70]. Such user-level access to machine specific registers (MSRs) is critical to allow in-band energy measurement. The Power API provides similar functionality and can utilize mechanisms like MSR-safe as plugins to allow for user-level access to privileged information. HP's iLO [75] is a proprietary out-of-band interface for taking measurements, including power/energy on HP clusters.

Previous work has been limited by the measurement capabilities available on extreme-scale HPC platforms used. For example, Leon et al. [48] used out-of-band measurement and application region marking. Additional work used in-line performance measurement counters (not power counters) to better understand the behavior of individual regions [49]. However, this work did not explore the capabilities nor trade-offs of in-band measurement, and used only traditional multicore CPUs but on a variety of architectures. The work described herein is the first known to address both in-band and out-of-band measurement techniques together on the same hardware and coupled with application profiling for each measurement. Furthermore, this work also contributes a detailed taxonomy to detail how, when, and why HPC application developers can accurately evaluate power consumption.

Power measurement/monitoring APIs have been developed in the past. Global Energy Optimization (GEO) is a energy optimization framework developed by Intel [17]. It manages job power bounds in a cluster while also attempting to increase performance by tuning the power consumption of systems involved in a job. GEO has a scalable collection mechanism that is based on MPI communication for individual job measurement collection. Unlike the

Power API, GEO's external interfaces are not proposed as a standard [17], though they are open-source. PAPI [13] is an example of a high-level, portable API for performance monitoring that seeks to solve similar problems to the PowerAPI, but in the performance realm instead of power.

Using Power Measurements for Optimization

The area of energy optimization is a mature area. Several energy saving runtime techniques [21, 35, 36, 50, 38, 72] having been proposed and implemented. Energy saving techniques such as DVFS [63] and clock throttling [58] have been used to provide power state governors for operating systems. Our previous work [27] has evaluated many different energy saving techniques in addition to providing a survey of these techniques. We provide a breif overview of these techniques here in order to demonstrate how power control mechanisms can be applied for overal system benefit from both a energy and performance stanpoint.

CPUSPEED [20] is the default power management system for most Linux distributions. It uses basic power-state governors to dynamically determine when to change power states on a system. CPUSPEED was not designed with HPC in mind and therefore is more aggressive than HPC energy saving techniques in terms of potential energy savings. It is the default power state adjustment mechanism in Linux. CPU MISER [21], is a solution that examines execution phases during runtime and makes decisions to lower CPU clock frequencies based on observed phase states. It provides the ability for a user to specify a maximum acceptable slowdown and attempts to adjust the power states such that the total runtime does not fall below the cutoff. As is the case with many runtime energy saving methods, the best energy savings occur for communication bound applications, with limited opportunities to save energy for CPU bound applications. PART [35] was an early entrant into the energy-saving HPC arena. It provided an algorithm for bounding the slowdown of applications while attempting to save energy using the runtime history of an application. It should be noted that the results for PART are only those for the CPU energy consumption, not total system energy. Therefore, the overall energy savings reported for this method will be higher than those for system level energy measurements, as only the CPU energy consumption is reduced using PART. ECOD [36] is another performance-bounding and workload predicting algorithm for energy saving. It is more accurate that past methods [21, 35], and provides tighter performance-bounding variance. Unlike PART, this algorithm does not consider the entire workload runtime history when making power state decisions. The method presented in [50] concentrates on communication periods and opportunities to reduce CPU power consumption during *blocking* MPI communication. This method can operate in two modes, one is on-the-fly and the other uses a priori information about the profile of an application. Although the profiling method is more slightly more efficient, the on-the-fly method is used for comparison here, as one cannot calculate the energy used in the profiling runs. Jitter [38] provides a method similar to that of the previously discussed method, in that it attempts to exploit slack in MPI programs. However, it does not exploit communication slack, only that caused by inter-node load imbalance. It requires manual changes to the application source

code, and only works for iterative programs. Green Queue [72] is a new (2012) energy savings approach for HPC designed for scalability. It utilizes an intra-node methodology using phase detection through profiling and offline simulation analysis of applications. It uses an SQL database to store profiles on past application runs and simulation analysis. This makes it somewhat different from the other approaches examined as it requires energy consumption to generate the profiling and simulation data. Adagio [68] is an integration of multiple methods, including Jitter, that exploits slack in MPI programs for both load imbalance and communication slack. It exploits advanced DVFS techniques to provide improved energy savings by allowing a "task" (a slice of execution between two MPI blocking operations) to be run over multiple frequencies, thereby better approximating the ideal frequency for which that code should be run. It is designed to minimize the performance impact of energy-saving, rather than finding a tradeoff between energy consumption and performance loss.

The Power API

The *"High Performance Computing - Power Application Programming Interface Specification"* [26] was developed at Sandia National Laboratories in collaboration with major vendors, laboratory and academic partners. The organizational structure behind the API's development follows those of several other very successful standards, with a vendor-neutral national laboratory funded through the federal government of the United States leading an effort that has community input and public review feedback with the goal of becoming a public community-led standard. The goal of developing a common API for power measurement and control was realized with the first specification release in 2014. The API continues to evolve and grow as further capabilities are added and new language bindings are supported. While most existing interfaces are mature, work continues on some of the highest level interfaces as research and the development of future systems better informs the high level reporting requirements. The Power API is specified primarily as a C API, as C is the preferred language for low-level software on HPC systems and is universally supported, however, alternative language bindings such as Python are provided. Implementations of the Power API may internally use whatever language is most convenient. For example the Power API reference implementation developed by Sandia is primarily written in C++ with user-visible C interfaces provided externally.

High-Level Design

The design philosophy behind the Power API is to allow for flexibility in future system architectures and power measurement and control capabilities. As such, it is designed to allow great freedom in describing system architectures and handling requests to many devices, including requests for information or capabilities that may not be supported in today's systems but are expected to be available in future Exascale class machines.

The Power API creates a system description in a hierarchical form, with basic sup-

ported, but not mandatory, objects starting with a platform and descending through the system to cabinets, boards, nodes, sockets, and core object types. Additional devices can be inserted where applicable in the system description, including memories, network interfaces, accelerators and more generic power plane objects. Power planes provide a useful control/measurement point for cases where power measurements or controls are aggregated amongst underlying objects. An example of this would be a power plane for a CPU where the individual cores do not have individual measurements available, but an aggregate measurement is available (two cores per power plane, for example. This hierarchical form can be expressed statically or can be built dynamically through a system description tool. The design of the hierarchy allows for current tools such as hwloc [12] to accurately describe current systems while still allowing for possible future system architecture changes that depart radically from contemporary systems. An example of this hierarchy in Figure 2.1 shows a simple small scale system using core Power API object types.

User interaction with the API follows the philosophy of making hard tasks possible to accomplish and easy tasks easy to accomplish. The Power API design also chooses potential complexity in implementation rather than in the user interface whenever possible. The rationale behind this approach is that the implementation can better deal with complexity once, where experts can be utilized more easily, and avoid complexity in the code that will be written more often, the Power API calls themselves at the user level.

Roles

The roles that users of the Power API can assume best illustrate the encompassing nature of the API. A diagram showing all of the roles and how they interact with different levels of the interface is shown in Figure 2.2. In the figure, role names are often proceeded with "HPCS" as this particular example is for High Performance Computing Systems. However, the Power API can be easily used on systems from large commercial data centers to individual desktops. One of the high level roles, Accounting, provides an interface for generating reports and metrics of the system at different levels of granularity, from the whole system down to individual components. The System Manager role is provided to represent the responsibility of dictating overall system-level policies, such as scheduling priorities and facility limitations. The Administrator role represents the traditional IT system administrator function, managing day to day operation of the system, but from a power and energy perspective. This interface is aimed at providing easy access to control power throughout the system on both a coarse and fine-grained basis as well as providing useful information on power measurements to better understand immediate and long term system needs. The administrator can choose between C and Python interfaces for these tasks, where Python scripts are desirable for quick unique scripting requirements, the C interface is useful for building command line tools requiring high-performance for frequently used operations. The Resource Manager role is oriented toward resource managers and job schedulers. Policy decisions communicated by the System Manager are translated into job policy on the running system, such as power caps that represent time of day differences in power costs. Interfaces are available for the Resource Manager to mine information or leverage information provided

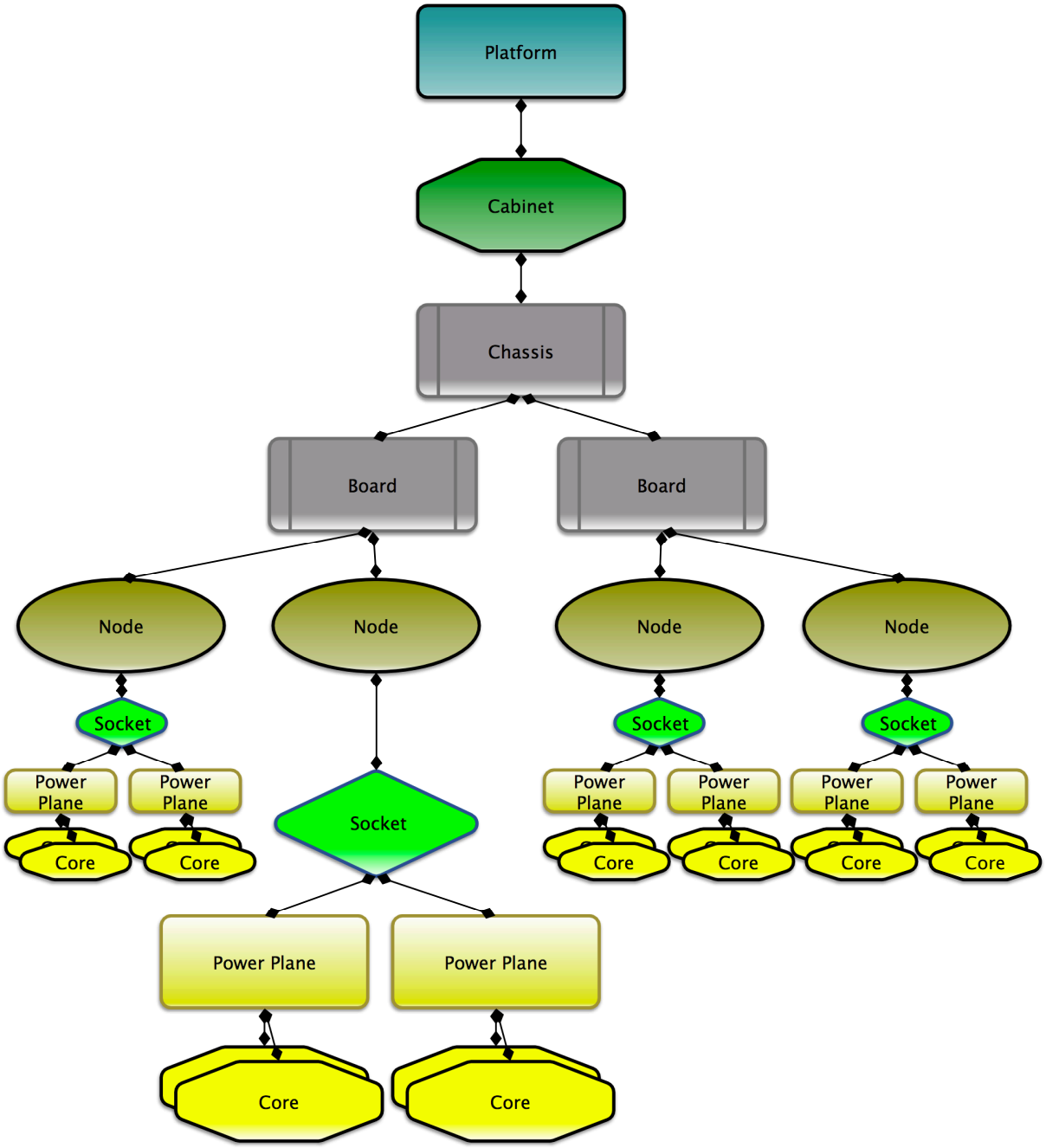


Figure 2.1. Example of a simple machine hierarchy for the Power API.

by the system from the Monitor and Control role (for example). The next role is the generic User role. This interface provides all of the capabilities potentially exposed to end-users of an HPC system, primarily taking measurements and potentially controlling power within

bounds enforced by the system administrators or resource manager. The Application role is the first-person interface for user applications running on the system. In many ways this is similar to the User role, but with lower-level requirements where necessary for describing the needs of HPC applications. The last two roles directly interact with hardware and expose the fundamental measurement and control capabilities of the system. While user-space level hardware interaction may be possible on some systems and therefore enable other roles to interact with hardware directly, the Operating System (OS) and Monitor and Control roles are required to interact with hardware on all systems. This is due to the high level of privileges required to interact with most hardware. The OS role is primarily a node centric role while the Monitor and Control role is a broader focused system level management role. The Monitor and Control role is largely analogous to traditional Reliability Availability and Serviceability (RAS) systems.

Using the Power API

The Power API provides many core functions shared by the different interfaces offered. Upon initialization, the user is presented with a context, basically the user's window into the functionality available to their role/user combination by the implementation. The system view exposed likewise depends on the combination of the role and the individual user. For example, an application may only have access to the hardware (the node) that it is currently executing on. A system administrator would commonly have access to all platform resources. Navigation functions allow any user (or role) to navigate to the device (object) in the system hierarchy with which the user seeks to interact. The API provides functions for creating groups of objects, which can then operated upon using group functions that mirror the capabilities of functions used to interact with individual system objects. Groups can also be combined using several different functions to create unions or intersections and differences of the two groups.

Each object in the system hierarchy has attributes associated with it, which correspond to measurement or control interfaces available, and exposed, for that individual object. For example, for a CPU core object, valid attributes may include power, energy, performance state, sleep state and low level measurements such as voltage and current.

The metadata about object-attribute pairs can be easily fetched using the Power API metadata interface. Metadata is particularly important for determining the utility of data obtained using the Power API interfaces, such as the frequency or accuracy of measured values.

Figure 2.3 demonstrates an example of using the Power API metadata and attribute interfaces. After initializing a Power API context, the `PWR_CntxtGetEntryPoint()` interface is used to get the object representing the caller's entry point for navigating the machine hierarchy. In the interest of space, this example assumes the entry point returned is the local node's object but in general the Power API's navigation interfaces would be used to find the desired object. Next, the `PWR_ObjAttrGetMeta()` metadata interface is used to retrieve the

expected accuracy of energy measurements obtained from the local node's `PWR_ATTR_ENERGY` attribute. Finally, the `PWR_ObjAttrGetValue()` attribute interface is used to measure the energy consumed by the `do_work()` function. Since `PWR_ATTR_ENERGY` is an energy counter, the difference of its value between calls is used to calculate the energy consumed.

Another powerful use case for the Power API is the collection of statistics. The API provides a statistics interface that allows the user to gather statistics on individual objects or groups of objects for individual attributes. These statistics, such as sum, max, min, and average, can then be further reduced if desired to provide averages of sums on multiple objects or find a maximum of maximums and the object that it occurred on.

High-level application interfaces are provided to allow the application to communicate to the system (the OS or potentially an intelligent run-time layer). These "hints" include informing the system about application phases such as serial or parallel regions that can be exploited at the node level to potentially deliver more performance and power savings. The application could also hint that it is in a communication phase on a particular node which would allow node level alterations but also allow an intelligent runtime system to coordinate between the nodes allocated to shift additional power to nodes which remain in computation phases, for example.

Implementation

While commercial vendor implementations of the Power API are in development, an open source reference implementation is available for early adopters. The Power API reference implementation is architected to support the core functions of the API in a single implementation, with multiple measurement device support implemented through a plugin architecture. This allows for rapid integration of new measurement devices as well as power control points. The current implementation supports many low-level hardware power measurement devices, from common off-the-shelf solutions such as WattsUp meters, to device/vendor specific methods such as Intel's RAPL. Support for more comprehensive out-of-band power measurement devices, such as PowerInsight from Penguin Computing, is also provided.

The reference implementation is mostly complete. Core functions, aside from historical data collection and a subset of statistics functions for certain objects, are implemented. The reference implementation is currently integrating and optimizing large-scale collection methods. The current functionality in the reference implementation is sufficient for most real-time data measurement and control use cases. The reference implementation is currently deployed as part of the Tri-lab operating system (TOSS) and is running on several test and production platforms at DOE laboratories. Reference implementation development and research is conducted at small scale on many of Sandia's Advanced Architecture Test Bed clusters [1]. Large scale testing and research is being accomplished on the production Skybridge cluster at Sandia National Laboratories.

The reference implementation incorporates a scalable framework for collecting measure-

ments from many different objects in a group at one time. Although aggregation of results is implicitly embedded in the object hierarchy of the Power API, distributing the aggregation at multiple points instead of a single aggregation point is an implementation optimization. The scalable distributed aggregation method for the implementation has shown good results at this early stage, before significant performance optimization has been completed. Figure 2.4 shows the initial scaling of collecting basic energy samples from a number of nodes in a large system. The microbenchmark used for the results in Figure 2.4 measures the time that 1000 `PWR_ObjAttrGetValue()` requests take to complete and divides by 1000. The test was performed on Chama, a production supercomputer at Sandia National Laboratories.

The Power API Reference Implementation was developed alongside the specification and is publicly available at `http://powerapi.sandia.gov`.

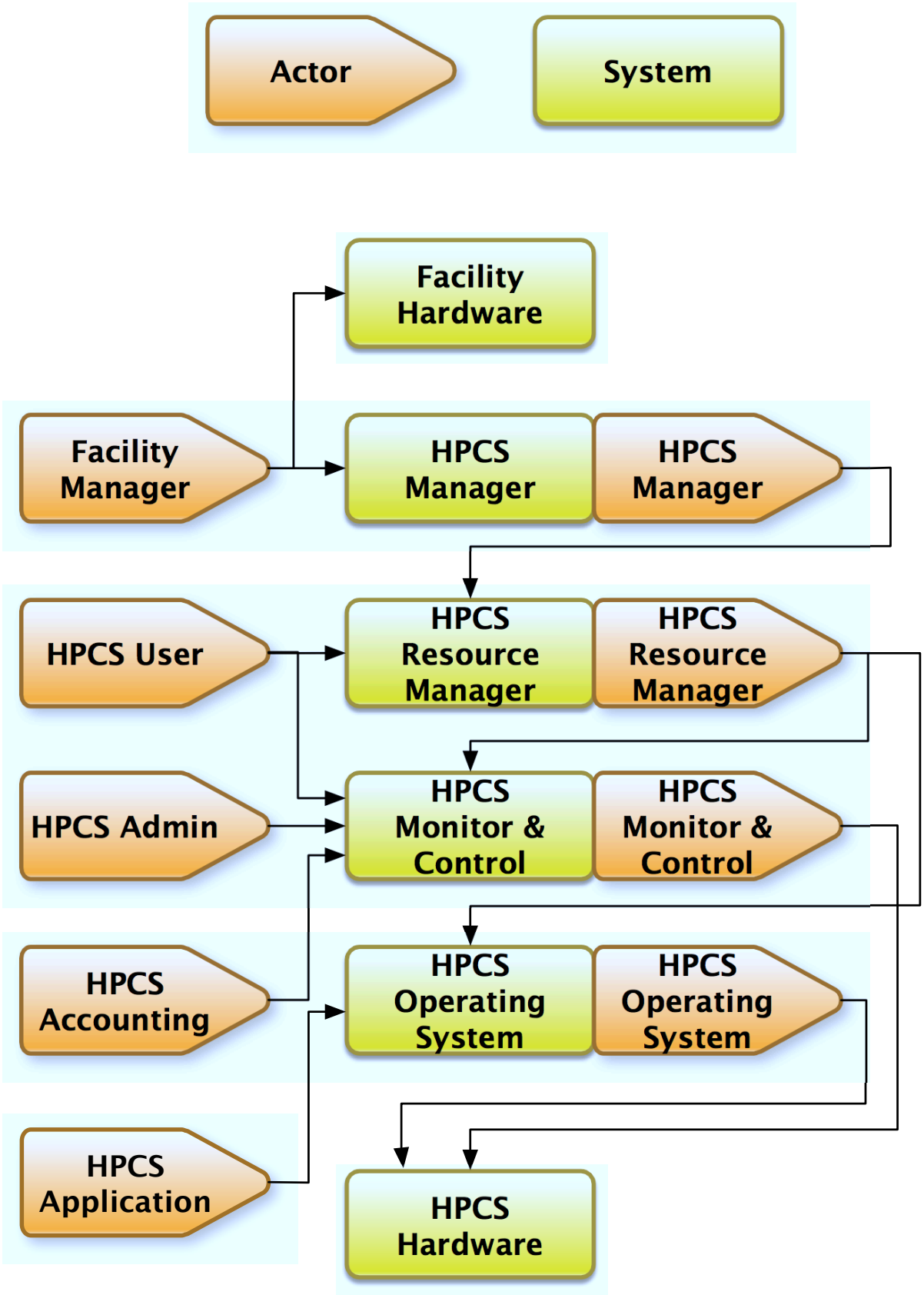


Figure 2.2. Top Level Conceptual Diagram representing the interaction of roles with different levels of the Power API interfaces.

```
PWR_Cntxt   context;
PWR_Obj      my_node;
PWR_Time     timestamp1, timestamp2;
double       energy1, energy2, accuracy;

// Initialize and get my node object
PWR_CntxtInit(PWR_CNTXT_DEFAULT, PWR_ROLE_APP, "MyContext", \
   &context);
PWR_CntxtGetEntryPoint(context, &my_node);

// Get accuracy of energy counter for my node
PWR_ObjAttrGetMeta(my_node, PWR_ATTR_ENERGY, PWR_MD_ACCURACY, \
   &accuracy);

printf("Accuracy +/- %f percent\n", accuracy);

// Measure energy consumed by do_work()
PWR_ObjAttrGetValue(my_node, PWR_ATTR_ENERGY, &energy1, \
   &timestamp1);
do_work();
PWR_ObjAttrGetValue(my_node, PWR_ATTR_ENERGY, &energy2, \
   &timestamp2);

printf("do_work() consumed %f J in %f ns\n", energy2-energy1, \
   timestamp2-timestamp1);
```

Figure 2.3. Example of using Power API to measure energy usage of a function.

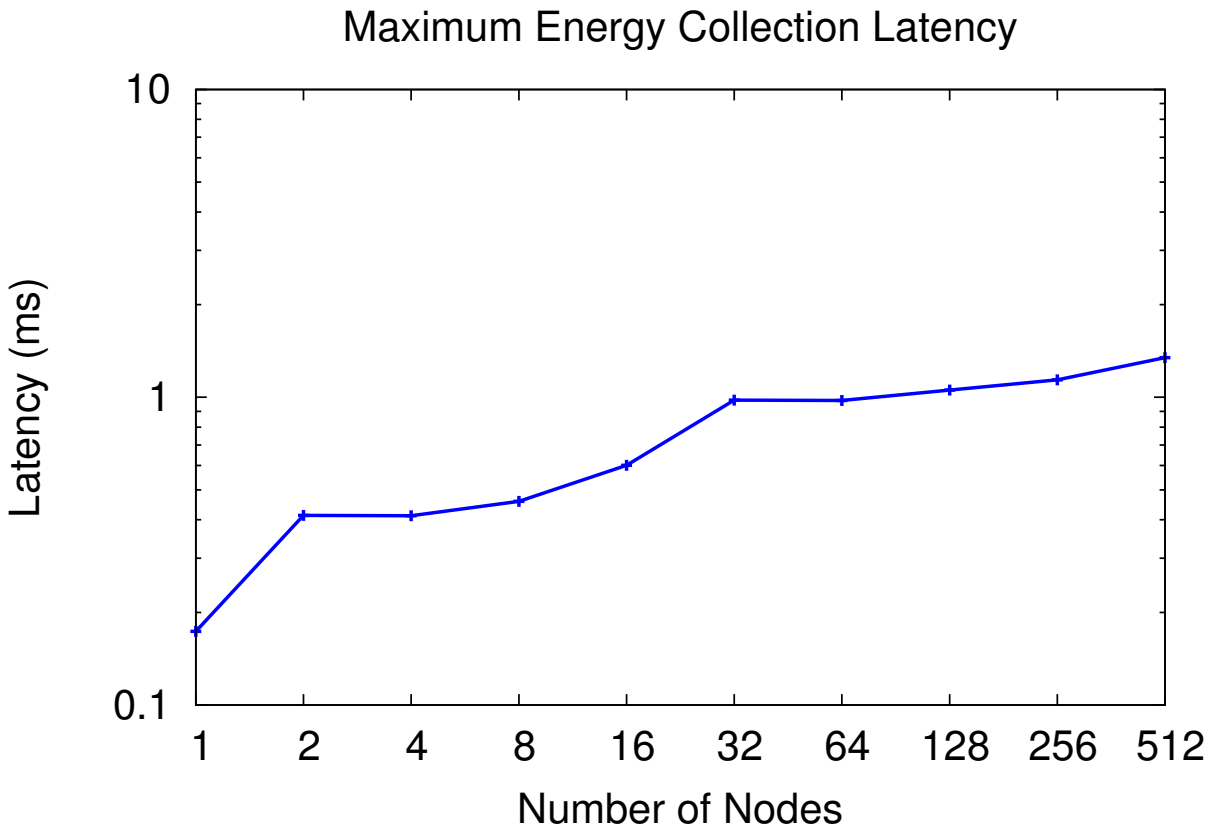


Figure 2.4. Power API Energy data collection latency using multiple nodes.

Chapter 3

Power Control Techniques

Taxonomy of Power Measurement

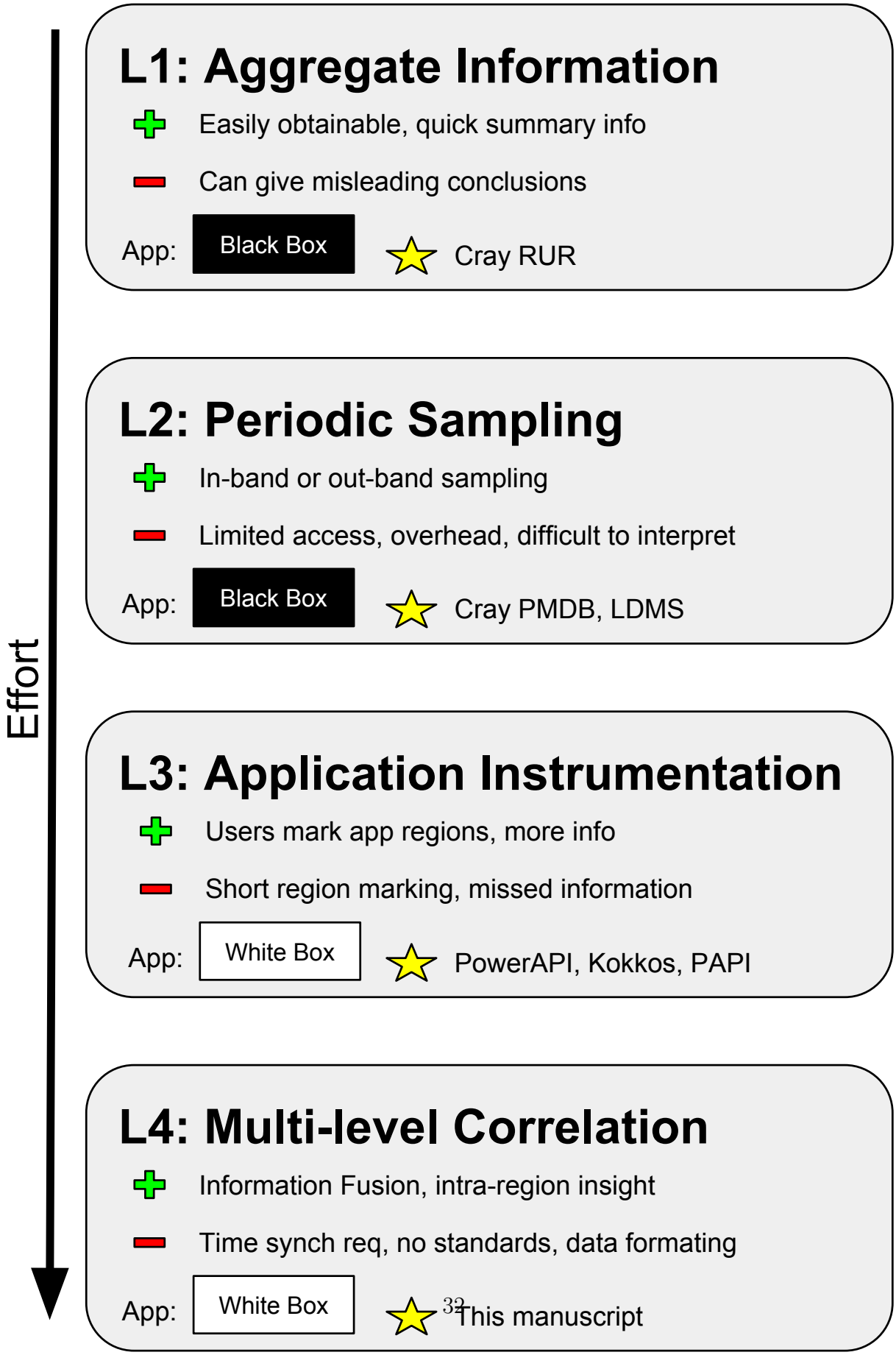
Modern large-scale HPC platforms have incorporated several forms of power measurement and energy accounting that expand the possibilities of gathering key data. However, it is often difficult to know where to start or, in many cases, difficult to access the information that is available. In this section, we describe a framework for understanding these capabilities and discuss the potential insight they can provide. We envision this taxonomy can be directly applied to the Trinity supercomputer [31], a 42 PF Cray XC40 at Los Alamos National Laboratories as part of the ACES collaboration and related smaller-scale testbeds at Sandia National Laboratories. Nevertheless, the framework detailed in Figure 3.1, as well as the various pros, cons, and examples provided for each level, are general and intended to be a useful tool for reasoning about the power measurement and control capabilities on current and future HPC platforms.

We use black box and white box terminology to assess the level of insight that can be gained with regards to power/energy consumption data within the region of interest. This means that approaches like aggregate sampling treat the power/energy rates during execution as a black box while approaches like out-of-band measurements provide insight in this area at their given sampling rates and therefore are defined as white box techniques.

Level 1: Job-wide Aggregate Information

Many platforms track coarse-grained power and energy usage continuously, such as total energy usage by each application executed. It may be broken down by component, e.g., separate CPU and memory energy values, but the information is usually aggregated over an entire application run rather than point-in-time samples or per-node information. The information collected may be available to users in a post-job report.

Job-wide energy information helps to understand how energy-to-solution changes for different application optimizations, algorithm choices, or run configurations. It can also be useful for performance tuning. High power usage levels (e.g., as a percent of the peak available budget) often indicate a well tuned application, whereas low power usage levels


Effort
L1: Aggregate Information
Easily obtainable, quick summary info
Can give misleading conclusions
App:
Black Box
Cray RUR
L2: Periodic Sampling
In-band or out-band sampling
Limited access, overhead, difficult to interpret
App:
Black Box
Cray PMDB, LDMS
L3: Application Instrumentation
Users mark app regions, more info
Short region marking, missed information
App:
White Box
PowerAPI, Kokkos, PAPI
L4: Multi-level Correlation
Information Fusion, intra-region insight
Time synch req, no standards, data formating
App:
White Box
This manuscript

may indicate room for further optimization. A downside to this technique is that it only provides insight into energy usage behavior in aggregate, not the varying rates of energy consumption throughout an application's execution.

This approach is very scalable as the number of sampling events is very low. It is also a local operation to each core and therefore does not require network communication that may incur overheads due to total process counts and data aggregation. Eventually all of this information must be aggregated for analysis, but the aggregation can take place after the application has finished.

Level 2: Periodic Sampling

Finer-grained detail is provided by periodically sampling power levels and energy usage over time. This is often how level 1 information is derived. Sampling may be implemented in-band or out-of band. With in-band, compute node resources are used to perform the sampling, reducing the resources available for application execution. With out-of-band, the platform's control system infrastructure is used to implement the sampling without using compute node resources. Many platforms store a short time window of power and energy samples in a database for use by administrators and workload managers. The information may be accessible to users, but obtaining access often requires administrator action.

This information can be used to plot power usage versus time. It is often possible to identify different application regions by looking for changes in power level. Activities such as idle periods, network polling, and I/O phases can sometimes be identified and used to diagnose load imbalance issues within an application. A downside to this technique is the potentially large volume of point-in-time sample information that must be retained and the difficulty of analyzing it.

A concern with in-band sampling is the performance degradation of the application itself due to frequent interruption of the CPU to read the hardware counters. This interruption not only pauses compute tasks but can also pollute the CPU cache, which may also impact total energy consumption and effect time measurements. While it may seem a small impact overall, previous experience with OS system noise shows that even minor interrumptions can induce larger slowdowns when processes must synchronize across the application [19].

Level 3: Application Instrumentation

While the first two levels treat the application as a black box, or as a gray box when application knowledge is used to interpret the recorded information, white box analysis is useful to understand an application's internal behavior at a finer level of detail. One can modify an application to instrument code regions of interest. The instrumentation points can then be used to record in-band power and energy samples during execution. This information can be analyzed to characterize each instrumented region's power and energy usage behavior.

The potential downsides of this technique are that it requires application modifications and may reduce performance due to instrumentation overhead. The effort needed to instrument an application can be reduced by using automated tools or amortized by leveraging the instrumentation points for other purposes, such as for input to an introspective runtime system.

Application profiling adds instrumentation to the start and end of any code regions of interest, and then as the application is executing, record power and energy information at each instrumentation point along with a timestamp. The information gathered can be stored for later analysis or processed on-line while the application is still executing. Another possibility is to modify the application to replicate level 1 or level 2 functionality, for example by starting a helper thread that performs periodic power sampling. However, we regard this as a workaround that should be avoided when the platform provides this functionality already.

Overheads from application instrumentation depend greatly on two different factors. The first is the complexity of the instrumentation in terms of the data that must be gathered. Some low overhead methods simply use time stamp and region tuples [48]. Timing has been used for a long time in HPC applications, especially using MPI [59] timing functions to instrument code regions of interest. Some instrumentation also incorporates information from sources like performance monitoring counters [71], which can even be used to estimate power consumption without hardware measurement support [37].

The second factor that impacts overhead is sampling frequency. Even with lightweight sampling, for short regions the timestamp can be called so often that it begins to impact performance. If a region is only 1000 cycles with 50 cycles to read and record the data then 5% overhead is incurred. Our evaluation uses regions long enough to amortize overheads for simple sampling like timestamp-region tuples. However, the cost must be taken into account when instrumenting applications to avoid high levels of overhead.

Level 4: Multi-Level Correlation

Finally, information gathered from previous levels can be cross-correlated and used to derive information that is not otherwise available, or would be too costly to obtain using a single level in isolation. For example, if the overhead of application instrumentation (level 3) is too high, power measurement at instrumentation points could be disabled with only timestamps kept. The timestamps could then be correlated with out-of-band periodic power samples (level 2) to obtain similar insight with less application overhead. As another example, if a particular application run experienced an unexplained performance degradation (e.g., a "slow run"), level 1 information could be inspected to look for anomalous power or energy usage behavior. To probe deeper, the level 1 job start and end timestamps could then be used to generate a power versus time plot from level 2 information. The plot may reveal clues to the reason for the slowdown, such as a concentrated idle period (e.g., a system I/O issue or network quiesce event).

Aligning application instrumentation timestamps with in-band and out-of-band periodic sampling measurements can be difficult. In-band measurements are typically the easier of the two to align as they can use the same timestamp as the application instrumentation (region timestamping). In this approach one can use system timestamps and reasonably expect them to line up with relative ease. Out-of-band measurements can be much more complex to align with region timestamps. Because the measurement hardware is separate and distinct from the CPU, absolute timestamps from measurements, applications timestamps, and hardware power samples are needed. To align these time samples, the user must find the minimum timestamp for both the application region profile output and the out-of-band measurements. Using this minimum, one can establish an offset for the individual timestamps and based on a common start point $t = 0$. From there timestamps can be lined up and produce useful data for analysis.

Chapter 4

Trinity Advanced Power Management

The Trinity supercomputing represents the latest in a series of joint LANL/Sandia ACES collaboration to host production ASC supercomputing resources. This ACES collaboration looks to satisfy mission requirements with more capable production-class platforms, and Trinity leads the way in driving next-generation HPC systems with many-core architectures, solid-state burst buffers, and of particular importance to this manuscript, advanced power management capabilities.

Trinity is in fact a single system based on a mixture of both Intel Xeon Haswell(HSW) and Intel Xeon Phi Knights Landing (KNL) processors. While the Haswell portion was installed first to meet FY16 mission needs, both node types are now connected across a single Aries interconnect with a Dragonfly topology. Specifically, Trinity TR1 Haswell portion includes 9436 nodes and 1.15PB of DDR memory, and the Trinity TR2 Knights Landing portion includes 9984 nodes with 0.91PB of DDR memory and 0.15PB of MCDRAM high bandwidth memory. [1] Combined, Trinity provides over 2PB of DDR memory to satisfy current mission requirements.

Platforms

Throughout the rest of this manuscirpt, a number of systems were used. The first, and most obvious is the Trinity system, which consists of both HSW and KNL TR1 and TR2 deployments. Due to the network and access restriction on Trinity as well as queue lengths, testbed systems were used for small-scale runs. Another platform, Volta, was a standalone Cray XC30 testbed system with dual socket IvyBridge E5-2695v2 CPUs and 64GB RAM and a max node power draw of 350W. The IvyBridge CPU has a frequency range of 1.2 - 2.4 Ghz with a turbo frequency of 3.2 Ghz. The Trinity testbed systems at Sandia, named Mutrino, are identical to the Trinity system, and as such represent the Trinity name for small-scale (sub-100 node jobs) experiments. As described previously, Trinity incorporates two partitions, the first comprising a traditional dual socket Haswell E5-2698v3 CPUs and

[1] While throughout the rest of this manuscript the terms MCDRAM and high bandwidth memory (HBM) are used interchangeably, they are in fact different. MCDRAM is a specific implementation from Intel, and HBM represents a JDEC standard for a new type of memory subsystem. Strictly speaking, MCDRAM in KNL is not an HBM spec, but for the purposes of this manuscript, are functionally equivalent.

128GB RAM, and the second consisting of a single socket Knights Landing (KNL) Xeon Phi 7260 with 96GB RAM and 16GB MCDRAM. The Haswell XC40 has a max power draw of 415W per node and a CPU frequency range of 1.2 - 2.3 Ghz with max turbo of 3.6 Ghz, whereas the KNL Phi has a max power draw of 345W per node and a frequency range of 1.0 - 1.4 Ghz, with 1.6 Ghz turbo frequency. All three systems utilize the same Cray Aries Interconnect. Here, the XC40 systems represent a small system that is identical to the hardware and software of the Trinity supercomputer, currently the tenth fastest computer on the Top500 list [55].

Table 4.1. Test Platform Specifications

	Volta "Ivy Bridge"	**Trinity "Haswell"**	**Trinity "Knights Landing" (KNL)**
System Architecture	Cray XC30	Cray XC40	Cray XC40
Interconnect	Cray Aries	Cray Aries	Cray Aries
Proessor Make	Intel	Intel	Intel
Processor	E5-2695v2 (x2)	E5-2698v3 (x2)	Phi 7250 (x1)
# Cores / Node	24	32	68
Frequency Range	1.2 – 2.4 GHz	1.2 – 2.3 GHz	1.0 – 1.4 GHz
Max Turbo Frequency	3.2 GHz	3.6 GHz	1.6 GHz
Memory Per Node	64 GB	128 GB	96 GB
Max Power / Node	350 W	415 W	345 W
Compiler	Intel 16.0.1	Intel 17.0.1	Intel 17.0.1
Cray Linux Release	5.2.UP04	6.0.UP03	6.0.UP03
Cray Mgmt. Release	7.0.UP03	8.0.UP03	8.0.UP03

Cray Systems Management Infrastructure

To facilitate goals of Reliability, Availability, and Serviceability (RAS), Cray HPC systems dating from the XT-series systems to the current XC-series systems utilize a separate, out-of-band management network in addition to the in-band high-speed network used by compute resources. Over this out-of-band network, a head node known as the System Management Workstation (SMW) is connected in a tree structure descending to embedded cabinet controllers (CCs) and from CCs to embedded blade controllers (BCs). This, along with the software that it supports, is known as the Hardware Supervisory System (HSS). HSS orchestrates power, booting, environmental monitoring, hardware health monitoring and logging, and response to hardware failures among other RAS-focused duties.

For power monitoring and management, Cray XC-series systems leverage the HSS infrastructure to:

- Monitor and store node-, blade-, and cabinet-level power, energy, and environmental telemetry,
- Set power "knobs" on sets of nodes including P- and C-states and setting power caps,
- Enable in-band monitoring on compute nodes using the PM counters interface,
- Support queries of historical power, energy, and environmental telemetry coupled with job and application data with a powerful PostgreSQL-based, time-series database, and
- Provide a backend system to support a RESTful interface for platform and power monitoring and control.

In the following two sections, we will overview the database and the RESTful interface for monitoring and control.

Power Management Database Overview

The Power Management Database (PMDB) is a round-robin, time-series database implemented leveraging PostgreSQL alongside Cray-custom software [51]. The PMDB was first released with SMW 7.0.UP02 in July 2013. Broadly, it stores node-, blade- and cabinet-level power and energy telemetry, job- and APID-level information and timings, and System Environmental Data Collections (SEDC) data, including thermals and hardware health data. Power and energy telemetry is captured system-wide by default at 1 Hz (i.e., one observation per second), but for a subset of the system, this frequency can be increased to 5 Hz. SEDC has long been part of the HSS infrastructure, existing prior to Cray's power management efforts but, targeting narrower hardware debugging use cases, had previously only been available in flatfile-form.

Because storage is unfortunately a finite resource, the PMDB is necessarily a round-robin database. That is, once a defined storage threshold is exceeded, the oldest data are dropped to make room for the newest data. These thresholds are defined on a per-table basis using an SMW-resident utility called *xtpmdbconfig*. Customers may use the *xtpmd hooks* interface to execute commands on rotation of old data, such as archiving the old data to a remote server [3].

Cray Advanced Platform Monitoring and Control

With an eye toward allowing workload managers to actively manage power and node configuration, Cray released the Cray Advanced Platform Monitoring Control (CAPMC) with SMW 7.2.UP02 and CLE 5.2.UP02 in the fall of 2014. With CAPMC, a remote (and authenticated) user may control the system by booting and shutting down nodes, setting P- and C-states (i.e., frequency and sleep-state limits), setting power caps. etc. This remote user may also monitor the system, by getting node state information, energy statistics

about sets of nodes, system- and cabinet-power information, etc. The CAPMC infrastructure implements a RESTful interface using *nginx* in one of its common deployment roles. It provides encryption and user authorization capabilities to an independent, application-specific server. In this case, that application-specific server is called *xtremoted*, a Cray-specific daemon residing on the SMW. This provides bridge between the external world and HSS using industry-standard security.

A full description of CAPMC functions and its API is documented in [2]. Some technical and use-case details about CAPMC are given in [52].

PowerNRE – PowerAPI for Cray

The decision of which area of the Power API to focus on involved many considerations. A complete implementation of the Power API would likely require more time and funding than available for the Trinity APM project. Since the team had to be more selective, we focused on high priority areas that aligned well with capabilities that appeared in Cray's roadmap in the Trinity time-frame, even if these capabilities required modification or acceleration to meet our combined goals. We also considered areas of the existing Cray systems management infrastructure that we could leverage and align with, see Section 4.

An important `System` for the purposes of this paper (and the Trinity Power NRE project) is the Monitor and Control system. The Monitor and Control system encapsulates the concepts of systems management or RAS systems (Reliability, Availability and Serviceability). Cray has been introducing measurement and control capabilities important for this topic for a number of years. Cray's Power Management Database (PMDB) is a collection point for a wide range of power and energy related information, along with data important to correlate this information with jobs that are and have executed on the platform (see Section 4). Exposing the information contained in the PMDB to the Admin Role is the first focus area of the Trinity APM NRE collaboration with Cray that will be covered in Section 4.

The second area of focus for the ACES/Cray collaboration is a compute node implementation [42]. The `Systems` (Figure **??**) relative to this area are Hardware and Operating System. In general, the focus is exposing power and energy relevant measurement and control knobs to `Roles` such as the Application and the Resource Manager. The compute node implementation is covered in Section 4.

The third focus area is power aware scheduling, a very broad topic. Adaptive and ACES are actively working towards finalizing the goals for this project. In Section 4 we will discuss some of the use cases that we hope to enable with this effort and some of the capabilities implemented as part of the ACES/Cray collaboration that will be exercised.

Power Management Database Implementation

Cray has recently introduced a capability to retain historic information related to power and energy called the Power Management Database (PMDB). See section 4 for a description of the PMDB and type of information retained in the database. One of most important aspects in any effort to modify a characteristic is to first understand the current condition of that characteristic. Trying to affect power on an HPC platform is no different. Cray's PMDB provides a repository of information that allows a user (some Power API `Role`) to mine power and energy relevant data (measure). The `Role` that Cray is initially implementing to interface with the PMDB (essentially part of the Monitor and Control `System`) is the systems administrator (Admin) `Role`. As mentioned previously, a system administrator typically has the need to understand the entire HPC system. In the PMDB implementation, the Admin `Role` will have a view of the entire HPC platform.

While the initial versions of the Power API were specified in the C language, systems administrators more commonly use scripting or interpretive languages to do their jobs. Python was selected due to its popularity for `Roles` like system administrators (Admin) and Resource Managers, for example. The PMDB implementation will include most of the core functionality of the specification. This includes the attribute interface which allow the user of the Power API to get (measure) information about specific objects or groups of objects. For example, the administrator may desire to get a point in time power measurement from a node (object) or a group of nodes (group of objects). Possibly more useful would be to monitor the energy use of a node or group of nodes. Using the attribute interface the administrator could request the energy reading from a node or nodes, wait a period of time and repeat the call to determine the energy used by that node or group of nodes over that period of time. Note the specification states that the time-stamp related to the sample returned be temporally as near as possible to when the sample was measured.

These low-level interfaces, while useful, are probably not as powerful when interacting with a database as they are at lower levels, like interfacing in real-time directly with the hardware. The PMDB implementation will additionally include the historic statistics interface of the Power API. This interface will allow the user to obtain information like the minimum and/or maximum of a power reading across a number of nodes (all of the nodes assigned to a particular job) over a period of time. The average power of the same group of nodes could be requested.

Probably the most common interaction with a data-base is generating a report. While the Power API specification has the beginnings of some high-level interfaces for this purpose, Cray and ACES are in the process of defining a flexible report interface that will enable the user to request reports for a range of information available in the PMDB. This information will contain job and system related information that the Power API does not currently address but is clearly closely related and necessary for many reasons. For example, Cray is working with ACES to develop two Python report programs that produce text output. The first uses some combination of job ID, application ID, and user ID as input to generate report output that includes data that are of general interest to the systems administrator (Admin)

`Role`. These data include: job ID, application ID, user ID, total energy, start time, end time, and node count. Verbose detailed data for this type of report may include per-node power and energy statistics. The second report type will deliver useful system- and cabinet-level power and energy information, perhaps over a 24-hour window. This report will include data targeted for data-center managers and site planning personnel. It will include statistics like daily and hourly minimum, average, median, and maximum power usage for each compute cabinet. Recall that the PMDB information is stored round-robin and information expires dependent on space available. Generating reports withing the bounds of data expiration is one way to retain important information on a more permanent basis.

An important value already realized by the ACES/Cray collaboration on the Trinity APM NRE is the improvement of the Power API specification. Cray has been instrumental in vetting the Power API from the implementation perspective. In the short time we have been collaboration we have discovered multiple opportunities for improvement that have been included in the latest three point releases of the Power API specification. The Python implementation of the Power API, when complete, will be included in the Power API specification as the first alternative language binding. We anticipate release later in 2016.

Compute Node Implementation

Any effort to understand (measure) or control power and energy for HPC platforms almost necessarily considers node level measurement and control. Early (and on-going) research in this area focused on the potential of manipulating CPU frequencies to reduce power or energy use, for example. For HPC, this is complicated by the need to maintain performance, or minimally affect it. ACES and Cray consider this area of focus to be of great importance in demonstrating and investigating advanced capabilities. Cray will be delivering a C based compute node implementation of the Power API as part of the Trinity APM NRE project. While we cannot cover every capability that will be implemented we will discuss some common and high value characteristics of the implementation in this section.

The `Roles` that will be initially developed are the Application and Resource Manger `Roles`. While these `Roles` could potentially have different needs from the perspective of how much of the system description is exposed upon initialization, the initial efforts will limit exposure to the node level, and below, to both `Roles`. As the collaboration proceeds an expansion of the use cases addressed related to the Resource Manager `Role` may be considered.

As with the Python PMDB implementation, the core functionality of the Power API will be implemented as part of this effort. For the compute node implementation, the core functionality has the potential of being of great value. For example, obtaining power or energy information for the node, or specific component of the node like the CPU, is something that any power-aware application or resource manager would require. The attribute interface (part of the core functionality) allows the user (the Application or Resource Manager `Role`

in this case) to obtain point in time power samples or energy over a given time period. With the exception of energy, measurement attributes, like power, are individual point in time samples. The user will have access to attributes representing power, temperature, frequency, and power cap, for example. The specification requires that the time-stamp returned with any sample accurately represents the time the sample was measured. In the case of energy, power over a period of time, two or more calls using the attribute interface are required. For example, an application that is interested in the energy used over a certain phase can make a call using the attribute interface to get the energy at the beginning of the phase. This value will typically be an accumulator with an associated time-stamp. At the end of the phase being examined the application can make a second call. In the typical use case, the energy over the period of time between the first and second call is represented by the difference between the first and second values returned.

One of the primary additions to typical production functionality that will be enabled by this effort is the ability to control CPU frequency from a user space process. This capability opens up a wide range of potential use cases. Currently, Cray, through the CAPMC interface enables the user (via the Resource Manager) to set CPU P-states that will remain constant through the life of the job execution. Through the Power API implementation, the ability to dynamically change CPU frequency will be exposed to user space processes. This will enable more granular dynamic control during the entire application execution. A power-aware application could, for example, run at a lower frequency P-state when it is in a communication phase. Figure 4.1 shows that lower frequency P-states can be used with little to no performance impact during communication phases [16] (given the network supports offloaded processing of network packets). While on-loaded networking solutions see significant impact from P-state changes, as the CPU is used to process network traffic, such systems can still benefit from P-state changes if applications use small messages that are latency sensitive as shown in Figure 4.2. These P-state changes allow significant power savings from a node perspective and in some cases may minimally impact application performance, making power saving P-states feasible to exploit for some applications. Capabilities like dynamic frequency/P-state and C-state control, implemented on the compute node can be made available to the Resource manager and/or the Application.

The compute node implementation will also include the statistics interface. In this case the real-time interfaces will be implemented (recall that the historic interfaces are being implemented to interact with the PMDB (see Section 4)). The user will be able to create statistics objects that represent a tuple of object, attribute and statistic that they wish to collect. Statistics objects can be started, polled while active and stopped to mine the particular time range of interest. The statistics interface is a very powerful tool that can leverage lower level telemetry capabilities like those provided by Crays HSS (see Section 4). For a complete description of the interface capabilities see the Power API specification.

The Power API specification contains the concept of application hints. The idea behind this is that the application best understands what it is doing at any point in time, or will be doing at some point in the future. While there are many open questions in implementing this type of capability, this is one of the areas that ACES and Cray are interested in inves-

tigating as part of this collaboration. Given the capability of dynamically controlling CPU frequency is available, the application is in a good position to provide hints regarding how some underlying layer might manage the CPU, or other components, to obtain an optimal balance between performance and power efficiency. Some of the hints available to the user via the high-level application interface are: serial, parallel, compute, and communicate[2]. As previously described we have shown potential power savings when using lower CPU P-states during an application communication phase (Figure 4.1). The application hints interface would be a convenient way for the application to communicate this to an intelligent run-time layer. Once the communication phase is complete, the application could again hint that it is about to enter a compute intensive region, for example. As applications adapt to evolving node architectures it has become increasingly important to exploit parallelism to take advantage of larger numbers of cores or accelerators. However, portions of current, and likely future, applications still have serial phases. If the application can provide hints that indicate a serial versus a parallel phase, the run-time could potentially deliver both greater performance and power efficiency. For example, when an application is entering a serial phase an intelligent run time could proactively shut down cores that will not be in use, enabling the core executing the serial portion of the application to run at the highest frequency available. This could result in greater performance for the application (serial phase is accelerated) while potentially saving power (cores not in use are put in a minimal power state). Once the serial phase is complete, the application can hint the end of the phase by sending the default hint or some other hint that could help optimize performance, power or both. The use case that was just described suggests that the application will provide appropriate hints. It is also possible that another layer, like the Message Passing Interface (MPI) layer, could send similar hints. This would allow this type of optimization without requiring modification of the application.

Power Aware Scheduling

ACES recently began a collaboration with Adaptive Computing as part of the Trinity APM NRE project. We will only discuss this briefly since we are in the very early stages of this work. Power ramp control, the ability to control the rate at which the system increases its power use, is being implemented by Cray as part of the compute node implementation (see Section 4) Trinity APM NRE. ACES will be working with Adaptive computing to develop, test and implement this capability which is controlled by the resource manager through the CAPMC interface and will leverage the Power API interface on each compute node. One of the challenges we see in the future is very large swings in power draw for our largest platforms. While this may not present a problem for the facility in the Trinity time-frame, it is likely to for the next ACES ATS platform, Crossroads. Refining this capability on Trinity will allow ACES to be prepared for platforms that can experience multiple megawatt swings in power in very short periods of time. Likewise, managing platform power within pre-determined, or pre-negotiated lower and upper limits can prove to be a huge cost savings for the facility. Using more or less of the pre-negotiated power results in much higher costs

[2]For a complete list please refer to the Power API specification.

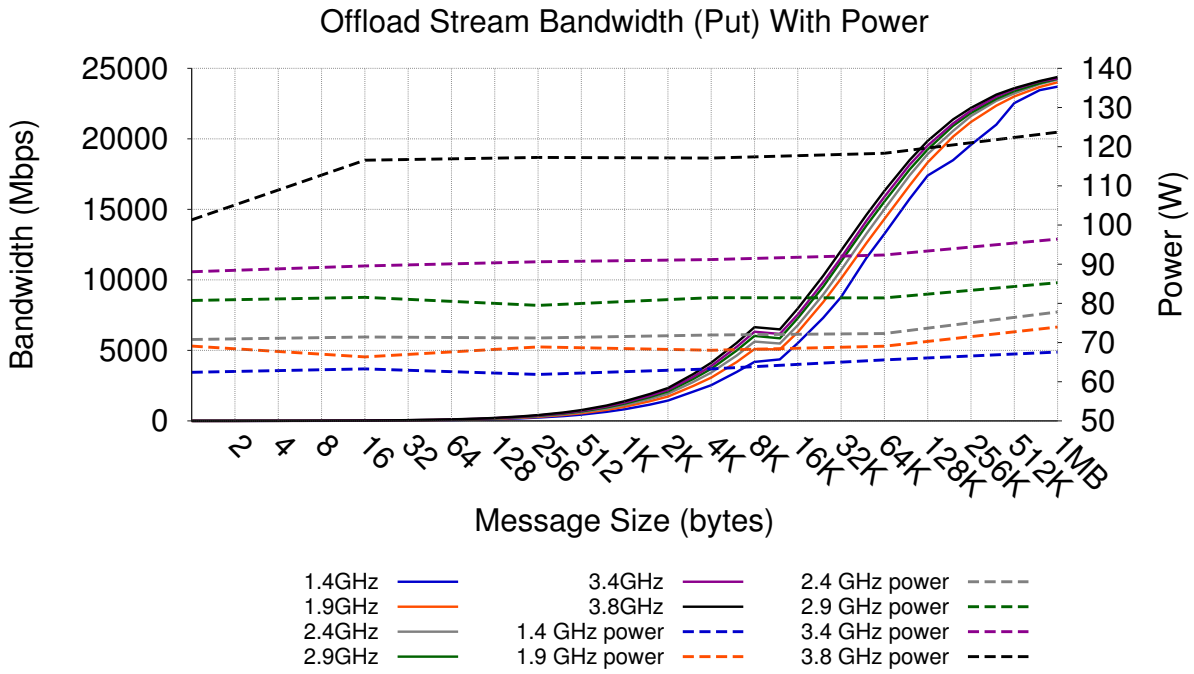


Figure 4.1. Offloaded network traffic stream bandwidth with varying CPU frequencies

for a facility. Similar to the power ramping effort, ACES will be working with Adaptive to exploit capabilities that Cray will expose, developed as part of the ACES/Cray Trinity APM NRE, to operate the platform within pre-determined upper and lower bounds, even bounds that differ throughout the day. In addition, we will be working with adaptive to execute individual applications within power constraints to maximize the science output within a given platform power constraint. The combination of the ACES/Adaptive and ACES/Cray collaborations should result in power-aware scheduling and management capabilities that have never been possible on a leader-ship class HPC platform.

Power Capping

Power Capping is another mechanism for controlling power utilization for a given job. The XC40 system power capping mechanism attempts to keep the node's power usage at or below a set power level. On-node firmware monitors draw and makes decisions based on an unspecified sliding time window. If a node's power usage begins to exceed its power cap, the node is throttled to a lower performance level – e.g., by running at a lower P-state or performing clock gating – until the node's power usage falls below the power cap for an unspecified period of time. Node-level power capping in Mutrino is implemented using the Intel Node Manager firmware. Each Node Manager instance operates autonomously and independently with no cross-node coordination.

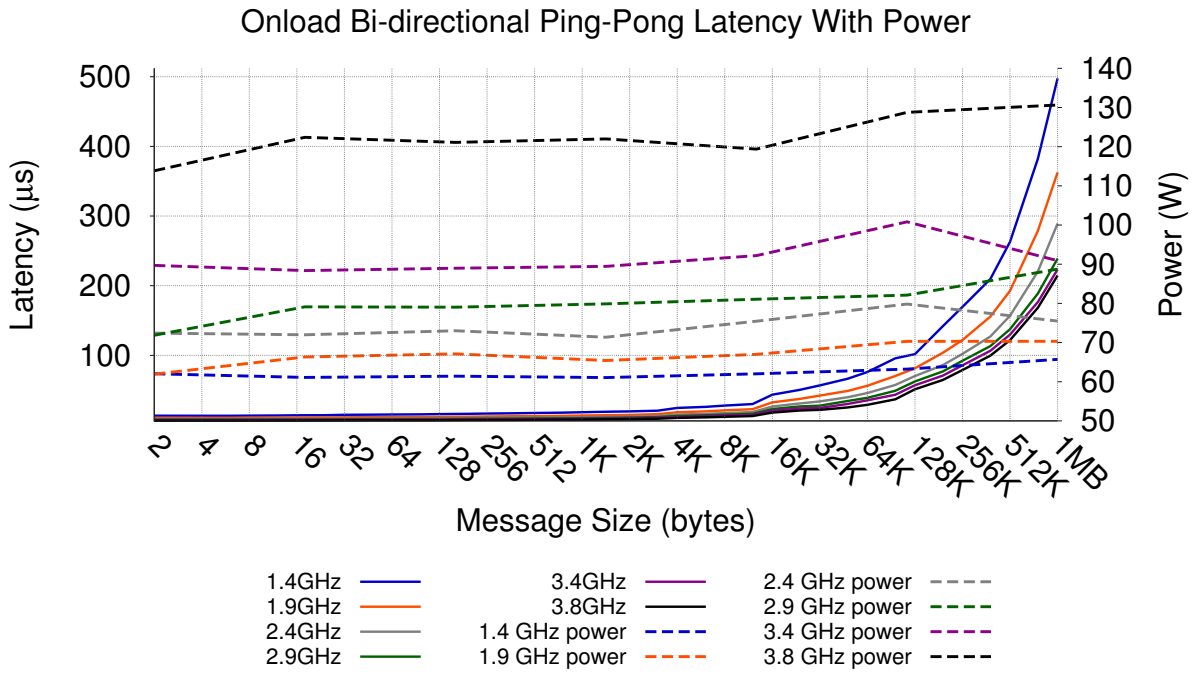


Figure 4.2. Onloaded network traffic stream latency with varying CPU frequencies

Table 4.2 describes the Power Capping mechanism and each setting for both Haswell and Knights landing in detail. While the power cap setting may range from 0 to 100%, it is important to note that there is a floor function associated with the minimum power a node can be set at. This is 230 and 200 Watts for HSW and KNL, respectively.

We have already put the Power API to use in studies focusing on understanding the power consumption of systems as well as understanding the effectiveness of power control methods on platforms. Testing the power capping mechanism on this system with CTH, a widely-used solid mechanics application, has shown the distribution of power samples under different power caps. Figure 4.3 shows the cumulative distribution function for power samples for several different node-level power caps for 96 nodes on the Trinity test system. The Power API data show that the power capping mechanism allows limited time periods where the power draw can exceed the power cap. These measurements reveal the consequences of the power cap mechanism's enforcement that is based only on an average of samples in a given time window [61]. The Power API has enabled collection of all measurements on this system and enables portable testing on other systems and with other applications.

Furthermore, we have begun to explore the use of power capping as a mechanism for enforcing power budgeting decisions made by the workload manager. Cray's CAPMC infrastructure enables workload managers to set a desired power budget for each compute node in the system, which is then enforced by firmware running on each compute node. For example, a workload manager may decide to power cap a given job's compute nodes to 200

Table 4.2. Power capping settings on Trinity XC40 Haswell and Knights Landing nodes.

Trinity Haswell			
Power Cap	Power Cap (Watts)	Savings Potential (Watts)	Savings Potential (Percent)
100%	415 W	0 W	0%
75%	369 W	46 W	11%
50%	322 W	93 W	22%
25%	276 W	139 W	33%
0%	230 W	185 W	45%
Trinity Knights Landing			
Power Cap	Power Cap (Watts)	Savings Potential (Watts)	Savings Potential (Percent)
100%	345 W	0 W	0%
75%	309 W	36 W	10%
50%	273 W	72 W	21%
25%	326 W	109 W	32%
0%	200 W	145 W	42%

W of the maximum 400 W per node and shift the 200 W difference elsewhere in the system where it can be better utilized. Our initial finding is that while node-level power capping on Trinity is effective at maintaining the desired average power usage over multi-second time windows, the scalability of some workloads is significantly impacted by the performance variability introduced by the power capping mechanism [62]. As an example, Figure 4.4 shows the performance of the CTH and S3D applications running under a 230 W per-node power cap setting on Mutrino, a small-scale Trinity testbed at Sandia. CTH performance behaves as expected, with performance degrading gracefully under the power cap. S3D, on the other hand, experiences significant performance degradation with scale when running at the default Turbo-On p-state (2.3 GHz base with dynamic scaling up to 3.6 GHz). In this case it is better to run S3D at a fixed 1.8 GHz p-state because it results in average power usage being below the 230 W cap, which avoids the power capping mechanism from being triggered. We are currently working to better understand this behavior and find ways to mitigate it, as well as examining how node-level power capping affects power usage at the facility level [10].

Instrumentation

The Cray platforms provide similar power measurement capabilities, all of which were utilized for the experiments in the following chapters. Following from the taxonomy defined in Chapter 3, we step through the instrumentation details of each level.

Level 1 information is provided by Cray's RUR (Resource Utilization Reporting) tool [6],

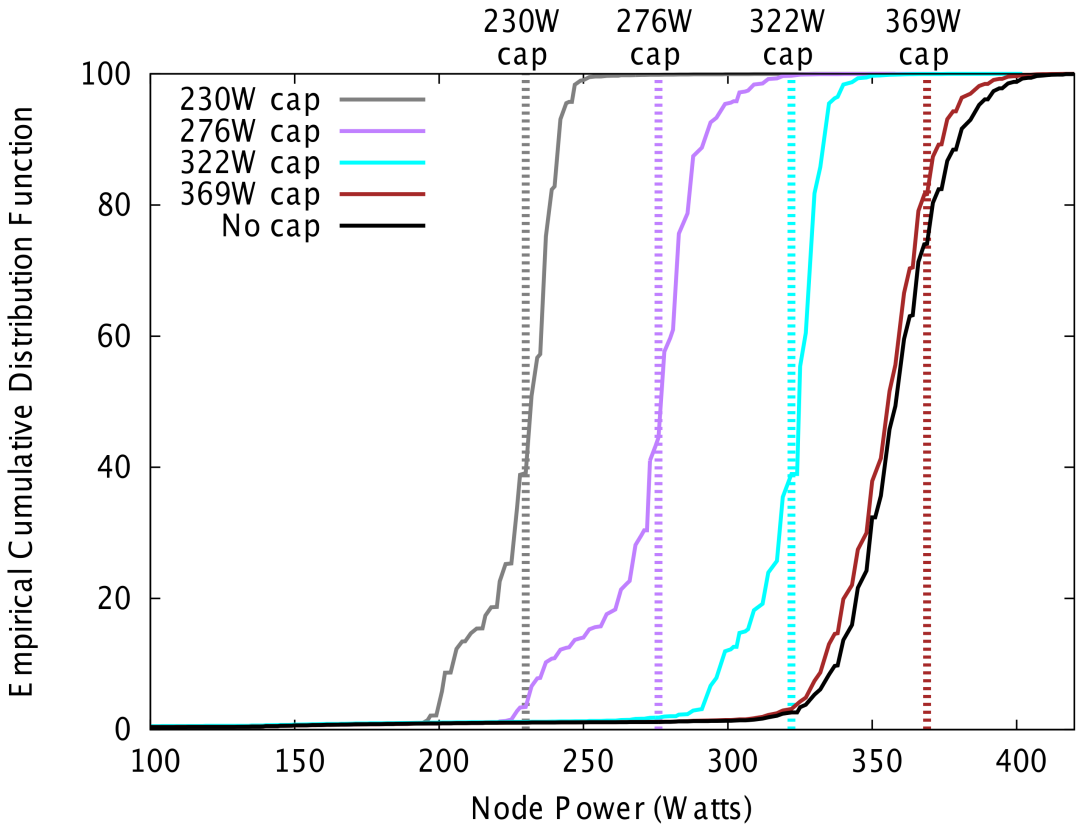
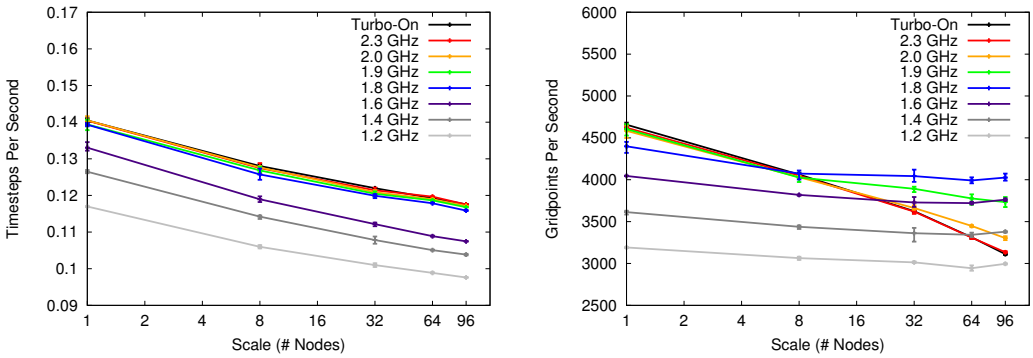


Figure 4.3. Power API power measurements used to understand node-level power capping behavior with production application, CTH. CTH is a strong shock wave, multi-material solid mechanics code

which records various aggregate statistics about each job, including start time, end time, and total energy consumed. Users can opt-in to receiving RUR reports, and in this case this was done for all runs. The overhead of using the Cray RUR tool is effectively zero.

Level 2 out-of-band data is provided by Cray's power monitoring infrastructure and Power Management Database (PMDB). Per-node power is sampled at 5 Hz and stored in the PMDB PostgresSQL database, with a rolling 4 hour window of samples kept for each system. Each compute node contains a power measurement device with accuracy of +/- 3% that is internally sampled at a rate higher than 5 Hz. We extract the relevant data from the database for each of the runs by application ID. The PMDB is not available to regular system users, it can only be accessed by users with root access to the system management workstation. For the Trinity platforms, each 5 Hz power sample is the average of the internal



(a) CTH Performance with 230 W Power Cap (b) S3D Performance with 230 W Power Cap

Figure 4.4. CTH and S3D application scalability when running under a node-level power cap.

samples since the previous 5 Hz sample. The XC30 (Volta) runs an older Cray management software release that does not perform this averaging, so the 5 Hz value recorded represents the most recent internal sample available.

The PMDB is located on a single node, the system management workstation (SMW). As such, it can be both difficult to insert many samples at high rate from many nodes and slow to extract data from many nodes, as the node is also busy recording current samples. Even extracting data for small numbers of nodes can be time consuming, and therefore this approach can be limited for very large scale jobs or long running jobs unless the database is routinely dumped to an alternative storage solution before the time window on the database rolls over and begins overwriting data.

In-band measurements on our system use hardware counters to regularly record measurements through register fetch operations. The granularity of the measurements is limited by the register refresh rate of [regrant: XX]%. In-Band measurements are available on a variety of hardware. We use the Running Average Power Limit (RAPL) measurements available on Intel processors for this paper. RAPL counters are similar to approaches by other vendors such as AMD's APM [4], so the conclusions drawn here regarding the capabilities of the measurement technique are generally applicable to x86 processors, and possibly even to other ISAs entirely.

The scalability of in-band measurement is good as each measurement is node local and recorded at that level. Consequently, the only concern with scalability is the collection of the measurements at the conclusion of an application run. The main concern with in-band measurement is the performance degradation of the application itself, which occurs due to frequent interruption of the CPU in order to read the hardware counters. This interruption not only causes a pause in compute tasks, but can also adversely impact the CPU cache depending on how the CPU is interrupted to read the data. While this may seem like a small impact overall, previous experience with OS system noise illustrates how the introduction of

small amounts of interruption in compute can cause larger slowdowns when processes must synchronize across the application [64]. This is why user-level frameworks, such as the Power API, consider additional metadata information necessary to pair with in-band measurement data.

In-band measurement through CPU counters can provide both CPU and memory subsystem energy measurement. This is not always possible with out-of-band measurement, depending on where the external measurement hardware is placed and its capabilities. Out-of-band measurement can capture whole node energy profiles easily, while this is not possible for in-band measurement that uses CPU counters. Whole node energy is can be useful when CPU external components consume a large amount of the power budget for a node, such as with networks or motherboard chipsets.

Level 3 application instrumentation for this study uses the KokkosP [74] runtime profiling hooks, part of the Kokkos Kernels library. KokkosP is a recent interface that provides low-overhead profiling and instrumentation of applications. We have used KokkosP exclusively to denote application region entry/exit with timestamps as described later. Some but not all applications described herein use the Kokkos programming model, demonstrating that the Kokkos lightweight profiling library can be used independently or in conjunction with Kokkos. We developed a Power API plugin for KokkosP that is used to collect in-band power and energy measurements.

Chapter 5

Testbed Mini Application Experiments

Workloads

We used three different proxy/mini applications, MiniMD, LULESH and MiniFE, to conduct initial experiments on Trinity tesbeds and the Volta machine.

MiniMD is a molecular dynamics simulation that is a proxy for the LAMMPS full featured molecular dynamics simulator. MiniMD solves a single problem type out of the LAMMPS suite [65], namely a Lennard-Jones liquid simulation. It is essentially equivalent to LAMMPS in terms of behavior for this particular type of simulation. LAMMPS was run with one MPI process per physical core and configured to run for 1000 cycles with a memory footprint of approximately 7 GB per node (`-n 1000 -s 280 -gn 0 --half_neigh 0`).

LULESH is a proxy application that solves an unstructured mesh Lagrangian explicit shock hydrodynamics problem. This is essentially the explicit hydrodynamics section of the larger ALE3D package [5]. LULESH was configured with 2 MPI processes per node with OpenMP used to utilize the remaining physical cores. LULESH was configured to run for 800 cycles with a memory footprint of approximately 1.2 GB per MPI process (`-s 100 -i 800`).

MiniFE is an implicit finite element conduction simulation using a conjugate gradient solver on a rectangular shaped problem. MiniFE, unlike other codes, is designed to represent a large swath of applications instead of a specific portion of a larger application. As such, it does not use a preconditioner, and uses a simple CG solver in order to approximate a larger range of finite solvers. MiniFE was configured with 2 MPI processes per node and OpenMP to utilize the remaining physical cores. MiniFE was configured to run with a ``-nx 800 -ny 1600 -nz 800'' input problem that resulted in a memory footprint of about 7 GB per MPI process.

These workloads were run on 32 nodes of each platform using all cores on each node. The same input problem was used across test platforms, enabling cross architecture comparisons. Input problems were chosen by consulting with subject matter experts to determine realistic configurations that would produce high performance across the three platforms studied.

Mini-App Power Profiling Experiments

Job-wide Aggregate Information

Aggregate counts of total energy consumed or average wattage over a sampling period are obtained using hardware registers that are polled for values only at the beginning and end of an application execution. This is technically in-band as the commands are issued to read the hardware counters from the CPUs executing the simulation code. Reading these counters can be done with negligible overhead, as the code reads the counter before and after the application execution, avoiding any interference while running the simulation.

To demonstrate the initial utility of HPC job aggregate information, we look at a common case study where a P-state sweep is performed to try to determine a configuration for the highest performing, the lowest overall power consumption, or the least total energy consumed per Figure of Merit (FOM). Given 3 architectures and turbo-boost frequencies, we also use the 1.2 GHz frequency results to compare different CPU architecture types at an identical clock speed. Results from this study are shown in Table 5.1. These results demonstrate trends that are predictable: The fastest clock speeds are almost universally the most performant mode of operation for all architectures. The exception is Haswell running MiniFE, where non-turbo max frequency is slightly better than turbo but within the margin of error. In terms of FOM per watt, the results are less straightforward. The Ivy Bridge system should be run in non-turbo highest frequency for power efficiency and performance, but for the Haswell and KNL systems the ideal frequency varies based on the architecture and application. The KNL and Xeon core architectures are vastly different x86 implementations. The results show that the many Intel Atom-based cores in the KNL architecture are better for both MiniMD and MiniFE. For the memory-intensive LULESH code, however, the high bandwidth, low latency memory subsystem in the Xeon allows it to slightly outperform KNL.

Unfortunately, we can also see in this case study the limitations of aggregate job information. First, this aggregated data cannot tell us which particular phase of an application is of concern or how energy is consumed throughout execution. Other techniques are necessary to answer these questions. Perhaps more alarmingly, the total job energy consumption may lead us to false conclusions about optimizing FOM per watt. With MiniFE, the Trinity Haswell portion shows 1.2Ghz to be the best FOM per watt. However, as we find later when application code regions are instrumented, MiniFE's FOM is based on a single region that is a small portion of the total runtime, effectively leading to a false conclusion regarding selecting an optimal P-state FOM per Watt.

Application Instrumentation

While total job aggregate data may be initially useful, it alone can also lead to false conclusions or miss more detailed information in regards to a given applications. Compared

Table 5.1. Power and Energy Efficiency Calculated from Cray RUR Aggregate Information

		FOM Per Node			**AVG Watts Per Node**			**FOM Per Watt**		
		Volta IVB	Trinity HSW	Trinity KNL	Volta IVB	Trinity HSW	Trinity KNL	Volta IVB	Trinity HSW	Trinity KNL
MiniMD	Turbo	**2.08e7**	**3.01e7**	**5.92e7**	269	334	246	7.73e4	9.01e4	2.41e5
	No Turbo	1.84e7	2.56e7	5.66e7	213	236	228	**8.64e4**	**1.08e5**	2.48e5
	1.2 GHz	9.45e6	1.39e7	4.93e7	138	142	194	6.85e4	9.79e4	**2.54e5**
LULESH	Turbo	**1.36e4**	**1.85e4**	**1.51e4**	291	346	218	46.7	53.5	69.3
	No Turbo	1.24e4	1.75e4	1.43e4	236	295	208	**52.5**	59.3	68.8
	1.2 GHz	6.75e3	1.09e4	1.25e4	156	175	180	43.3	**62.3**	**69.4**
MiniFE	Turbo	**1.24e4**	1.42e4	**3.00e4**	185	212	138	67.0	67.0	217
	No Turbo	1.23e4	**1.43e4**	2.93e4	145	152	133	**84.8**	94.1	**220**
	1.2 GHz	8.37e3	1.41e4	2.76e4	104	104	127	80.5	**136**	217

to simplistic aggregate job data, application profiling requires significantly more effort and knowledge of the applications under study than aggregate counter or out-of-band power measurement. It can yield more insight into the power and energy characteristics of applications. However, the detail and frequency at which application instrument is implemented can have cascading impacts and considerations.

For more insight into application behavior we need to understand the phases of the applications themselves. Table 5.2 shows the region breakdown for each of the three applications. The number of occurrences of each region are identical across architectures and only the region timings vary. MiniMD has 5 main regions. Not all of these regions occur in every timestep of the simulation. For example, NeighborBuild is only run every 20 timesteps. However, when it does occur it is a significant portion of the execution time for that timestep. Like MiniMD, LULESH also has 5 regions in its main solve. Regions 2 and 5 are the most significant in terms of time, while region 3 is very short. Like Exchange or Communicate regions for MiniMD, region 3 is very difficult to profile for energy/power. This is because the region is so short that measurement may not be possible inside of the region. For MiniFE, we have only instrumented the assembly and CG solves as regions. Since both regions are large, the power profile clearly differentiates these two regions in Figure 5.1c.

Power/energy profiling can be done inline in the application directly through in-band measurement, such as calling the PowerAPI or interfacing with RAPL directly. However, the overhead of in-band measurement can be significant when sampling at high frequency. To quantify the potential overhead of inline application profiling, multiple experiments with both power/energy readings and timestamps, as well as with only timestamps enabled, are shown in Table 5.3. Our initial investigation for KNL yielded significant overheads of in-band sampling with region profiling for MiniMD and LULESH, ranging form 4-8%, whereas MiniFE in-band sampling was negligible compared to no sampling. These overheads are significant even at a small scale. Given prior knowledge of how asynchronous noise in parallel

Table 5.2. Application Profiling Region Durations For Trinity KNL

			Region Durations (seconds)			
	Region Name	Count	Mean	SD	MIN	MAX
MiniMD	Exchange	1632	0.005	0.004	0.003	0.025
	CommBorders	1632	0.010	0.008	0.006	0.085
	Communicate	30400	0.003	0.002	0.002	0.032
	Force	32000	0.039	0.038	0.036	0.155
	NeighborBuild	1600	0.600	0.600	0.595	0.624
LULESH	IntegrateStress	25600	0.011	0.010	0.010	0.035
	HourglassControl	25600	0.039	0.039	0.038	0.067
	VelocityForNodes	25600	0.002	0.002	0.001	0.002
	LagrangeElements	25600	0.018	0.018	0.017	0.066
	MonotonicQ	25600	0.031	0.031	0.023	0.063
MiniFE	Assemble	32	135.243	134.816	125.582	149.543
	CGSolve	32	14.209	14.209	14.208	14.210

applications can have cascading effects at a large scale, these overheads could increase when moving to an extreme scale such as the Trinity supercomputer.

When we disable profiling and use only timestamping, the performance overheads become essentially non-observable when accounting for normal application runtime variance in measurement. Timestamping is currently the best way to couple application profiling with out-of-band power samples. Since out-of-band samples are detached from the application or compute infrastructure entirely, the use of timestamps within the application becomes the only feasible way to couple out-of-band data. While it requires significant additional effort and synchronized clocks, the result is a near complete lack of perturbation or added overhead due to profiling.

Out-of-band Periodic Sampling

Out-of-band power sampling can provide detailed information about application phases and the impact of varying clock frequency on the CPU. Continuing with the case study detailed in Table 5.1, we next look at P-states across the KNL system. In Figure 1, the power versus time plots include level 2 power samples taken at 5 Hz for each of the 32 nodes for the respective run, with a solid horizontal line added to visualize the job-wide average power calculated from level 1 information. MiniMD's out-of-band measurements in Figure 5.1a show expected behavior in the main solve of the application. The periodic power consumption corresponds with known solver phases, and each P-state shows the expected number of phases. Using a slower CPU frequency lengthens the phases but does not alter any observable power consumption trends.

Table 5.3. Overhead of Application Profiling For Trinity KNL

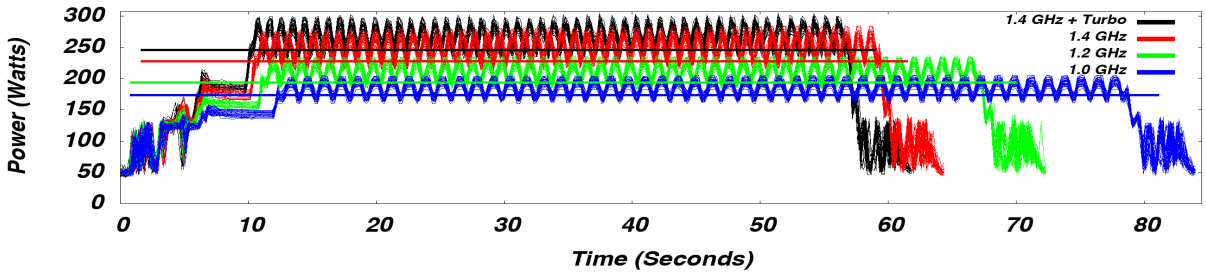
		Power + Energy Region Profiling	Timestamps Only
MiniMD	Turbo	6.84%	-0.08%
	1.4 GHz	7.49%	-0.08%
	1.2 GHz	7.71%	0.08%
	1.0 GHz	8.15%	0.07%
LULESH	Turbo	4.84%	0.24%
	1.4 GHz	4.94%	0.35%
	1.2 GHz	5.23%	0.22%
	1.0 GHz	4.73%	-0.08%
MiniFE	Turbo	-1.22%	0.15%
	1.4 GHz	-0.59%	-0.95%
	1.2 GHz	-1.50%	-1.42%
	1.0 GHz	-1.26%	-1.96%

LULESH's power consumption with varying CPU frequency for the KNL is shown in Figure 5.1b. Its power consumption is much less periodic than MiniMD but shows similar patterns of lowered power consumption and lengthened runtimes with different P-states. Unlike MiniMD, where phases are obvious in the power consumption graph, LULESH has 5 phases, but all of them are similar in power consumption, even though the time periods of the individual phases are not equal (some are very short, others are longer than average).

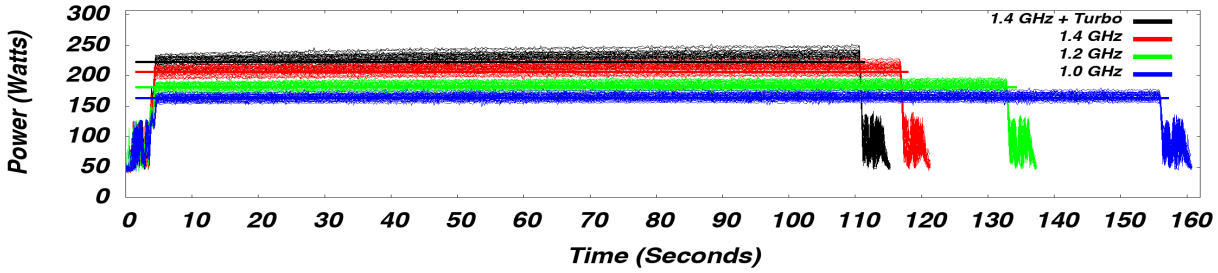
MiniFE shows multiple different phases throughout execution in Figure 5.1c. The first long, low-power phase is the assembly phase while the increased power consumption regions are the problem solve (CG). We can observe that power usage throughout the assembly phase is slightly improved by lowering the CPU frequency. However, the difference is not as large as the power savings during the main CG solve region. This will impact the performance per watt of the lower frequency results as the assembly phase is the longest phase.

In addition to an analysis of P-states on KNL only, we can also directly compare all three of our platforms with out-of-band measurement. The results for MiniMD, shown in Figure 5.2, illustrate some differences between the out-of-band measurement techniques across the 3 system platforms. The out-of-band measurement capabilities between these systems are very similar, except that the Ivy Bridge system reports only point-in-time power samples, while the Haswell and KNL systems average results between sampling reading points. This leads to a much less noisy power profile for the newer systems, avoiding spurious data not significant to the analysis of the power consumption of the system.

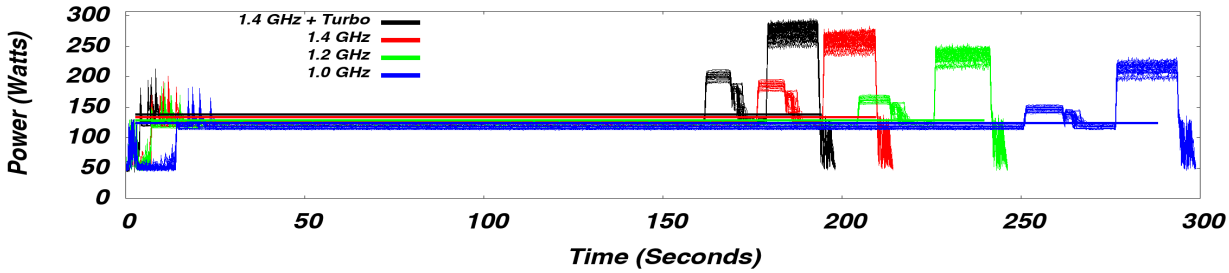
LULESH does not have periodic signals hidden in the noise of its power samples. This can be contrasted to the results for MiniMD. The results can be significant for identifying periodic power behavior in an application. In Figure 5.2 we observe that the KNL system



(a) MiniMD

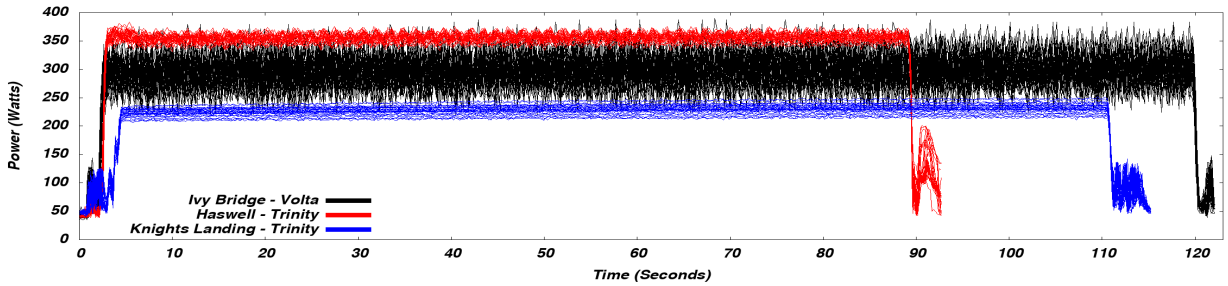


(b) LULESH

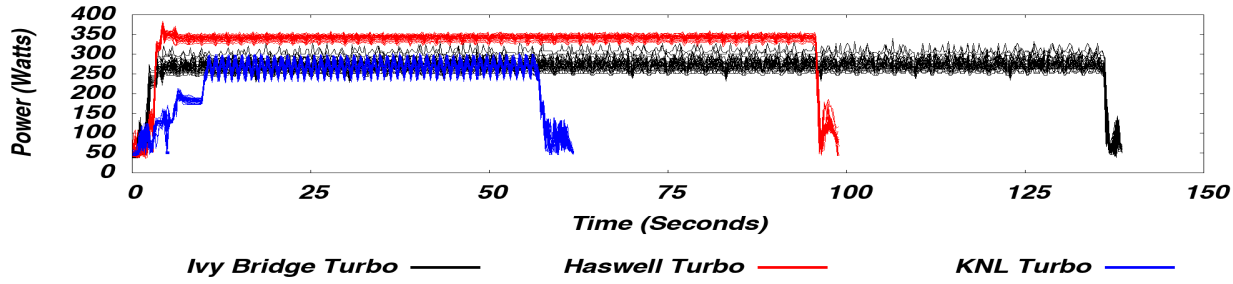


(c) MiniFE

Figure 5.1. Out-of-band power sampling for workloads running on Trinity Knights Landing at different CPU frequencies.



(a) LULESH



(b) MiniMD

Figure 5.2. Out-of-band power sampling for LULESH and MiniMD running on the different test platforms.

shows clear periodic behavior that we can relate back to known phases of the application and timesteps in the simulation itself. Such results on the Ivy Bridge system yield a noisy signal, and the periodicity of the underlying code regions was lost. This is not the case when we have intra-sample averaging like on the KNL/Haswell system. While LULESH does not have the same region periodicity, the noise from LULESH running on IvyBridge was even greater, to the point of obscuring useful data.

Combining Out-of-band Periodic Sampling and Application Instrumentation

Region profiling with timestamps and in-band power/energy profiling paired with the collection of out-of-band data allows significantly more insight than any one technique. Specifically, the method of power data paired with profiling may have drastically different resolutions that effect the perceived accuracy of the interpreted results. The out-of-band results are samples taken through the PMDB on the platform, whereas the in-band results show real samples taken alongside the application at region entry and exit.

For MiniFE in Figure 5.3, the regions are very coarse-grained, therefore the assumed behavior throughout the region is very different than the out-of-band sampling throughout

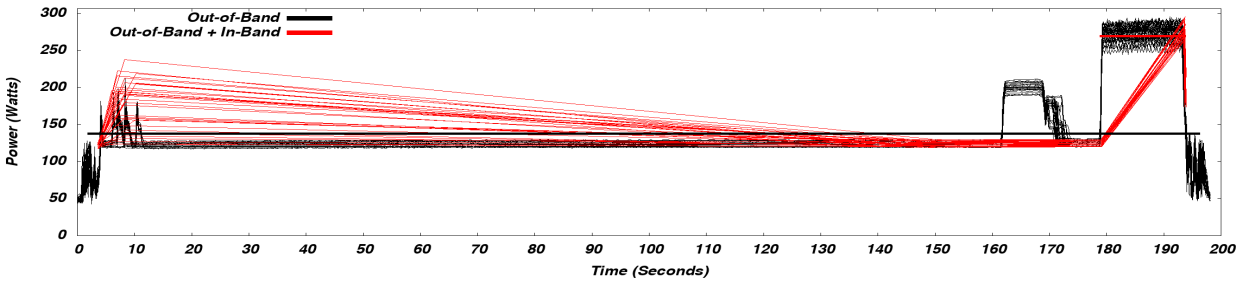


Figure 5.3. MiniFE correlating out-of-band power sampling with in-band application region profiling.

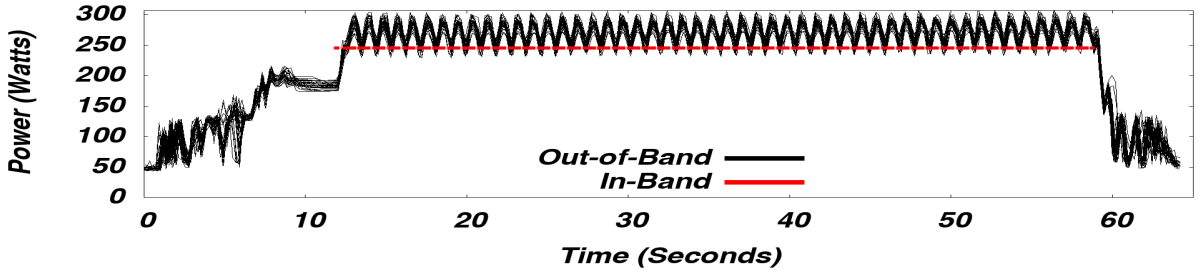


Figure 5.4. MiniMD correlating out-of-band power sampling with in-band application region profiling.

the region. This illustrates a key issue with in-band measurement at the beginning and end of a region: Region behavior is not guaranteed to be uniform. We can see notable differences between the perturbation in out-of-band data compared to the simple measurements and the assumptions they would lead to from in-band profiling.

Looking at MiniMD in Figure 5.4, the results show an issue of resolution that can be introduced by power/energy profiling only at region entry and exit. The Figure shows that all of the periodic data with timesteps (in black) of the application are not observable in comparison to in-band measurements (in red), which are taken along region boundaries. We can examine this closer in Figure 5.5, which provides a zoomed-in view of Figure 5.4. The out-of-band measurements capture the periodic behavior from MiniMD and, when combined with timestamps, application profiling clearly illustrates which regions correspond to the periodic behavior. While the in-band measurement does accurately report the total energy usage, the rate of usage throughout the region is incorrect as it is treated as a uniform power/energy draw, seen by level line of red dots in Figure 5.5. Effectively, this in-band approximation does not match the fine-grained out-of-band measurements or the higher frequency periodic behaviors observed. Other studies have validated in-band measurements

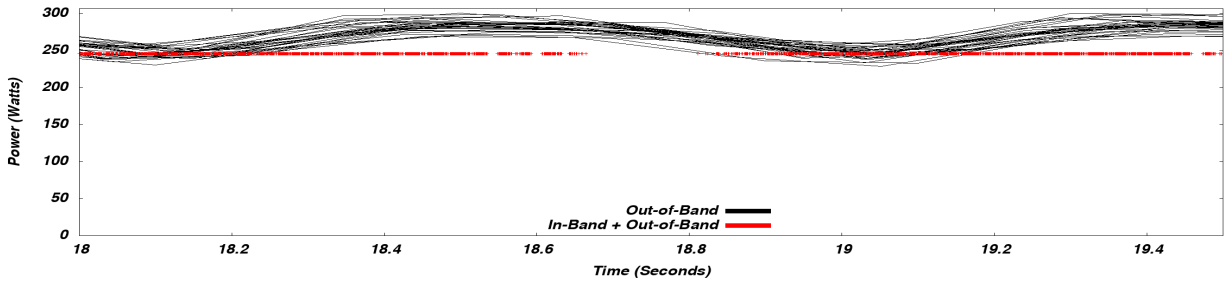


Figure 5.5. Zoomed-in MiniMD correlating out-of-band power sampling with in-band application region profiling.

using out-of-band measurement [54], so while in-band measurements are accurate, it is the lack of the resolution with sampling rate that fails to properly illustrate the application's true power profile.

In-band sampling resolution could be increased by interrupting the application to query the energy counters throughout the execution. One could also define finer-grained regions if appropriate for the code under study, however as seen in Table 5.3, doing so would also add overhead and potential perturbation to the application itself. Furthermore, for course-grained application regions like MiniFE, the assumed behavior throughout the region is very different than the out-of-band sampling throughout the region. This illustrates a key issue with in-band measurement sampling only between regions: the behavior within a region is not guaranteed to be uniform.

Chapter 6

Trinity TR2 Experiments

This chapter describes the dual-sided analysis of Advanced Power Management Trinity system at scale. Specifically, this includes running the mini appliations described in Chapter 5 beyond what is possible on the testbed systems, scaling to thousands of nodes and hundreds of thousands of cores. This includes investigating per-node power usage as scale increases, as well as P-state configuration and its effect on total application performance. Furthermore, we also look at a snapshot of an in-the-wild power analysis of real applications run on Trinity in April 2017 to illustrate the effectiveness of continual power management on a large-scale system.

Mini Applications on TR2

As part of the APM NRE project for Trinity described in Chapter 4, Cray's advanced power management features were enabled for the Trinity system. This includes updates to the Cray RAS software and PMDB that allows for receiving detailed power measurements on such a large-scale system. In this, we can evaluate vary P-states, power caps, and measure total energy consumed, both at a node level and a job level. These capabilities will allow us not only to discern the best performing application, but perhaps provide the best energy-to-solution metric given a certain time envelope, or even a new way to evaluate overall application efficiency as a function of current vs potential power draw.

Looking first at MiniMD in Figure 6.1, we can see a few things occur as we vary P-state and scale. The first observation is that as P-state frequency is lowered, average node power and performance (as a percent decrease of FOM) both decrease as well. What is interesting is that this is not a linear progression. Instead, initial ramp-down from Turbo to the highest base frequency (1.4Ghz for KNL), there is a generally smaller impact on FOM and a larger decrease in node power draw, compared to the lower frequencies, whereby performance takes deceases more than node power.

Another observation that as as scale increases, per-node power draw decreases. This is expected as the amount of communication necessary increases with scale. The observation of reduced power usage at increased node count may indicate room for potential overlap of commutation and communication, or some other potential for algorithmic refinements,

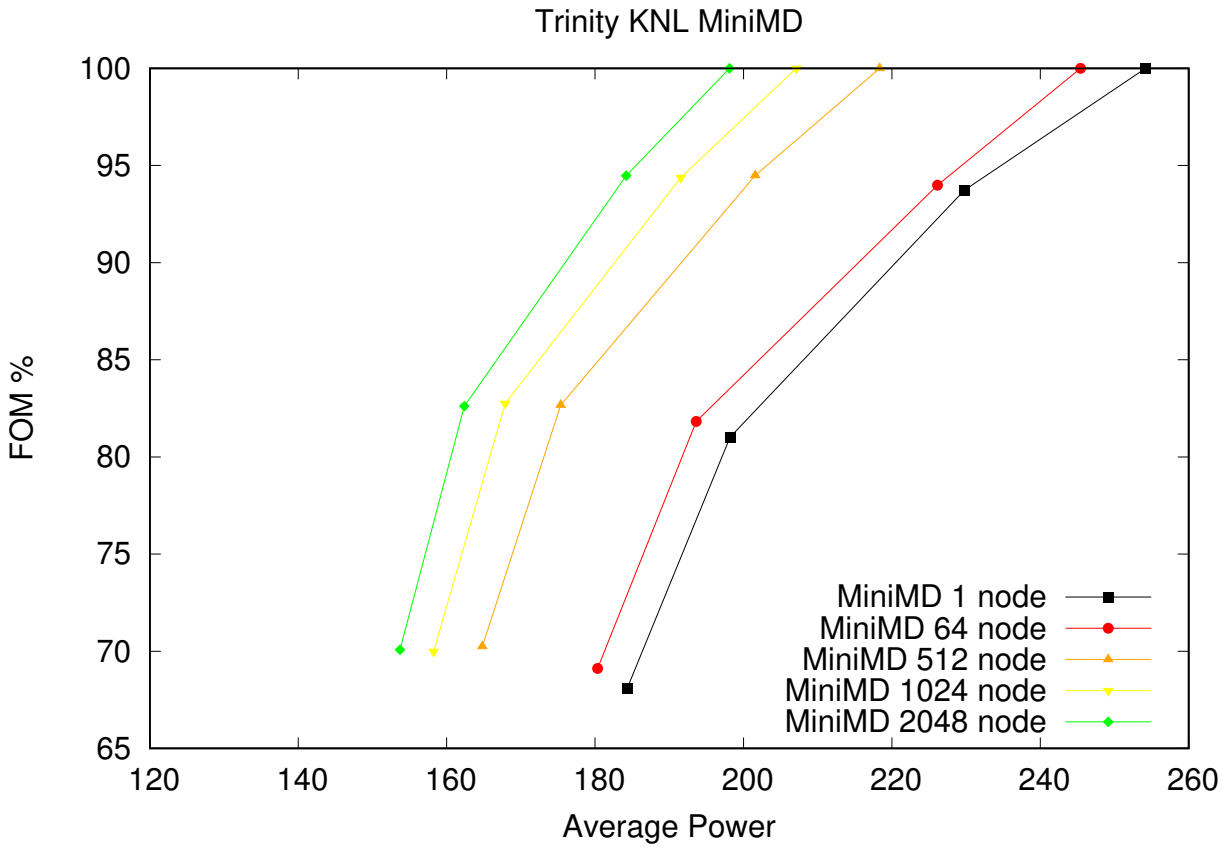


Figure 6.1. Average node power of MiniMD, as a percent form Turbo frequency FOM.

however this is application specific and may not apply to all systems. These initial observations confirm our expectations from testbed small-scale experiments, that small changes in P-state equate to substantial power savings at some minimal decrease in performances, but diminishing returns exist beyond.

Viewing MiniMD next in Figure 6.2, we see a number of interesting differences between job node count. First, single-node MiniFE runs consume substantially more node power than others at scale. However, once multi-node MiniFE runs exist, their per-node power draw does not vary considerably from 64 to 2048 nodes, indicating that the performance impact of multi-node parallelization is hit early and is potentially independent of job size. Furthermore, we again see that low frequency P-states (such as 1.0 Ghz) have a considerable impact on total application performance, as measured by FOM. Comparing 1.0 Ghz to Turbo frequencies of MiniFE at large scales, we see there is only roughly a 8% power reduction per node, yet up to a 20% impact on performance. As also observe with MiniFE that large-scale runs generally have a low per-node power draw, operating at just 40 % of peak power draw per node. With MiniFE, we expect this is due to the long assembly phase in MiniFe which is

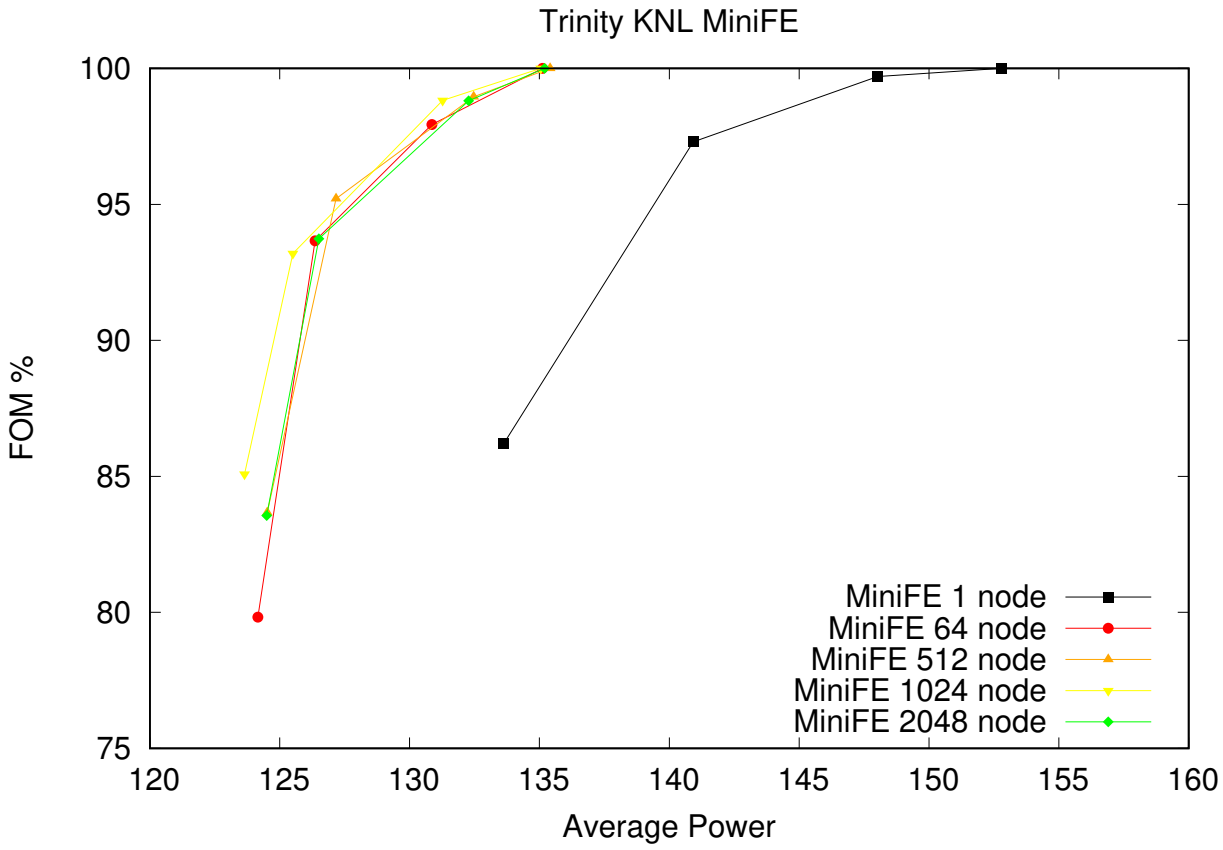


Figure 6.2. Average node power of MiniFE, as a percent form Turbo frequency FOM

not properly parallelized, but this example still shows how peak power draw may not match expected power draw for HPC applications of interest.

Moving to LULESH in Figure 6.3, we again see a similar comparison between application performance, normalized to Turbo frequency figure of merit, and power per per node. With LULESH, the power curves are flattened, indicating that performance is more directly tied to CPU frequency. Furthermore, varying CPU frequency has a predictable result independent of scale, as per-node energy only changes by a few watts as scale increases from 1 node to 2197 nodes (13 cubed). The difference between Turbo and non-turbo frequencies is also interesting, as giving up roughly 5% of performance gains around 7% decrease in power utilization.

With LULESH, we also can evaluate the impact of scaling as a factor of FOM per node, seen in Figure 6.4. Here, we can see first that FOM per node does not vary much as scale increases, which is a strong factor for high scalability of LULESH itself. Second, we see there is only a little decrease in FOM moving form Turbo to 1.4 Ghz P-state, independent of scale. This again confirms that it is likely we can evaluate power profiles of applications

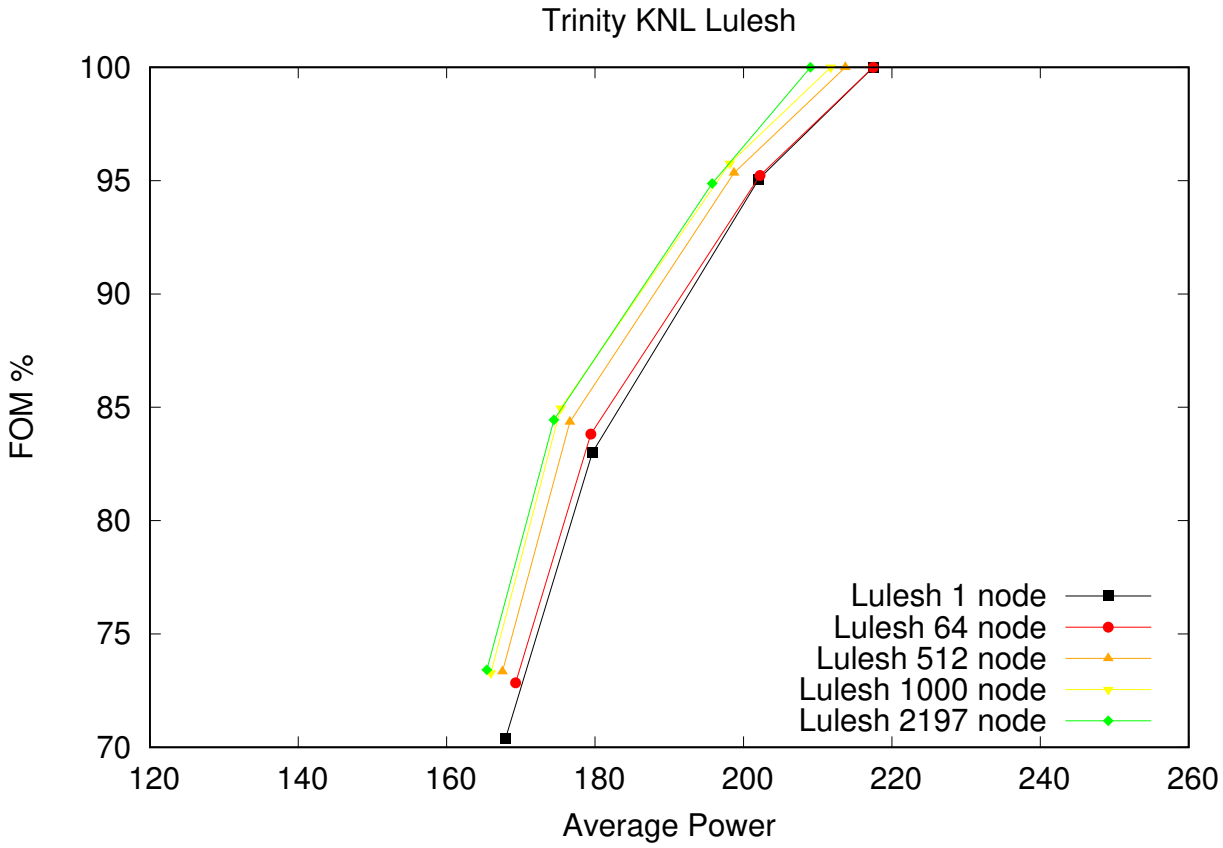


Figure 6.3. Average node power of Lulesh, as a percent form Turbo frequency FOM

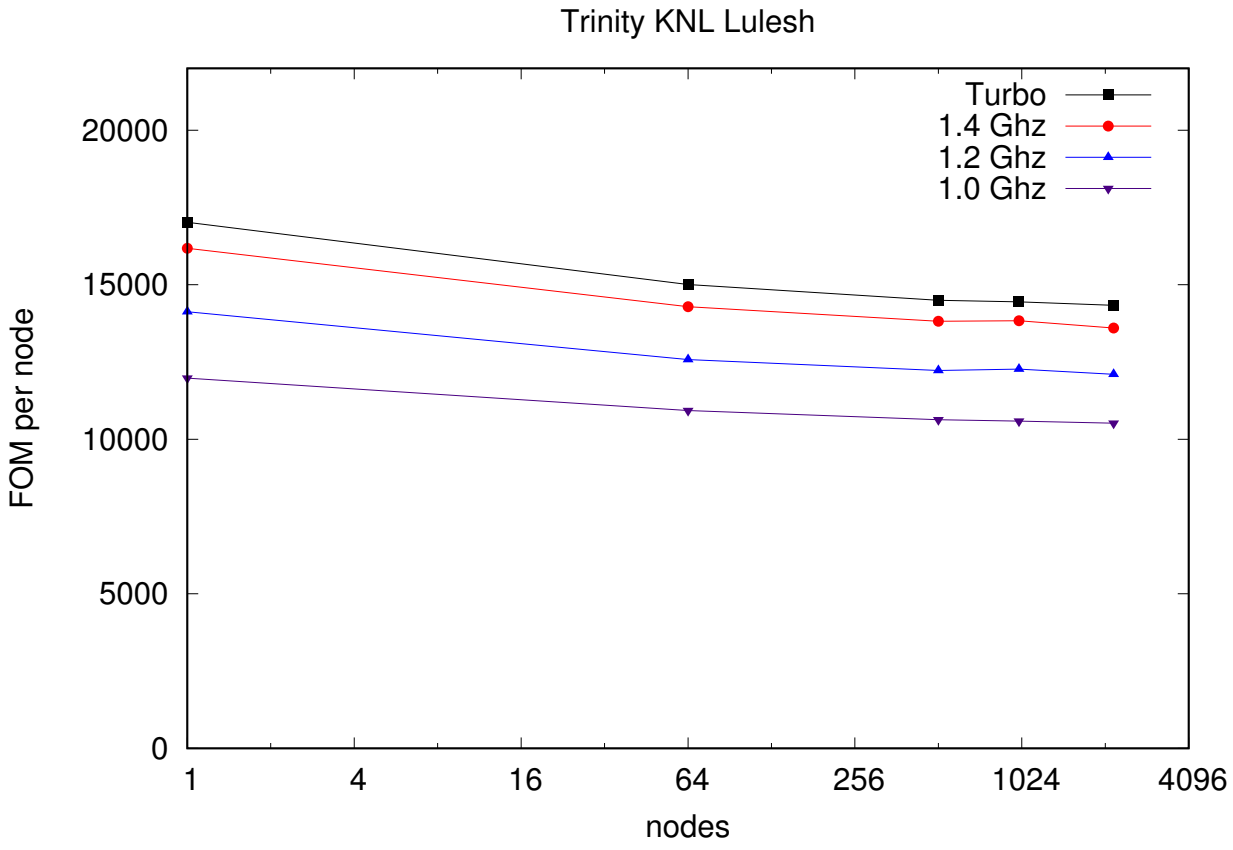


Figure 6.4. Lulesh Figure of Merit per node, scaling to 2197 nodes

at a smaller scale using testbeds with tens or hundreds of nodes, with expected conclusions when scaling is increased to thousands of nodes.

In-the-Wild Analysis

As part of this milestone and in collaboration with LANL and Cray, a method was developed to collect and archive 1 Hz node-level power samples for the duration of the three month Trinity TR2 open science period from March to May 2017. This power data is available for post analysis together with job scheduler logs and other system logs, enabling per-job energy usage over time to be evaluated. The volume of data collected was approximately 30 TB (2.7 TB compressed), making this a challenging data set to work with. Work is currently underway to analyze this data set in more detail, but our initial analysis focused on a series of large-scale application runs performed during a dedicated system time on April 18, 2017.

Two workloads consisting of multiple simultaneous application runs were evaluated. The first workload, shown in Table 6.1, consisted of four 1024 node CTH runs, two 1024 node SPARC runs, and one 2048 node SPARC run that were all launched more or less simultaneously on the TR2 system. The second workload, shown in Table 6.2 consisted of the same four CTH jobs as workload 1 running together with four 1024 node PARTISN runs. The tables include summary power and energy statistics for each run.

Run-to-run performance variation was higher than expected in several cases. For example, in Workload 1 (Table 6.1) the final CTH run was 14% slower than the fastest CTH run, even though each of the four CTH runs were configured identically. More alarmingly, the Baseline1 run of the final PARTISN run, Workload 2 (Table 6.2), was 32% slower than the others, with no obvious explanation. To investigate further, we analyzed the archieved power and energy information collected during each run to look for possible explanations.

We focused on comparing the fastest and slowest PARTISN runs in Workload 2, "PARTISN 2" and "PARTISN 4" respectively, as these had the largest run-to-run variation observed. As can be seen in Table 6.2, there was not a significant difference in average power per node between PARTISN 2 (201.90 W) and PARTISN 4 (202.75 W). The average power was calculated by dividing the total energy used by the job, as recorded by Cray's RUR tool, by the run's total execution time and then dividing by the number of nodes (1024). RUR additionally breaks down the total energy into CPU and memory components. At this level, there is a more significant 12% difference in memory power between the two jobs. However, PARTISN was configured to run exclusively out of on-package MCDRAM, which is counted in the CPU energy measurement rather than memory energy (external DIMM slots). This suggests the 12% difference may be more a result of part-to-part differences in idle external memory power for the different set of nodes used by each run.

The aggregated job-wide energy usage values reported by RUR obscure the individual node-level details. It could be the case that one node out of the 1024 nodes has a very different power usage behavior than the others, possibly suggesting a "slow node". To probe further, we plotted the 1 Hz power samples recorded for each individual node, shown in Figure 6.5. The plot includes 1024 separate curves, but they largely overlap making it difficult to see the fine detail. There are roughly 100 spikes evident in each plot, which likely correspond to the 100 cycles that PARTISN was configured to run. In the PARTISN 4 run, the spikes are more spread out than in PARTISN 2, indicating that the slow down in PARTISN 4 is spread out over the entire run rather than centralized to a single time period. If there were slowdowns due to network or I/O contention, we would expect there to be large dips in power usage during each cycle. This is either not the case or the dips are obscured by the overlapping waveforms for the 1024 nodes. As analyzing each of the waveforms by hand is not practical, it might be useful to apply a clustering algorithm or some other automated technique to look for outliers. This is a possible area for future work.

As a final effort to understand the run-to-run variation, we plotted histograms of the per-node average power usage for each run. The histograms count the number of nodes that had the average power usage shown on the x-axis, with 50 bins used. Separate distributions for total node power, CPU power, and memory power are plotted. In general, the histograms

for the two runs look very similar. The primary difference seems to be in memory power, with the slower PARTISN 4 run having a narrower range of values, except for one outlier node that is barely visible with an average memory power of 27 W. This is further confirmed by plotting the 1 Hz memory power samples, shown in Figure 6.6. We have not yet been able to confirm the reason for this outlier node, but it could be due to a miscalibrated power sensor or the node's memory mode somehow being misconfigured (e.g., set to quad-cache instead of quad-flat). A second run of Workload 2 produced the same outlier behavior, but in the second run the PARTISN 4 performance was as expected (i.e., not significantly slower than the other runs). Hence, we do not believe the high memory power outlier node is the reason for the slower PARTISN 4 run, but we continue to investigate.

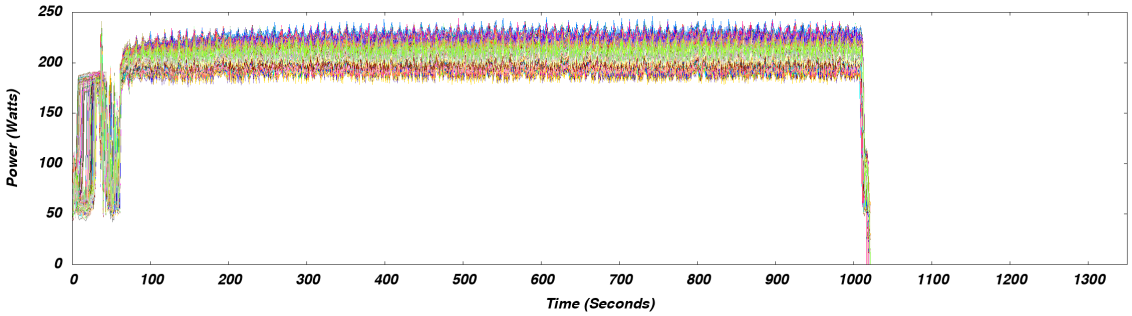
While this analysis of the archived power samples did not provide a "smoking gun" for the observed performance variability, it does demonstrate how this information could be useful. In this case, the time vs. power plots clearly showed that the slowdown in the PARTISN 4 run was evenly spread out across its entire runtime. This provides valuable information and allows certain classes of problems, such as a long I/O or network interruption, to be ruled out.

Table 6.1. Workload 1 Power and Energy Usage

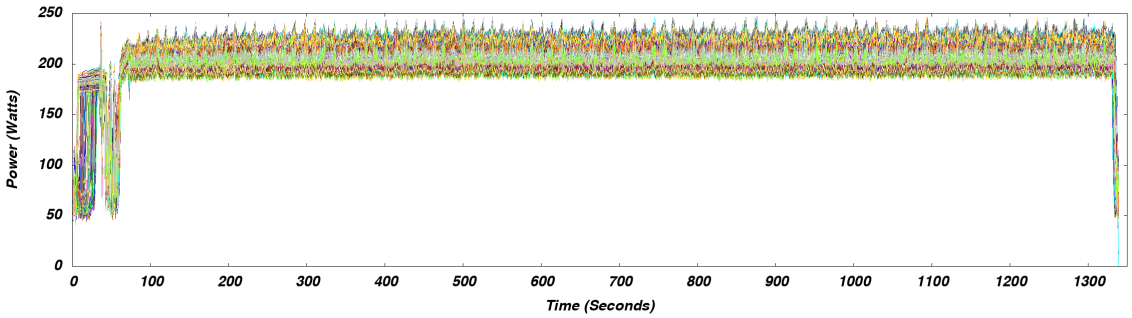
	Nodes	Runtime (s)	Total Energy (J)	Avg Power Per Node (W)	Avg CPU Power Per Node (W)	Avg Mem Power Per Node (W)
CTH 1	1024	1343	278268114	207.28	148.84	12.39
CTH 2	1024	1339	281224066	205.10	147.28	12.03
CTH 3	1024	1304	274096754	205.27	148.24	12.31
CTH 4	1024	**1485**	310102646	203.93	146.73	12.20
SPARC 1	1024	1371	312268221	222.43	145.78	28.54
SPARC 2	1024	1369	306805951	218.86	145.02	26.58
SPARC 3	2048	1512	685787490	221.47	145.82	27.89

Table 6.2. Workload 2 Power and Energy Usage

	Nodes	Runtime (s)	Total Energy (J)	Avg Power Per Node (W)	Avg CPU Power Per Node (W)	Avg Mem Power Per Node (W)
CTH 1	1024	1236	264790220	209.21	149.87	12.30
CTH 2	1024	1249	264785934	207.03	148.33	11.95
CTH 3	1024	1196	254463798	207.78	149.72	12.23
CTH 4	1024	**1424**	299234211	205.21	147.28	12.12
PARTISN 1	1024	1120	233431163	203.54	146.73	11.57
PARTISN 2	1024	1019	210674495	201.90	146.32	10.53
PARTISN 3	1024	1039	215450107	202.50	146.94	10.87
PARTISN 4	1024	**1343**	278823301	202.75	145.69	11.81

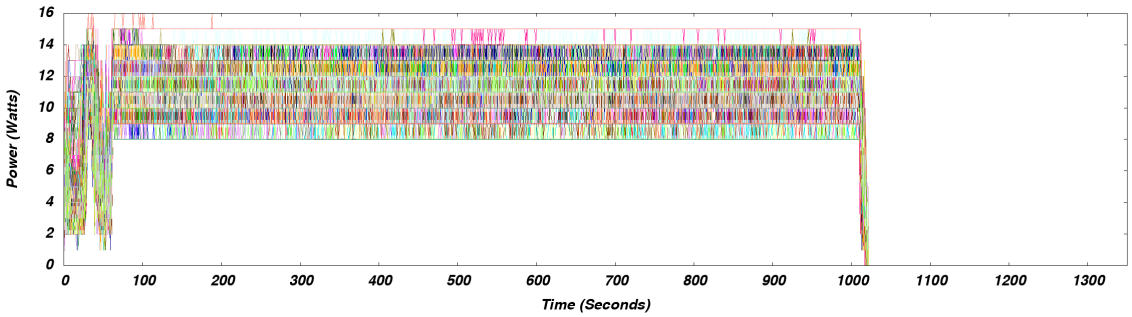


(a) PARTISN 2

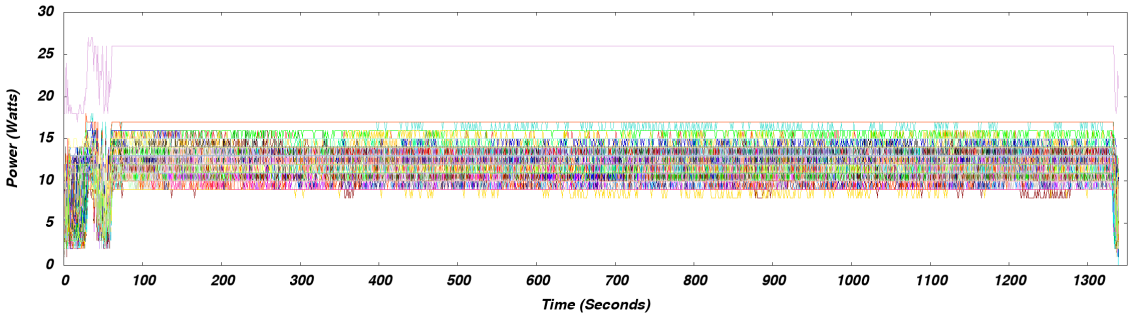


(b) PARTISN 4 (runtime +32%)

Figure 6.5. Node-level power over time for two of the PARTISN runs in workload 2. Spikes likely correspond to application cycles indicating that any slowdown in PARTISN 4 is not a large localized event.

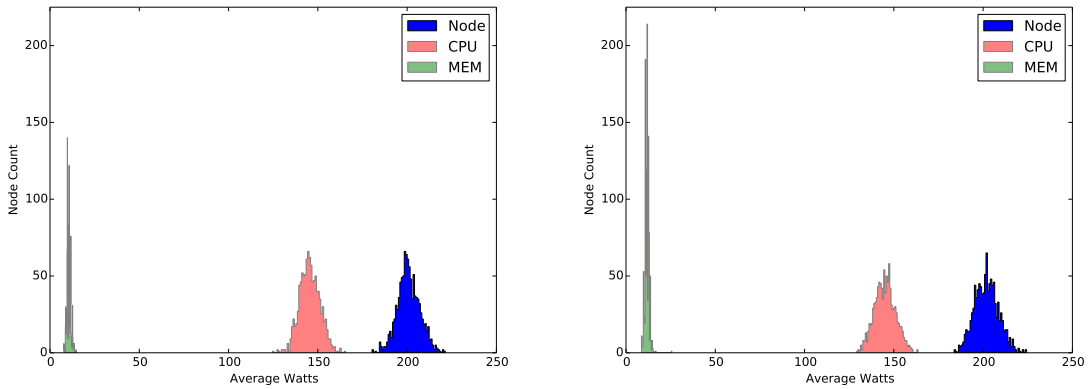


(a) PARTISN 2



(b) PARTISN 4 (runtime +32%)

Figure 6.6. Memory power over time (external DIMMS, does not include MCDRAM) for two of the PARTISN runs in workload 2.



(a) PARTISN 2

(b) PARTISN 4 (runtime +32%)

Figure 6.7. Histogram of node-level average power for two of the PARTISN runs in workload 2. The histograms include average power for each of the 1024 nodes in each run, calculated from the per-node 1 Hz power samples recorded during each run. There is an outlier node value in the PARTISN 4 allocation.

Chapter 7

SPARC Experiments

This chapter presents our analysis of the power and energy usage characteristics of the SPARC application running on ATS-1 Trinity hardware. The Advanced Power Management (APM) capabilities of the Trinity platform are utilized to measure and control the power usage of SPARC running on Trinity's Haswell and Knights Landing compute nodes. We perform controlled experiments on a small-scale Trinity testbed system to better understand the sensitivity of SPARC to compute node type, OpenMP configuration, static P-state control (CPU frequency), solver configuration, Knights Landing on-package memory configuration, and static node-level power capping control.

SPARC: A Performance Portable Compressible CFD Code

SPARC is a next-generation compressible computational fluid dynamics (CFD) code being developed by Sandia National Laboratories as part of the NNSA's Advanced Technology Development and Mitigation (ATDM) subprogram. Howard et. al. [33] describe SPARC as follows:

> SPARC is a compressible computational fluid dynamics (CFD) code being developed to solve aerodynamics and aerothermodynamics problems primarily for NNSA's nuclear security programs. In its present state, SPARC solves the Navier-Stokes and Reynolds-Averaged Navier-Stokes (RANS turbulence models) equations on structured and unstructured grids using a cell-centered finite volume discretization scheme and is targeted towards the transonic flow regime to support gravity bomb analyses and the hypersonic flow regime to support re-entry vehicle analyses. SPARC also solves the transient heat equation and associated equations for non-decomposing and decomposing ablators on unstructured grids using a Galerkin finite element method. One and two-way multiphysics couplings exist between the CFD and ablation solvers within the code. Current development is being driving by the re-entry application space, which nominally involves hypersonic flows, ablator/thermal response, and structural response of the vehicle under normal and hostile environments.

A key goal of the effort to develop SPARC is to produce a performance portable code, meaning that it should run well on a diversity of HPC architectures with as little application code as possible being aware of architectural differences. Specific targets include traditional multi- and many- core CPU-based systems (e.g., ATS-1 Trinity Haswell and Knights Landing compute nodes, respectively) and accelerator-based systems (e.g., ATS-2 Summit Volta GPUs). The Kokkos programming model is being used by SPARC to assist with performance portability, enabling application developers to focus on core algorithm design and exposing parallelism while Kokkos takes care of mapping computation and data structures to the underlying hardware architecture [74].

In this work, we analyze SPARC running on ATS-1 Trinity hardware. This platform provides an interesting case study because it includes two types of compute nodes that otherwise provide the same execution model — multiple traditional CPU cores per node — and a similar x86 instruction set architecture, but significantly different implementations. The Trinity Haswell compute nodes represent an evolution of Intel's traditional multi-core server line of processors. The Trinity Knights Landing compute nodes represent a new "many-core" design that provides a larger number of cores with relatively lower performance per core compared to Haswell. Additionally, Knights Landing includes 16 GB of on-package high bandwidth memory (HBM [1]) that provides over 4x the memory bandwidth of Haswell's DDR4-based main memory. These architectural characteristics were chosen to target highly-parallel HPC workloads, particularly those with working sets that fit within the Knights Landing's on-package memory. A key question is whether the new design of Knights Landing benefits realistic NNSA workloads such as SPARC.

Test Setup and SPARC Configuration

We performed a series of controlled experiments on the 'Mutrino' Cray XC40 testbed system at Sandia. Mutrino consists of 100 Haswell and 100 Knights Landing compute nodes that are identical to those used on Trinity. Due to the system being in high demand, experiments were limited to 32 nodes of each compute node type. This was deemed large enough to provide a reasonable test case while not significantly impacting system availability to application code teams.

We obtained SPARC from the developer git repository in early August, 2017. The Intel Parallel Studio XE compilers version 17.0.1 were used to build an optimized executable for each compute node type by following the instructions provided on the SPARC developer website. The default Trinity programming environment was used with the appropriate `sparc-dev` module loaded.

A "Generic Reentry Vehicle" (GRV) input problem was used for all testing. This problem was recommended by SPARC developers as a good test case and has been used in previous

[1] We use the term high bandwidth memory (HBM) in the generic sense, not the JEDEC standard. Knights Landing's HBM is implemented with MCDRAM technology.

```
# Trinity Haswell (HSW) Configuration
#SBATCH --nodes=32 --ntasks-per-node=32 --partition standard
export OMP_NUM_THREADS = 1
export OMP_PLACES      = threads
export OMP_PROC_BIND   = close
srun --nodes 32 --ntasks 1024 --cpu_bind=cores sparc.exe

# Trinity Knights Landing (KNL) Configuration
#SBATCH --nodes=32 --ntasks-per-node=32 --partition knl \
   --constraint quad,flat -S 4
export OMP_NUM_THREADS = 8
export OMP_PLACES      = threads
export OMP_PROC_BIND   = close
srun --nodes 32 --ntasks 1024 --cpu_bind=cores \
     --cpus-per-task=8 numactl --membind=1 sparc.exe
```

Figure 7.1. Baseline run configurations used for SPARC GRV problem on 32 nodes.

studies [33]. The GRV problem was configured for 1000 time steps with 10 linear iterations per timestep and no norm calculations. All testing was performed on 32 nodes with 32 MPI ranks per node (1024 MPI processes total), resulting in a memory footprint of 13 GB per compute node. This enables the problem to fit entirely within the Knights Landing's 16 GB of on-package memory.

Figure 7.1 shows the recommended run configuration that were used for each compute node type. From this baseline, we experimented with changing OpenMP configuration, static P-state selection, solver configuration, and static power cap configurations. Additionally, for Knights Landing nodes we compared running the problem from on-package memory vs. off-package DDR4 memory.

Experimental Results

Sensitivity to OpenMP Configuration

Prior experience with running SPARC on Trinity has shown that it benefits significantly from using multiple hardware threads per physical core on Knights Landing nodes but not Haswell nodes. Table 7.1 shows the results of our OpenMP configuration experiments, which confirm this is the case. On Haswell nodes, using two OpenMP threads per physical core results in a slowdown of 2.4% for the total elapsed time reported by SPARC. Energy and

average power per node are also slightly increased.

The situation is reversed for Knights Landing nodes, where running four threads per physical core (OMP-8, each MPI process is allocated two physical cores that run four threads each) results in a 1.2x performance speedup compared to running one thread per core (OMP-2, each MPI process is allocated two physical cores that run one thread each). However, since energy-to-solution remains about the for these two cases, there is no energy-efficiency penalty for the increased performance.

It is interesting to observe that running two threads per core (OMP-4, each MPI process is allocated two physical cores that run two threads each) achieves essentially the same performance as running four threads per core. Furthermore, energy-to-solution improves by 5%, resulting in the average power draw per node dropping by 12 W. This suggests that there is little benefit from using the extra two threads per core. Nonetheless, since most runs of `SPARC` on Knights Landing in practice are using four hardware threads per core, we continue to use this configuration throughout the rest of our experiments.

Table 7.1. Sweeping OpenMP configuration for `SPARC` GRV problem on 32 nodes.

			Job-wide Aggregate Information			
	OpenMP # Threads	**SPARC Total Elapsed (s)**	**Wall Time (s)**	**Energy (MJ)**	**Power (W)**	**% Peak Power**
HSW	OMP-1	445.92	458	5.15	352	85%
	OMP-2	456.56	468	5.31	354	85%
KNL-HBM	OMP-1	797.26	804	4.72	183	53%
	OMP-2	431.59	439	3.15	224	65%
	OMP-4	355.81	364	2.97	255	74%
	OMP-8	355.71	366	3.13	267	77%

Sensitivity P-state and Solver

We performed a set of experiments to explore three dimensions simultaneously: compute node type, solver configuration, and P-state setting (CPU frequency). `SPARC` is undergoing active development to improve solver efficiency. In particular, we wished to evaluate the Trilinos block-tridiag solver that has been in development over the past year and compare it to the block-triag solver built into `SPARC`.

Figure 7.2 shows an overall comparison of three different solver configurations running on Haswell nodes (in black) and Knights Landing (in red) nodes. The curves labeled 'SPARC' present results for the built-in sparc/block-tridiag solver, which is the default for our input problem. Next, the curves labeled 'Trilinos' present results for the new tpetra-blockcrs/block-tridiag solver. This solver has been designed to provide performance similar to or better than

the SPARC built-in solvers, while being less application-specific, making it better suited for inclusion in a general-purpose library suite such as Trilinos. Finally, the curves labeled 'Naive' present results for the existing tpetra/belos solver. This solver was originally incorporated into SPARC to provide support for GPU-based systems. It was only intended to be used for unit testing on CPU-based systems. The data points in the figure are labeled with the static P-state (CPU frequency) that was used for the duration of the respective run, set via the `--cpu-freq` option to SLURM srun. For example, a label of '1.2' means that the application run using a fixed CPU frequency of 1.2 GHz. The 'Turbo' label means up to 3.6 GHz on Haswell and up to 1.6 GHz on Knights Landing. Turbo is the default run configuration on Trinity.

As can be seen in Figure 7.2, the new Trilinos solver consistently provides the best performance at a given P-state. It also consistently achieves the best energy efficiency, as shown in Figure 7.3. Compared to the built-in SPARC solver, the new Trilinos solver improves overall runtime by approximately 9% for Haswell and 8% for Knights Landing compute nodes with approximately the same average power draw per node (-1.7 W for HSW, +1.2 W for KNL).

Comparing the node types head-to-head for the best observed performance (Trilinos/-Turbo), Knights Landing provides 15% higher performance, 24% lower average power per node, and 41% lower energy-to-solution than Haswell. This demonstrates the energy-efficiency improvements of the Knights Landing architecture for a realistic NNSA-relevant workload.

It is interesting to point out the similarity and overlap of the 'Naive' curve for Haswell and Knights Landing in Figure 7.2. This indicates that both architectures are achieving roughly the same FLOPS/clock with this solver. This is likely because it is poorly vectorized (or not vectorized at all) and more CPU bound than memory bound. The SPARC and Trilinos solvers have been heavily optimized to be memory bandwidth bound, as indicated by their more horizontal slopes. This is shown more clearly in Figure 7.4, which plots P-state vs. linear equation solve time. For the SPARC built-in and Trilinos solvers, the absolute solve time is nearly flat, suggesting these solvers are highly memory bandwidth bound. The performance of the Naive solver, on the other hand, is highly dependent on CPU frequency.

Figure 7.5 shows the percentage of the overall solve time that is spent in the linear equation solver. For Haswell, this increases with CPU frequency because assembly is sped up significantly while the linear equation solve time reamins roughly constant because it is a memory bandwidth bound operation. This effect is present but less pronouced for Knights Landing. With Naive, both assembly and linear equation solve are equally sped up by increases in clock frequency.

Sensitivity to Knights Landing Memory Configuration

The Trinity Knights Landing have a two-level memory consisting of 16 GB of on-package high bandwidth memory and 96 GB of external DDR4 memory. The on-package memory

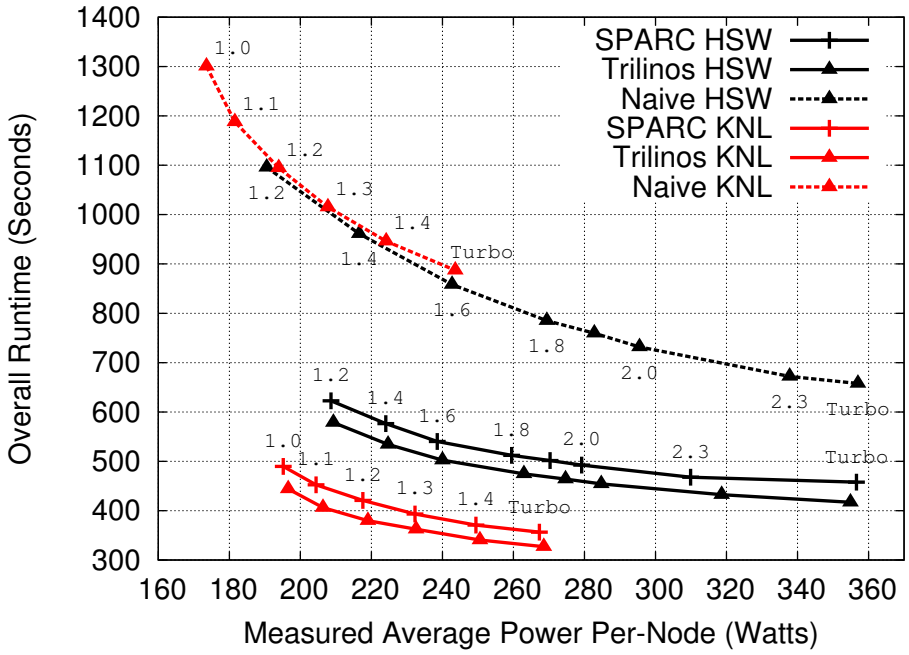


Figure 7.2. Power vs. time for different solver configurations for `SPARC` GRV problem on 32 nodes.

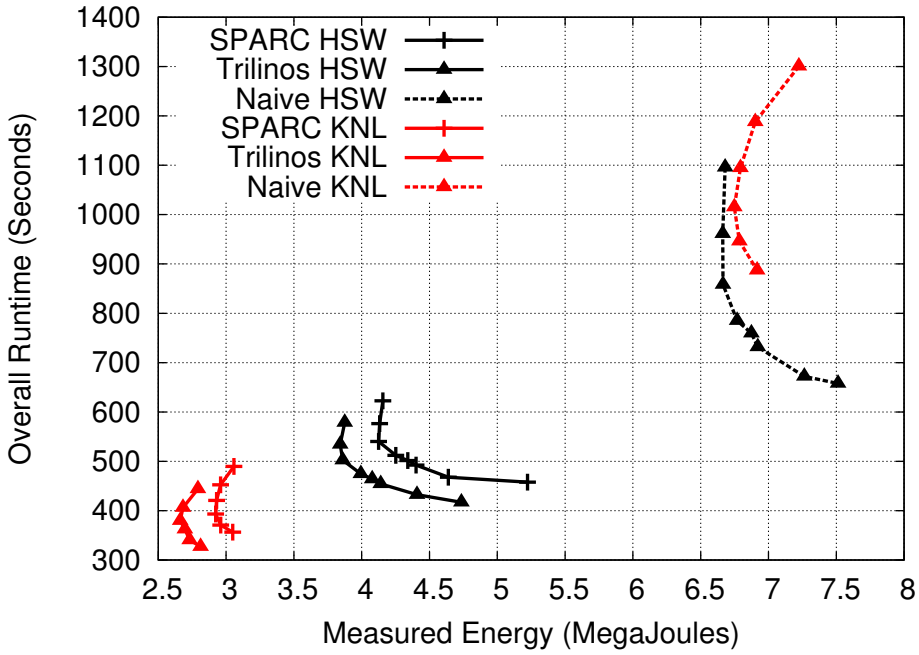


Figure 7.3. Energy vs. time for different solver configurations for `SPARC` GRV problem on 32 nodes.

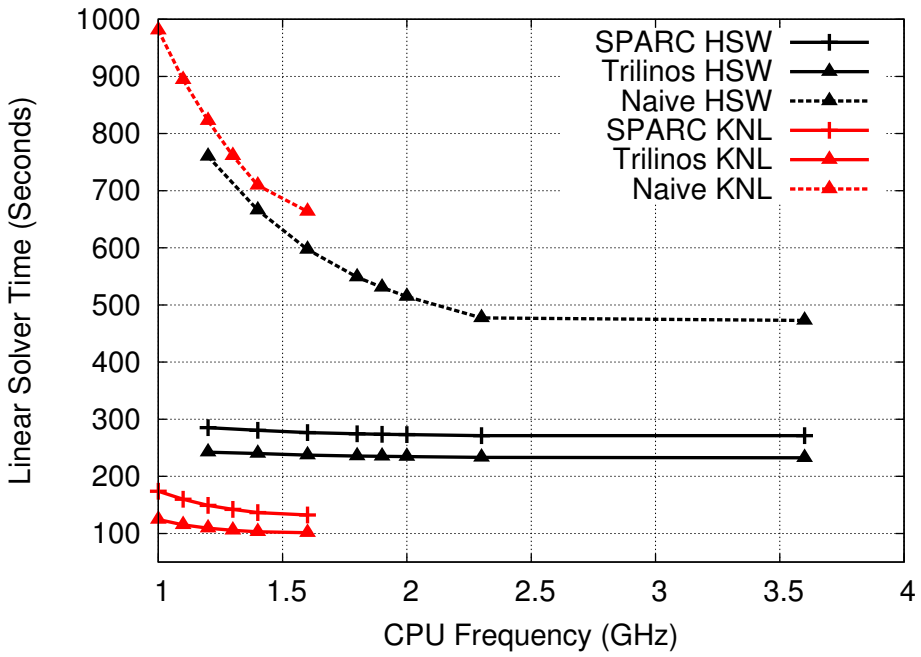


Figure 7.4. Linear equation solve time for different solver configurations for `SPARC` GRV problem on 32 nodes.

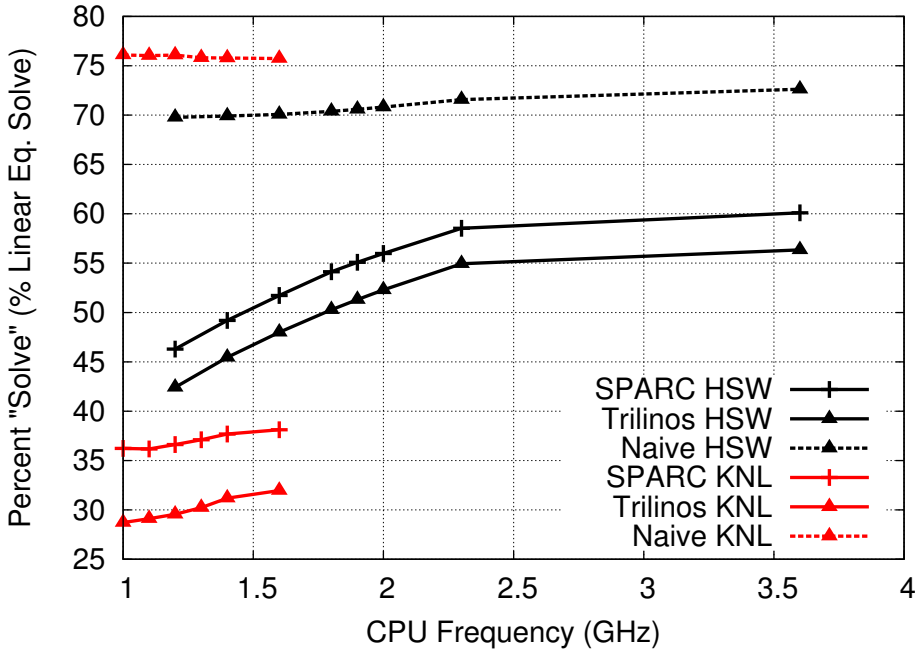


Figure 7.5. Percentage of linear equation solve time of overall solve time for different solver configurations for `SPARC` GRV problem on 32 nodes.

achieves roughly 400 GB/s of memory bandwidth on the STREAM memory bandwidth microbenchmark, which is about 4x better than memory bandwidth to the external DDR4 memory. This should result in significantly improved performance for a memory bandwidth sensitive code such as `SPARC`, especially for the linear equation solver portion of the code.

Figure 7.6 shows the results of our experiments running our test problem using solely on-package memory (labeled HBM, solid lines) and using solely external DDR4 memory (labeled DDR, dashed lines). The Knights Landing on-package memory was configured in the quad-flat mode and the `numactl` tool ws used to bind memory allocation to the appropriate memory type. As can be seen, both the `SPARC` and Trilinos linear solvers exhibit little performance sensitivity to CPU frequency for both types of memory. Using the on-package memory with these solvers provides roughly a 4.2x performance advantage, which closely matches the 4x improvement measured by `STREAM`. The Naive solver, in contrast, only sees a 1.2x advantage, suggesting it is not as memory bandwidth bound as the other solvers. This is further indicated by its greater dependence on CPU frequency.

For the `SPARC` and Trilinos solvers running with HBM, there is very little performance difference between the fastest (Turbo, up to 1.6 GHz) and slowest (1.0 GHz) P-states, however, there is a 27% average power draw difference. This indicates there is significant potential for energy savings, and reduced node-level power usage, during the linear solver portion of `SPARC`'s runtime. We have measured the overhead of changing P-state configurations to be less than 100 microseconds using Cray's compute node Power API implementation, which is three orders of magnitude less than the fastest linear solve time we measured (100 ms per timestep). This suggests dynamic P-state switching may be feasible for `SPARC` for certain input problems with minimal induced performance overhead.

We aslo examined running the Knights Landing's on-package memory in cached mode, which uses the 16 GB of on-package memory as a direct mapped last-level cache. This mode of operation should result in our test problem, which has a 13 GB memory footprint per node, being cached fully by the on-package memory. Our experimental results shown in Table 7.2 confirm this. Explicitly using the Knights Landing's on-package memory (KNL-HBM) provides roughly the same performance and power usage as using it as an implicit cache (KNL-CACHE). The Haswell results do not include finer-grained CPU and Memory (external DDR memory, does not include on-package memory) breakdowns because these components are not measured separately on Trinity Haswell nodes. Figure 7.7 provides a visualization of the results in Table 7.2.

Out-of-band 5 Hz Power Sampling

All of the aggregate power and energy information presented for `SPARC` thus far has been obtained by post-processing Cray's out-of-band node-level power sampling. This infrastructure samples each node's power usage at 1 Hz intervals and records this information in a SQL database on a separate system administration network. We increase the sampling rate to the maximum available, which is 5 Hz for each node. Analyzing the raw 5 Hz samples enables

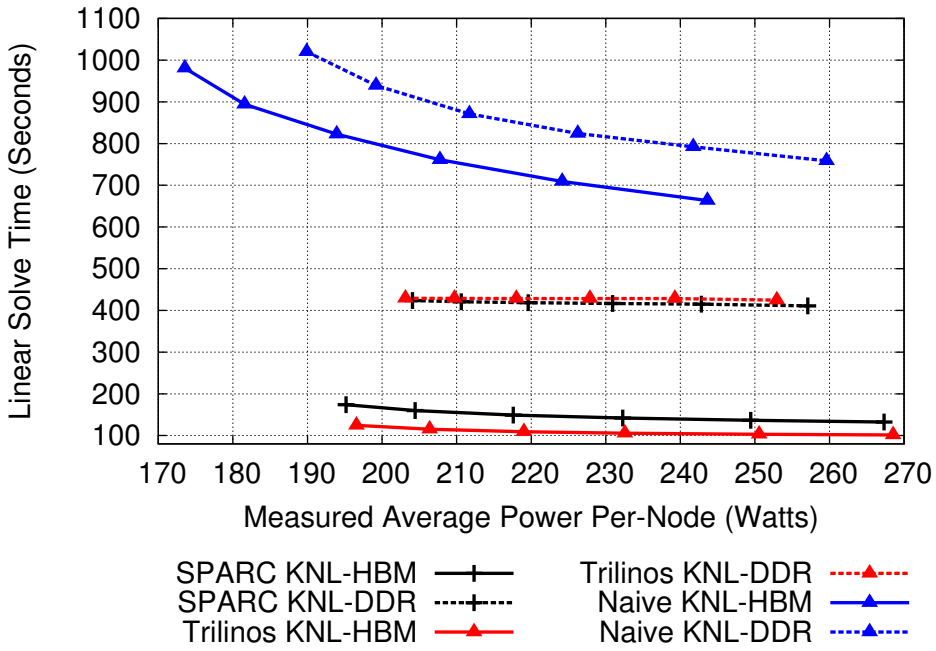
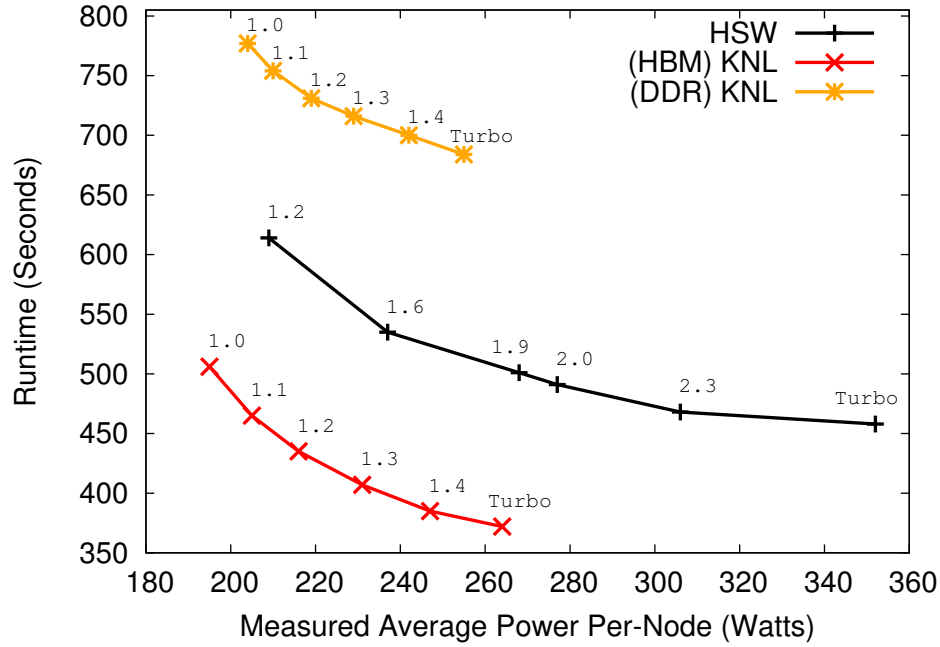


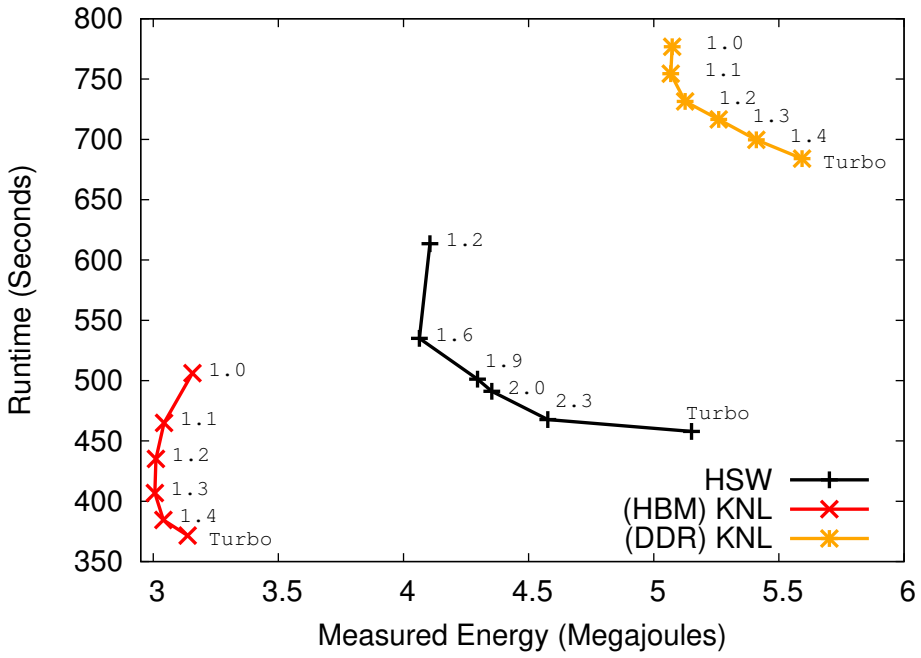
Figure 7.6. Comparison of SPARC GRV problem on 32 nodes running from KNL on-package memory vs. off-package DDR memory.

Table 7.2. Aggregate results for static p-state selection for `SPARC` GRV problem running on 32 nodes.

			Average Power Per-Node		
	P-state (GHz)	**Job Runtime (s)**	**Node (W)**	**CPU (W)**	**MEM (W)**
HSW	2.3+Turbo	458	352	–	–
	2.3	468	306	–	–
	2.0	491	277	–	–
	1.9	501	268	–	–
	1.6	535	237	–	–
	1.2	614	209	–	–
KNL-HBM	1.4+Turbo	372	264	195	13
	1.4	385	247	181	13
	1.3	407	231	169	13
	1.2	435	216	157	13
	1.1	465	205	148	13
	1.0	506	195	140	13
KNL-DDR	1.4+Turbo	684	255	166	39
	1.4	700	242	154	39
	1.3	716	229	145	38
	1.2	731	219	136	38
	1.1	754	210	129	37
	1.0	777	204	125	37
KNL-CACHE	1.4+Turbo	382	277	199	19
	1.4	381	249	184	13
	1.3	406	232	170	13
	1.2	432	218	159	13
	1.1	462	205	149	13
	1.0	506	196	141	13



(a) Power



(b) Energy

Figure 7.7. Aggregate results for static p-state selection for `SPARC` GRV input running on 32 nodes.

additional statistical values to be calculated, such as high-water marks and percentiles.

Figure 7.9 show the point-in-time power draw for a sweep of static P-state settings for each Trinity node type. We have examined many more power traces for `SPARC` and all exhibit similar behavior—relatively constant power draw over the entire runtime with progressively reduced average power draw and lower performance as P-state (CPU frequency) is reduced. Figure 7.9 presents a zoomed in view of this same data. The periodic nature of `SPARC`'s timesteps are visible for several P-state configurations. When this is the case, there are roughly 1000 peaks for our test problem.

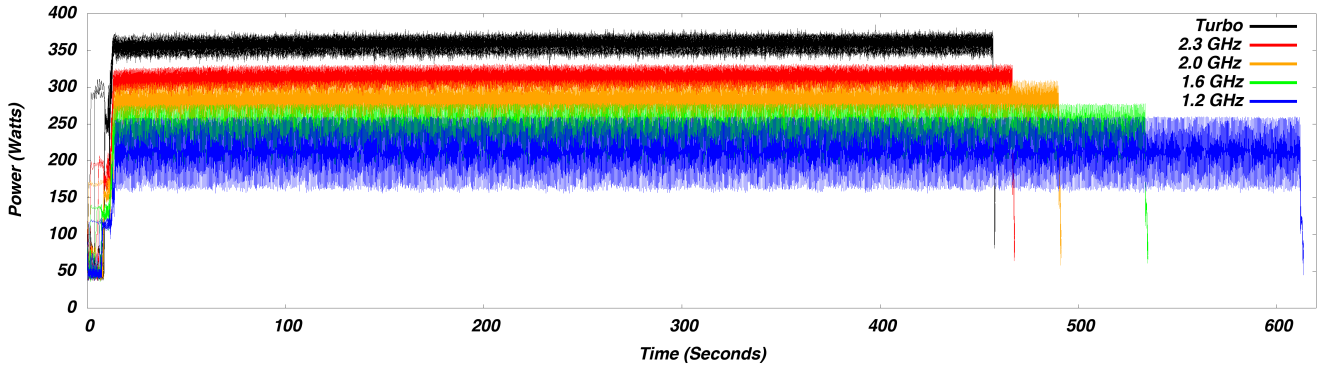
Access to the SQL database that stores the recorded power samples requires system administrator privilege and is not easily automatible. In the future it would be useful to provide a way for users to obtain this information automatically. Many of the users we have shown this information to would be interested in getting this information themselves, without needing to involve an administrator, in particular for diagnosing "slow node" problems and other issues that may be visible in their application's power usage signature. The tools and infrastructure developed by the Trinity Advanced Power Management NRE effort help to enable this, but more effort is required to deploy these capabilities on Trinity and make them accessible to users.

P-state vs. Power capping control

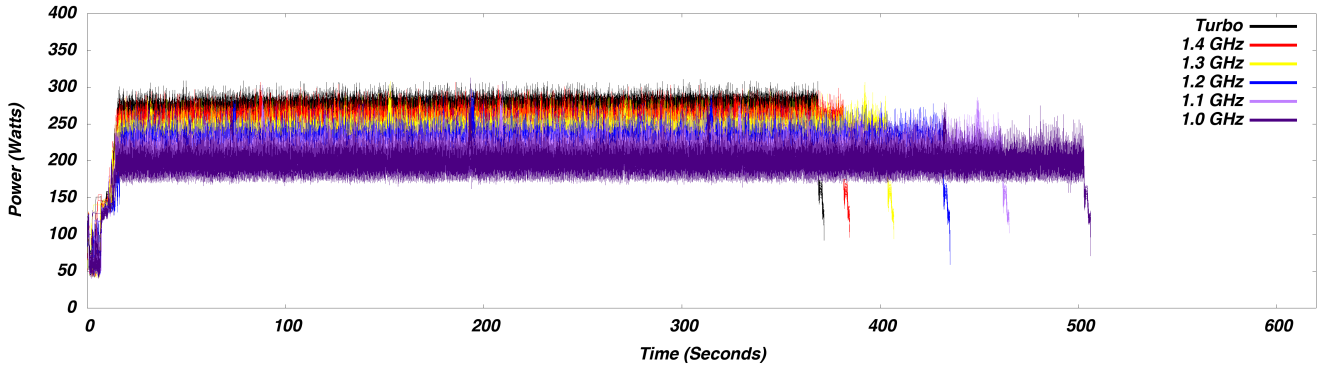
The Trinity platform includes a mechanism for setting the desired power budget for each compute node. This capability could be used, for example, by a power-aware resource manager to enforce power usage limits on specific jobs or the system as a whole. In a power constrained system—one where facility power and cooling infrastructure limits how much computing equiment can be powered at full speed—an intelligent workload manager or runtime system may be able to shift the available power budget to jobs where it will have the greatest performance impact. In prior work we performed an initial evaluation of power capping on Trinity's Haswell nodes [61], which highlighted the potential for significant performance degradation for some workloads when running under a power cap.

We performed a series of experiments to evaluate Trinity's node-level power capping mechanism for our `SPARC` workload. We analyzed the 5 Hz power samples from previous uncapped `SPARC` experiments for the default Turbo P-state to determine target power cap settings to test, including the maximum power value recorded for an uncapped run, the 95th percentile across all samples recorded on all nodes, the 75th percentile, 50th percentile, 25th percentile, and minimum power cap setting possible for each node type (230 W on Haswell nodes, 200 W on Knights Landing nodes). The range of node-level power cap savings available on each compute node with potential power savings are shown in Table 4.2.

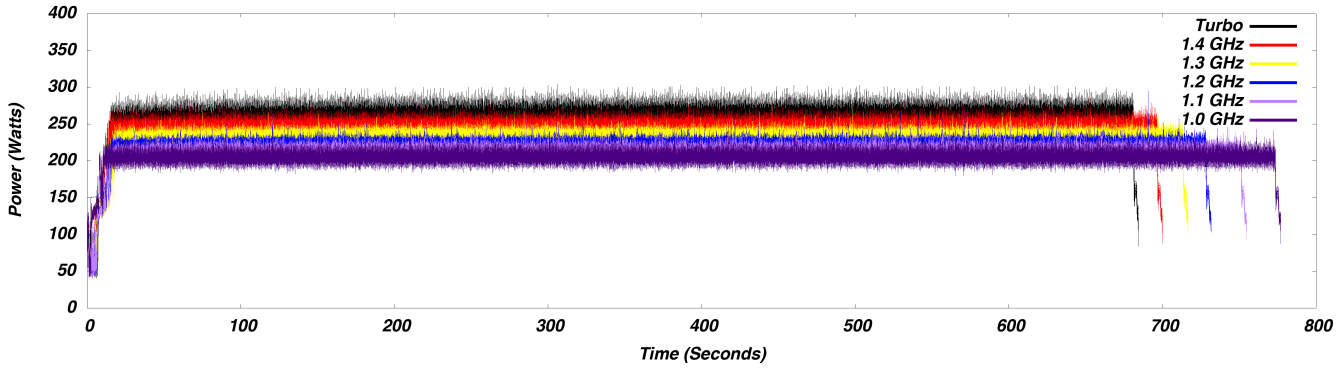
Figure 7.10 summarizes the results of our experiments. For Haswell nodes, there is little performance impact when capping at the 25th percentile (354 W) and higher. This matches our expectation that capping near or above an application's natural power usage



(a) HSW

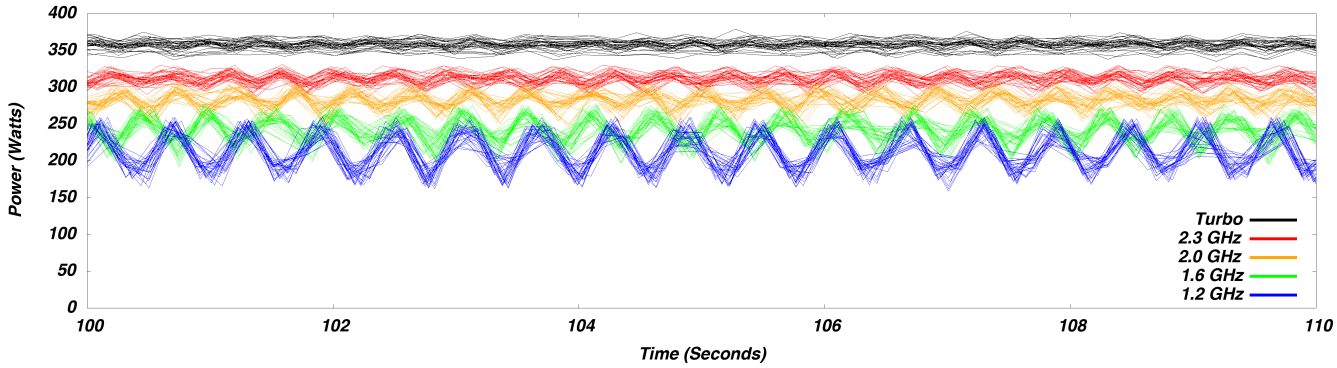


(b) KNL-HBM

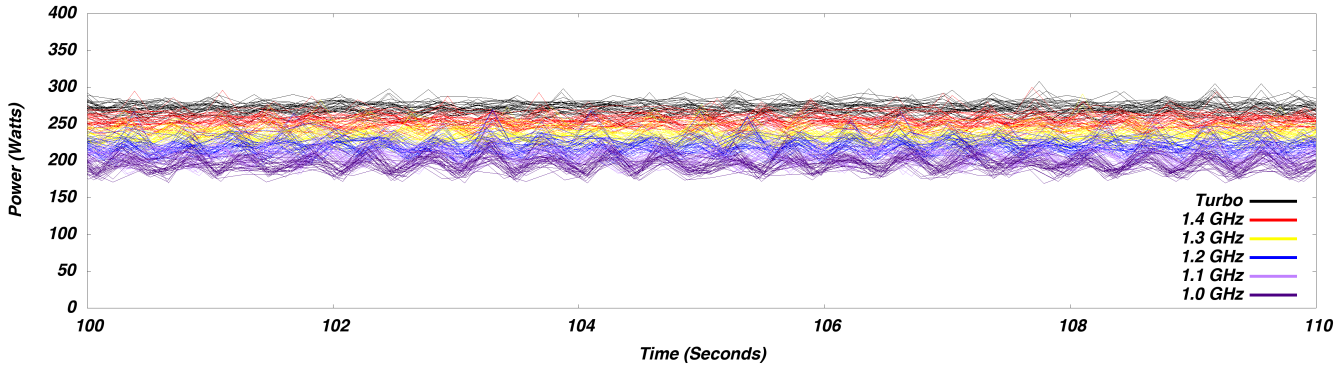


(c) KNL-DDR

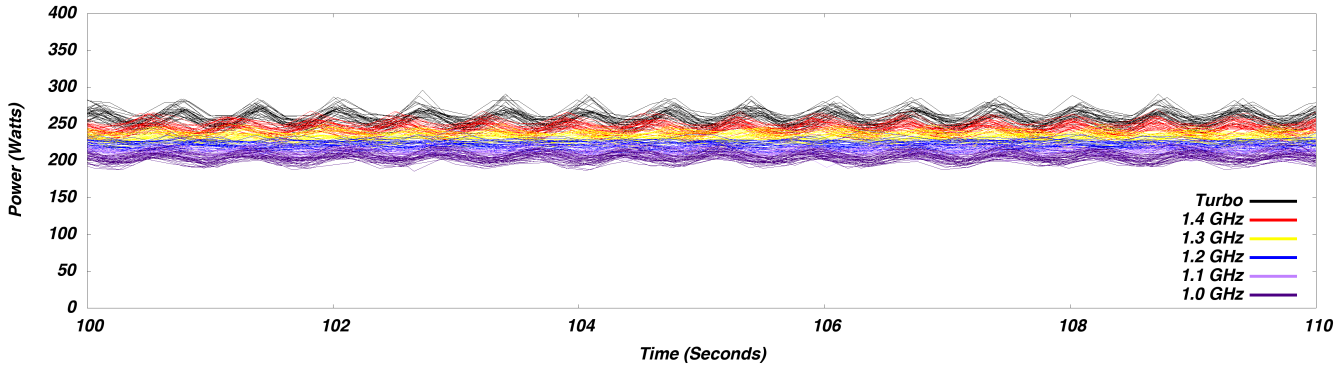
Figure 7.8. Time vs. power for static p-state selection for `SPARC` GRV input running on 32 nodes. Note that the x-axis limits in 7.8c are different than in 7.8a and 7.8b.



(a) HSW



(b) KNL-HBM



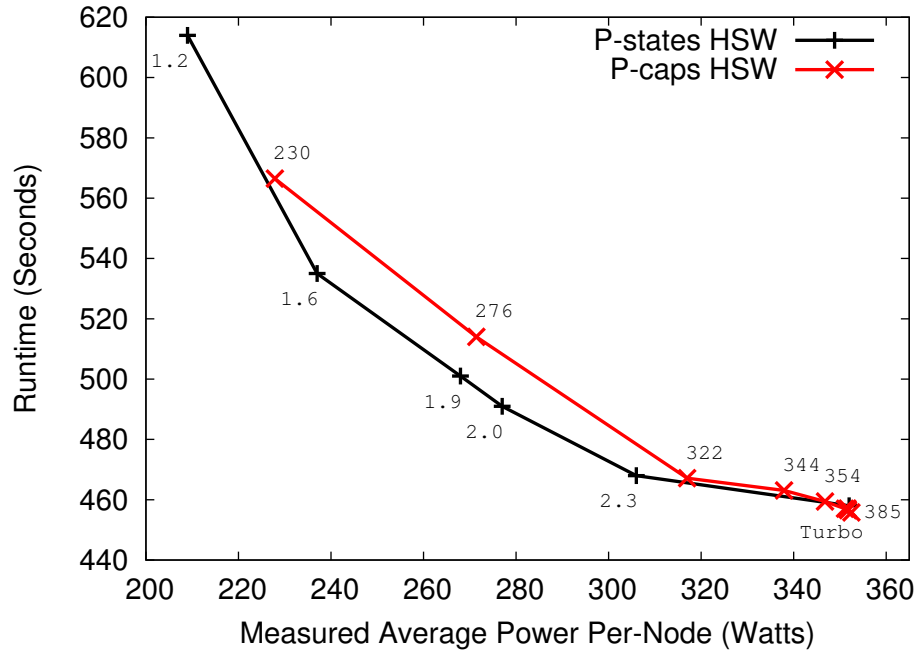
(c) KNL-DDR

Figure 7.9. Zoomed-in time vs. power for static p-state selection for `SPARC` GRV input running on 32 nodes.

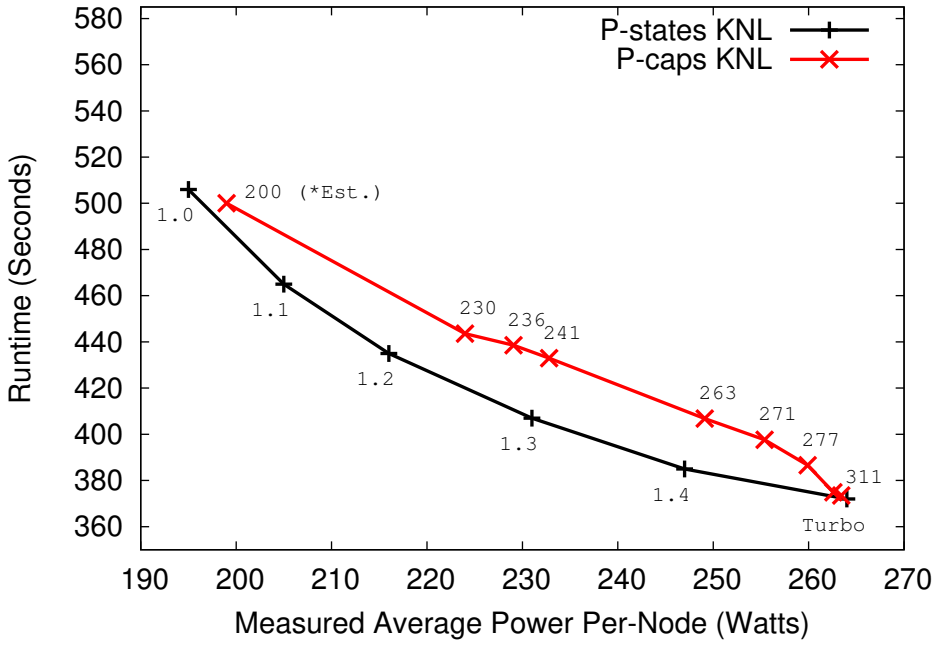
should not cause significant performance degradation. The capping mechanism will only be triggered, and hence potentially reduce performance, when the average power draw over an approximately 1 second rolling window exceeds the desired limit. Capping at lower levels, for example 276 W and 230 W on Haswell, leads to the capping mechanism being frequently triggered. As can be seen in the figure, this results in lower performance than using P-state control for a given average node-level power draw.

For Knights Landing nodes, there is a more noticeable performance penalty when power capping at `SPARC`'s natural power usage levels. Capping at the 75th percentile (277 W) and below leads to reduced performance compared to static P-state control. Furthermore, capping we observed that capping far below that level, for example below the 5th percentile of uncapped power samples (241 W), led to seemingly random node crashes that required a reboot. The KNL 230 W and 236 W power cap settings required many tries to get a full run in and runs using the 200 W setting were never successful. This could be due to broken hardware on our test system, but it could also be indicative of a less mature node-level power capping implementation on Knights Landing.

Figure 7.11 show point-in-time power plots for several of the power cap configurations tested. The power cap mechanism is visible as acting to reduce power at application startup time (far left) and then maintains a relatively steady power draw for the remainder of the run. The 200 W cap configuration for KNL failed shortly after startup at timestep 57.

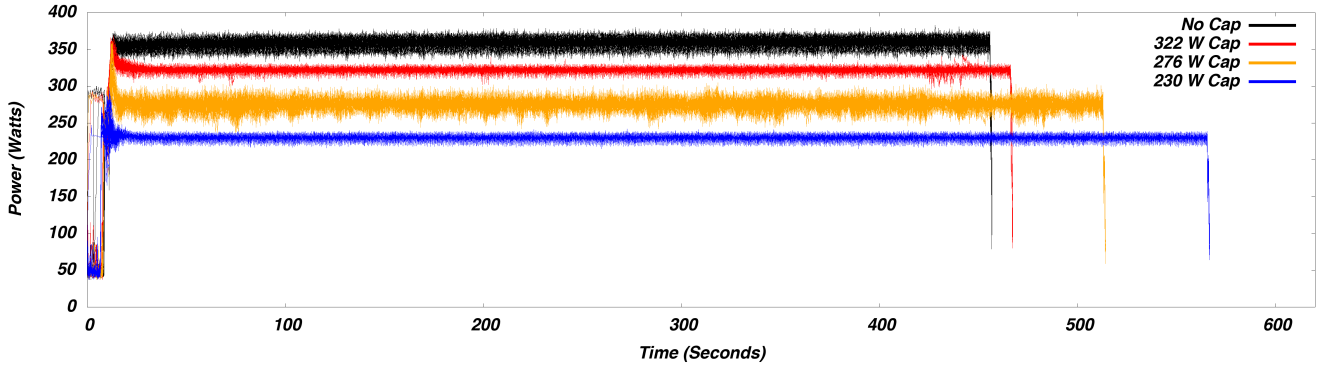


(a) HSW

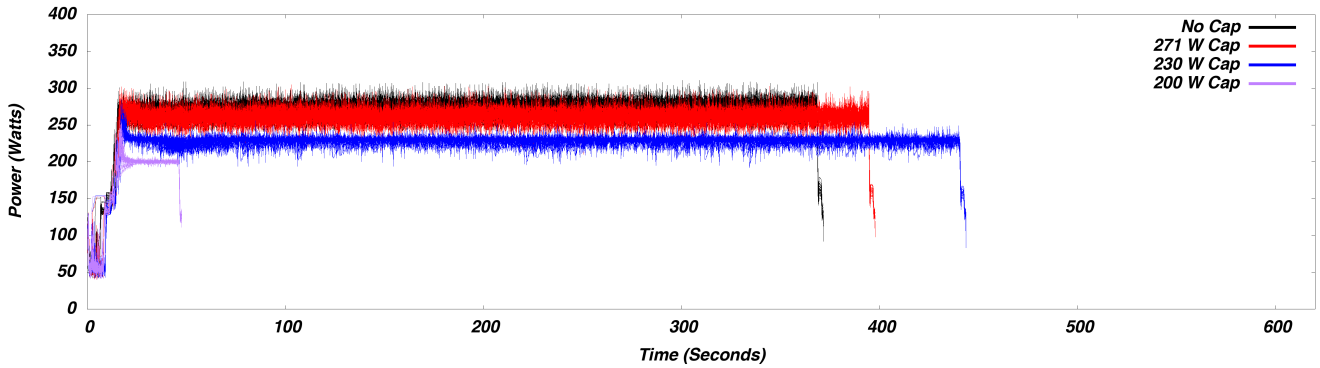


(b) KNL-HBM

Figure 7.10. Comparison of static p-state selection to static node-level power cap selection for `SPARC` GRV input running on 32 nodes.



(a) HSW



(b) KNL-HBM

Figure 7.11. Time vs. power for static node-level power cap selection for `SPARC` GRV input running on 32 nodes.

Chapter 8

Discussion

One of the key aspects of this L2 milestone is the vast knowledge and experience gained with advanced power measurement and control on large scale leadership class supercomputing resources. Without this effort, there are a number of key aspects to energy efficiency that would be left undiscovered. This section summarizes some of the key advancements and lessons learned as part of this L2 milestone.

Power Profiling Lessons Learned

The key lessons learned from our study can help to guide HPC application profiling for measuring power and energy. One must first decide what level scope is necessary when conducting power profiling. The taxonomy given in Section 3 assists in determining the level of detail desired, the amount of data available, and the amount of effort required.

If only total energy usage or average power for an application are needed but not phase data or periodic consumption rates, aggregate data collection is the best initial option. This is also the first approach to use when detailed application information is not available, such as in-depth knowledge of code regions. Aggregate data has low overhead and requires less storage space and analysis time than the other studied measurement techniques. If more data is required than aggregate collection can provide, then determine what further level of detail is needed. If application code knowledge is limited, out-of-band data collection or timed in-band periodic counter polling are the best next options. Out-of-band data is the preferred collection method but is often not available since it requires specialized out-of-band hardware like that of the Cray platforms. Moreover, out-of-band data may be available only to administrators, an active policy issue to be resolved at HPC facilities. Time-averaging of samples, if supported in the measurement system, is helpful to discover trends in the application. Phases are more easily discernible if adjacent regions of code do not have similar power profiles.

Application developers will often be able to correlate high-frequency out-of-band data measurements with specific details. Experts in the application domain can sometimes identify features in power profiles that correspond to specific code regions. In both cases where regions were clearly identifiable in our results (miniFE and miniMD), consultation with the

application developers resulted in immediate feedback on the behavior of the code and the observed phases. This feedback is useful both to the researcher conducting the test and for the application developer in determining new information about the intensity of given code regions with respect to the power consumption of different system components.

If detailed information on application code regions is necessary and sufficient knowledge of the application is given, then in-line application profiling should follow. Application profiling can be aligned to in-band data through measurement tools like RAPL to sample values surrounding code regions. However, in-band application instrumentation may significantly perturb performance, especially for short regions. Therefore, the best form of measurement for the most in depth understanding is to combine code regions with timestamps and correlate with out-of-band measurements. This method provides low-overhead, fine-grained data over the entire execution to enable useful observations that would not be visible otherwise. However, this approach requires the greatest amount of effort and knowledge of both the system and application, as well as specialized out-of-band hardware support that is still not yet commonplace. Automating this process is difficult, and the proprietary nature of out-of-band hardware can be limiting. This situation motivates standardization efforts, such as the Power API [26], that can provide a common interface for both application instrumentation timings and out-of-band data. The Power API [26] provides such functionality and if available for the platform it can ease the burden of developing tools for data alignment in the future.

In order to understand adjacent code regions with similar power profiles, researchers need to understand more about the code to be tested than data from out-of-band or in-band counter polling measurements alone. Application instrumentation provides an easy route for application developers and domain experts, but can be difficult for those unfamiliar with potentially large code bases. Investing the time to understand the code enables better understanding of power profiles and allows identification of different regions of code that share similar power profiles. When these regions are adjacent, they can be indistinguishable from a single region without application instrumentation. This is especially true for highly optimized code regions that operate near peak power, which can be common for many HPC bulk synchronous parallel codes. If timestamping in applications is used, times should be absolute rather than relative to allow for proper data alignment with external measurement hardware that may have timestamp drift from the local compute hardware.

To summarize, no single contemporary measurement technique is sufficient to gather power measurements for all HPC use cases. Instead, the measurement taxonomy introduced can be applied to determine the appropriate level of detail and effort, as demonstrated by our thorough investigation of multiple proxy HPC applications across multiple production platforms using both in-band and out-of-band data. In particular, we that show that the combination of application region profiling and out-of-band power measurement provides an accurate view of application power profiles with negligible overhead. Our recommendations provide actionable guidance for HPC application profiling to better understand power and energy usage on HPC systems.

Power Profiling and Energy Efficiency

Throughout the L2, we have investigated methods and functionality to control and measure the efficiency of HPC applications of interest. The APM and NRE efforts effectively implemented techniques for power control such as P-state and Power capping control at a job-level and node-level. Furthermore, both in-band and out-of-band measurement was possible, allowing not only for aggregate job data, but also fine-grained analysis of HPC codes that can be in-lined with application code regions. However, there are no silver bullets found for increasing application energy efficiency; instead there are guidelines and best practices developed that can help.

First, we've learned that power capping, or limiting node-level or job-level power can have a number of consequences. While power capping is relatively easy for users and administrators to use, we see that basic power capping is less effective than anticipated. First, power caps may actually exceed their specified cap for small periods of time, as illustrated in Figure 4.4. Second, even small power caps can have a considerable impact on performance, and not necessarily correlated to optimal energy-to-solution. Basic all-node power capping may be less effective than simple P-state controls. However, what still may be useful is dynamic power capping, whereby certain nodes can be capped at different rates compared to others. This could be used for reducing iteration idle time and inconsistencies between runs to help minimize computation and communication overlap. Effectively, we expect future research to take advantage of "dynamic power allocation" based on a per-job allocation, whereby Power caps can be adjusted on a per-node basis across a large parallel job to meet iteration barriers or communication regions.

P-state control methods, or the process by which CPU frequency and voltage are varied, can have a large impact on potential energy efficiency, however again no simple rules are illuminated. In most cases, utilizing the Turbo boost setting for an entire application run, which allows for CPU frequencies to extend beyond base designed frequencies, often comes at a energy efficiency cost, with potentially only minor improvement in FOM. Turbo boost max frequencies, which in the case of Haswell is over 1 Ghz of base max frequency, is often rarely achieved for a sustained period of time due to the thermal limits of the CPU and the parallel nature of codes effective at saturating CPU thermal design power(TDP). However, turbo P-states may be useful for small yet targeted code regions. For instance, a small serial code portion may benefit most from turbo mode, or it could be used selectively on some nodes when aligned with iterations, again to speed-up collectives or barriers.

Table 8.1 investigates in detail the utility of Turbo mode for the SPARC application. Here, we learn that the default turbo mode is only 2% or 3% faster than non-turbo yet use 11% and 3% more energy for the total run for Haswell and KNL, respectively. If we consider a future Exascale supercomputing system that is constrained by a 20MW facility power limit rather than compute resources, effectively over-provisioning of hardware, these small power savings become substantial. at 20MW, we can utilize 15% more nodes within the same power envelope. While a detailed SPARC run at this scale is currently infeasible, it is likely that performance will benefit far more form an additional 15% or 7% node count,

Table 8.1. SPARC potential energy savings of lower P-state at scale.

	Runtime	Energy	AVG Power/Node	#Nodes @ 20MW
HSW Turbo	458 s	5222854 J	357 Watts	56 K
HSW No-Turbo	468 s	4639255 J	310 Watts	65 K
% Diff	+2 %	-11 %	-13 %	+15 %
KNL Turbo	372 s	3137315 J	264 W	76 K
KNL No-Turbo	385 s	3040528 J	247 W	81 K
% Diff	+3 %	-3 %	-6 %	+7 %

rather than the small boost in performance from turbo frequencies alone due to the scalable nature of bulk synchronous parallel applications.

Given the extensive experimentation with P-states throughout the L2 milestone, we've learned that a P-state sweep, where all available P-states are evaluated, can be the most useful investigatory technique for evaluating the optimal energy-to-solution of a given application. This is because various applications can be either compute, memory, or I/O bound, or any mixture in between, and varying P-state can have a non-linear impact depending on resource bounds. For instance, codes with high levels of AVX vectorization often are most efficient running near or at the AVX frequency, which is currently at 1.9 Ghz and 1.2 Ghz for Haswell and KNL, respectively. When P-state is reduced to match AVX frequency of highly vectorized codes, there is often little performance impact yet substantial energy savings. This may not be the case, however, for large portions of serial code which may benefit more from a Turbo P-state. Only when a complete P-state sweep is done and the total aggregate energy consumed for the job is calculated can the best energy-to-solution be chosen. For the ATDM SPARC workload, we found that this is often near the AVX frequency and less than the base maximum frequency, and that applying turbo boost to SPARC results in little performance improvement with significant power cost.

Beyond just P-state sweeps of an entire application, the abilities of the techniques created by this L2 allow for further investigation of P-state within specific code regions of applications. Looking into ATDM SPARC workload's P-state sweep for the Solve region in Figure 7.6, a key observation is made that decreasing P-state for the Solve region of SPARC does has only a small impact, while total node power can be reduced by an upwards of 27%. As the Solve region within SPARC can dominate total application runtime, decreasing P-state to the a lower CPU frequency for just the Solve region could have substantial energy savings without impact on total application performance. Given the tools available as part of this APM and NRE effort and the Power API, as well as the small P-state control delay compared to the total Solve region runtime, we expect in future work to demonstrate the effect of sub-region P-state having dramatic reduction in total energy consumed without impacting performance. However, further investigation is still needed to confirm this hypothesis.

Another factor that can effect total energy efficiency of an application is the number of

threads per MPI rank used with the KNL architecture. As KNL represents a real many-core node architecture, it is often advisable to use as many OpenMP threads as possible to keep each small KNL core busy with computation. This is a key aspect in utilizing complete memory bandwidth for particular applications. However with SPARC in section 7, we see that the difference in performance between 2 and 4 threads is effectively negligible at under 1 second, however there is in fact a 12 Watts per node power saving in running only 2 OMP threads. With a SPARC run of 32 nodes, this equates to an every savings of 384 Watts, or enough energy to power a new Cray XC40 KNL compute node. Similar to the consideration at a facilities 20MW power limit, this could lead to substantially more nodes that can be allocated to a job at full scale.

Architecture - Haswell v. Knights Landing

Throughout this investigation, a number of architectural differences between Intel's Haswell and Knights Landing CPUs have been identified that would have not been possible without the APM and NRE efforts. First, we've been able to directly compare total energy usage between mini applications of both Haswell and KNL, seen in Table 5.1. Not only do we see here that KNL nodes are often (with MiniMD, MiniFE, and SPARC) offer the best total performance per node in terms of FOM, all applications evaluated (including LULESH) have found the KNL system to offer the best overall performance per Watt. For future supercomputing platforms that are expected to be primarily limited by available facility power, the KNL many-core architecture may be an ideal choice for Sandia applications.

However, analysis between architectures does not stop there. With KNL, we've also investigated how differences between HBM and DDR RAM have drastic improvements as well. While it has been well known and documented that HBM can offer large improvements in memory bandwidth, this report also confirms that such memory bandwidth improvements also translate to improved total energy-to-solution. For SPARC, observe that the total energy-to-solution for using HBM on KNL is just 3 Megajoules, compared to at best 5.1 Megajoules using DDR on KNL and just over 4 Megajoules for HSW. This leads to a 25% energy savings over Haswell. Given that the restraints in supercomputing resources at Exascale will be power-limited, using HBM with a many-core architecture like Knights Landing can allow for considerable additional hardware provisioning compared to traditional server-class CPUs and DDR memory.

In summary, we confirm that not only do we find KNL to be more performant than Haswell for most cases, we've also confirmed KNL to be more energy efficient as well for many of Sandia's benchmarks and applications of interest. This bodes well for some of the architectural changes brought by KNL, most notably including a low-power many-core architecture coupled with HBM memory availability and the ability of future architecture to drive towards Exascale computing within constrained power envelopes.

Considering running the ATDM SPARC workload at current testbed scale, KNL reduces

runtime by 19%, energy-to-solution by 40%, and individual node power by 26%. It is currently expected that Exascale demands will require strict facilities power limits that reduce resource provisioning of such supercomputers. From these findings, if a hypothetical 20MW facility were to be built today to meet the needs of SPARC workloads at extreme scale, the KNL node architecture would provide 35% more nodes than compared to the Haswell architecture.

Algorithmic and System Software Advances

Throughout the L2, a number of different applications, benchmarks, runtime configurations, and hardware platforms were all considered. While this lead to some of the findings above, there are also some other important lessons learned beyond initial P-state and Power capping measurement and controls.

First, we've found that while mini applications are useful for initial investigation of performance and energy, in fact they may not always be representative of power behavior of real applications. For instance, MiniFE's runtime is mostly a long, low-power setup and assembly phase with only a small portion of total runtime as a factor in the FOM. As seen in chapter 5, this can lead to potentially false conclusions. As a result, special region marking through the PowerAPI or Kokkos Profiling is necessary to identify only the region(s) contributing to FOM, and matching power and energy information to that, resulting in significantly more effort.

This L2 also inadvertently became more involved in a longstanding debate within the research community about the best ways in which software can help energy efficiency. This debate is as to whether improving algorithmic efficiency or improving system software will have a more drastic impact on overall power and energy for a given application. After a detailed analysis with SPARC, it becomes clear the answer is c) All of the above. Specifically, we found that power capping and P-states can improve energy efficiency and that total energy-to-solution minimization is best found at lower P-states, as seen in Figure 7.3. Furthermore, we've illustrated that careful consideration of on-node runtime parameters can also have an effect, as detailed in the previous section regarding OpenMP thread counts on KNL. However, we've also seen that improvements into the Solver on SPARC have also made drastic improvements in overall application performance, which in turn improve energy efficiency. Taken all these factors together, it is clear that a dual-edge approach of algorithmic advancements couple with system software and runtime improvements can together have a greater collective impact than either effort individually, effectively strengthening an argument for application and system software co-design at Exascale.

One of the major challenges we faced in completing the milestone was the move to the SLURM job scheduler roughly 2/3rd of the way through the year. While the move to SLRUM has generally been considered a good one, there was some impact on the power NRE efforts. This included job-wide power capping features that were being developed in conjunction

with Cray and Adaptive Computing. While developing these features were successful, the switch to SLURM meant that such extensive features were never evaluated or tested as originally desired or intended. Furthermore, some initial features, especially Cray's RUR data as generated during job runs with Moab, was subsequently lost with SLURM. As the RUR job data is the main key aspect of initial job aggregate data collection within the power profiling taxonomy in Chapter 3, missing this data with the SLURM upgrade limits researcher abilities to even conduct an initial investigation of application power profile. While this data can be easily re-created, there is still integration to be done with SLURM to do so.

Chapter 9

Conclusion

In anticipation of practical power consumption limits on extreme-scale HPC platforms, the ASC program requires guidance for power management of future platforms and applications. This becomes especially concerning as the DOE envisions supercomputing resources to top 40 MW for the first Exascale systems. The Trinity supercomputer, the ASC's ATS-1 Peta-scale leadership class system, provides the ability to initiate practices regarding HPC energy efficiency at scale today.

The Trinity program's Advanced Power Management (APM) Non-recurring Engineering (NRE) project delivers integrated power measurement and control capabilities, building on the prior work of developing the Power API. This milestone has utilized the capabilities of the NRE and APM techniques on the power usage characteristics of ASC proxy applications as well as an ASC production workload running on Trinity and its associated testbed systems. The specialized techniques developed as part of these combined efforts have lead to a number of key contributions. First, we have created a taxonomy for profiling HPC applications of interest for energy efficiency, defining a methodology for how to investigate power metrics for large-scale HPC systems. Second, we have demonstrated the ability to apply this taxonomy to analyze a number of mini applications on Trinity and testbed systems, finding ideal energy-to-solution configurations, conducting scalability studies, and characterizing power profile differences between applications. We also apply a detailed analytical evaluation of energy usage of the ATDM SPARC production workload, producing a number of insights into the application itself as well as how to best understand the application's energy efficiency. This experimentation and evaluation is distilled into a discussion of lessons learned on how to best evaluate energy efficiency of HPC systems and applications.

The methods developed in this work effectively demonstrate the ability to measure and control power at a detail and specificity for HPC systems and applications that has yet to be seen. Furthermore, this L2 milestone illustrates how the techniques of power profiling and management described herein can have direct impact on workloads today and are applied to assess the potential impact of power-constraints in future ASC platforms. Furthermore, this milestone lays groundwork for addressing the long-term goal of determining how to best use and operate future ASC platforms to achieve the greatest benefit subject to a constrained power budget.

References

[1] Sandia national laboratories advanced architecture test beds.

[2] CAPMC API documentation release 1.1, 2015.

[3] Monitoring and managing power consumption on the Cray XC system, 2015.

[4] AMD. BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h-0Fh processors, January 2013.

[5] A Anderson, R Cooper, R Neely, A Nichols, R Sharp, and B Wallin. Users manual for ALE3D — an arbitrary Lagrange/Eulerian 3D code system. Technical report, Lawrence Livermore National Laboratory, 2003.

[6] Andrew Barry. Resource utilization reporting. In *Proc. Cray Users' Group Technical Conference (CUG)*, 2013.

[7] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proc. of the IEEE Region 3 Southeast Conference 2010 (SoutheastCon)*, pages 479–484. IEEE, 2010.

[8] Ramon Bertran, Yutaka Sugawara, Hans M Jacobson, Alper Buyuktosunoglu, and Pradip Bose. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. *IBM Journal of Research and Development*, 57(1/2):4–1, 2013.

[9] William Lloyd Bircher and Lizy K John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2012.

[10] Jim Brandt, David DeBonis, Ann Gentile, Jim Lujan, Cindy Martin, Dave Martinez, Stephen Olivier, Kevin Pedretti, Narate Taerat, and Ron Velarde. Enabling advanced operational analysis through multi-subsystem data integration on trinity. *Proceedings of the Cray User Group (CUG)*, 2015.

[11] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: A framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[12] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[13] Shirley Browne, Jack Dongarra, Nathan Garner, Kevin London, and Philip Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 42–42. IEEE, 2000.

[14] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: memory power estimation and capping. In *ACM/IEEE Intl. Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[15] Jack Dongarra, Bernard Tourancheau, Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W Cameron. Energy profiling and analysis of the HPC challenge benchmarks. *The Intl. Journal of High Performance Computing Applications*, 23(3):265–276, 2009.

[16] Matthew GF Dosanjh, Ryan E Grant, Patrick G Bridges, and Ron Brightwell. Re-evaluating network onload vs. offload for the many-core era. In *Cluster Computing (CLUSTER), 2015 IEEE Intl. Conference on*, pages 342–350. IEEE, 2015.

[17] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Federico Ardanaz, Brad Geltz, Asma Al-Rawi, Fuat Keceli, and Kelly Livingston. Global extensible open power manager: A vehicle for hpc community collaboration toward co-designed energy management solutions. In *Intl. Conference on Supercomputing (ICS)*, 2017.

[18] Electronic Educational Devices. Watts up PRO, 2009.

[19] Kurt B Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2008.

[20] Rong Ge, Xizhou Feng, and Kirk W Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proc. of the 2005 ACM/IEEE Conference on Supercomputing*, page 34. IEEE Computer Society, 2005.

[21] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W Cameron. CPU miser: A performance-directed, run-time system for power-aware clusters. In *Intl. Conference on Parallel Processing (ICPP)*, pages 18–18. IEEE, 2007.

[22] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.

[23] Neha Gholkar, Frank Mueller, and Barry Rountree. Power tuning hpc jobs on power-constrained systems. In *Proc. of the 2016 Intl. Conference on Parallel Architectures and Compilation*, pages 179–191. ACM, 2016.

[24] Ryan E Grant and Ahmad Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *20th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, pages 8–pp. IEEE, 2006.

[25] Ryan E Grant, Michael Levehagen, Stephen Olivier, David DeBonis, Kevin Pedretti, and James H. Laros. Overcoming challenges in scalable power monitoring with the power api. In *Proc. 20th IEEE Intl. Parallel & Distributed Processing Symposium, Workshop on High-Performance Power-Aware Computing (HPPAC)*. IEEE, IEEE, 2016.

[26] Ryan E Grant, Michael Levenhagen, Stephen L Olivier, David DeBonis, Kevin T Pedretti, and James H Laros III. Standardizing power monitoring and control at exascale. *Computer*, 49(10):38–46, 2016.

[27] Ryan E Grant, Stephen L Olivier, James H Laros, Ron Brightwell, and Allan K Porterfield. Metrics for evaluating energy saving techniques for resilient hpc systems. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 790–797. IEEE, 2014.

[28] Taylor Groves and Ryan Grant. Power aware, dynamic provisioning of hpc networks. *Sandia National Labs report*, 21, 2015.

[29] Marcus Hahnel, Bjorn Dobel, Marcus Volp, and Hermann Hartig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[30] Alastair Hart, Harvey Richardson, Jens Doleschal, Thomas Ilsche, Mario Bielert, and Matthew Kappel. User-level power monitoring and application performance on Cray XC30 supercomputers. *Proceedings of the Cray User Group (CUG)*, 2014.

[31] K Scott Hemmert, Michael W Glass, Simon D Hammond, Rob Hoekstra, Mahesh Rajan, Shawn Dawson, Manuel Vigil, Daryl Grunau, James Lujan, David Morton, et al. Trinity: Architecture and early experience. In *Cray Users Group*, 2016.

[32] Hewlett Packard Enterprise. Redfish API implementation on iLO RESTful API for HPE iLO 4. Technical report, 2016.

[33] Micah Howard, Andrew Bradley, Steven W. Bova, James Overfelt, Ross Wagnild, Derek Dinzl, Mark Hoemmen, and Alicia Klinvex. Towards performance portability in a compressible cfd code. In *Proc. 23rd AIAA Computational Fluid Dynamics Conference*, 2017.

[34] Mingyu Hsieh, Kevin Pedretti, Jie Meng, Ayse Coskun, Michael Levenhagen, and Arun Rodrigues. SST + gem5 = a scalable simulation infrastructure for high performance computing. In *Proc. of the 5th Intl. ICST Conference on Simulation Tools and Techniques*, pages 196–201. ICST, 2012.

[35] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proc. of the 2005 ACM/IEEE Conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[36] S Huang and W Feng. Energy-efficient cluster computing via accurate workload characterization. In *Proc. of the 2009 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*, pages 68–75. IEEE Computer Society, 2009.

[37] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proc. of the 2001 Intl. Symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[38] Nandini Kappiah, Vincent W Freeh, and David K Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *2005 Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, page 33. IEEE Computer Society, 2005.

[39] Steve Kaufmann and Bill Homer. Craypat-cray x1 performance analysis tool. *Cray User Group (May 2003)*, 2003.

[40] Michael Knobloch, Maciej Foszczynski, Willi Homberg, Dirk Pleiter, and Hans Böttiger. Mapping fine-grained power measurements to HPC application runtime characteristics on IBM POWER7. *Computer Science - Research and Development*, 29(3-4):211–219, 2014.

[41] James H. Laros, Ryan E. Grant, Micheal Levenhagen, Stephen Olivier, Kevin T. Pedretti, Lee Ward, and Andrew Younge. High performance computing – power application programming interface specification version 2.0, 2017.

[42] James H. Laros, Kevin Pedretti, Ryan E. Grant, Olivier Stephen, Michael Levenhagen, David DeBonis, Scott Pakin, Steven Martin, Matthew Kappel, and Paul Falde. Aces and cray collaborate on advanced power management for trinity. In *Cray User's Group*, 2016.

[43] James H Laros, Kevin T Pedretti, Suzanne M Kelly, John P Vandyke, Kurt B Ferreira, Courtenay T Vaughan, and Mark Swan. Topics on measuring real power usage on high performance computing platforms. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[44] James H Laros, Phil Pokorny, and David DeBonis. PowerInsight - a commodity power measurement capability. In *2013 Intl. Green Computing Conference (IGCC)*, pages 1–6. IEEE, 2013.

[45] James H Laros III, , Suzanne M Kelly, Steven Hammond, Ryan Elmore, and Kristen Munch. Power/energy use cases for high performance computing. *Sandia National Laboratories, Tech. Rep. SAND2013-10789*, 2013.

[46] James H Laros III, David DeBonis, Ryan Grant, Suzanne M Kelly, Michael Levenhagen, Stephen Olivier, and Kevin Pedretti. High performance computing-power application programming interface specification version 1.0. *Sandia National Laboratories, Tech. Rep. SAND2014-17061*, 2014.

[47] James H Laros III, Kevin T Pedretti, Suzanne M Kelly, Wei Shu, and Courtenay T Vaughan. Energy based performance tuning for large scale high performance computing systems. In *Proceedings of the 2012 Symposium on High Performance Computing*, page 6. Society for Computer Simulation International, 2012.

[48] Edgar A León, Ian Karlin, and Ryan E Grant. Optimizing explicit hydrodynamics for power, energy, and performance. In *IEEE Intl. Conference on Cluster Computing (CLUSTER)*, pages 11–21. IEEE, 2015.

[49] Edgar A León, Ian Karlin, Ryan E Grant, and Matthew Dosanjh. Program optimizations: The interplay between power, performance, and energy. *Parallel Computing*, 58:56–75, 2016.

[50] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *2006 Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, pages 14–14. IEEE, 2006.

[51] S Martin and M Kappel. Cray XC30 power monitoring and management. *Proceedings of CUG*, 2014.

[52] S Martin, D Rush, and M Kappel. Cray advanced platform monitoring and control (CAPMC). *Proceedings of CUG*, 2015.

[53] SJ Martin, D Rush, and M Kappel. Cray advanced platform monitoring and control (CAPMC). In *Proc. Cray Users' Group Technical Conference (CUG)*, 2015.

[54] Abdelhafid Mazouz, Benoît Pradelle, and William Jalby. Statistical validation methodology of CPU power probes. In *European Conference on Parallel Processing*, pages 487–498. Springer, 2014.

[55] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 supercomputing sites, 2017.

[56] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proc. 1st Intl. Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[57] Bryan Mills, Taieb Znati, Rami Melhem, Kurt B Ferreira, and Ryan E Grant. Energy consumption of resilience mechanisms in large scale systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 528–535. IEEE, 2014.

[58] Millind Mittal and Robert Valentine. Performance throttling to reduce IC power consumption, February 17 1998. US Patent 5,719,800.

[59] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1, June 2015.

[60] Tapasya Patki, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *2013 Intl. Conference for High Performance Computing, Networking, Storage and Analysis*, pages 173–182. ACM, 2013.

[61] Kevin Pedretti, Stephen L Olivier, Kurt B Ferreira, Galen Shipman, and Wei Shu. Early experiences with node-level power capping on the Cray XC40 platform. In *Proc. of the 3rd Intl. Workshop on Energy Efficient Supercomputing*, page 1. ACM, 2015.

[62] Kevin Pedretti, Stephen L. Olivier, Kurt B. Ferreira, Galen Shipman, and Wei Shu. Early experiences with node-level power capping on the cray xc40 platform. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing (E2SC)*, 2015.

[63] Trevor Pering, Tom Burd, and Robert Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Power Driven Microarchitecture Workshop, attached to ISCA98*, pages 96–101, 1998.

[64] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, page 55, 2003.

[65] Steve Plimpton, Paul Crozier, and Aidan Thompson. LAMMPS - large-scale atomic/-molecular massively parallel simulator. Technical report, Sandia National Laboratories, 2007.

[66] Mohammad Rashti, Gerald Sabin, David Vansickle, and Boyana Norris. WattProf: A flexible platform for fine-grained HPC power profiling. In *2015 IEEE Intl. Conference on Cluster Computing (CLUSTER)*, pages 698–705. IEEE, 2015.

[67] J Reinders. *Vtune performance analyzer essentials: Measurement and tuning techniques for software developers*. 2005.

[68] Barry Rountree, David K Lownenthal, Bronis R de Supinski, Martin Schulz, Vincent W Freeh, and Tyler Bletsch. Adagio: making DVS practical for complex HPC applications. In *Proc. of the 23rd Intl. Conference on Supercomputing*, pages 460–469. ACM, 2009.

[69] Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis De Supinski. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *IEEE Intl. Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, 2013.

[70] Kathleen Shoga, Barry Rountree, Martin Schulz, and Jeff Shafer. Whitelisting MSRs with msr-safe, 2014.

[71] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[72] Ananta Tiwari, Michael Laurenzano, Joshua Peraza, Laura Carrington, and Allan Snavely. Green queue: Customized large-scale clock frequency scaling. In *2012 Second Intl. Conference on Cloud and Green Computing (CGC)*, pages 260–267. IEEE, 2012.

[73] Ananta Tiwari, Michael A Laurenzano, Laura Carrington, and Allan Snavely. Modeling power and energy usage of HPC kernels. In *IEEE 26th Intl. Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 990–998. IEEE, 2012.

[74] Christian Robert Trott, Harold C Edwards, Nathan David Ellingwood, and Simon David Hammond. Kokkos - portability performance productivity. Technical report, Sandia National Laboratories, USA, 2016.

[75] Adrian Richard White. Methods and apparatus for diagnosing and correcting faults in computers by a support agent at a remote location, April 2 2002. US Patent 6,367,035.

[76] Michal Witkowski, Ariel Oleksiak, Tomasz Piontek, and J Weglarz. Practical power consumption estimation for real life HPC applications. *Future Generation Computer Systems*, 29(1):208–217, 2013.

[77] Huazhe Zhang and H Hoffman. A quantitative evaluation of the RAPL power control system. *Feedback Computing*, 2015.

DISTRIBUTION:

	MS	,
		,
1	MS 0899	Technical Library, 9536 (electronic copy)


Sandia National Laboratories