APPLICATION-LEVEL MODELING AND ANALYSIS OF TIME AND ENERGY FOR OPTIMIZING POWER-CONSTRAINED EXTREME-SCALE APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Eric Anger

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2016

APPLICATION-LEVEL MODELING AND ANALYSIS OF TIME AND ENERGY FOR OPTIMIZING POWER-CONSTRAINED EXTREME-SCALE APPLICATIONS

Approved by:

Dr. Sudhakar Yalamanchili, Advisor
Professor, School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ada Gavrilovska
Professor, School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
Professor, School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Linda M. Wills
Professor, School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. George F. Riley
Professor, School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: December 2016

ACKNOWLEDGMENTS

This thesis would not be possible without the concerted effort of many people. Foremost is my adviser Dr. Sudhakar Yalamanchili, who took me under his wing and taught me not only the technical aspects of my work, but also how to conduct myself in formal research and with my colleagues. Thank you for guiding me on how to be a better researcher. I would also like to thank Dr. George Riley and Dr. Rich Vuduc for their feedback and insight as my reading committee members, as well as Dr. Ada Gavrilovska and Dr. Linda Wills for their input on my thesis and during the defense.

Through the years I have had the great fortune to work with individuals not associated with the university. I would like to thank Dr. Curtis Janssen and Dr. Gilbert Hendry for mentoring me at Sandia National Labs in California, and especially Dr. Jeremiah Wilke for his advice and leadership for the many research projects I undertook in conjunction with him. I am grateful as well for the mentorship of Dr. Kevin Barker and Dr. Darren Kerbyson at Pacific Northwest National Lab and Patrick McCormick and Dr. Scott Pakin at Los Alamos National Lab. These internships were invaluable in exposing me to the outside world of academic research.

Many, many thanks to my friends in the CASL lab; Andrew Kerr, Jeffrey Young, Chad Kersey, Will Song, Si Li, Minhaj Hassan, Jin Wang, Haicheng Wu, Hugh Xiao, Xinwei Chen, Karthik Rao, and all the rest were there when I needed information, advice, commiseration, sympathy, motivation, and cheer. I could not have gotten through it without you all.

Last, I would like to thank my close friends and family. You are all my rock, giving me nothing but encouragement and support through this process. This is as much your creation as mine.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of the proposed research is to create a methodology for the modeling and characterization of extreme-scale applications operating within power limitations in order to guide optimization. While extreme scale systems get larger, the amount of power they may draw at one time is changing very little. This places immense pressure on ensuring that the design is energy-efficient. To combat this pressure, the hardware is being designed to expose unprecedented degrees of parallelism and complexity, necessitating new programming models. However, efficient hardware is not enough to reach exascale performance goals; it is necessary to understand the energy behaviors of applications to guide their development. Optimizing extreme-scale applications to operate within power limitations will require new techniques for understanding the relationships between application characterization, performance, and energy. The main contributions of this work are:

- A method for modeling execution time and energy for regions within applications.
- Models of energy scaling and the trade-off between performance and power.
- Power-constrained performance scheduling of synchronous and asynchronous applications.

The foundation for addressing these problems is the modeling of time and energy consumption of applications. The different forms these models take provide the necessary insight to approach these challenges. The first challenge is addressed with the development of automated techniques for the construction of fast models of time and energy parameterized by application-level information. These models are used to drive macro-scale simulations, giving developers feedback on the performance of their applications on future large-scale systems. The second challenge is addressed by improving existing analytical energy scaling models to show how time and energy are related as the system scales. This leads into the use of Pareto frontiers to describe a reduced set of configurations that achieve the highest performance for a given power. The third challenge is addressed with a set of runtime

heuristics, guided by performance and power models, for driving the configuration selection of tasks in both bulk-synchronous parallel and asynchronous, task-based programming models. Collectively, these techniques provide the necessary insight into the energy consumption behavior of applications to ensure that they are designed in a way that achieves the highest performance when constrained by power limitations.

This thesis illustrates how the appropriate application of models for energy can help ensure the energy-efficient design of next-generation extreme scale systems.

CHAPTER 1

INTRODUCTION

High-performance computing is trending towards extreme scale where the quantity of data operated on is growing dramatically [11]. In order to reach exascale—10^{18} operations per second—it is projected that machines will need millions of execution cores. It is likely that these new machines will operate with stringent power caps [80] where the amount of power the systems may draw at one time is fixed. Likely this will manifest as an overprovisioning of compute resources (e.g., processor cores, memory channels, clock frequency states) that cannot be used concurrently [31]. This imposes a new challenge for balancing performance and energy consumption in the design of these next-generation systems as the performance of the systems is tied to their energy-efficiency. These new architectures require that applications be written with new programming models to ensure efficiency gains, particularly as the development of future exascale systems is driven in large part by the rapid feedback cycle termed *codesign* [75]. In the codesign regime the hardware and software communities work together, communicating design trade-offs across the entire system stack. It is founded upon effective methods for describing the requirements and effects of design decisions as they pertain to the performance of the system. As a result, it is necessary to understand the energy behaviors of applications and the trade-offs imposed by future systems. This is synthesized in the following problem statement: *performance optimization of extreme-scale systems that operate within power limitations require new techniques for understanding the relationships between platform characterization, performance, and energy consumption as a function of the applications being executed.*

Existing techniques at the architecture level to address this problem have focused on hardware design, presenting mechanisms for observing their energy behaviors and tuning their architectures to provide the highest performance per watt. However, these techniques are insufficient to address the rigorous energy-efficiency demands of exascale systems. The

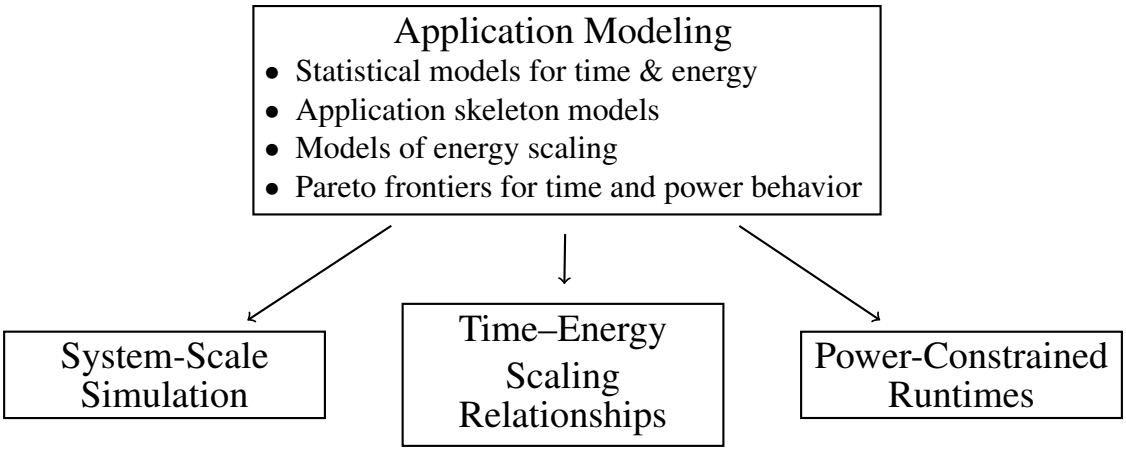


Figure 1: Application-level models provide a foundation for the three challenges addressed in this thesis.

increased synergism between the hardware and software created through the codesign process necessitates explicit consideration of the specific applications used when designing energy-efficient systems. Writing energy-efficient applications is itself a challenge, especially as the industry shifts to more dynamic programming models such as asynchronous task-based models. In order to fully address the problem of energy-efficient exascale systems, there needs to be a methodology for characterizing the relationships between energy and performance of the application's component parts, as well as how to quickly iterate over application designs to converge on a design that fits within the power constraints for a target system. This characterization is necessary to guide the developer to ensure energy efficient applications.

To that end, this dissertation presents a methodology for modeling and characterizing applications in order to both steer the design of energy-efficient applications as well as improve the energy consumption and performance of applications at runtime. At a high level, this thesis addresses the following challenges:

1. How can system-level models for the execution time and energy behavior of applications be developed?

2. What is the relationship between time and energy scaling and how does configuration (i.e., number of cores, clock frequency) affect performance and power scaling behaviors?
3. How can these models of time and energy drive the decision-making process during execution to speed up applications under a power cap?

The relationships between the contributions of this work are shown in Figure 1. The foundation for addressing these problems is the modeling of time and energy consumption of applications. The different forms these models take provide the necessary insight to approach these challenges. The first challenge is addressed with the development of automated techniques for the construction of fast models of time and energy parameterized by application-level information. These models are used to drive macro-scale simulations, giving developers feedback on the performance of their applications on future large-scale systems. The second challenge is addressed by improving existing analytical energy scaling models to show how time and energy are related as the system scales. This leads into the use of Pareto frontiers to describe a reduced set of configurations that achieve the highest performance for a given power. The third challenge is addressed with a set of runtime heuristics, guided by performance and power models, for driving the configuration selection of tasks in both bulk-synchronous parallel and asynchronous, task-based programming models. Collectively, these techniques provide the necessary insight into the energy consumption behavior of applications to ensure that they are designed in a way that achieves the highest performance when constrained by power limitations.

The next section describes the key contributions of this work, followed by a summary of the structure of this document. The key contributions have also been presented in several publications [53, 15, 9, 10, 8].

1.1 Contributions

This thesis makes the following major contributions.

1. **Modeling of execution time and energy for regions within applications**

   The first contribution of this thesis approaches the problem of creating time and energy models of scientific applications that can scale to a large number of nodes. Traditional high-accuracy modeling techniques such as cycle-level simulation can take hours to days of time to run. However, moving towards exascale will require system-level simulations which is only achievable with composable models that can be evaluated quickly. This thesis addresses this concern with a method for constructing models of applications based on profiling data. In leveraging profiling data, this technique generates statistical models for behaviors of applications as a function of the program parameters. The model generation procedure is customizable, allowing for domain-specific knowledge about the behavior of the application or the architecture to inform the modeling process. These models are much faster to evaluate than cycle-level simulations, providing a mechanism to replace regions of computations within large-scale applications for fast large-scale simulation. *Application skeletons*, stripped-down versions of applications originally used for network studies [88], are used as the vehicle for these simulations. By replacing the time-intensive computation within these skeletons with analytic models of time and energy behaviors, the accuracy of the simulation is maintained while reducing simulation execution time.

   In order to evaluate this modeling methodology, this work contributes *Eiger*, a framework for the creation of statistical models of execution time and energy for regions of computation [53]. These models represent execution behavior based on empirical data, parameterized by source code-level information. The Eiger framework has three main components: reading in instrumentation data, iterative model generation, and output of the model for use in large scale simulation. The first step is achieved with the Lightweight Performance Data Collectors (`lwperf`) tool to facilitate the collection of performance data. This tool is robust in the instrumentation

sources it can draw from; it has been used with real hardware, simulators, and emulators. This data is then fed into Eiger for model generation. This model generation procedure is semi-automated, learning an appropriate model from the input data, but able to leverage domain-specific information. It uses stepwise regression to adaptively add terms to the model so long as they improve the model's fit. As well, the instrumentation data can be passed on to the R programming language [68] for more complex model generation. Once the model has been constructed, it is output into a structured format for use in external tools. The use of these models have been shown to accelerate simulation of high-performance applications on both CPUs and GPUs with minimal error, when predicting both execution time and energy consumption [9].

2. **Models of energy scaling and the trade-off between performance and power**

   While the prior work demonstrated the ability to estimate energy consumption for regions of applications, it does not address how energy consumption scales with parallelism. In the pursuit of high performance, computer systems are being developed with an increasing degree of parallelism. While time behavior under parallelization is well studied, energy behavior has received significantly less attention. While existing analytical models of energy scaling have been presented [90], they have been used almost exclusively as high level estimates with little corroborating measurement data.

   This thesis presents the key observation that although execution time can be overlapped in time through the use of parallelization, energy cannot; energy is the cumulative cost for all the *work* performed, regardless of whether that work happens sequentially or in parallel. Total energy consumption does have a time-dependent portion, relating to the time a system has to be powered on. In general, total energy consumption is decomposed into two parts: the *idle energy*, which is the energy cost

associated with running an inactive system, and the *dynamic energy*, which is all energy that is consumed by the application performing computation. Breaking down total energy into its two parts forms the basis for the energy scaling model. For this model, idle energy scales proportionally to time speedup, where as the work (dynamic energy) remains constant. This thesis shows how characterizing the platform and the application are necessary to determine the ratio of idle to dynamic energy. Experimental data on modern shared memory multicore architectures demonstrates how this model accurately represents the energy scaling behavior of an application as a function of time scaling behavior.

The selection of degree of parallelism is just one way in which the configuration of an application affects performance and energy behaviors. Other features, such as dynamic voltage and frequency scaling (DVFS), can also be used to present different trade-offs between performance and power for a target platform. However, the combinatorial explosion in the number of configurations makes selecting the appropriate configuration challenging. This thesis proposes reducing the state space with the use of *Pareto frontiers* to identify those configurations that achieve the highest performance for a given amount of power. Importantly, these frontiers are dependent on the application and platform. This thesis proposes the identification of Pareto-optimal configurations to describe the trade-offs between performance and power. These frontiers are used as the model for time and power behaviors as a function of configuration. An experimental analysis of Pareto frontiers is performed on a set of tasks taken from HPC applications run on a large shared memory multicore system.

3. **Power-constrained performance scheduling of synchronous and asynchronous applications**

   The prediction of application execution time and energy behaviors can also be

leveraged to make decisions at runtime for systems constrained to operating under a power cap. This thesis presents two complimentary techniques for increasing performance of applications both across nodes and within a node. First, this thesis presents *Dynamic Power Steering*, a method for optimizing execution of load-imbalanced bulk-synchronous applications in power-limited environments [15]. In the bulk-synchronous programming model, computation is periodically synchronized at global barriers. However, there may be imbalance in the amount of work assigned to the nodes in the system, which may change over time. Because of this, some nodes may finish their work and reach the barrier before others. This idle time is called *slack* [50]. The Dynamic Power Steering approach uses DVFS to slow down nodes that would reach the barrier too quickly and speeds up nodes that are the last to reach the barrier, so long as the system remains under the power cap. This technique uses system-level models of node power and performance to direct the application of the available power. This work explores the effect of Dynamic Power Steering for three synthetic workloads that represent typical workload imbalance in BSP applications. Experimental results are shown for a power-instrumented cluster.

The second technique presented in this thesis addresses the power-constrained scheduling of applications on shared memory, high core count multicore processors. An important programming model is the task graph-based application, which mixes coarse-grained, asynchronous tasks with fine-grained intra-task data parallelism. A major challenge with this application model is dynamically selecting which tasks to schedule and how many cores to assign to each task. This decision is further complicated when a power cap is imposed, since there is a non-trivial relationship between number of threads and power consumption.

This thesis contributes a set of scheduling heuristics that determine which of the available tasks should run as well as how to configure their execution. In order to make this decision, the heuristics leverage the previously described Pareto-optimal

configuration information to evaluate the trade-offs between performance and power consumption. This work demonstrates the benefits of power-aware scheduling over baseline static techniques across a wide range of power caps on a high core count, shared memory system.

1.2 Organization

This dissertation is organized as follows.

Chapter 2 describes the landscape for application modeling and power-constrained optimization. It contains an overview of techniques used to model the behavior of applications, both as it pertains to simulation at scale as well as modeling energy consumption. In addition, this chapter includes a description of analytical modeling techniques for applications as they scale, and the major existing methods for optimizing the performance of both bulk-synchronous parallel and asynchronous task-based programming models. Additional related work is included toward the beginning of each chapter as appropriate.

Chapter 3 presents the method for constructing models of time and energy behaviors of applications as a function of program parameters. These techniques, encapsulated in the Eiger and `lwperf` tools, are used to construct statistical models of execution time and energy from instrumentation data. These models replace regions of computation within *Application skeletons*, the vehicles for macro-scale simulations, which are shown to increase the accuracy of system-level energy and performance models while reducing simulator execution time.

Chapter 4 examines the creation of energy scaling models and formulating the trade-offs between performance and power as Pareto frontiers. These models describe the increasing energy cost for executing applications in parallel as a function of the execution time speedup as well as a characterization of application and platform. This chapter then describes some methods for improving the fidelity of the energy scaling model, as well as an evaluation of the model for HPC workloads on a server-grade system. Additionally, this

chapter presents the characterization of system configurations into Pareto-optimal points in the performance–power space. By providing this formulation, Pareto frontiers represent the configurations of the application, such as the degree of parallelism or DVFS state, as a trade-off between performance and power.

Chapter 5 builds on the energy and performance modeling work to provide heuristics for improving performance of applications when power constrained. This begins with a description of the Dynamic Power Steering technique for bulk-synchronous applications, with experimental evaluation on a test bed cluster system. This chapter then expands on this methodology for asynchronous task-based application scheduling. Several heuristics are presented which leverage the Pareto frontier formulation of power–performance trade-offs to increase performance while remaining under a power cap. Several task graph-based HPC applications are demonstrated to achieve greater performance while power constrained compared to baseline static limiting techniques.

Finally, Chapter 6 summarizes the contributions and conclusions from this thesis as well as contemplating future areas of research.

CHAPTER 2

RELATED WORK

This chapter documents the important research that frames the work in this thesis. It is broken down into several parts, broadly broken into the three main themes from this dissertation: performance and energy estimation applications, modeling of time and energy, and methods for optimizing performance of parallel applications in the face of power limitations. These prior works lay out the framework for the thesis, outlining current trends in application modeling. The sections below will walk through the relationships between existing works, as well as indicating the gaps in those works, questions left open, and how the goals of those works are different from those presented in the subsequent chapters of this thesis.

2.1 Performance and Energy Estimation

In high performance computing, a wide range of performance prediction techniques are used depending on the quality of estimate desired, as shown in Figure 2. Simulators range from high-fidelity and computationally expensive simulators for measuring performance between two nodes [85] to lower-fidelity and lower-cost simulators that can estimate performance on large-scale machines[88]. Often emphasis is on MPI emulation or simulation, with prominent simulators including BigSim [93] and SIMGRID [20] in addition to SST/macro [71]. There are notable examples of simulators that fit application runtime to analytical models [40]. The following sections describe key efforts in the literature that address energy and performance modeling across the fidelity–cost spectrum.

2.1.1 Large-scale Simulation Techniques

Three well-known approaches have been investigated for estimating performance at scale. The most common approach is direct execution of the full application on the target system [66, 70]. This simulation approach uses virtual time unlike normal benchmarking that

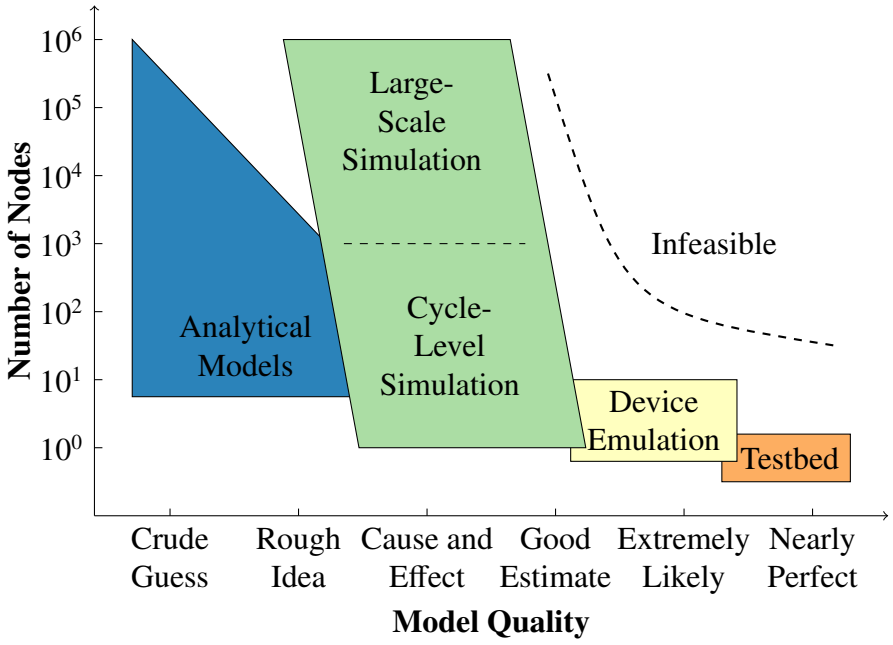


Figure 2: Spectrum of different modeling granularity and the associated quality.

uses real time. Another approach requires tracing the program in order to collect information about how it communicates and executes [93]. The resulting trace file contains computation time and actual network traffic. Still, tracing does not scale to a different number of processors or new problem sizes. A third approach is to implement a model skeleton program as a simple, curtailed version of the full application but complete enough to simulate realistic activity [5]. This approach has the advantage that the bulk of the complex computation can be replaced by simple timing information. The skeleton application provides a powerful method for evaluating the scalability and efficiency over various architectures of moderate or extreme scales.

Some approaches investigate automatically synthesizing skeletons from communication traces [4], but this requires extensive trace collection and may not capture behavior produced with extrapolated application parameters outside the calibration range. Static analysis techniques have been used to identify computations that have no impact on control flow or communication and replace them with symbolic estimates for time [5]. Automated methods have been demonstrated for transforming scientific codes through the use

of both static analysis and runtime information in the form of MPI communication patterns [67, 77].

The use of application kernels in performance analysis is well established. Two widely-used collections of kernels are the NAS Parallel Benchmarks (NPB) [12] and the PARKBENCH suite [30]. The kernels in these suites represent common computational patterns that are found in many full-scale applications. While these suites provide simpler implementations of important algorithms than full scientific codes, they represent generic algorithms that lack any nuances that would be found in specific application implementations.

2.1.2 Performance Prediction Using Models

While simulation can give accurate predictions of execution behavior, it is a tool used late in the design stage. Instead, approximate behavior is encapsulated into models that provide a reasonable estimate while remaining intuitive and fast to evaluate. The most general of these models, known as *analytical models*, provide a mathematical expression to describe performance. These models have been used to predict network performance [25] and to describe how Amdahl's Law can be extended to energy in multicore processors [90]. Another example is the roofline model which relates the computational intensity of a workload—the ratio of computation to memory demand—to performance [89] and energy [23]. The roofline model focuses only on the characterization of the application under the fixed behavior of the platform, under the goal of energy optimization. A more nuanced approach is to examine the energy behavior of individual instructions [76]; this modeling technique allows for individual characterization of the hardware and software, which can be combined orthogonally.

In addition to predicting execution behavior, models have been used to quickly search large design spaces. Jia et al. [46] present such a technique which simulates a random subset of GPU designs from a very large design space then applies a stepwise regression modeling algorithm to construct a performance estimator. These models tend to be powerful but hard to introspect. Cook et al. [24] describe a Monte Carlo method for design

space exploration and performance prediction using sampling and statistical methods. Automated regression analysis [33] constructs an analytic model determined from a set of training samples. In order to manage model complexity and optimize training error rate, a forward-stepwise procedure is used to aggregate basis functions based upon adjusted coefficient of determination, a modification of the coefficient of determination that adjusts for the number of terms in the model [55].

2.1.3 Statistical Modeling of Performance

Various techniques from the field of machine learning have been used to predict the performance of applications, rather than to select an optimal design from a large design space. Statistical techniques are often used to characterize the behavior of applications; Genbrugge et al. [35] describe a method for constructing a synthetic trace of a program execution bearing the same statistical properties as a complete execution but of much shorter overall length. These techniques provide methods for reducing design space, but struggle to disclose why one particular design may be better than others, providing little insight to the application developers.

Applications have been characterized using principal component analysis [42] and hierarchical clustering [37] in order to describe the differences between applications in benchmark suites as well as to indicate how applications stress different parts of the hardware pipeline. One technique used to predict execution time, power, and energy consumption is Artificial Neural Networks (ANNs) [43, 83]. While this method provides high quality predictions, it obscures the meaning and interpretation of the model while prohibiting the designers from informing the modeling process. Some works [81] use linear regression models to predict performance and energy for the entire execution of the application. Hong et al. [41] propose a predictive analytical performance model for GPUs along these lines.

In addition to performance prediction, other techniques explore how to estimate the performance of applications by collecting data from existing architectures to predict for previously unseen architectures. Kerr et al. [54] explore an approach based on Principal

Component Analysis [47] to automatically select the correct metrics to model GPU workloads executing on either GPUs or CPUs. This method provides a good framework for understanding the execution behavior of kernels of execution, setting the stage for adding statistical models to skeleton applications presented in this work.

2.1.4 Power Measurement Facilities

The previous subsections describe estimation techniques that rely on accurate input data. While measuring performance for these systems is typically straight forward, the ability to measure power draw has only been recently added as a feature. CPU-specific methods have lead the way, such as the model-specific registers provided by recent Intel parts through the Running Average Power Limit (RAPL) infrastructure [1] and corresponding AMD tools [6]. Accelerators are often the goal of power-measurement, with tools like NVIDIA's NVML library for accessing GPU power information [63], or Intel's `libmicmgmt` library for accessing information about the Xeon Phi coprocessor cards [2]. Cray has provided an system for monitoring and measuring the power consumption of their clusters [59]. Other methods for energy measurements leverage external meters [34]. These measurement tools have been aggregated by wrapping libraries such as PAPI [61] and the PowerAPI specification [18].

2.2 Analytical Models of Time and Energy Scaling

The majority of scalability work has focused on understanding performance behaviors, both in predicting scaling behavior as well as determining performance bottlenecks. The energy scaling model has been explored from the perspective of clock frequency [22], giving a measure for the relationship between scaling and energy. This has been extrapolated to an optimization function for selecting configurations given die area constraints [21]. The Amdahl's Law-style model for performance and power have motivated exploring energy scaling on multicore processors, concluding that with current trends it is untenable to keep all devices powered up at the same time, known as *dark silicon* [31]. Rather than purely

analytical models, several simulation frameworks have been used to predict both performance and energy [78, 82] under scaling. This includes work that adds models of energy consumed during regions of computation to large-scale simulations [9].

2.3 Optimizing Performance Under Power Limitations

Going forward, not all computing resources will be active at any one time as insufficient power will be available. In the HPC space, power constraints limit the extent to which large-scale systems can grow. To combat this, several techniques have been developed to increase energy efficiency, resulting in better performance. Recent work includes designing energy efficient runtimes for hybrid programming models [57], and graph algorithms [73]. The Adagio runtime system predicts execution 'slack' to provide energy optimization for applications [72]. However, the approach assumes Bulk Synchronous Parallelism (BSP) in which the model of execution consists of multiple steps each containing a compute followed by communication that is often global [36]. This does not provide the best energy efficiency for applications with complex and time- dependent processing patterns including those in wavefront applications that have benefited from the Energy Template approach [52].

The work by Etinski et al. [32] optimizes job scheduling to make best use of an available power budget. A framework that exploits barriers for energy efficiency has also been explored [58]. Green building blocks and methodologies for hybrid execution for providing energy efficiency in the runtime system was proposed in [56]. Several techniques that use Dynamic Voltage Scaling, including a just-in-time method [50], enable a processor-core to slow down if the assigned computation is lower than on others. Methods for designing energy efficient collective communication primitives using MPI have been studied for a range of applications [49].

Tackling the problem of power limits for HPC systems has presented many techniques for increased energy efficiency. Understanding when imbalance occurs in an application in order to reduce energy when executing some applications under a power cap has been

demonstrated [15]. These works apply to only bulk-synchronous programming models, whereas the work presented in this thesis is more general, presenting techniques that work for both bulk-synchronous as well as the more complex asynchronous task graphs.

Zhang [92] provides a survey of methods for enforcing a power limit, both by the hardware and software. However, the surveyed techniques are viewed from the perspective of administration, ensuring that limits are met within a timely manner. Methods for increasing energy efficiency by adjusting the configuration either of the number of threads given to each task [79] or the configuration of the hardware, tweaking either the frequency states or number of available cores for use [13] have been demonstrated. Additionally, work by Patki et al. [64] shows the effect of power-aware job scheduling depending on user behavior. While not focused on executing task graphs, that work provides a foundation for applying power limitations to entire clusters, ensuring fast task turnaround time based upon user requirements.

2.4 Power-Aware Parallelism for Task-Based Applications

The problem of effectively scheduling task graphs has been shown to be difficult, particularly when additional constraints, such as limited power or number of cores, need to be considered. The work by Buttari et al. [19] describes the formulation of linear algebra algorithms including Cholesky and LU decompositions into tasks and describing their performance improvements over more traditional algorithms. The DAGuE runtime [17] provides a programming model and framework for the execution of task graphs for the HPC domain, focusing on scale and speed of execution for large problem sizes, and has been used to implement the DPLASMA library of dense linear algebra functions for distributed systems [16]. This formulation of asynchronous DAGs has been expanded by Wu [91] to add an additional layer of hierarchy, distinguishing between the coarse-grained task parallelism and the fine-grained data parallelism within a task which has been termed *elastic* or *parallel* tasks [74, 86]. To address the scheduling of graphs when constrained by the

number of cores available, Barbosa el al. [14] present a static method and Vydyanathan et al. [87] present a dynamic method for assigning cores to tasks to minimize execution time. These works use analytical performance models based on the task workload. In contrast, this thesis focuses on the empirical behavior of tasks to classify their execution.

Recent work has explored measuring the power behavior of task-based applications [60], paving the way for appropriately scheduling individual tasks so as to optimize performance under power caps [13]. One large focus of work in this area is to understand the way parallelism affects time and energy. Cho et al. [22] examine the theoretical relationship between these three built off of Amdahl's Law-style analysis. The technique proposed by Curtis-Maury et al. [26] leverages machine learning to select the degree of parallelism that optimizes performance and power consumption. Work by Vydyanathan et al. [87] expands upon this notion, providing a method for optimizing the runtime assignment of parallelism either within a task or across tasks.

Research has been shown to estimate the performance, power, or energy behavior of application tasks as a function of frequency state [27]. The models in Chapter 5 leverage Pareto frontiers to describe the power and performance for each of the tasks. Pareto-optimal configurations have been used to characterize system designs across multiple cores [31].

2.5 Concluding Remarks

The evolving landscape of HPC application and system development has created a large corpus of work addressing the issues of performance and energy modeling, operating under a power cap, and scheduling task graphs. Moving forward into the exascale regime, the demands for understanding the behaviors of applications, particularly as they scale, will become necessary. While existing work has isolated key issues within these domains, this thesis focuses on treating energy consumption as a first-class resource, which has so far been typically considered only secondarily. By providing an understanding of the energy behaviors of applications, new models can be constructed to help estimate behaviors at

scale, as described in Chapter 3, easily describe the trade offs between performance and energy efficiency, as in Chapter 4, and provide a mechanism for making runtime decisions to increase performance under a power cap, discussed in Chapter 5.

CHAPTER 3

APPLICATION-LEVEL MODELING OF TIME AND ENERGY

Both the scientific and enterprise computing domains are seeing the rise of extreme scale systems to handle explosive growth in problem sizes. These systems will need to be designed without the luxury of prototyping to fit within the tightening constraints of power, heat dissipation, memory bandwidth, and processor speed as discussed in Chapter 2. Hardware–software codesign will play a key role in their development. The power constraints of future systems will place greater emphasis on quickly evaluating the energy consumption of applications in addition to their execution time.

Developing accurate models of the time and energy consumption of applications is a complex task [11]. Cycle-level hardware simulation is infeasible at scale motivating the need for more abstract models of the *effects* of computation on the system rather than the *mechanism*. A key contribution of this work is the use of statistical modeling to learn the relationships between application-level parameters and the time and energy behaviors on the target platform through analysis of instrumentation data taken from the application running on real hardware. This modeling technique can flexibly handle model construction for complex architectures and applications while reducing domain expertise required by users building the application skeletons. This method is used across execution models and language selection to predict both execution time and energy. This work presents the Eiger Statistical Modeling Framework [53] as a tool for reading instrumentation data, iteratively constructing models, and emitting them for use.

This chapter proposes a large-scale modeling methodology based on the notion of *application skeletons*. These are application implementations where regions of code (such as compute-intensive loop nests) can be replaced by analytic models of their physical properties (e.g., execution time and energy). A second contribution of this work is a method for the creation of these skeletons, collecting instrumentation data, and feeding it into the

Eiger framework to generate models. After the models are generated, calls are inserted to replace the computation with queries to the models. This instrumentation functionality is encapsulated in the `lwperf` instrumentation infrastructure.

These skeletons form the vehicle for *macro-scale simulation*, where candidate applications and hardware configurations can be simulated at the scale of hundreds or thousands of cores to understand the impact of system design at scale. Inter-node communication is modeled as flows on the network, which is analyzed for congestion behavior and its impact on system performance. Such simulations have demonstrated high scalability [88], capable of reaching tens of thousands of nodes. However to ensure high-fidelity simulations, these abstracted communication models must be coupled with fast, accurate end-point computation models. This work contributes an analysis of the use of Eiger models in application skeletons for macro-scale simulation. Included is a demonstration of the increased accuracy of the application skeleton, as well as the trade-offs in simulator overhead.

The main contributions described in this chapter are:

- The generation and use of statistical models to predict how different input parameters translate into execution time and energy for regions of computation.
- A technique for converting an application into a skeleton.
- The application of computation models in application skeletons to permitting high-fidelity simulation at scale.

3.1 Statistical Models of Application Execution Time and Energy

This section describes an automated statistical approach for modeling program behaviors on diverse architectures. The objective is to *design and implement a methodology* for discovering and synthesizing analytic performance models of the execution time and energy consumption of applications running on target heterogeneous processors.

This approach discovers analytic relationships between the static and dynamic parameters of an application and performance metrics. For example, one may wish to capture

the impact of the sparsity of input data structures, dynamic execution count, and number of function calls on the execution time. Or one may wish to discover a relationship between the number of double precision load operations, number of DMA calls, and occurrence of unconditional branches on energy consumption. These are usually complex relationships that elude manual discovery or effective application of off-the-shelf models. This complexity is magnified in modern and emerging heterogeneous processors. Broadly, this methodology is comprised of 1) experimental data acquisition and database construction, 2) data analysis passes over the database (possibly creating new higher order data), and 3) model selection and construction. The analysis passes can utilize existing data analytics techniques. The last phase automates the creation of model files for use with external tools.

Eiger is a methodology for constructing statistical models from instrumentation data. Eiger provides a standardized infrastructure and API for adding instrumentation to applications, a relational data store for collecting and managing that data, and a workflow for the creation and analysis of models. The original version of the Eiger tool was used to estimate performance across GPU models [54]. The main goal of this second generation of the Eiger project is to facilitate intelligent, semi-automated model construction without requiring expert knowledge of hardware or software. Instead of meticulous domain analysis, the models are learned from the measurement data. The infrastructure is designed for flexibility in adding new parameters controlling model construction and trading accuracy for performance.

The current implementation achieves these goals via statistical methods for dimensionality reduction and automated model selection. Eiger constructs coarse-grain predictive models when trained with results of similar applications on similar machines. The regression models may be as simple or complex as desired using metrics ranging from fine grained counts of instruction distributions to coarse grain estimates of computation working set size. The infrastructure is easily extensible to new sources of measurement data and supports the incremental addition of new experimental data. Its modular structure supports

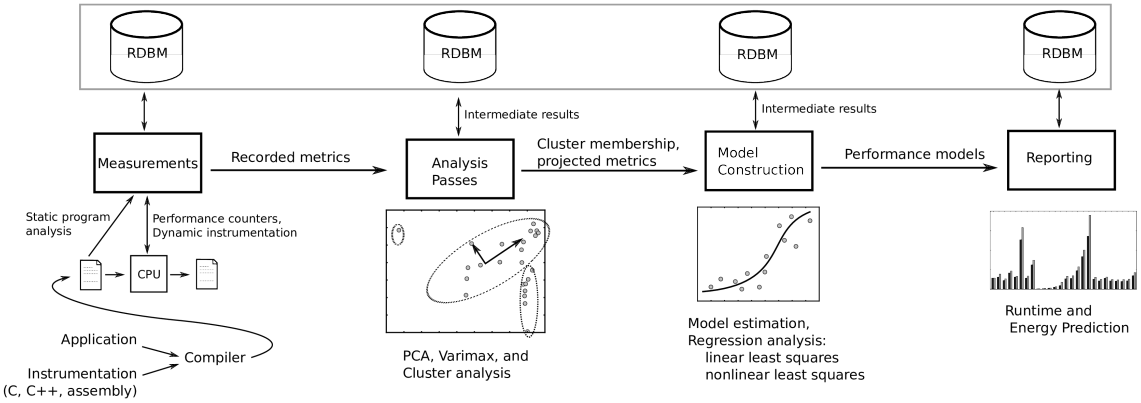


Figure 3: Implementation details of Eiger Statistical Model Creation framework.

the easy addition of new analysis and model construction passes. Consequently, the infrastructure will benefit from other explorations in the community by lowering the barriers to entry in performance model generation or synthesis.

A detailed illustration of the Eiger framework is provided in Figure 3. The framework encompasses an automated process in which application profiling data is collected via a standardized interface and ultimately used to construct a model of execution time, energy, or any other dependent result metric. The resulting statistical model may be then composed with other tools and applications such as simulation environments, heterogeneity-aware process schedulers, and reporting tools. The following are the key components of this infrastructure.

1. **Measurements**

   Eiger uses a relational database to manage the storage of all data accumulated during profiling runs as it allows asynchronous insertions while allowing for rigorous relational specifications. Additionally, Eiger provides support for the scenario where execution runs of multiple tools are required to construct a single data point (trial). This data can be drawn from many sources, including real hardware, simulators, and emulators. This data is internally stored within the Eiger framework for the subsequent analysis and construction passes.

2. **Analysis**

After the measurement process, the data is fed through a flexible series of analysis passes to prepare it for model generation. An analysis may in turn generate new data that is also stored within the Eiger framework to be the candidate for future analyses or model construction passes. One example of an analysis pass is Principal Component Analysis (PCA) which is a well-known dimensionality reduction technique. Benefits of reducing the dimensionality of the input dataset are manifold; it speeds up the model generation process, improves the clarity of the resulting model, and allows for intuition into the correlations between input metrics. A more in-depth examination of the entire modeling process is described below.

It is important to note that PCA is an *unsupervised* learning technique in that it does not take the performance metric into account when choosing dimensions to eliminate. It is entirely possible that a dimension with low variance may have a larger affect on application performance than one with high variance. For example, number of cores may not have as large a variance as memory bandwidth for a range of machines but a greater impact on runtime for compute-bounded applications. PCA does not explicitly specify how many dimensions to retain; rather it relies upon the user to make the final decision.

3. **Model Construction**

   After initial analyses are performed, the data is ready to move into the model construction pass. This pass provides a built-in method for constructing statistical models to show how a general technique can be used to generate models of application execution time and energy. This modeling process is outlined in the following subsection.

   As well, an interface to the R programming language [68] is available, allowing for data to move back and forth between the two environments. R is a powerful language for performing sophisticated data analysis and modeling; making this interface available enables more modeling techniques from the domain of machine learning

and data science to be applied to execution data.

4. **Reporting**

   Completed models consist of a set of transformation matrices from dimensionality reduction and cluster analysis as well as a vector of functions and their associated weights. Reporting passes over this data will format the information in a method easily consumed by the user, including plotting and statistical results. This phase also allows for the serialization and memoization of the finished models for later consumption. Finally, this phase produces model descriptions in a format that can be imported by system simulation tools and software modules such as run-time schedulers.

3.1.1 Model Generation Procedure

The framework does not predefine what should constitute an input parameter or a performance metric. Input parameters may include hardware (e.g., cache size, cache line size, register file size) and software features (e.g., input data size, stencil size, DMA block size) while performance metrics are any measurable metric (e.g., execution time, energy, failure rates). The general procedure for model construction can be seen in Figure 4 and consists of three main components: Principal Component Analysis, Clustering, and Forward Model Selection.

3.1.1.1 Principal Component Analysis

Before any modeling takes place, Eiger constructs a single, large matrix $\boldsymbol{D}_{m\times n}$ representing all instrumentation data. Each column is a different input parameter and each row is a separate data point corresponding to a unique application execution. In its effort to minimize domain expertise to construct models, Eiger recommends liberally including input parameters, even if they ultimately will not be used in the model. This step, Principal Component Analysis (PCA) [47], aims to compress the number of columns, eliminating unnecessary

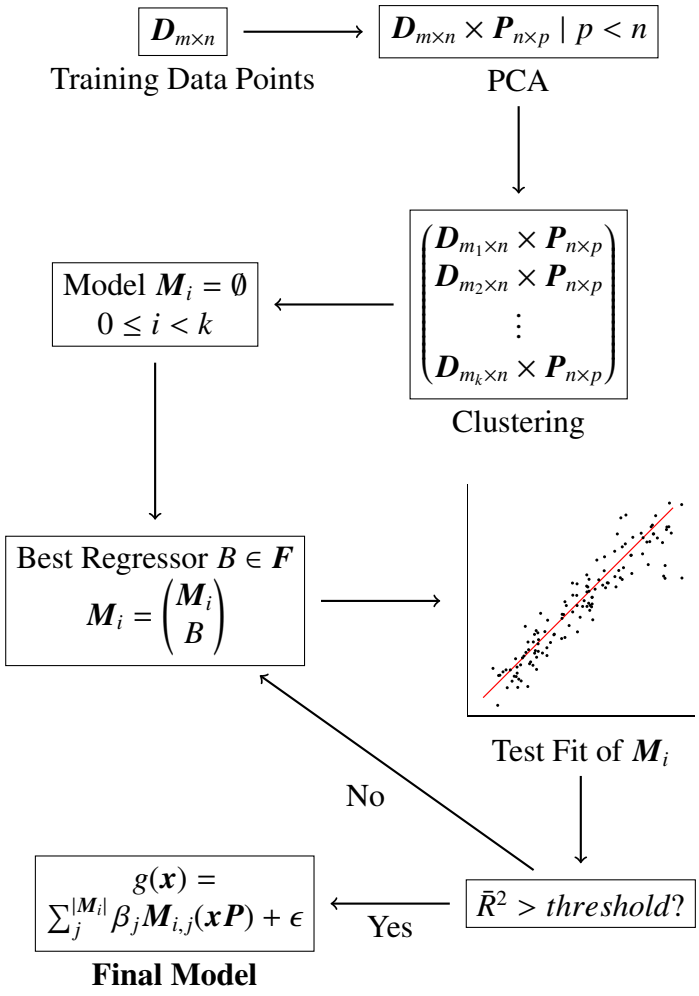


Figure 4: The steps of model construction.

data and simplifying model generation. A linear transformation $\boldsymbol{P}_{n\times p}$ is produced, converting the set of possibly correlated input parameters into a derived set of uncorrelated parameters. Derived parameters with low variance can be eliminated. A concern [48] with this technique is that even parameters exhibiting low variance may relate strongly to the dependent variable, and would be valuable information to discard. For this reason, PCA is leveraged to eliminate parameters with *zero* variance, simplifying the data set without throwing away information.

3.1.1.2 Clustering

The next modeling step is Clustering, where similar data points (i.e., rows of the output matrix from the PCA step) are grouped together. Designing processors to accelerate general sets of workloads would be significantly simpler if all workloads exhibited similar performance characteristics. In contrast, real applications demonstrate varied performance behavior. Dense linear algebra workloads with regular control properties and compute-intensive inner loops differ significantly from irregular workloads with data-dependent branch behavior and load imbalance across threads. Goswami et al. [37] analyze the diversity of CUDA workloads and present a prioritized tree of benchmark applications sorted by how much each increases total variance of a given set of metrics.

Rather than using all the data points in the model fitting step, each cluster goes through fitting separately, under the assumption that this composite of fits will perform better. Applications exhibiting high correlation among principal components are clustered together. A single model trained from profiling data from all applications in a comprehensive benchmark is unlikely to yield high accuracy. Rather, the best model is likely to be obtained from training data gathered by a set of applications that are "similar" to the experimental application. Kerr et al. [54] provide empirical evidence that clustering and partitioning improves model accuracy.

Eiger uses *k-means* clustering, where a data point belongs to the nearest of k clusters. The center point of the cluster is defined as the mean of the input parameter values for all

Table 1: Model pool.

Functions							
x_i^{-2}	x_i^{-1}	$x_i^{-1/2}$	$\log_2(x_i)$	$x_i^{1/2}$	x_i^1	x_i^2	$x_i * x_j$

the data points in that cluster. Eiger performs the next step of model construction for each of the clusters, creating k different analytical expressions. When evaluating the model, the closest cluster must be found and its associated analytical expression used. Choosing the value for k is left to the user.

3.1.1.3 Forward Model Selection

The last transformation fits an analytical expression of the input parameters in a cluster to calculate the result metric. This is achieved with a linear combination of functions, called *regressors*, that are applied to the input parameters. These regressors come from a pool $\boldsymbol{F}$ of candidates. The model pool defines a set of possible basis functions which may be mapped to principal components and whose linear combination yields the resulting performance model. The model pool must be selected by the experimenter and should offer sufficient variety for maximizing goodness of fit of the resulting model. The model pool should include functions that closely model the space and time complexity of dominant algorithms within the applications of interest as well as non-linear combinations of several metrics. For example, compute-bound applications may demonstrate a very strong correlation between the product of clock frequency and dynamic instruction counts. Table 1 describes the basis functions used for this work.

In the Eiger framework a selection approach is taken to progressively include regressors in the final model only if they improve the quality of the fit. This approach is *forward* in the sense that it begins with an empty model and adds regressors to it, in contrast with a *backward* approach in which the model starts out with every possible function and unsatisfying elements are removed.

The stepwise procedure is outlined in Algorithm 1. Beginning with an empty model,

Algorithm 1 Selects a model that minimizes error over a cluster

```
1: profile, performance, modelPool = ... // initialize training data
2: threshold = ... // specification from user
3: finalModel = [] // empty set
4: currentRsquaredAdj = −∞
5: while not done do
6:   maximum = −∞
7:   for each each function left in modelPool do
8:     add function to finalModel
9:     U = apply finalModel to profile
10:    beta = leastSquares(U, performance)
11:    RsquaredAdj = ... // calculate adjusted rsquared
12:    if RsquaredAdj > maximum then
13:      maximum = RsquaredAdj
14:      newFunction = function
15:      newBeta = beta
16:    end if
17:    remove current function from finalModel
18:  end for
19:  if maximum - currentRsquaredAdj > threshold then
20:    add newFunction to finalModel
21:    remove newFunction from modelPool
22:    currentRsquaredAdj = maximum
23:  else
24:    done = True // there are no more useful functions
25:  end if
26: end while
27: return finalModel, beta, currentRsquaredAdj
```

regressors are chosen from the pool and added one by one to select the one that increases the quality of the fit the most. The coefficient of determination (R^2) gives a measure of how well a model is able to map predicted values to their associated training value; it ranges from 0, indicating no correlation between the prediction and the observed valued, to 1, indicating exact replication by the model. Eiger uses the adjusted coefficient of determination ($\bar{R}^2$), a modification of R^2 taking into consideration the number of terms in the model [55]. $\bar{R}^2$ will decrease if the added regressor results in a model that performs worse than would be expected by adding a random regressor. To control over-fitting, in which the predictability of new data points is sacrificed in order to ensure the model approaches the training data as closely as possible, the winning regressor is only added if the amount by which it increases the fit is larger than a user-provided threshold. In this work, selecting the threshold value is an empirical procedure. Model construction finishes when there are no more models left in the pool or when the winning regressor does not surpass the threshold. The final model is comprised of each selected regressor $\boldsymbol{F}_i$ and an associated weight β_i, plus an error term ϵ.

There are instances where the user has some intuition about the relationship between input parameters and performance and wish to generate a model based on this intuition, (i.e., based on a specific set of regressors and input parameters). For example, the user may wish to construct a linear model or a model that is logarithmic in the input data set size. Consequently, Eiger enables the user to *a*) specify the set of regressors to be used, and *b*) specify the set of input parameters to be used. For example, even though the set of input parameters is quite extensive, the user may wish to construct a model based on two input parameters and two specific regressors.

3.1.2 Formal Specification

Let $m \in \mathbb{Z}$ refer to the number of *trials* executed for a multiplicity of applications, datasets, and machine configurations. Let $n \in \mathbb{Z}$ refer to the total number of metrics (measurements) acquired per trial. These may include static application metrics, dynamic metrics acquired during the execution of the trial, and machine configuration parameters.

Define $X \in \mathbb{R}^{m \times n}$ as an input dataset and $R \in \mathbb{R}^{m \times 1}$ as a result set. Each row in X corresponds to a trial instance with result (e.g., execution time) in the corresponding row of the column vector R. Together, (X, R) captures sufficient data describing the application, machine, and performance characteristics to construct a model for R.

$$X = \begin{bmatrix} \vdots & & \\ m \text{ trials} & & \\ \vdots & & \\ \dots & n \text{ metrics} & \dots \end{bmatrix} \quad R = \begin{bmatrix} \vdots \\ m \text{ results} \\ \vdots \end{bmatrix}$$

Principle component analysis (PCA) yields a projection P from X onto $U \in \mathbb{R}^{m \times p}$, where $p \in \mathbb{Z}$, $p \leq n$ is the number of principle components, such that all axes are orthogonal and are sorted in decreasing amount of variance.

Clustering analysis enables a down-selection of trials used in model computation. This analysis yields a subset of rows such that $U' = S\,U$ where $U' \in R^{m' \times p}, m' \leq m$ and S is a selection matrix. This work applies k-means clustering which partitions a set of points into k clusters such that each point in a cluster k_i is closest to the mean of k_i than any other mean. The distance metric used in this work is *squared Euclidean distance*, which gives increasingly greater weight to the distance between two elements.

Model selection yields a function $f : \mathbb{R}^{1 \times p} \rightarrow \mathbb{R}$ that maps individual trials onto a predicted result value. f is the performance model, and this work yields one model per cluster. Model selection leverages linear regression, a commonly-used and well-behaved form of regression that evaluates the dependent variable y as the weighted linear combination of independent variables x plus an error term ϵ, representing any deviation of the expected value of the model from the real value.

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon \tag{1}$$

The method of model estimation is least squares, in which the set of coefficients β is chosen to minimize the residual sum of squares

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 \tag{2}$$

Model selection itself composes a performance model as the linear combination of a set of non-linear functions on the projected profiling data. The set of possible functions is known as a *model pool*, which can include basis expansions, mathematical transformations, and varaible interactions, among others. To allow for greater generalizability, this work can iterate over a set of model pools and select the one that minimizes squared error.

In order to manage model complexity and optimize training error rate, a forward-stepwise procedure is used to aggregate basis functions based upon adjusted coefficient of determination, a modification of the coefficient of determination that adjusts for the number of terms in the model [55].

$$Adjusted R^2 = \bar{R}^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1} \tag{3}$$

It is important to note that $\bar{R}^2$ does not have the same interpretation as R^2; while R^2 is in the range $0 \rightarrow 1$, $\bar{R}^2$ is in the range $-\infty \rightarrow 1$. The intuition is that any value of $\bar{R}^2$ less than zero implies a fit worse than could be expected by chance.

3.2 Modeling Performance of Applications at Scale

While existing techniques for performance prediction such as cycle-level simulation provide an accurate mechanism for estimating execution behavior, they are typically too slow and too fine-grained to observe the system-level phenomenon of extreme scale applications. By composing fast approximations of behavior for regions of computation, full systems can be simulated quickly, estimating both time and energy. The following sections describe the process of creating application skeletons and the tools developed to facilitate this process.

Simulating the execution of applications on a large number of cores requires the abstraction of application software with minimal compromises in simulation accuracy. A recent development is *macro-scale simulation* models, which contrasts with cycle-level or micro-scale simulation models. The macro-scale simulation models are driven by *application skeletons*: a full application, modified to retain its control flow and structure, but with code segments replaced by analytic models to compute the resources consumed by the corresponding code segment. In general the *effect* of execution characteristics are modeled rather than the *mechanism*, i.e., the procedure a code goes through during execution. The use of application skeletons enables modeling to be applied at a larger scale that otherwise feasible. In principal, this technique can be used to model any resource consumed by the system; this thesis applies this technique to execution time and energy, but the methodology is general enough to model resources such as memory or network bandwidth.

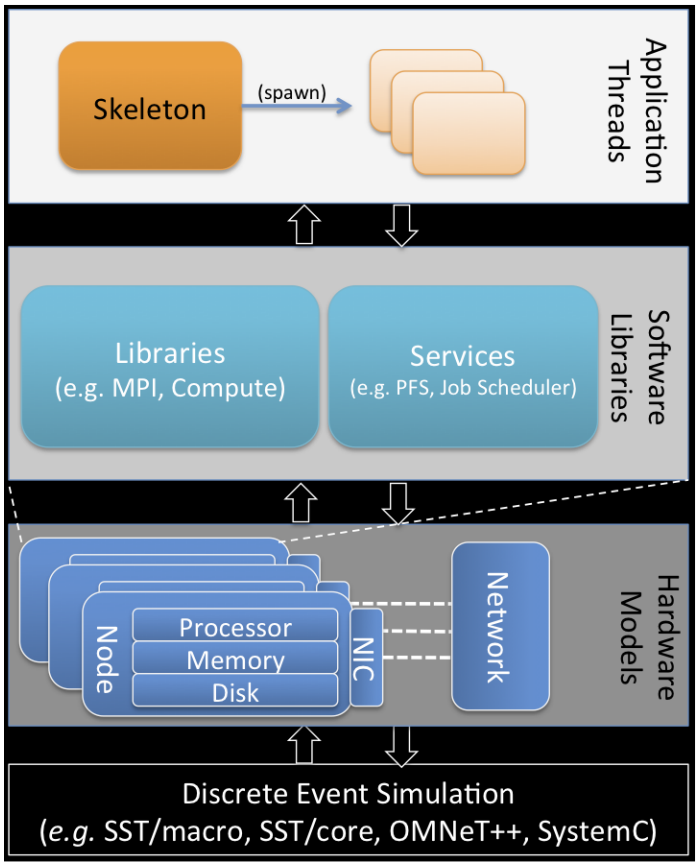


Figure 5: The structure of the SST/macro simulator.

This work leverages the SST/macro simulator, one branch of the SST project [71], for coarse-grained macro-scale simulations. The general organization of SST/macro is shown in Figure 5. SST/macro provides the foundation for large scale systems research and has previously been used in high performance network-related studies [44, 45]. The application skeleton is comprised of communication calls (e.g., MPI, HPX, SHMEM) and models of computation, where control flow structure is preserved. Application skeletons serve as a more accurate (from an execution perspective) representation of the time and energy consequences of application execution. This work seeks to improve the models of computation, including the addition of energy models.

The timing/energy behavior of application skeletons should be as close to the original application as possible; any deviations degrade the accuracy of the simulation. The skeletons should retain control flow characteristics as close to that of the original application as possible. Take for example the simplified structure of an extreme-scale application as shown in Figure 6a, where local and global work is performed on a periodic basis. In this application, computation is performed locally several times before synchronization across the system. For a skeleton to accurately reflect the behavior of its source application, it must retain the control flow and model the execution effects of computation regions.

The SST/macro simulator replaces all communication directives with its own models. Such a skeleton, shown in Figure 6b, would correctly capture the number and duration of all communication functions. However, it misses the execution behavior of potentially lengthy computation between communication calls; the behavior of this application may depend on the loads of each processor and how well the synchronization step is performed. The work in this paper enhances SST/macro with *compute models*, taking the place of all code segments within the skeleton application, as shown in Figure 6c. These models estimate the consumed time or energy taken by a computation region.

```
for(int iteration = 0; iteration < MAX_ITERS; ++iteration){
    local_computation();
    if(iteration % SYNC_PERIOD == 0){
        local_aggregation_step();
        global_sync();
    }
}
```

(a) Original application.

```
for(int iterations = 0; iteration < MAX_ITERS; ++iteration){
    // REMOVED: local_computation();
    if(iteration % SYNC_PERIOD == 0){
        // REMOVED: local_aggregation_step();
        model_communication(global_sync);
    }
}
```

(b) Skeleton application with communication calls replaced with a model, while retaining correct control flow.

```
for(int iterations = 0; iteration < MAX_ITERS; ++iteration){
    model_computation(local_computation);
    if(iteration % SYNC_PERIOD == 0){
        model_computation(local_aggregation_step);
        model_communication(global_sync_model);
    }
}
```

(c) Skeleton application with the inclusion of both communication and computation models.

Figure 6: Creation of a compete application skeleton from its original.

3.2.1 Data-Dependent Computation

Ensuring correct flow of the application skeleton is important when considering computation that is data-dependent; there may be computation regions whose execution is prescribed by the data being processed. These types of computations manifest in areas such as iterative algorithms, where termination is based on convergence criteria, and in sparse data representations, where meta-data such as dimensionality is not fixed.

There are several approaches that can be used to ensure that the execution behavior of a skeleton matches its parent application. The most direct, but most costly route, is to preserve in the skeleton any calculations that dictate the termination conditions of execution. A typical manifestation of this is when some internal data structures must be initialized before computation is performed on them. The exact sizes of these structures are not known until after the initialization has completed; here their construction is permitted, a time- and resource-intensive process, to access these dimensions.

When data-dependent computation is performed throughout the application execution, more care must be taken. An example is when the loop iteration count is not known statically, such as operating on sparse matrix representations. When these matrices are multiplied, only the nonzero elements are iterated through. A trade-off must be made here between accuracy of the simulation and runtime expense in maintaining execution-dependent information.

3.2.2 Compute Modeling

Making models of computation begins with finding the places in the application skeleton where computation takes place. Typically this involves nested loops or other forms of iterative computational load, such as found in matrix algebra. Standard performance profilers like `gprof` are invaluable for this task; they break down where time is spent in the application and the function call hierarchy. Converting the most time-consuming functions in the profile into models works well; these regions should avoid containing network communication calls.

This work proposes a method for generating high-quality analytic models for these regions through statistical inference. A corpus of execution data from instrumented executions of the application, including source-level parameter values, is used as training data to construct an analytic model of execution time or energy. This work leverages the Eiger modeling framework. This modeling approach provides many benefits over traditional approaches to constructing analytic models, including removal of the need for deep domain expertise (e.g., in energy consumption) during model construction, increase in the diversity of what can be modeled, and significant reduction in manual tuning.

3.2.3 Skeletonization Procedure

The original source code of the candidate application is augmented so that compute intensive code regions can be easily replaced by the model thereby producing an application skeleton. These modified applications are then executed by linking with the SST/macro simulator that simulates the execution of this application on a large scale parallel architecture model. The collection of measurement data is done with the Lightweight Performance Data Collectors (`lwperf`) tool[1], a collection of simple, portable macros aimed at making it easy to gather high-level algorithm features and performance numbers from real applications. A single code markup scheme is provided which, based on compilation flags, can record performance data for code regions from parallel application execution. The performance data is recorded in an Eiger database. `lwperf` supports C, C++, and modern Fortran. Figure 7 shows the different aspects of model generation, the typical procedure for which is:

1. Place appropriate `lwperf` initialization (`PERFINIT`) and finalization (`PERFFINALIZE`) calls at the beginning and end of the application, respectively.
2. Locate the compute regions to be modeled. Typically this involves nested loops or accelerator kernels, but can be at user-defined granularity.
3. Surround the compute regions with appropriate calls to `lwperf` macros `PERFLOG`

[1]The source code and documentation for `lwperf` is available at `https://github.com/gtcasl/lwperf`

and `PERFSTOP`, indicating a region name as well as any parameters which may be important determinants of the performance of this region, such as loop bounds.

4. Compile the application with the data collection flags set and run the program over various problem sizes. This will fill the Eiger database with training data.
5. Run Eiger model generation for each compute region.
6. Rebuild the application with the simulation flags set. During execution, the marked compute regions will be replaced with with calls to the models, skeletonizing the application.
7. Simulate the skeleton with SST/macro. This requires linking with SST/macro and providing a configuration file describing the machine model to use. More details on the changes needed to make an application skeleton run in SST/macro can be found in its documentation[2].

Handling heterogeneous or parallel node architectures is straightforward with Eiger models. Eiger makes no assumptions about the execution model, instead learning the relationships between input parameters and performance for each marked code region. As long as the entire computation region is captured and profiled, including any intra-node communication, Eiger will attempt to learn a model of performance from the data. This simplifies the modeling process, reducing the need for expert knowledge about the execution model of the simulation target and eliminating any hand-tuning.

At any code region where data collection is desired, the collection point is defined by a unique name and the set of parameters characterizing the workload. Figure 7a shows the same function from Figure 8a with `lwperf` instrumentation calls inserted. The region is uniquely identified by the first argument and indicates two metrics of interest, the matrix dimensions `M` and `N`. A single data point is then recorded every time this region of the program is entered, along with the time elapsed as shown in Figure 7b.

[2]The source code and documentation for SST/macro is available at `https://bitbucket.org/sst-ca/sstmacro`.

```
void foo(int M, int N, int** matrix){
    PERFLOG(foo, M, N);
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            int temp = (matrix[i][j] + i) % 10;
            matrix[i][j] += temp; } }
    PERFSTOP(foo, M, N);
}
```

(a) Annotated Original

```
void foo(int M, int N, int** matrix){
    double start = MPI_Wtime();
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            int temp = (matrix[i][j] + i) % 10;
            matrix[i][j] += temp; } }
    double stop = MPI_Wtime();
    EIGER_record("foo", M, N, stop - start);
}
```

(b) Data Collection

```
void foo(int M, int N, int** matrix){
    /*
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            int temp = (matrix[i][j] + i) % 10;
            matrix[i][j] += temp; } }
    */
    double foo_time = EIGER_compute_time("foo", M, N);
    SSTMAC_compute_time(foo_time);
}
```

(c) Skeleton

Figure 7: Transformation of a simple function into a compute model. The original application is transformed into data collection and skeleton versions during compilation.

To facilitate skeletonization, `lwperf` can be configured to remove and replace all instrumented code with calls to Eiger models for running within the SST/macro simulator. Each collection point is replaced with calls to an Eiger model under the assumption that each characterizing parameter is used as an input feature to the model. Figure 7c shows how computation is replaced with a call to the model for execution time. This technique allows for the application to serve as both the instrumentation source and the skeleton.

3.3 Experimental Results

To demonstrate the benefits of a comprehensive modeling tool like Eiger, a simple model is constructed by hand that tries to replicate the structure of loop nests. Computation is represented by a linear function of the loop bounds scaled by a *loop factor* describing the amount of "work" performed per iteration. For example, the loops in the simple function in Figure 8a might be replaced with a single call such as in Figure 8b, specifying the nested lower and upper loop bounds, and in this case, a loop factor of 2. The product of the loop bounds and loop factor is scaled by a set of empirically derived constants to provide an estimate of the performance metric.

The loop model is simple and intuitive. However, it is a manual process and can only represent linear relationships. Therefore it is not easily adapted to the parallel implementations of nested loops or execution on heterogeneous architectures where performance is a more complex function of many interacting microarchitectural features. In this paper it serves as a baseline for comparison for cases where such simple models would suffice.

3.3.1 Experimental Setup

This work uses four scientific application codes familiar to the HPC community to demonstrate adding Eiger models to application skeletons:

1. **MiniMD** is a molecular dynamics mini-application from the Mantevo project [39], created to investigate improving spatial-decomposition particle simulations as a simpler, more accessible version of LAMMPS [65]. Parameters to miniMD include

```
void foo(int M, int N, int** matrix){
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            int temp = (matrix[i][j] + i) % 10;
            matrix[i][j] += temp; } }
}
```

(a) Example function performing computation.

```
void foo(int M, int N, int** matrix){
    /*
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            int temp = (matrix[i][j] + i) % 10;
            matrix[i][j] += temp; } }
    */
    SSTMAC_compute_loops2(0, M, 0, N, 2);
}
```

(b) Computation replaced with model.

Figure 8: Example function (a) and the same function where the computation is replaced by the loop model (b).

problem size, atom density, temperature, time step size, number of time steps, and particle interaction cutoff distance. This application offloads computations to GPU accelerators using the CUDA parallel programming language [62].

2. **HPCCG** is a simple conjugate gradient benchmark code for a 3D chimney domain, also from the Mantevo project. It generates a 27-point finite difference matrix with a user-prescribed sub-block size on each processor, and weak scales well to a large number of processors.
3. **MiniFE** is a proxy application from the Mantevo suite for unstructured implicit finite element codes. It is similar to HPCCG but provides a much more complete vertical covering of the steps in this class of applications. This application also uses CUDA for GPU acceleration.
4. **LULESH** discretely approximates explicit hydrodynamics equations [51] found in complex software packages like ALE3D by partitioning the spatial problem domain

Table 2: Machine configurations for collecting time and energy training data.

Use	Parameter	Value
Time	CPU	Intel i7-920
Time	CPU threads	8
Time	CPU clock rate	2.67 GHz
Time	Memory size	6GB
Time	GPU model	NVIDIA GTX 660 Ti
Time	GPU cores	1344
Time	GPU clock rate	915 MHz
Time	GPU memory size	2GB
Energy	CPU	Intel i7-4770
Energy	CPU clock rate	3.40 GHz
Energy	CPU threads	8
Energy	Memory size	16GB

into a collection of volumetric elements defined by a mesh. This application uses OpenMP [28] to parallelize computation across CPU threads.

Constructing Eiger models relies upon a corpus of training data taken from real executions, including the performance metrics associated with each region of computation. This work measures application energy consumption for CPU-only applications using the Running Average Power Limit (RAPL) hardware performance counters available on recent Intel processors. These counters are measured using the Performance Application Programming Interface (PAPI) [61] and cover energy consumption for the entire processor package. Infrastructure for measuring energy consumption of the entire node (i.e., memory modules, accelerators, network interface components) can be used to better describe the energy characterization of a node, but is outside the scope of these experiments. The systems used for these experiments are described in Table 2.

3.3.2 Model Fit

This experiment examines how closely the models predict the training data. For these experiments, each application was executed unmodified for several different input sizes, using the data collection procedure described in Section 3.2.3. This training data was then

used to generate the models; one model is generated for each region of computation.

The improved fit of the Eiger models over the baseline loop models can be clearly seen in the region of computation in MiniFE where the Dirichlet boundary conditions are imposed; this region requires calls to two GPU kernels. The fits of the models are shown in Figure 9. The loop models were hand-tuned until they converged as close as possible to the training data. This process required significantly more time to construct, requiring a thorough examination of the original application source code. Even then, the performance of these models was in general poorer than Eiger. The loop model statically subdivides the number of elements on the boundary across all the GPU threads, achieving an R^2 value of 0.912 and an average error of 6.73%. In contrast, Eiger generates a model that scores an R^2 value of 0.983 and an average error of 2.43%, without manual intervention.

As this computation is performed on the GPU, factors such as how long it takes to transfer all the data to the device, how work is assigned to threads, and how those threads get scheduled on the device at runtime contribute to variability in performance unrelated to the size of the vectors. To increase the quality of the loop models, more rigorous attention to the inner workings of the device could be harnessed, but would require expert knowledge of the architecture.

3.3.3 Simulation Accuracy

One concern with the models is that they may be too strongly tied to the training data. This phenomenon is called *overfitting*, indicating that the model is biased towards the training data. While a model may be a strong predictor of the training data, there might be ancillary effects that go unobserved as the problem size changes. One example of this happens with the region of computation in HPCCG where two vectors are scaled and added together. For smaller problem sizes, the vector length strongly correlates with execution time. However as the vectors get larger, the structure of the memory hierarchy affects the average access latency per element. This causes the loop model to diverge from the training data.

In contrast, the model generation process in Eiger avoids overfitting through use of

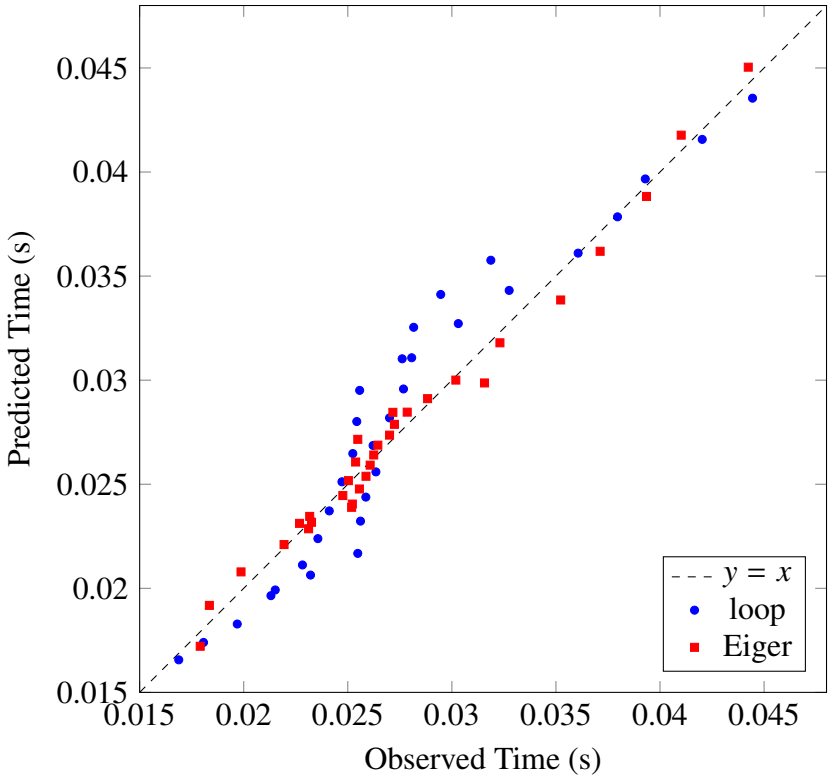


Figure 9: Observed training values and their associated loop model predictions for MiniFE Dirichlet boundary condition region of computation.

a threshold parameter. This tunable parameter prevents the inclusion of elements in the model that would result in overfitting. Figure 10 shows the effect of the threshold value (i.e., how biased the model is to the training data) on the average number of regressor functions in the model and the resulting simulation error. There is a middle ground where increasing the threshold no longer reduces overfitting but rather begins to eliminate necessary model terms, increasing overall error. Selecting the correct threshold requires performing a series of independent experiments, ensuring that the model constructed with a given threshold is sufficiently general for the types of inputs the model sees.

With the models trained, this experiment shows how the individual models interact when simulating an entire application skeleton. Having accurate application skeletons, across input problem sizes, forms the foundation for macro-scale simulation. To eliminate

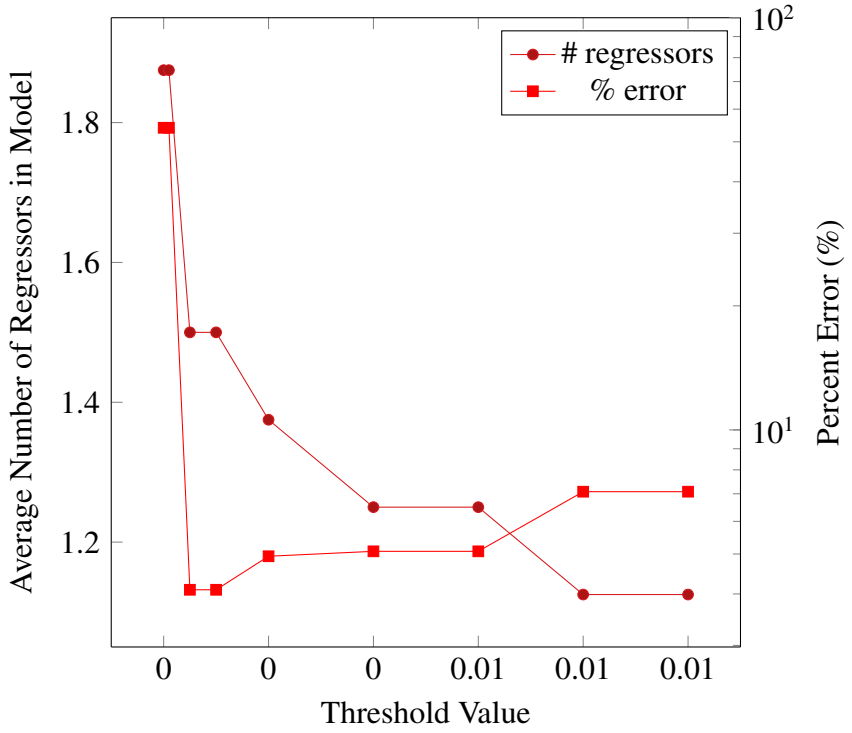


Figure 10: Effect of threshold value on overfitting for HPCCG.

any error caused by the communication models in SST/macro, only a single node is simulated. This test demonstrates that all computation performed by the application is modeled and to guarantee that any code removed during the skeletonization process does not change the overall structure of the application. Figure 11 illustrates how close the simulations are to correctly predicting the execution times of all four applications and the energy consumption of HPCCG and LULESH. On average, the models generated by Eiger result in 6.08% error, where the loop models incur a 9.17% error.

3.3.4 Simulation Speedup

SST/macro has been demonstrated to simulate parallel systems with tens of thousands of nodes in minutes [88]. To enable scalable macro-scale simulation, the models of computation added to application skeletons must be fast. These experiments compare the execution time of the original application with its application skeleton. Where as testbed systems

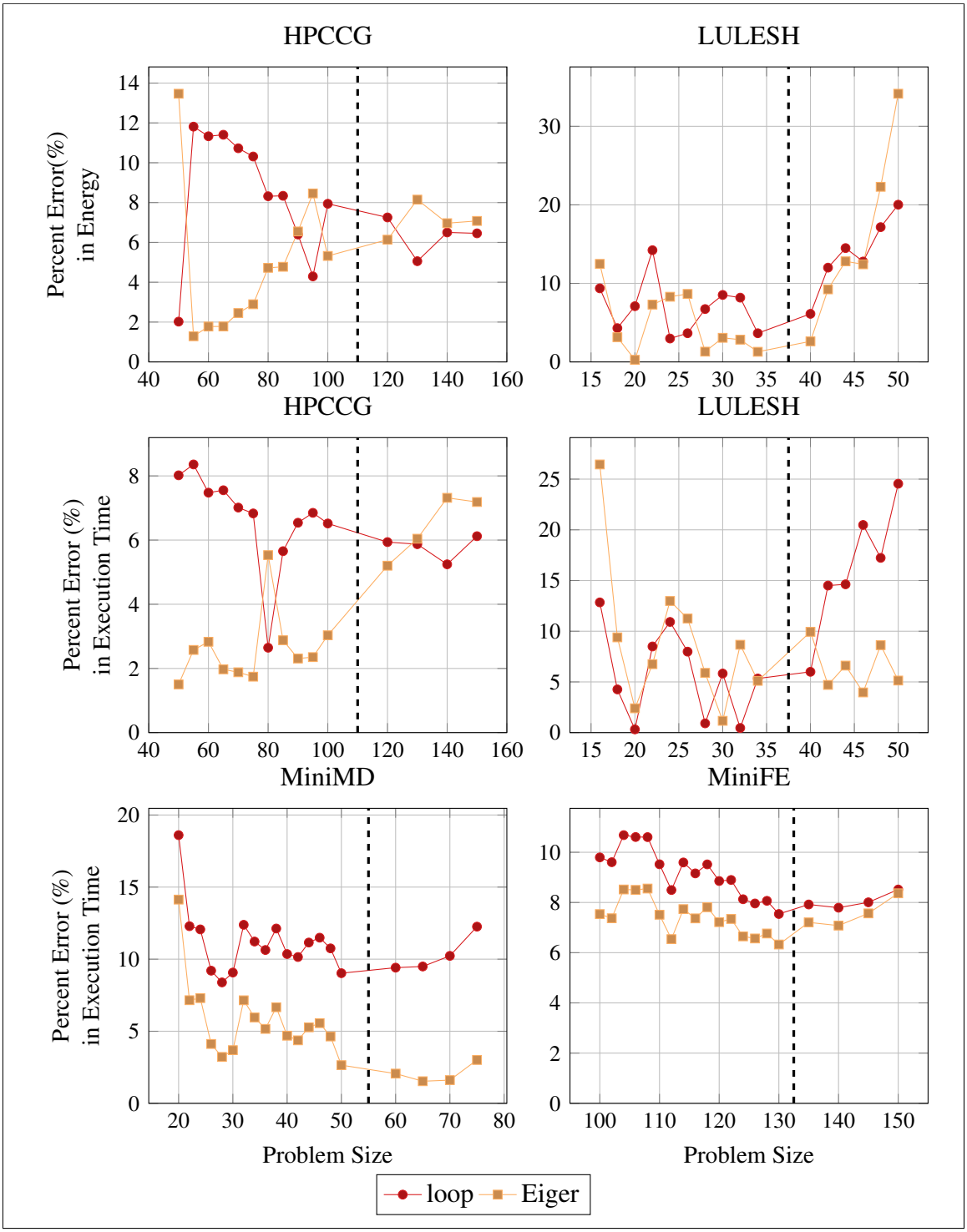


Figure 11: Error in simulation predictions of application runtime.

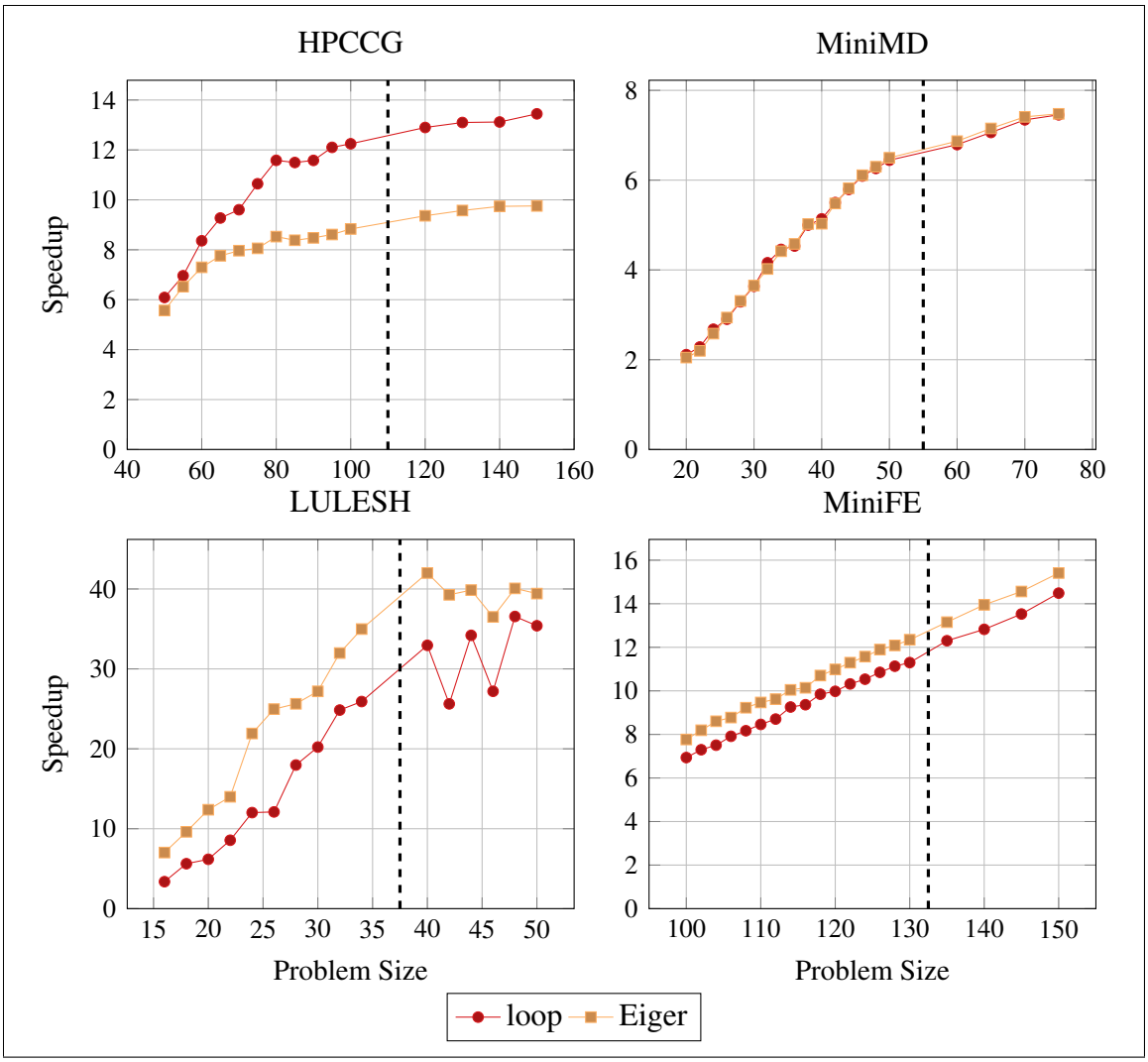


Figure 12: Speedup of simulator runtime over native execution.

and cycle-level simulators tend to run orders of magnitude slower than on the native machine, simulating the application skeletons results in a *speedup* of simulator runtime over the native execution time of the application, as shown in Figure 12. These speedups tend to increase with the problem size; the predicted computation time increases as the problem size increases, but the time it takes to poll those models remains constant, resulting in an overall increase in the speedup of simulator runtime.

Eiger models are capable of increased simulator speedup over loop models due to their higher level of abstraction; for similar accuracy, more granular loop models are required.

Take, for example, creating a loop model for the execution time of a GPU kernel. Memory transferred from the host to the device is represented by the amount of data to be transferred and the time to transfer an individual unit. The computation in the kernel is modeled, requiring a non-trivial understanding of how tasks are scheduled and the breakdown of tasks to be performed.

This is significantly easier to model with Eiger: the entire kernel can be abstracted into a single model that is learned from training data, encompassing all aspects of execution of the kernel, including data transfers. This reduces the total number of times the model needs to be polled, decreasing the running time of the simulation. This is the case for the MiniMD, MiniFE, and LULESH skeletons. The HPCCG skeleton behaves differently; simulation speedup is lower when using Eiger models compared to loop models. This is due to the simplicity of computation regions within the application; there are few opportunities for abstracting large regions of computation into Eiger models, resulting in a similar number of calls to Eiger models as loop models. Eiger models are more computationally expensive to evaluate than loop models, resulting in lower speedup. On average, the skeletons with loop models ran 12.1 times faster than the original application and skeletons with Eiger models ran 12.5 times faster.

3.4 Concluding Remarks

This chapter demonstrates the process for automating the generation of statistical performance models. Eiger provides an automated method in which designers may profile and characterize workloads, automatically construct performance models, and evaluate performance sensitivity to processor configurations. Application skeletons, the vehicles of macroscale simulation, require high-fidelity models of computation. This chapter presented a method for modeling time spent in compute regions: statistical models from Eiger that learn the relationship between input parameters and the performance metric (e.g., execution time or energy) through analysis of instrumentation data from execution of applications

on real hardware. On average, the statistical models added to four scientific proxy applications resulted in simulations that are 12.5 times faster than the original application with only 6.08% error. This is on average 12.5% faster and 33.7% more accurate than baseline models.

CHAPTER 4

CHARACTERIZING RELATIONSHIPS BETWEEN TIME AND ENERGY SCALING

The previous chapter described how to predict the execution time and energy consumption of regions of applications as a function of program parameters. This was used to accelerate macro-scale simulation, where scaling is achieved by adding more nodes to the network. However, there is a significant range of configuration states that may affect the trade-offs between time and energy in an application, at the node level. In particular, there has been an enormous growth in the number of execution cores per processor node, as well as greater flexibility in the frequency states to set for each core. Understanding how these configurations affect the behaviors of energy and how they differ from time behaviors is the problem addressed in this chapter.

Addressing this issue has two parts. The first part examines the way that applications behave as the number of cores increases, i.e., constructing a model for energy scaling behavior that relates the execution hardware and the application characteristics. While there is a large corpus of work analyzing the time scaling of applications, little work has explored how applications scale in energy as the number of cores increases. The main observation described in this piece of the chapter is that although execution time can be overlapped in time (e.g., through the use of parallelization), energy is always incurred to do the work and cannot be hidden. Energy is the cumulative cost for all the *work* performed, regardless of whether that work happens sequentially or in parallel, and as a result scales differently with parallelism than time. This chapter decomposes energy consumption into two parts: idle energy, that is proportional to the amount of time the system is on, and dynamic energy, that is proportional to the amount of work the application performs. An analytical model is constructed by combining these with a notion of time scaling, providing a framework for

understanding the relationships between platform characterization and application modeling. This thesis includes an analysis of applications, comparing the accuracy of the model with empirical measurements on server-grade multicore CPU systems.

The second part of the solution builds on this model, presenting a technique for selecting an appropriate configuration to maximize performance for a given amount of power dissipated. These configurations include the degree of parallelism and DVFS state. Depending on the configuration, an application exhibits different performance and power consumption behaviors. This work proposes the selection of Pareto-optimal configurations from a combination of settings, where performance is maximized for a given power cost. This formulation reduces the size of the configuration space, simplifying the trade-off analysis. This modeling method increases the understanding of application energy behaviors and paves the way to performance optimization under a power cap, described in Chapter 5.

The main contributions described in this chapter are:

- A model for understanding the relationships between time scaling and energy scaling.
- An evaluation of the scaling model on real architectures.
- The construction of Pareto frontiers to characterize the trade-offs between performance and power consumption as a function of system configuration in shared memory multicore architectures.

4.1 Theoretical Energy Scaling

This section lays out the foundations of the energy scaling model by presenting the framework used to describe performance scaling, then describing a breakdown for energy consumption and the model for its behavior. This model has been described for use in exploring the benefits of hybrid processor architectures, but this behavior is generalized to describe the behavior of applications on all types of platforms.

The term *strong scaling* is used to describe the execution time T_p of a parallel application as the degree of parallelism p increases, relative to the execution time of the sequential

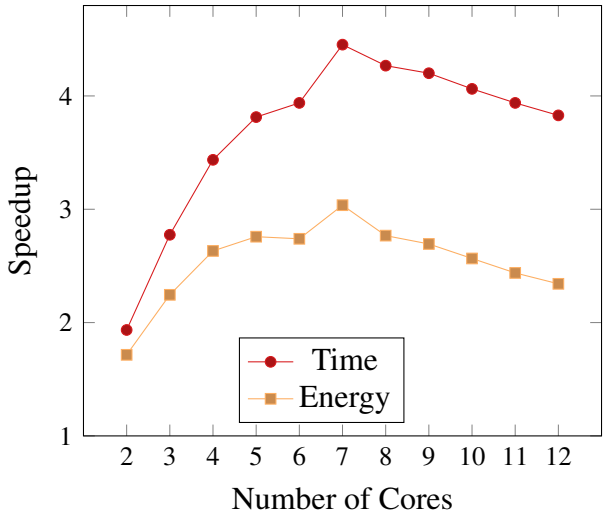


Figure 13: Energy and time speedup for strong-scaling study of HPCCG mini-app on a Sandy Bridge-EP platform.

version T_1, for a fixed size of input data. The *speedup* $S_t = T_1/T_p$ describes how well the parallel application, for the same input, scales to multiple cores. Define the parameter f to represent the fraction of the program that is unable to be parallelized, such as critical sections or communication overhead. In a perfectly strong-scaled application, the speedup is equal to p; intuitively, this is because all work is evenly distributed across cores. A nonzero value for f caps the achievable lower-bound, resulting in the following relationship known as Amdahl's Law [7]:

$$S_t = \frac{1}{\frac{(1-f)}{p} + f} \tag{4}$$

The value $\frac{S_t}{p}$, defined as η_t, is called the *time efficiency* of the application, indicating how close to ideal the application strong scales.

In a similar way, the energy consumption behavior of the system can be modeled with E_p and E_1 representing the energy consumed by the parallel and sequential executions of the application, respectively. *Energy speedup*, analogous to time speedup, is defined as $S_e = \frac{E_1}{E_p}$. Understanding how energy changes as a function of parallelism is the main contribution of this piece of the thesis.

4.1.1 The Baseline Energy Scaling Model

Figure 13 shows the measured speedup in both time (S_t) and energy (S_e) for the HPCCG mini-app [39], where speedup is defined as the sequential time or energy divided by the parallel time or energy. Although the two curves have a similar shape, they are not the same. If energy behaved the same as time, the energy speedup of an application would mirror the time speedup. The key insight into the difference between the two is that energy can never be reduced by concurrent execution: while execution can be overlapped in time, all computational "work" must still be paid for with energy. In an ideal machine, where energy is only paid for the work performed, total energy consumption would be invariant to the degree of parallelization, resulting in an energy speedup always equal to one.

When these processors are turned on, there is always power draining, even if no computation is taking place. At the device level, some current always flows between transistor terminals, known as *leakage current*, resulting in a constant but non-zero draw of power. This is the cost of keeping the devices turned on. Similar costs are associated with the software, such as the operating system performing some background operations intermittently while an application is running, artificially increasing the observed energy consumption of the application's execution. Both of these phenomenon can be characterized as the energy cost associated with running an inactive system, in this work called *idle energy*, E_i. All energy that is consumed by doing computation by the application is called *dynamic energy*, E_d. The energy behavior during perfect strong scaling can be modified with this behavior in mind. Energy consumed by the sequential version is defined as $E_1 = E_i + E_d$.

These two components for energy can describe how energy consumption changes when running in parallel. As described, dynamic energy can never be reduced; it is the cost of solving the problem. However, the idle energy for the parallel execution decreases at the same rate that execution time does, since the machine does not need to be powered on for as long. Parallel application energy consumption is specified by Equation 5.

$$E_p = \frac{E_i}{S_t} + E_d \tag{5}$$

This equation shows that any loss in time speedup is applied only to the idle energy; intuitively, any time the program is serialized, all cores must remain on but idle. In this way any penalties due to poor strong scaling in time also affect the strong scaling in energy, as a function of the relationship between idle and dynamic energy. The *idle fraction s* is defined as $\frac{E_i}{E_i+E_d}$. This serves as an analogy to the serial fraction in Amdahl's Law; the closer the idle fraction is to zero, the more efficient the hardware. Energy speedup can then be reformulated as in Equation 6.

$$S_e = \frac{1}{(1-s) + \frac{s}{S_t}} \tag{6}$$

This formula indicates that energy speedup ranges from one to the value of time speedup, depending on the idle fraction. Energy speedup will only be greater than one, implying improvements in energy consumption with parallelization, if both the idle fraction and time efficiency are high. Figure 14a shows the modeled energy speedup as a function of the idle fraction, for various time speedup values. Only when the idle fraction is high can there be any opportunity for energy speedup. Intuitively, this is because the idle energy is high during the sequential execution, implying an energy penalty just to run the application on an inefficient platform. When the idle fraction is low, energy speedup is limited since there is less idle energy to be reduced. Similarly, Figure 14b shows the modeled energy speedup as a function of time speedup for different idle fractions. This shows how improvements to time speedup directly relate to energy speedup only when the idle fraction is high.

This theoretical model make assumptions about application behaviors on real architectures. The follow subsections lay out what effects are overlooked and how to incorporate them into the model.

4.1.2 The Effect of Processor Idle States on the Model

The general trend in the model is significant under-prediction of energy scaling, indicating that the true idle fraction is greater than initially assumed. This implies that it is incorrect

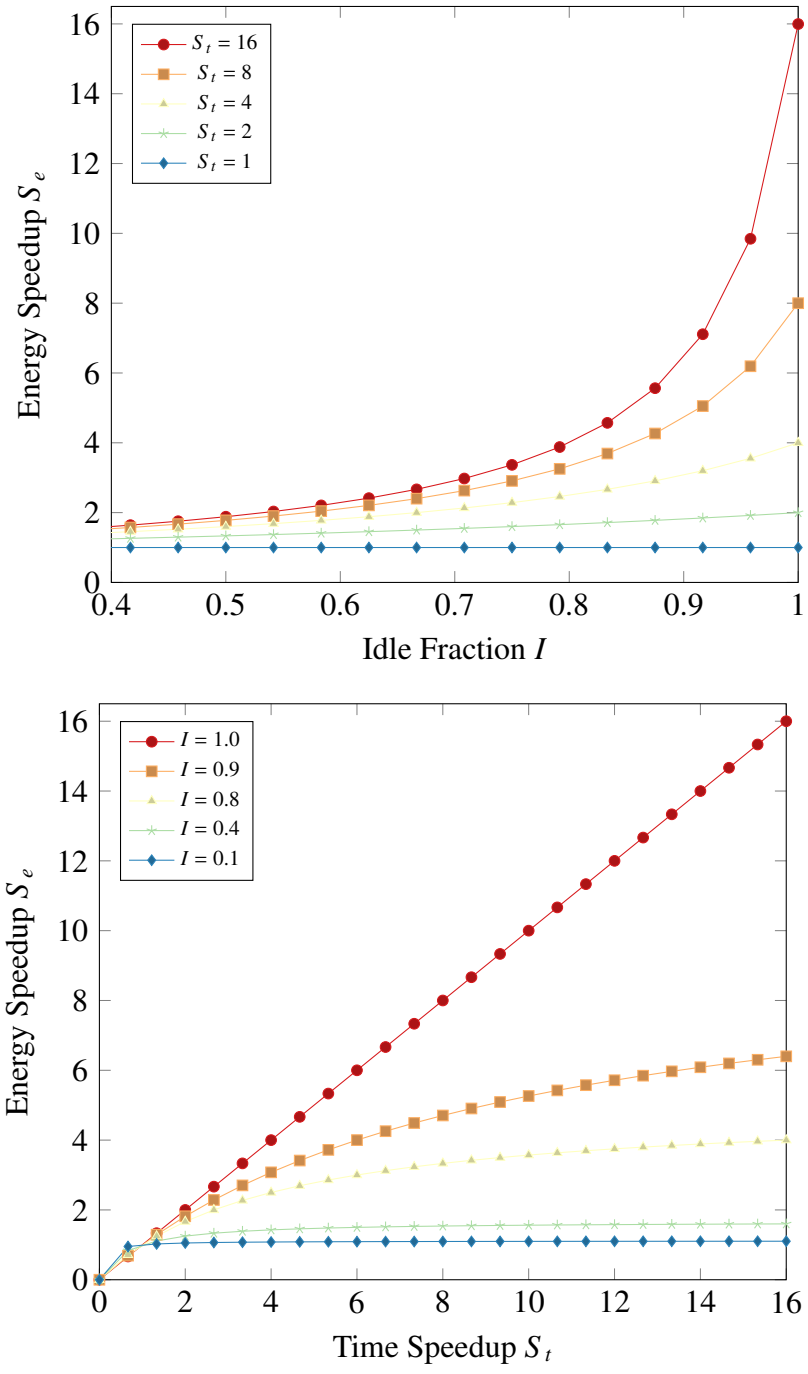


Figure 14: Achievable energy speedup as a function of: a idle fraction of energy for different time speedup values, and b time speedup for different idle fractions of energy.

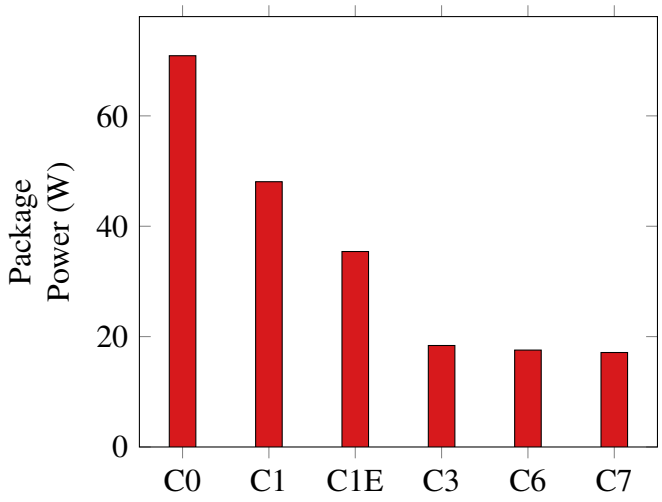


Figure 15: Idle power for each core C-state on the Sandy Bridge-EP platform when running a stress microbenchmark.

to select the idle fraction based on the power consumed by the system when no programs are running. Recent innovations in processor architecture allow for the processor to enter into many different low-power states, called C-states, in order to reduce energy consumption [1]. This complicates the measurement of idle energy, since it is no longer invariant in time; increasing the C-state number has the effect of reducing power consumption while increasing the latency to return to C0, the active state where instructions are executed.

The power consumed by each C-state can be measured by limiting the permissible maximum latency for returning to C0 through the Power Management Quality of Service (PMQOS) interface. This prohibits the processor from entering into a C-state with a longer latency than our choice. We measured the package power for each state on the Sandy Bridge-EP platform when the system is idle, as shown in Figure 15. The original package idle power number comes from all cores being in state C7, which is the lowest power state available to this platform.

Since the processor is at liberty to select the C-state for inactive cores, it is not possible to know a priori which state's power to use in calculating the idle fraction. Instead, this can be determined empirically with a microbenchmark that stresses individual cores at a time, associating power consumption with the number of cores active. Then a curve is fit

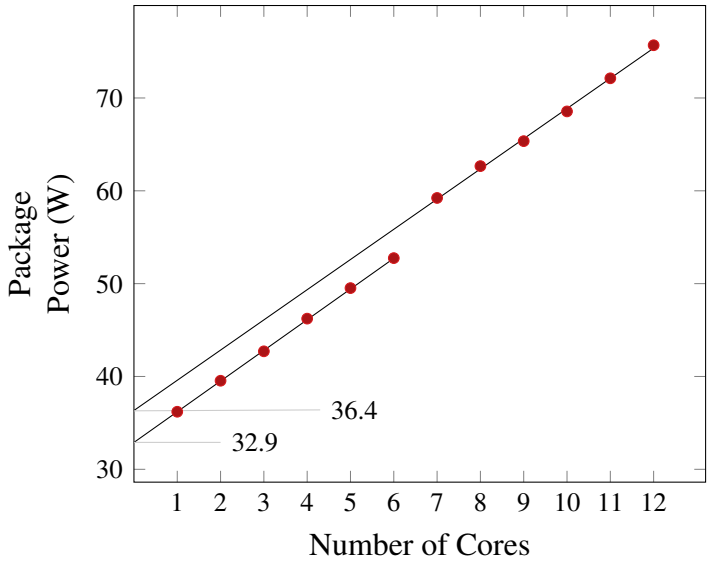


Figure 16: Sandy Bridge-EP platform power as the number of compute tasks increases, extrapolating the idle power.

to these points. The slope represents the incremental power consumed by each additional active core, and the intercept represents the idle power for the system. Figure 16 shows the data points and trend lines for the Sandy Bridge-EP platform. Note that two lines are fit, splitting up the first and second sockets. In addition to core C-states, there are processor C-states that affect the entire package. Because of this, when only a single core and therefore a single package is active, the other package is allowed to remain in a low-power state. The transition from the sixth to seventh core requires the second package to move out of the low-power state. By fitting two separate curves, we demonstrate the incremental costs of *a*) moving all cores out of the lowest-power idle state, *b*) moving the first package out of its low power-state, and *c*) moving the second package out of its low power-state.

4.1.3 Application-Dependent Active Power

The second method to improve the fidelity of the model is to consider the choice of total power. Up until this point, the model has been using a fixed value for the peak power consumed by the sequential execution of the benchmarks. However, this obscures the application-dependent nature of dynamic energy; the total power varies from application

Table 3: Configuration of Intel Sandy Bridge-EP platform used for experimental data collection.

Parameter	Value
CPU	Xeon E5-2620
CPU threads per socket	6
CPU Sockets	2
CPU clock rate	2.0 GHz
DRAM DIMMs	4
Total memory size	64GB

to application. This extends from the assumptions made in the model, in that dynamic energy is perfectly divisible among concurrent threads, with the assumption that the dynamic energy is the total cost to solve the problem, independent of time. Since each workload is solving a different problem, the total energy should vary. Rather than using a fixed cap to indicate the best-case scenario for energy scaling, the model uses the calculated single-threaded active power to determine the idle fraction.

In summary, the energy scaling model requires several adjustments to accurately capture the relationship between time and energy as the application scales. This framework for understanding the way that configuration affects time and energy different sets the stage for future work that will expand this analysis to processor clock frequency, to ultimately construct Pareto frontiers that delineate the optimal configuration to achieve desired time and energy requirements. The first demonstration of this optimization work is described in the next section.

4.1.4 Experimental Results

There has been little work so far that measures the energy scaling on existing platforms to validate the behavior of the model. This work measures the processor socket and memory energy values as the scalability changes on a shared memory multicore platform, described in Table 3.

To measure energy, this work leverages the Intel Running Average Power Limit (RAPL) [1]

Table 4: Descriptions of the applications examined in this work, along with their input configuration.

Type	Application	Description	Size
Mini-app	HPCCG	Conjugate gradient	150^3
	MiniFE	Finite element	100^3
	Lulesh	Shock hydrodynamics	30^3
Benchmark	graph500	Graph search	$-s$ 20
NAS solver	bt	Block tri-diagonal	*A*
	lu	Lower–upper Gauss–Seidel	*A*
	sp	Scalar penta-diagonal	*A*
NAS benchmark	dc	Data cube	*A*
	ua	Unstructured adaptive mesh	*A*
NAS kernel	cg	Conjugate gradient	*B*
	ep	Embarrassingly parallel	*B*
	ft	Discrete 3D FFT	*B*
	is	Integer sort	*B*
	mg	Multigrid mesh	*B*

hardware performance counters. For these experiments both the processor package and DRAM energy consumption are measured. Although our experimental platforms have two sockets, they are treated uniformly as a single node by summing the measured energy values for each socket or DRAM power plane. Reading these performance counters is done with the PAPI [61] interface. Processor frequency state changes are disabled by fixing the cores at their peak frequency, as well as disabling any turbo-boost states.

This work examined the energy scaling behavior of a wide spectrum of applications and benchmarks of interest to the high performance community. The work in this chapter uses the mini-applications HPCCG [39], Lulesh [51], and miniFE [39], the Graph 500 benchmark "Search" [84], and ten benchmarks from the NAS Parallel Benchmark Suite [12]. More information on the programs and input sizes is shown in Table 4.

These applications are all executed on the CPU and parallelized using the OpenMP parallel programming API [28] to give fine-grained control over the degree of parallelism. All threads are bound to cores for the duration of the program's execution, with threads

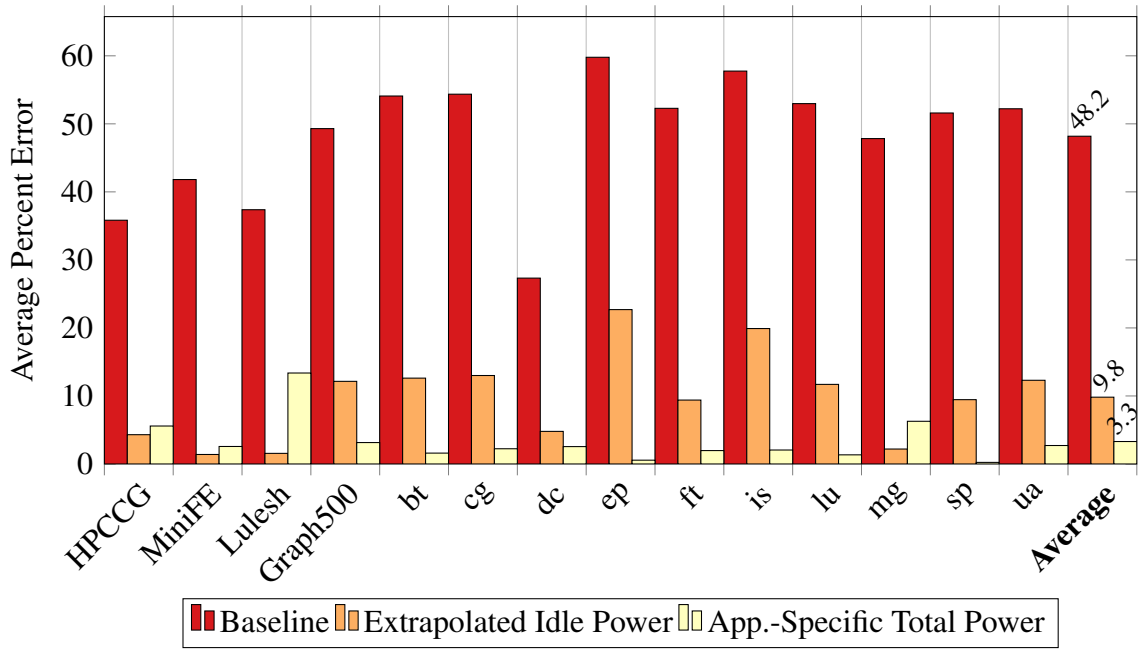


Figure 17: Error between measured energy speedup for the baseline model and progressive improvements due to extrapolated idle power and per-application active power.

added sequentially to a socket until it is full before assigning to the second socket.

With the baseline setup as described, the model significantly underpredicts the energy scaling behavior for all fourteen experimental workloads, as shown in the first series in Figure 17. Measuring the idle fraction of the application is significantly better at predicting energy scaling. The second series in Figure 17 shows the predicted and actual energy scaling values with the extrapolated idle power for each socket. Since this only affects the idle power for the processor, the memory energy scaling is unchanged. The third series in Figure 17 shows the case when, in addition to idle power, the application-specific total power is incorporated into the model. This is the result of a more accurate characterization of the work performed by each application, which contributes to the dynamic energy.

By adding these two techniques, model error can be reduced from an average of 48.2% for the baseline model to 9.8% with the measured idle power and down to 3.3% when modeling the application-specific power.

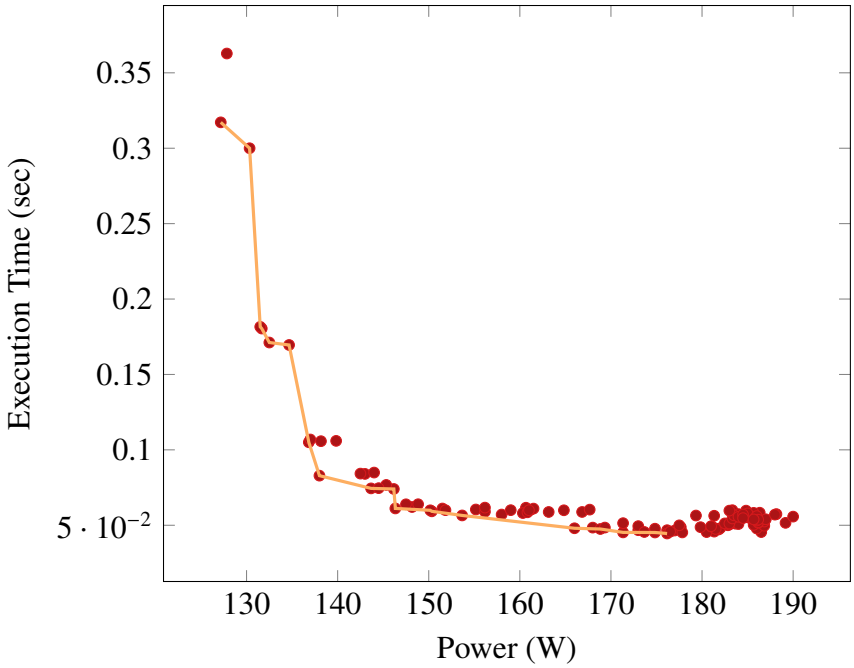


Figure 18: Task configuration points and resulting Pareto frontier for the `DPOTRF` task from Cholesky factorization.

4.2 Pareto-Optimal Scaling in Time and Power

Making scheduling decisions depends on the relationship between task configuration—in this case, number of processor cores—and the resulting performance. Existing theoretical models, such as Amdahl's Law, demonstrate a monotonically increasing relationship between number of cores and performance. However this may not always be the case due to the effects such as communication overhead. As a result, a parallel task may have higher performance with a smaller number of processors than larger. The number of cores a task uses may also change the amount of power consumed. That is, each task can be characterized by a set of points corresponding to the execution time and power consumed by the task for a given configuration. Figure 18 shows this for a single task; each point represents running the task with a specific number of processor cores.

From these figures it is possible to find configurations that achieve the same performance for different amounts of power, or conversely different performance for the same amount of power. The set of configurations that achieve the highest performance for a

given power consumption are called *Pareto-optimal* and lie on the *Pareto frontier*. In other words, a configuration is Pareto-optimal if it is not strictly dominated by any other. The line in Figure 18 is drawn through the Pareto frontier. By selecting a configuration on the Pareto frontier, the scheduler knows that it is choosing the best performing configuration for the given amount of power. This work proposes the offline construction of these frontiers for use during task scheduling, allowing the selection of the most power-efficient configurations.

4.2.1 Machine Model

This work examines the issue of scheduling under a power cap from the perspective of a single, homogeneous system, but may be adapted to distributed or heterogeneous systems. The task graphs used in this work exhibit an extremely high degree of parallelism both coarse-grained (task-level) and fine-grained (data-level). Due to this degree of expressed parallelism, this work leverages an Intel Xeon Phi coprocessor [69] to provide a homogeneous computing system with a large core count. The Intel Xeon Phi architecture allows for up to 72 in-order x86 cores connected by a ring interconnect. Each core can support up to four hardware threads. The instruction scheduler issues from each hardware thread in a round-robin fashion, but with the limitation that it is unable to issue from the same thread back-to-back. As a simplification, this work assumes each core is fungible. Rather than modeling the interactions between threads when they must compete for resources on a core, this work limits each core to two simultaneous threads, which can each provide a continuous instruction stream to the core.

Each core has its own L1 and L2 cache, with device-local memory and a large vector processing unit. Importantly, this architecture retains a traditional programming model similar to existing CPU workloads, allowing for compiler and runtime control over the degree of parallelism per task and across tasks. As such, existing workloads need little modification in order to run on these devices.

Table 5: Machine configuration used for experimental evaluation.

Component	Parameter	Value
Host	CPU	Intel i7-975
Host	CPU cores	4
Host	CPU clock rate	3.33 GHz
Host	Memory size	6GB
Coprocessor	CPU	Intel Xeon Phi 3120A
Coprocessor	CPU clock rate	1.10 GHz
Coprocessor	CPU cores	57
Coprocessor	Memory size	6GB

4.2.2 Frontier Construction

This section describes the experimental construction of the Pareto frontiers. It describes the execution environment as well as the structure of each task's Pareto frontiers used as the power and performance models to guide the power-aware scheduling decisions discussed in Chapter 5.

4.2.2.1 Experiment Setup

Table 5 shows the hardware configuration for the host and coprocessor card. To enable launching tasks asynchronously on the device, this work leverages the Intel Compiler's Heterogeneous Offload programming model. The main scheduling thread executes on the host, launching tasks to the device and checking on the execution progress. When a task needs to be launched, a new worker thread on the device is spun up, which then uses the `offload` compiler pragma to issue task execution to the coprocessor. The main scheduler then waits for this worker thread to complete. The offload framework takes an entire coprocessor to choreograph offload requests, so the total number of hardware threads a scheduler can assign tasks to at any time is 112.

This work leverages the `libmicmgmt` library provided by Intel which accesses Xeon Phi parameters [2] to poll the onboard power sensors. These measure the current and voltage for the device at a rate of $50Hz$. To collect the profiling data to construct the Pareto frontiers, each task graph was executed by a profiling scheduler that would repeatedly

launch a task for a set period and read the power of the coprocessor, recording the highest observed power and the average task execution time. This power number includes the idle power, measured as 106.9 Watts, for the coprocessor, which must be subtracted in order to determine the incremental power cost for launching a task.

For the validation experiments, the applications are executed by each scheduler and the execution times compared to the Fair Scheduler. Due to the coarse power measurement granularity, this work uses the values from the Pareto frontiers to represent the task power behavior.

4.2.2.2 Asynchronous Task Graph Applications

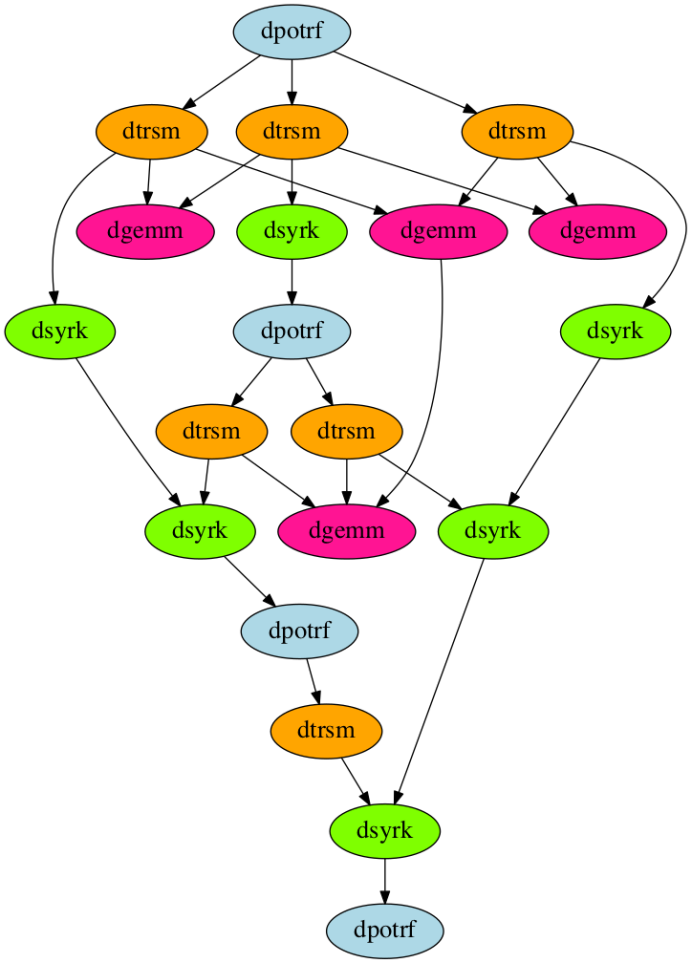


Figure 19: Task graph showing task dependencies for a 4×4 tile-base right-looking Cholesky decomposition.

The schedulers in this work are fed by a directed acyclic graph (DAG) of tasks. Instead of constructing synthetic tasks this work uses three different task graphs taken from current HPC workloads. Each application can be parameterized to the size of the input problem as well as the degree of tiling. The tasks from these applications are linear algebra kernels provided by the Intel MKL library [3]. Each task can be configured to use a different number of OpenMP threads to control their degree of data-level parallelism.

Conjugate Gradient (CG) The conjugate gradient example is adapted from the HPCG benchmark for solving a matrix equation of the form $Ax = b$ for a given sparse matrix A and vector b for unknown x. The task-based version is derived from the HPCG benchmark [29], using a compressed sparse row (CSR) representation of A. The sparse structure of the matrix A is derived from a structured 3D stencil with X-Y-Z linearized to form the column and row indices. This work assumes a +1/-1 connectivity in each dimension, resulting in a a total of 27 nonzeroes for each row (fewer for boundaries). The current example does not use a preconditioner. The individual kernels required for conjugate gradient are `DDOT` and `DAXPY` and a sparse matrix-vector multiply (`SpMV`).

Cholesky Decomposition The Cholesky decomposition calculates the decomposition of a positive-definite matrix A into the form $A = LL^*$ where the matrix L is a real lower-triangular matrix with positive elements in the diagonal. This uses a tiled-based right-looking algorithm, as described in [38]. Cholesky is done in double precision, requiring the BLAS kernels `DPOTRF`, `DSYRK`, `DTRSM`, and `DGEMM`. The task graph for a 4×4 tile-based Cholesky is shown in Figure 19.

LU Decomposition The Lower-Upper (LU) decomposition factorizes a matrix A into the form $A = LU$ where L is a lower triangular matrix and U is an upper triangular matrix. As with Cholesky, the LU Decomposition uses a tiled, right-looking algorithm [38]. The tasks for this application are `DTRSM`, `DGETRF`, and `DGEMM`.

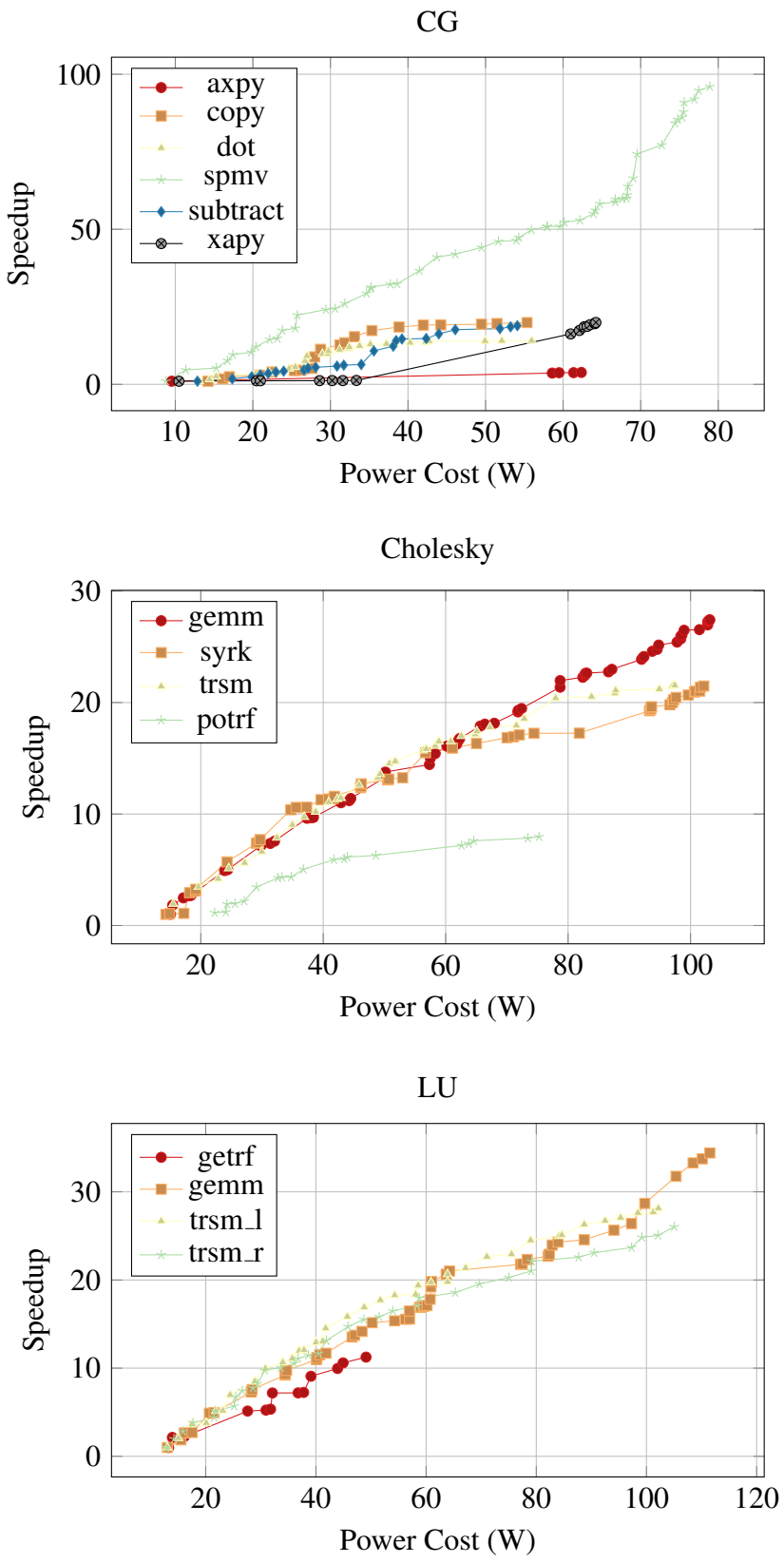


Figure 20: Pareto frontiers for each task in CG, Cholesky, and LU task graphs.

4.2.3 Power–Performance Pareto Frontiers

The use of Pareto frontiers to characterize application behavior are generalized into Pareto surfaces of multiple dimensions, where each dimension describes the behavior of one configuration parameter. For example, one dimensions may represent an application as it changes in response to the number of cores, while another considers the DVFS state. When more than one dimension exists, moving along the Pareto frontier involves walking along the gradient in this multidimensional space. This allows for examining multiple behaviors concurrently and reducing the configuration space into a more manageable set. The experimental validation in this section describes only the use of a single dimension, the number of cores available, but the procedure is identical as the number of dimensions increases.

Figure 20 shows the different Pareto frontiers for each task in the three task graphs examined in this work. The curves pass through the Pareto-optimal points that express the highest relative speedup over single-threaded execution versus power consumed over idle. Some tasks have a strong correlation between power and speedup, such as `SpMV` in CG and `DGEMM` in Cholesky and LU, as seen by the relatively straight shapes of the Pareto frontiers. On the other hand, tasks such as `DAXPY` in CG and `DPOTRF` in Cholesky have few Pareto-optimal points, which indicates poor performance scaling.

4.3 Concluding Remarks

In this chapter, a novel model for energy consumption as a program scales was presented. It builds upon the notion that energy is comprised of two parts: idle energy that is spent when a machine is powered on but idle, and dynamic energy, which is the cost to perform the work of an application. The first component is directly proportional to the time the machine is powered on, however the second cannot be reduced through parallelization. By characterizing the hardware platform as well as the application behavior, the model error can be reduced from 48.2% down to 3.3%.

As well, a new method for describing the trade-offs between performance and power

were presented, taking the form of Pareto frontiers. These frontiers reduce the set of configurations (i.e., the number of processors a task can use, or the frequency state) into only those that give the highest performance for a given power. The models will power the scheduling decisions described in the following chapter.

CHAPTER 5

POWER-CONSTRAINED PERFORMANCE SCHEDULING

Sea changes are imminent in both system hardware and software architecture as well as application design in response to the pressures imposed by this new emphasis on energy efficiency. On the system side, restrictive power budgets imply that it may be the case that not all architectural components may be utilized at their full capabilities simultaneously. In turn, fine-grained power allocation and measurement capabilities will allow system software and applications to closely monitor power consumption across the system and adapt the power distribution to characteristics of the executing workload. In other words, the parallel system may be used in a "throttled-down" configuration, or, alternatively, an asymmetric power distribution may be employed across the machine.

In response to these needs this thesis evaluates *Dynamic Power Steering* whose goal is to maximize inter-node performance within a fixed power budget and thereby optimizing energy consumption. Dynamic Power Steering routes power to resources (processor-cores) that are assigned the most work and thus lie along the performance-critical path of the application. Such situations arise in most applications that exhibit load-imbalance and that often require complex load balancing that approximately equalizes work across the system resources but at the cost of increasingly complex load-balancing operation and with extensive data movement. In order to keep the overall system within the prescribed power budget, power is diverted away from other resources in such a way as to not negatively impact overall performance. Of particular interest is the following open question: what application characteristics will enable Dynamic Power Steering approaches to be most successful? This work addresses that question through the analysis of load-imbalanced applications on a test bed server cluster.

On the application side, it is expected that applications will become increasingly more adaptive and asynchronous, varying over time and across the parallel system in ways that

are input dependent. The evolving nature of the ongoing physical simulations will lead to dynamic and input-dependent load imbalance, making static performance prediction and power allocation difficult and thus necessitating the need for dynamic mechanisms to respond to the evolving workload. Modern applications with irregular and unstructured control and data flows exhibit significant inefficiencies in the BSP model inspiring the emergence of the asynchronous task graph model for programming extreme scale applications to achieve new performance peaks [17]. However, this performance must be achieved subject to increasingly stringent power caps, as these systems will need to be designed to fit within the tightening constraints of power and scalability.

This work presents a set of scheduling strategies for task graphs that improve intra-node performance when running under a power cap. This work compares a static core-limiting system, analogous to the way systems are provisioned for the worst-case power behavior, with dynamic, power-aware schedulers that can flexibly decide when resources can be used so long as the power cap is not exceeded. This work also proposes the offline construction of Pareto-optimal power–performance profiles of data-parallel tasks, built from the work in Chapter 3, to guide the dynamic schedulers, providing a set of trade-offs between power consumption and the number of cores. This chapter will examine how leveraging program slack, an existing technique from the BSP model [15], performs in asynchronous models, showing how it is *ineffective* at reducing average power.

The main contributions described in this chapter are:

- Dynamic Power Steering, a system-wide approach to optimize performance within a fixed power budget and thereby optimize energy consumption.
- An exploration of Dynamic Power Steering for a range of application workload characteristics on a medium sized, 1K-processor-core system, that allows for the emulation of a constrained power-budget.
- A series of heuristics that leverage Pareto frontiers to drive energy-efficient scheduling decisions for asynchronous task graphs.

- An analysis of the scheduling heuristics for asynchronous task graphs, demonstrating performance improvements of on average 18.8%, 15.4%, and 38.4% better than the statically constrained schedulers when limited by the power cap for the CG, LU, and Cholesky applications and perform as well as the state-of-the-art when power is no longer restrictive.

5.1 Dynamic Power Steering For Bulk-Synchronous Applications

Dynamic Power Steering addresses a key challenge envisioned with future extreme scale systems: restrictive cluster-level power budgets will mean that not all system components may be fully utilized simultaneously. Combined with increased software complexities that will lead to adaptive and dynamic applications, whose behavior and resource requirements will evolve with simulation progression, will create the need for novel methodologies that will require the optimization of energy in order to extract maximum performance. The Dynamic Power Steering approach addresses these challenges by routing power to components across a system where it provides the most benefit in response to changing demands imposed by the executing workload.

Traditional techniques to optimize execution time typically involve dynamic load balancing, in which computation tasks or data are migrated from over-loaded processors to under-loaded ones in an attempt to dynamically smooth out variations in workload across a parallel system. Thus processors will step through the execution of an application at approximately the same rate. This often requires significant movement of data from one memory domain to another, a cost which increases with system scale. In addition, carefully constructed data distributions may be altered to the detriment of performance. In successful load balancing techniques, the cost (in terms of time) of both evaluating a load-balancing decision-making algorithm as well as data movement is less than the idle time lost to load imbalance and thus provides an overall reduction in runtime. However, the complexity for determining optimum (or even reasonable) balanced distributions, as well as determining

which data to migrate, is increasingly complex for irregular data such as that used in many adaptive applications.

Given the tight power constraints of future Exascale systems, there will be more resources available than can be actively powered at any one time. Thus there is a clear need for active power management to ensure that maximum power thresholds are not exceeded. The Dynamic Power Steering approach optimizes the power consumption in two primary ways:

1. Minimization of the power associated with data movement by eliminating the load-balancing of data between computation resources
2. Determining how power can be assigned to those resources that have more work to perform.

Dynamic Power Steering is most suited where the static calculation of an ideal power distribution is impossible such as to applications that are naturally load-imbalanced and whose load varies dynamically over time in an input-dependent manner. Further, applications whose performance is impacted by changes to the node or core p-state are most amenable to this approach; routing more power to over-loaded resources should have caused a significant improvement in performance.

Dynamic Power Steering results in a *power-optimized* system in which power is directed to the work being performed instead of to the data movement associated with load-balancing. This goes some way towards allowing applications to be tolerant of power constraints while still enabling the optimization of performance and thereby increasing energy efficiency.

5.1.1 Power Assignment to Processor Cores

Central to Dynamic Power Steering is the routing of power to processor-cores that are assigned more work. This can be achieved at a local-level by assigning each processor-core a suitable p-state that is in proportion to the amount of work whilst also satisfying the constraint of not exceeding a global-power budget. Determining the correct p-state to

use for each processor-core depends on several elements. The first is the notion of the global power budget, the system-wide quantity of power capable of being assigned to the system at any one point. Only the available power in the global power budget may be allocated across the system. By lowering the p-state of under-utilized cores, the savings in power draw may be reallocated to other processor-cores with more assigned work which improves the performance of those processor-cores that are in the performance critical-path. The processor-core with the highest work will always be the one limiting the performance of the application, due to all other cores waiting idle at a global synchronization. If enough power is freed by other cores to allow this processor-core to move to a higher p-state, overall execution time will improve.

Algorithm 2 Power assignment heuristic used by Dynamic Power Steering.

1: PWR_{max} = maximum globally available power
2: $p-state_{max}$ = fastest $p-state$
3: $N_{work\_max} = \max(N_{work\_i})\ \forall i \in \{P_i\}$
4: $t_{work\_max} = N_{work\_max} \times t_{work}(p-state_{max})$
5: $\forall i \in \{P_i\ | P_i \neq P_{work\_max}\}$ find slowest p-state such that $t_{work\_i} < t_{work\_max}$
6: $PWR_i = t_{work\_i}(p-state_i)$
7: $PWR_{global} = \sum_{i=0}^{p} PWR(p-state_i)$
8: If $PWR_{global} > PWR_{max}$ then reduce $p-state_{max}$ and repeat 3
9: Assign p-state calculated to each processor-core

The heuristic shown in Algorithm 2 is used in this work to select the p-state for all processor-cores in an iteration of an application. Before the processor-cores are released for the next compute iteration, the heuristic is run, under the assumption that the workload distribution for the current iteration is similar to the previous iteration. The maximum amount of work is calculated over all processors P_i (step 3), and its associated time cost using a performance model or empirical measurements (step 4). The *p-state* for all other processor-cores is calculated as being the slowest that does not impact the overall execution time (step 5). If the global power were to be exceeded then the *p-state* of the highest loaded processor is reduced and the assignment heuristic is repeated from step 3.

This algorithm optimizes the selection of p-states so that the most heavily loaded processor-core has the highest performance possible. Since applications are typically synchronized per iteration, the slowest processor-core is the limiting factor in the overall performance of the application.

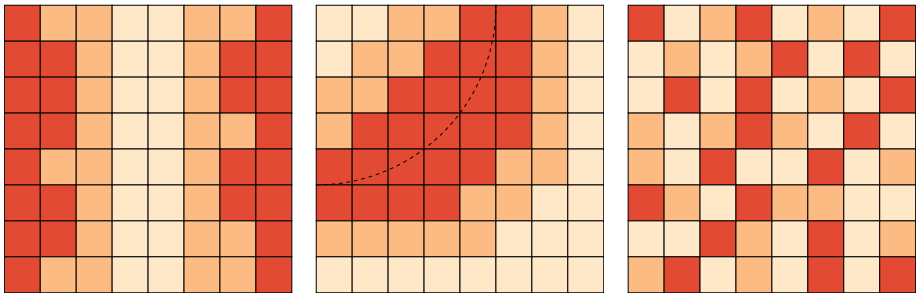
Most processor-cores will enter a polling state when they reach a global synchronization. This strategy reduces the latency of collective operations, but drastically increases processor utilization and by extension power. For this work an estimate is made for how long each iteration will require so that any processor-cores that reach the barrier early can be put to sleep, as below:

$$T_{sleep} = T_{MaxWork} - T_{LocalWork} \tag{7}$$

In addition, every processor-core enters the lowest-power p-state before sleeping. This ensures that all processor-cores are consuming as little power as possible while they wait to proceed. Lowering power consumption at collectives allows for exploring the maximum potential benefit of power steering.

5.1.2 Workload Case Studies

This work considers two types of workload that represent the underlying processing characteristics of many large-scale applications: the processing of a wavefront whose position varies over time, and a particle-in-cell processing for a charged field between two electrical plates. In addition a third random assignment of work across the processors in a parallel system is used as a control case. To explore the impact of Dynamic Power Steering, each of the workloads are kept as general as possible without reference to a particular application. Their processing characteristics are generated through a workload generator that allows for compute, load-imbalance as well as temporal aspects to be easily changed. One assumption is that the processing flow in each step of the workload consists of concurrent computation performed by each processor-core, followed by a global synchronization. This flow corresponds directly to most large-scale applications. Each processor-core determines its own



(a) Charged field. (b) Wavefront processing. (c) Random distribution.

Figure 21: Load levels for the three workloads on an 8×8 processor configuration. Color indicates relative processor load from light (low) to dark (high).

work assignment locally, which is then shared globally; *p-state* settings are determined based upon the local work in relation to the global work state to optimally assign power. *p-states* are then set according to each processor-cores assigned work, and the processing of the step begins.

Work is assigned based on the three workload distributions, is shown in Figure 21 for an 8x8 processor configuration; two represent patterns typically manifest in HPC workloads, while the third captures a control case. The first HPC pattern represents the density of charged particles in two-dimensional space placed between two charged plates. Due to the electric field caused by the charged plates, particles move over time toward the plates. Given that the global spatial domain is equally decomposed across the available, the initial load of randomly distributed particles is well balanced. However, as the simulation progresses and particles migrate towards the charged plates, the load becomes imbalanced. Figure 21a shows an example of this load imbalance; the darker the color indicates processor-cores with the highest concentration of particles. These processor-cores are the ones that would most benefit from an increase in performance afforded by an additional power allocation.

Rather than perform the actual calculations of this workload, an analytic model is used. As time approaches infinity, points will follow a Gaussian distribution in distance from the

closer of the two plates:

$$Position(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{8}$$

Work assignment to each processor-core is determined by weighing a sample from each distribution that puts more weight on the even distribution earlier in time and on the Gaussian distribution later in time, as in Equation 9. The *Weight* value monotonically increases with each application iteration.

$$Work = Weight \times Position(x, \mu, \sigma) + (1 - Weight) \times SteadyState \tag{9}$$

The second type of workload replicates the propagation of a wavefront through a two-dimensional space (Figure 21b). The wavefront begins in the upper-left corner of the two-dimensional global domain and expands with each subsequent iteration. As with the charge field, the space is partitioned along two dimensions with a single subdomain assigned to each available processor-core. The amount of work assigned to each core is inversely proportional to the distance of that core's subdomain from the wavefront according to the Gaussian distribution curve defined in Equation 8.

The final type of workload serves as a "control case" and assigns a random distribution of work across all processor-cores (Figure 21c). Note that this is not a random uniform distribution of particles, but is instead a random distribution of load across processor-cores, meaning that some are assigned proportionally higher or lower values of work in a random fashion.

The specific parameter space is controlled by two parameters. The first is termed *computational intensity* and reflects the ratio of computation to memory access within the sequential computation. The computational intensity is uniform across all processor-cores. The computational intensity is able to reflect the characteristics of a multitude of applications; as the value is increased more work is performed in the core. Performance of a compute-bound application is more sensitive to changes in core *p-state* than an application

that is memory bound. The second parameter is load-imbalance. The degree of load-imbalance across the system can be varied so as to again represent multiple applications. The lower the load-imbalance, the closer each processor-core's assigned work is to a set maximum per-core load value.

5.1.3 Experimental Setup

Since current systems are not power constrained, an artificial constraint system was imposed for these experiments. As described below, the power constraint is set to be the power use when setting all processor-cores' p-states to the mid-point of the available p-states. This allows for some cores to be allocated more power (when more heavily loaded), and at the same time for some cores to be allocated less power (when less loaded) with the same overall power budget.

This work utilized the PAL cluster located at Pacific Northwest National Laboratory (PNNL). This system contains power instrumented Power Distribution Unit (PDU) rails with which power consumption can be measured on a per-outlet basis at a frequency of 0.3 Hz. The cluster contains 144 nodes in total, each comprised of two sockets of AMD Opteron 6272 Interlagos processors with a maximum clock frequency of 2.1 GHz. Each socket consists of a dual-chip module with each chip containing four AMD Bulldozer dual-core modules (i.e., eight cores in total per chip). Because each dual-core Bulldozer module contains only a single floating-point unit, the experiments utilize a total of 16 of the available 32 cores per node, with one application-level rank per module. Each 8-module chip contains 8×64KB instruction caches, as well as 16×16KB L1 data cache, a 4×2MB L2 data cache, and a 2×8MB L3 data cache. Each node has 64GB DDR3 memory connected to the processors via a 3200MHz front-side bus. The Thermal Design Point (TDP) of each processor socket is 115W.

PAL is housed in three racks, with each rack containing 48 nodes physically organized as 12 quad-node units. Each quad-node contains two power supplies and is supplied by two outlets from the instrumented PDU rails. No direct power measurement of the InfiniBand

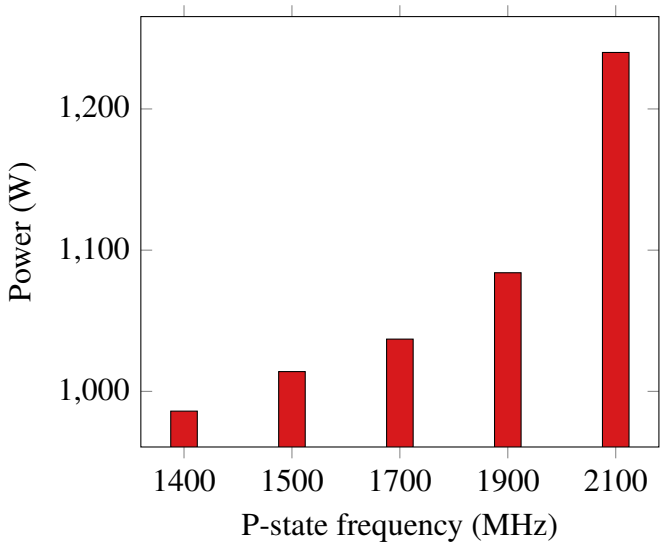


Figure 22: Measured power states for an AMD Interlagos quad-node on the PAL system running a stress microbenchmark at the given frequency state.

switches was performed. Monitoring and storage of the power use of all nodes was enabled at the system's normal operating frequency of 0.3 Hz. PAL's installed O/S was Red Hat Enterprise Linux version 5.7 distribution containing the Linux kernel version 2.6.32. Pathscale version 4.0.10 compilers were used, along with the OpenMPI version 1.5.4 MPI libraries.

DVFS is available on PAL at available clock frequencies of 2.1, 1.9, 1.7, 1.5, and 1.4 GHz. The frequency domain is a single dual-core Bulldozer module. A "baseline" power budget is assumed corresponding to all processor-cores set at a 1.7 GHz. This baseline defines the power budget which may not be exceeded across and enables improved performance on some cores by increasing the clock frequency to 2.1 GHz at the expense of requiring other cores to be reduced to 1.4 GHz to compensate.

In order to analyze the impact on power consumption caused by changing the processor-core *p-state*, the active power is measured under load. A characterization of the power consumption for a quad-node as a function of frequency is shown in Figure 22. It can be clearly seen that higher frequencies correspond to higher power draws, and also to shorter runtimes. However, the impact on runtime is dependent on the sensitivity of the benchmark

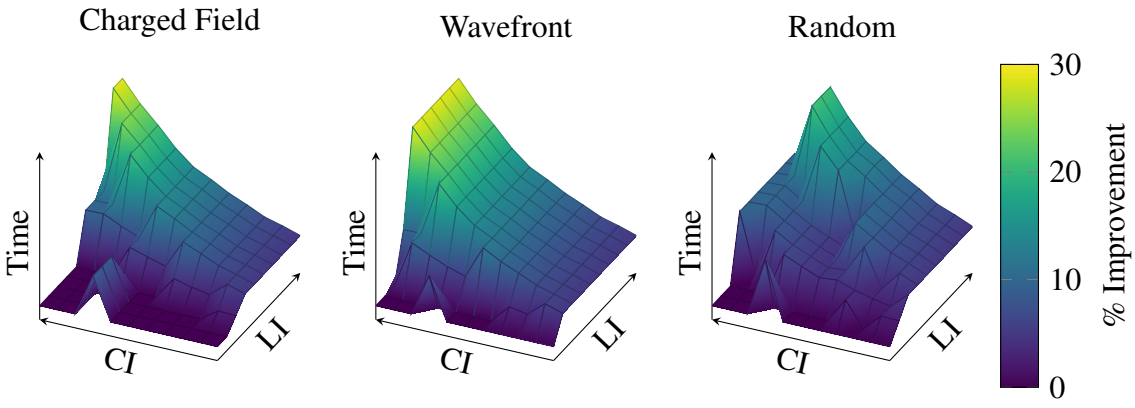


Figure 23: Relative runtime improvement for three workload types when using Dynamic Power Steering as a function of compute intensity and load imbalance.

to clock frequency.

5.1.4 Results and Discussion

A set of experiments were conducted using PAL and the three workload case studies. For all application configurations, 576 processor-cores were utilized across 36 nodes of the PAL cluster. Results are presented relative to executing the workloads without Dynamic Power Steering thus enabling its impact to be assessed. Positive values indicate improvement over the baseline and thus the advantage of using Dynamic Power Steering. Below the results of the experiments are presented and a discussion of the more general observations is provided.

Figure 23 indicates the improvement in runtime improvement when using Dynamic Power Steering for all three application configurations with varying degrees of load imbalance and compute intensity. For each graph, the *x-axis* denotes the level of computational intensity, on a scale from zero (entirely compute bound, executing in cache) to one (memory bound with very few arithmetic operations). The *y-axis* denotes the load-balance across processor-cores in the parallel system, again with zero indicating a high degree of load-imbalance and one indicating a perfect load-balance across all processor-cores.

In the case of the Charge Field synthetic workload a clear trend emerges. First, when using Dynamic Power Steering and allowing each of the processor core's *p-state* rto change

results in a greater change in performance when the workload exhibits a higher level of computational intensity. This is due to the fact that, on PAL, changing processor core *p-state* equates with altering clock frequency. Workloads that are bound by memory performance are sensitive to processor-core frequency, and thus less impacted by changes in *p-state*. Compute intensive workloads therefore benefit from increased clock frequency on those processor cores that are overloaded.

Second, the performance improvement increases with increased load-imbalance, although the effect is more dramatic than for increased computational intensity. One reason for this is the coarse granularity of the available *p-states* on the PAL system. The workload has to exhibit a fairly high level of imbalance before lowering a processor core's *p-state* does not place it onto the performance critical path, thereby increasing overall runtime.

It is important to note that the performance improvement for the Wavefront workload is non-zero even in cases of load balance. This is due to the load not being perfectly distributed due to work being distributed according to a Gaussian distribution for a given distance from the wavefront; however, the Gaussian distribution always assigns more work along the wavefront itself so that a degree of load imbalance always exists. Given a smooth load distribution, the performance improvement for a well-balanced load distribution would be negligible.

Although the Random load distribution distributes load levels across processor cores randomly, ranging from under-loaded to over-loaded load levels, there is still sufficient load imbalance across the system (in some configurations) to afford a performance improvement of over 25%. An notable side-effect of the Dynamic Power Steering methodology is that, by eliminating the data movement typically associated with dynamic load balancing, data locality is preserved to the greatest extent possible. In the case of the Random load distribution example, this is especially relevant as data may be required to migrate a great distance away from its originating processor core to evenly distribute the load.

Improved performance combined with nearly equivalent power consumption will result

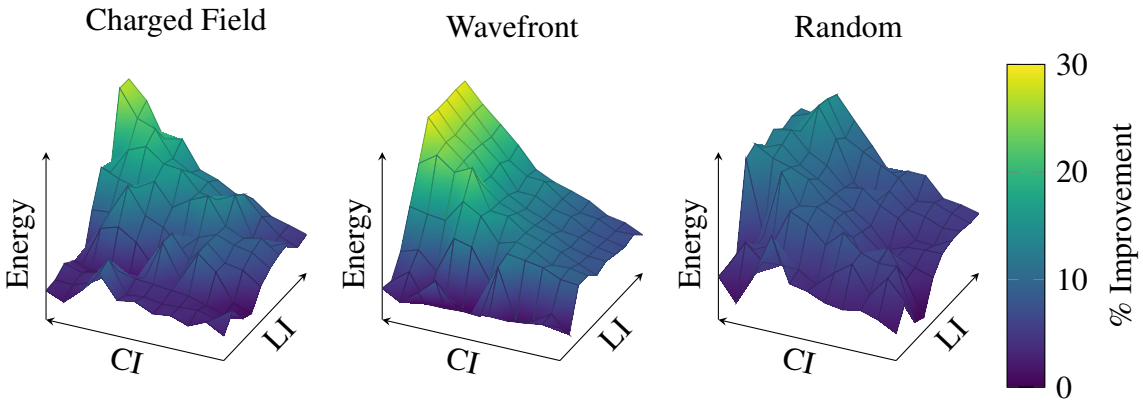


Figure 24: Relative energy improvement for three workload types when using Dynamic Power Steering as a function of compute intensity and load imbalance.

in improved energy efficiency, and that is indeed what is shown in Figure 24. For all three workload types, energy efficiency is improved; in the case of the Wavefront workload, energy efficiency is improved by nearly 45%, for the charged field by 27%, and for the random load by up to 20%. The greatest improvement in general corresponds to the cases of greatest compute intensity and load-imbalance. In future systems with more available *p-states* and a finer degree of control in power routing, the Dynamic Power Steering approach offers greater improvements.

5.2 Power-Constrained Scheduling for Asynchronous Task Graphs

This work examines applications built using the asynchronous task-graph programming model, where individual operations, called *tasks*, form the nodes in a directed acyclic graph (DAG) with the edges formally specifying the data dependencies between tasks. Figure 19 shows the task graph for Cholesky decomposition; a task in the graph is only allowed to execute once all of its predecessors have completed. A *runtime* monitors the progress of tasks as well as the utilization of system resources, such as the number of cores already assigned to tasks.

The design of the runtime must take into consideration the complicating issue of hierarchical levels of parallelism provided by the task graph. There is coarse-grained parallelism

for tasks that may execute concurrently provided there is no data-dependence between them. Tasks may have fine-grained data parallelism within a task so that a node within the DAG may be executed on more than one processor core. Such tasks have been termed *elastic* [74]. This trade-off between degrees of parallelism makes the scheduling decision to be made by the runtime more complicated, as it may chose between running multiple tasks or a single task on multiple processor cores.

5.2.1 Power-Efficient Scheduling

During execution, the runtime decides which tasks to run as well as the number of processor cores they may use. The system is provisioned with a certain number of cores that may be used globally across all tasks at any moment in time; the scheduler makes the decision as to how these cores get assigned, as well as which tasks to assign them to. As soon as a task is capable to execute, a call to the scheduling heuristic decides which tasks in the set of ready tasks to run, as well as the configuration to use for each of these tasks. This work presents four different scheduling heuristics to demonstrate the ability to increase performance while operating under a power cap, with each heuristic providing a successively more comprehensive approach to minimize execution time of the task graphs while remaining below the power limit. This work builds upon the Pareto frontier construction from the previous Chapter; here, the Pareto frontiers serve as the mechanism to characterize performance–power trade-offs that the heuristics must make, ensuring that the highest performing configuration (such as number of cores or clock frequency state) is selected for a given amount of power.

5.2.2 Scheduling Heuristics

This work examines four different scheduling heuristics, described below. The input to each heuristic is the set of tasks that are ready to run, as well as the set of available resources, which includes the amount of power left in the budget, as well as the current number of unassigned processor cores.

5.2.2.1 Fair Scheduler (FS)

Algorithm 3 Fair Scheduler

1: $numCores$ = getAvailCores()
2: $numTasks$ = size($pendingTasks$)
3: $coresPerTask = numCores/numTasks$
4: Assign $coresPerTask$ to each Task in $pendingTasks$

The Fair Scheduler (Algorithm 3) is the most simple heuristic which assigns cores in a round-robin manner to all of the ready-to-execute tasks. For this heuristic, the goal is to maximize utilization of the cores with no regard given to the power consumed during execution. Rather, the power utilization will be enforced statically by selecting the highest number of total cores that results in a peak power consumption that does not exceed the power cap. This is analogous to provisioning the scheduler for the worst case, preventing any cores from being used if they will, at any time, go over the power cap. This scheduler forms the baseline for this work.

5.2.2.2 Core Constrained Scheduler (CCS)

Algorithm 4 Core Constrained Scheduler

1: Set each task to highest performance configuration
2: **while** $\sum_{i=0}^{numTasks} cores_i > numIdleCores$ **do**
3: **for each** Pending task T **do**
4: Determine makespan with $cores_T = cores_T - 1$
5: **if** Makespan is largest seen yet **then**
6: Set T as losing task
7: **end if**
8: **end for**
9: Reduce threads for losing task
10: **end while**

The second scheduler is called the Core Constrained Scheduler (4). This heuristic, adapted from [14], attempts to minimize execution time of the task graph by assigning cores to the tasks that lie on the critical path. To achieve this, an estimate must be made of the total time to complete a specific task plus the time to complete the longest chain of tasks to the bottom of the graph, called the *makespan*. To calculate the makespan, the scheduler

assumes that all subsequent tasks after the current one to be scheduled will use the highest performance configuration possible.

As with the Fair Scheduler, this scheduler does not consider power consumption to make its scheduling decisions. Instead, a similar static limitation will be placed on the number of idle cores to ensure that at no time it goes over the power cap. This represents the current state of the art in achieving high performance, but it is not cognizant of the power cap. The goal is to examine how well these heuristics manage when power limitations are put into place.

5.2.2.3 Power Aware Scheduler (PAS)

Algorithm 5 Power Aware Scheduler

```
1: Set each task to highest performance Pareto point
2: while ∑ cores_i > numIdleCores or ∑ power(cores_i) + currentPower > powerCap do
3:   for each Pending task T do
4:     Move to next less powerful Pareto-optimal point
5:     Determine the new makespan
6:     if Makespan is shortest seen yet then
7:       Set T as losing task
8:     end if
9:   end for
10:  Walk down Pareto frontier for losing task
11: end while
```

The Power Aware Scheduler (5) is the first scheduler that leverages the Pareto frontiers to model the power behaviors of the executing tasks, providing a mechanism for dynamically tuning the power behavior of the system. Instead of statically limiting the number of idle cores available to the scheduler, this heuristic is aware of the power cap, as well as the amount of power used by each task. The insight is that, informed by the Pareto frontiers, this heuristic is able to exceed the number of cores in the prior two schedulers so long as it guarantees that it remains under the power cap. In this way, it is able to accelerate some tasks by giving them more cores, reducing the makespan of the graph, so long as there is power available.

The key insight for this scheduler is that power can be attributed dynamically, allowing tasks to leverage cores so long as, for the currently scheduling tick, the operation remains under the power cap. In practice, this results in a higher average core utilization than the static schedulers, as the scheduler can assign more cores to tasks than the static limit, so long as there is power headroom.

5.2.2.4 Slack Aware Scheduler (SAS)

Algorithm 6 Slack Aware Scheduler

```
1: Call PAS
2: for each Pending task T do
3:   if T is not on critical path then
4:     while Slack ≠ 0 do
5:       Move to next less powerful Pareto-optimal point
6:     end while
7:   end if
8: end for
```

The Slack Aware Scheduler (6) is an improvement over PAS. The key insight for this heuristic is that a task may be slowed down so long as it does not affect the critical path of the graph. The additional amount of time a task can run before affecting the schedule is called *slack* [50]. This scheduler operates like the PAS, but with the additional step that allows tasks to move into lower power configurations so long as there is slack. The scheduler determines if there is slack by slowing down the thread and recalculating the graph's makespan; there is slack if the makespan does not change. The goal for this scheduler is to achieve the same performance as the PAS, but with a lower average power. This builds off of existing techniques for achieving energy efficiency from slack [15], but expands their use to asynchronous task graphs.

5.2.3 Results

Figure 25 shows the performance of each scheduler relative to the performance of the Fair Scheduler, as a function of an imposed power cap. The power cap starts at 35 Watts, which is guaranteed to contain at least one Pareto-optimal point for each task. The CCS performs

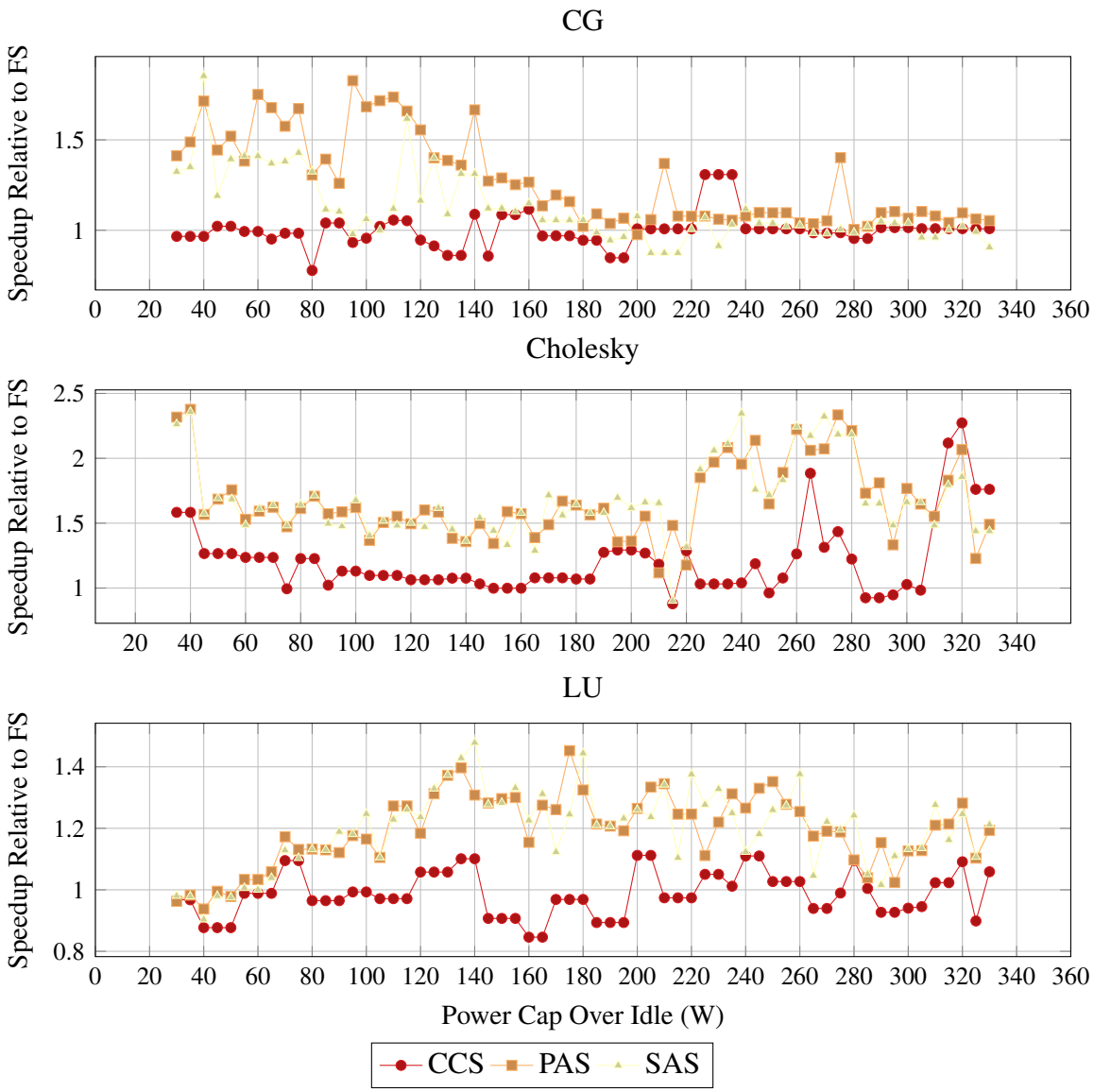


Figure 25: Relative performance of the CCS, PAS, and SAS schedulers over the Fair Scheduler for the three task graphs as a function of the power cap.

similarly to the baseline. For a fixed number of cores, CCS has a higher performance, but also a higher power consumption. When the power cap is imposed, the CCS must use fewer total cores than the FS, even though it uses them more efficiently.

However, the two power-aware schedulers, due to their dynamic nature, are more power-efficient than the baseline and the CCS. For Cholesky and LU, the PAS and SAS schedulers perform better than FS and CCS across all power caps. This becomes less pronounced as the power cap increases for CG in particular; at this level of power cap, the FS and CCS are not statically restricted by the number of cores they can use. That is, they can use all of the cores available and still remain underneath the cap. Compared to Cholesky and LU, the tasks in CG scale poorly (except for `SpMV`), so that even though there is more headroom under the power cap, the tasks would not exhibit a speedup.

An important observation is of the parabolic shape to the performance improvement of PAS and SAS over both the baseline and CCS, seen most clearly with LU: when the power cap is low, none of the schedulers perform well since very few cores can be used. There is a middle region where the dynamic, power-aware schedulers achieve the greatest speedup attributed to their higher average core utilization. Then finally, the power cap gets high enough to allow all cores to be used, even for the static schedulers, so performance normalizes. This underscores the need to understand what power caps ought to be feasibly set at from the perspective of applications, possibly across phases. Certain power cap–application relationships can elicit the greatest improvements in efficiency and others elicit little.

The slack-aware scheduler performs comparably to the power-aware scheduler. However, SAS has an average power of only 0.43% less than PAS for Cholesky and has an increase in power of 0.15% and 1.2% for CG and LU, respectively. This goes against the hypothesis that taking advantage of slack can lower average power while retaining performance. The reason for this behavior can be attributed to the difference between the asynchronous programming model and the bulk-synchronous model (that can leverage slack).

Tasks that slow themselves down free up cores for use in subsequent scheduling ticks. The schedulers only look at the current set of tasks to be scheduled. This may not include the tasks that lie on the critical path of the overall graph; still, the scheduler minimizes the distance between the current tasks to be scheduled and the end of the graph, using up the power headroom as necessary. In the end, the limiting task is run at a higher power, but does not affect the execution time of the graph. This results in a similar or higher average power. A more intelligent scheduler that can determine the true execution-limiting tasks may be better informed to leverage any slack during scheduling.

For this work, the Pareto frontiers are constructed from profiling runs, using the average execution time and peak power. However, tasks such as `SpMV` are dependent upon the input data. As well, the interactions between the tasks is not modeled, such as the behavior of scheduling more than two hardware threads per core. While this work does not attempt to model these behaviors, they would provide greater accuracy for the schedulers, potentially increasing the power-efficiency and performance of the graphs.

5.3 Concluding Remarks

This chapter presented methods for improving the performance of applications when they are constrained by power limitations. The focus was on two types of applications: bulk-synchronous parallel applications running on clusters and asynchronous task graphs on a highly-parallel shared memory multicore processor.

The first approach is Dynamic Power Steering, a method by which performance and energy efficiency of load-imbalanced applications may be optimized on power-constrained large-scale systems. In contrast with traditionally dynamic load balancing, data migration is eliminated, conserving the power expenditure associated with data movement. Instead, power is dynamically routed to processing resources with the greatest load allowing them to execute at a higher rate of performance. To compensate, power is routed away from under-loaded resources, thereby minimizing idle-time. This method is shown for three synthetic

work loads, demonstrating an average performance improvement of greater than 11% and an average improvement in energy efficiency of 13%.

The second contribution is the use of performance–power Pareto frontiers to inform the scheduling heuristics for asynchronous task graph. This work shows four scheduling heuristics used by the runtime to determine the configuration state for each task: a Fair Scheduler that assigns cores in a round-robin fashion, a Core-Constrained Scheduler that attempts to minimize the distance from the current tasks to the end of the graph, a Power-Aware Scheduler that leverages the Pareto frontiers to pick the tasks to best utilize the available power budget, and a Slack-Aware Scheduler that attempts to slow down those tasks that are not on the critical path, reducing average power consumption. An experimental validation of Conjugate Gradient, Cholesky Decomposition, and LU Decomposition shows how the dynamic, power-aware schedulers are able to increase performance while remaining underneath a power cap.

CHAPTER 6

CONCLUSION

This dissertation concerns itself with addressing the problem of modeling and analyzing applications for power-constrained, extreme scale systems. While these systems get larger, the amount of power they may draw at one time is changing very little. This places immense pressure on ensuring that the design is energy-efficient. To combat this pressure, the hardware is being designed to expose unprecedented degrees of parallelism and complexity, necessitating new programming models. However, efficient hardware is not enough to reach exascale performance goals [11].

A new technique called *codesign* presents a design scheme where the hardware and software communities work together, communicating design trade-offs quickly to ensure convergence on a high performance and energy optimal system. This sets the framework for the problem statement addressed by this thesis: *performance optimization of extreme-scale systems that operate within power limitations require new techniques for understanding the relationships between platform characterization, performance, and energy consumption as a function of the applications being executed.*

This thesis addresses this problem through three main contributions:

1. Modeling execution time and energy for regions within applications.
2. Modeling energy scaling and the trade-off between performance and power.
3. Power-constrained performance scheduling of synchronous and asynchronous applications.

The first topic is comprised of a semi-automated method for constructing statistical models of performance and energy and composing them in *application skeletons* for fast simulations of applications at scale. This technique learns the relationships between application-level parameters and time and energy behaviors on the target platform with instrumentation taken from any of a wide variety of sources. This work showed how this methodology can

accelerate system simulation. While this research provides a method for predicting application performance on very large systems, no significant large-scale validation efforts have taken place, which is a future avenue of research that would greatly increase the appeal of these techniques. As well, there is a growing issue with the modeling of input-dependent applications; the execution behaviors of these applications evolve as the application runs, making performance estimates more challenging. Coming up with methods for describing the data-dependent changes in the application is also a ripe area for future research.

The second topic addresses the construction of an analytical model of energy consumption for an application as the number of cores increases. This model leverages the fact that energy, unlike time, cannot be "overlapped" by running in parallel. Decomposing total energy into the idle fraction, that scales with the time the system is running, and dynamic fraction, that is invariant across scale, provides the basis for the model. This method for comparing an application's *configuration* (e.g., the number of cores, or the frequency state) to its time and power behavior is then generalized into a *Pareto frontier*, which delineates the minimal set of configurations that achieve the highest performance for a given amount of power. For this work, the tasks are characterized in isolation to construct their Pareto frontiers. An open research question is if these frontiers retain their accuracy once applications begin executing concurrently on the same system. It seems likely that there are adverse reactions between tasks, such as memory bandwidth contention or thermal effects, that may not be visible when characterizing the tasks independently. This opens up new areas of research, which can potentially build upon existing practices such as pertains to Quality of Service and ensuring fair utilization of resources.

Finally the third topic examines the performance scheduling of applications both across a cluster and within a node when there is a power cap. This work presented a method called *Dynamic Power Steering* for bulk-synchronous parallel applications. By allowing nodes with low workload to clock down, the power can be assigned to high workload tasks, minimizing the penalty of load imbalance. This method is demonstrated to increase

performance and decrease energy consumption when clusters are power-constrained. As well, this third topic leveraged the Pareto frontiers from the second topic to ensure that tasks from asynchronous task graphs are scheduled in a way that increases performance when constrained by power. The work on these runtimes examined the best-case scenario for the use of the presented heuristics. A future area of potential research could examine the overheads involved with choreographing nodes on a cluster, when using the Dynamic Power Steering technique. As well, each of the tasks in the graphs examined had their own Pareto frontiers constructed a priori. An open research question is if these frontiers can be generalized by task classification, so that less work would need to be done for preparing a task graph for execution. Should a method for classification arise, it would also seem possible to perform the classification on-line, so that a new graph can be executed and the correct configuration settings be determined dynamically based on data collected during runtime.

By examining these topics, this thesis sets the framework for understanding the energy behaviors of applications on extreme scale systems. The evaluation presented within this work helps guide the design of the next generation of applications and systems, ensuring that they can reach their performance goals.

This dissertation opens the door to future work in increasing the quality of energy models as well as their application to solving design problems. New techniques in machine learning will allow for more robust and expressive statistical models, which can be used to augment application skeletons. The effect of parallelization on workload may become more nuanced in multicore systems where resources are shared. Input-dependent applications will require methods for characterizing when performance and energy consumption will change with the input. Hopefully the work in this thesis helps to lay the foundation for these types of studies.

REFERENCES

[1] "Intel®Xeon®Processor E5-1600/ E5-2600/E5-4600 Product Families Datasheet - Volume One." http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-1-datasheet.html, May 2012.

[2] *Intel Xeon Phi Coprocessor System Software Developers Guide*. Mar. 2014. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[3] *Intel Math Kernel Library Developer Reference*. Aug. 2015. https://software.intel.com/en-us/articles/mkl-reference-manual.

[4] Adve, V. S., Bagrodia, R., Browne, J. C., Deelman, E., Dube, A., Houstis, E. N., Rice, J. R., Sakellariou, R., Sundaram-Stukel, D. J., Teller, P. J., and Vernon, M. K., "POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 1027–1048, Nov. 2000. http://dx.doi.org/10.1109/32.881716.

[5] Adve, V. S., Bagrodia, R., Deelman, E., and Sakellariou, R., "Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures," *Journal of Parallel and Distributed Computing*, vol. 62, no. 3, pp. 393 – 426, 2002. http://www.sciencedirect.com/science/article/B6WKJ-45F4YNW-2W/2/30693cc41f5423a5f6e3eb566ada9998.

[6] AMD, *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*. AMD, Inc., Feb. 2015.

[7] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967. http://doi.acm.org/10.1145/1465482.1465560.

[8] Anger, E., "Understanding Performance–Energy Tradeoffs: When Is Energy ≠ Time?," Aug. 2015.

[9] Anger, E., Dechev, D., Hendry, G., Wilke, J., and Yalamanchili, S., "Application Modeling for Scalable Simulation of Massively Parallel Systems," in *2015 IEEE International Conference on High Performance Computing and Communications (HPCC)*, Aug. 2015.

[10] Anger, E., Wilke, J., and Yalamanchili, S., "Power-Constrained Performance Scheduling of Data Parallel Tasks," in *Energy Efficient Supercomputing Workshop (E2SC), 2016*, Nov. 2016.

[11] Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., Dongarra, J., Kothe, D., Lusk, R., Messina, P., Mezzacappa, T., Moin, P., Norman, M., Rosner, R., Sarkar, V., Siegel, A., Streitz, F., White, A., and Wright, M., "The Opportunities and Challenges of Exascale Computing—Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee," *US Department of Energy Office of Science*, Fall 2010.

[12] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K., "The NAS parallel benchmarks–summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 158–165, ACM, 1991. http://doi.acm.org/10.1145/125826.125925.

[13] Bailey, P., Lowenthal, D., Ravi, V., Rountree, B., Schulz, M., and de Supinski, B., "Adaptive Configuration Selection for Power-Constrained Heterogeneous Systems," in *2014 43rd International Conference on Parallel Processing (ICPP)*, pp. 371–380, Sept. 2014.

[14] Barbosa, J., Morais, C., Nobrega, R., and Monteiro, A. P., "Static scheduling of dependent parallel tasks on heterogeneous clusters," in *Cluster Computing, 2005. IEEE International*, pp. 1–8, Sept. 2005.

[15] Barker, K. J., Kerbyson, D. J., and Anger, E., "On the Feasibility of Dynamic Power Steering," in *Energy Efficient Supercomputing Workshop (E2SC), 2014*, pp. 60–69, Nov. 2014.

[16] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J., "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, pp. 1432–1441, May 2011.

[17] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J., "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, pp. 1151–1158, May 2011.

[18] Bourdon, A., Noureddine, A., Rouvoy, R., and Seinturier, L., "Powerapi: A software library to monitor the energy consumed at the processlevel," *ERCIM News*, vol. 2013, no. 92, 2013.

[19] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J., "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, Jan. 2009. http://www.sciencedirect.com/science/article/pii/S0167819108001117.

[20] Casanova, H., Legrand, A., and Quinson, M., "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in Iccms 2008: 10th Int. Conf. On Comput. Model. Simul., pp. 126–131, 2008.

[21] Cassidy, A. and Andreou, A., "Beyond Amdahl's Law: An Objective Function That Links Multiprocessor Performance Gains to Delay and Energy," *IEEE Transactions on Computers*, vol. 61, pp. 1110–1126, Aug. 2012.

[22] Cho, S. and Melhem, R., "On the Interplay of Parallelization, Program Performance, and Energy Consumption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 342–353, Mar. 2010.

[23] Choi, J. W., Bedard, D., Fowler, R., and Vuduc, R., "A Roofline Model of Energy," in *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 661–672, May 2013.

[24] Cook, J., Cook, J., and Alkohlani, W., "A statistical performance model of the opteron processor," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 75–80, Mar. 2011. http://doi.acm.org.prx.library.gatech.edu/10.1145/1964218.1964231.

[25] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T., "LogP: Towards a Realistic Model of Parallel Computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, (New York, NY, USA), pp. 1–12, ACM, 1993. http://doi.acm.org/10.1145/155332.155333.

[26] Curtis-Maury, M., Singh, K., McKee, S., Blagojevic, F., Nikolopoulos, D., de Supinski, B., and Schulz, M., "Identifying energy-efficient concurrency levels using machine learning," in *2007 IEEE International Conference on Cluster Computing*, pp. 488–495, Sept. 2007.

[27] Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M., "Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 250–259, ACM, 2008. http://doi.acm.org/10.1145/1454115.1454151.

[28] Dagum, L. and Menon, R., "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[29] Dongarra, J. and Heroux, M. A., "Toward a New Metric for Ranking High Performance Computing Systems," in *Toward a New Metric for Ranking High Performance Computing Systems*, 2013.

[30] Dongarra, J., Hey, T., and Strohmaier, E., "PARKBENCH: Methodology, Relations and Results," in *HPCN Europe*, pp. 770–777, 1996.

[31] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011. http://doi.acm.org/10.1145/2000064.2000108.

[32] Etinski, M., Corbalan, J., Labarta, J., and Valero, M., "Parallel job scheduling for power constrained HPC systems," *Parallel Computing*, vol. 38, pp. 615–630, Dec. 2012. http://www.sciencedirect.com/science/article/pii/S0167819112000610.

[33] Farkas, K. L., Chow, P., Jouppi, N. P., and Vranesic, Z., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," pp. 149–159, 1997.

[34] Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., and Cameron, K., "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 658–671, May 2010.

[35] Genbrugge, D. and Eeckhout, L., "Chip Multiprocessor Design Space Exploration through Statistical Simulation," *IEEE Transactions on Computers*, vol. 58, pp. 1668–1681, Dec. 2009.

[36] Gerbessiotis, A. V. and Valiant, L. G., "Direct Bulk-Synchronous Parallel Algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 251–267, Aug. 1994. http://www.sciencedirect.com/science/article/pii/S0743731584710859.

[37] Goswami, N., Shankar, R., Joshi, M., and Li, T., "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, 2010.

[38] Haidar, A., Ltaief, H., YarKhan, A., and Dongarra, J., "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," Concurr. Comput. : Pract. Exp., vol. 24, pp. 305–321, 2011.

[39] Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W., "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[40] Hoisie, A., Johnson, G., Kerbyson, D. J., Lang, M., and Pakin, S., "A Performance Comparison Through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple," in *SC '06: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2006.

[41] Hong, S. and Kim, H., "An integrated GPU power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 280–289, ACM, 2010.

[42] Hoste, K. and Eeckhout, L., "Microarchitecture-Independent Workload Characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, 2007.

[43] Ipek, E., Supinski, B. R. D., Schulz, M., and Mckee, S. A., "An approach to performance prediction for parallel applications," in *Euro-Par, Springer LNCS*, p. 2005, 2005.

[44] Janssen, C. L., Adalsteinsson, H., Cranford, S., Kenny, J. P., Pinar, A., Evensky, D. A., and Mayo, J., "A Simulator for Large-scale Parallel Architectures," *International Journal of Parallel and Distributed Systems*, vol. 1, no. 2, pp. 57–73, 2010.

[45] Janssen, C. L., Adalsteinsson, H., and Kenny, J. P., "Using simulation to design extremescale applications and architectures: Programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, Mar. 2011. http://doi.acm.org/10.1145/1964218.1964220.

[46] Jia, W., Shaw, K., and Martonosi, M., "Stargazer: Automated regression-based GPU design space exploration," in *2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 2–13, 2012.

[47] Jolliffe, I. T., *Principal Component Analysis*. Springer Series in Statistics, 2 ed., 2002.

[48] Jolliffe, I. T., "A Note on the Use of Principal Components in Regression," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 31, no. 3, pp. pp. 300–303, 1982. http://www.jstor.org/stable/2348005.

[49] Kandalla, K., Mancini, E., Sur, S., and Panda, D., "Designing Power-Aware Collective Communication Algorithms for InfiniBand Clusters," in *2010 39th International Conference on Parallel Processing (ICPP)*, pp. 218–227, Sept. 2010.

[50] Kappiah, N., Freeh, V. W., and Lowenthal, D., "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005.

[51] Karlin, I., Keasler, J., and Neely, R., "LULESH 2.0 Updates and Changes," Tech. Rep. LLNL-TR-641973, Lawrence Livermore National Laboratory, Livermore, CA, Aug. 2013.

[52] Kerbyson, D., Vishnu, A., and Barker, K., "Energy Templates: Exploiting Application Information to Save Energy," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 225–233, 2011.

[53] Kerr, A., Anger, E., Hendry, G., and Yalamanchili, S., "Eiger: A framework for the automated synthesis of statistical performance models," in *2012 19th International Conference on High Performance Computing (HiPC)*, pp. 1–6, 2012.

[54] Kerr, A., Diamos, G., and Yalamanchili, S., "Modeling GPU-CPU Workloads and Systems," in *Third Workshop on General-Purpose Computation on Graphics Procesing Units*, (Pittsburg, PA, USA), Mar. 2010.

[55] Kutner, M. H., Nachtsheim, C. J., and Neter, J., *Applied Linear Regression Models*. McGraw-Hill/Irwin, fourth international ed., Sept. 2004. Published: hardcover.

[56] Li, D., de Supinski, B., Schulz, M., Cameron, K., and Nikolopoulos, D., "Hybrid MPI/OpenMP power-aware computing," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, Apr. 2010.

[57] Li, D., de Supinski, B., Schulz, M., Nikolopoulos, D., and Cameron, K., "Strategies for Energy-Efficient Resource Management of Hybrid Programming Models," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 144–157, Jan. 2013.

[58] Liu, C., Sivasubramaniam, A., Kandemir, M., and Irwin, M., "Exploiting Barriers to Optimize Power Consumption of CMPs," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, Apr. 2005.

[59] Martin, S. and Kappel, M., "Cray XC30 Power Monitoring and Management," *Proceedings of CUG*, 2014.

[60] McCraw, H., Ralph, J., Danalis, A., and Dongarra, J., "Power monitoring with PAPI for extreme scale architectures and dataflow-based programming models," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 385–391, Sept. 2014.

[61] Mucci, P. J., Browne, S., Deane, C., and Ho, G., "PAPI: A Portable Interface to Hardware Performance Counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.

[62] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40–53, Mar. 2008. http://doi.acm.org/10.1145/1365490.1365500.

[63] NVIDIA, *NVML API Reference Manual*. NVIDIA Corporation, Apr. 2012.

[64] Patki, T., Lowenthal, D. K., Sasidharan, A., Maiterth, M., Rountree, B. L., Schulz, M., and de Supinski, B. R., "Practical Resource Management in Power-Constrained, High Performance Computing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, (New York, NY, USA), pp. 121–132, ACM, 2015. http://doi.acm.org/10.1145/2749246.2749262.

[65] Plimpton, S., "Fast Parallel Algorithms for Short-range Molecular Dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, Mar. 1995. http://dx.doi.org/10.1006/jcph.1995.1039.

[66] Prakash, S., Deelman, E., and Bagrodia, R., "Asynchronous Parallel Simulation of Parallel Programs," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 385–400, 2000.

[67] Preissl, R., Köckerbauer, T., Schulz, M., Kranzlmüller, D., de Supinski, B. R., and Quinlan, D. J., "Detecting Patterns in MPI Communication Traces," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, (Washington, DC, USA), pp. 230–237, IEEE Computer Society, 2008. http://dx.doi.org/10.1109/ICPP.2008.71.

[68] R Core Team, *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2013. http://www.R-project.org/.

[69] Rahman, R., *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Berkely, CA, USA: Apress, 1st ed., 2013.

[70] Riesen, R., "A Hybrid MPI Simulator," in *IEEE Inter. Conf. on Cluster Computing 2006*, pp. 1–9, Sept. 2006.

[71] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., and Jacob, B., "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, Mar. 2011. http://doi.acm.org/10.1145/1964218.1964225.

[72] Rountree, B., Lownenthal, D. K., de Supinski, B. R., Schulz, M., Freeh, V. W., and Bletsch, T., "Adagio: Making DVS Practical for Complex HPC Applications," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 460–469, ACM, 2009. http://doi.acm.org/10.1145/1542275.1542340.

[73] Satish, N., Kim, C., Chhugani, J., and Dubey, P., "Large-scale Energy-efficient Graph Traversal: A Path to Efficient Data-intensive Supercomputing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 14:1–14:11, IEEE Computer Society Press, 2012. http://dl.acm.org/citation.cfm?id=2388996.2389015.

[74] Sbirlea, A., Agrawal, K., and Sarkar, V., "Elastic Tasks: Unifying Task Parallelism and SPMD Parallelism with an Adaptive Runtime," in *Euro-Par 2015: Intl. Conference on Parallel and Distrib. Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[75] Shalf, J., Quinlan, D., and Janssen, C., "Rethinking Hardware-Software Codesign for Exascale Systems," *IEEE Computer*, vol. 44, pp. 22–30, Nov. 2011.

[76] Song, W., Yalamanchili, S., Mukhopadhyay, S., and Rodrigues, A., "Instruction-Based Energy Estimation Methodology for Asymmetric Manycore Processor Simulations," ACM, 2012. http://eudl.eu/doi/10.4108/icst.simutools.2012.247770.

[77] Sottile, M., Dakshinamurthy, A., Hendry, G., and Dechev, D., "Semi-Automatic Extraction of Software Skeletons for Benchmarking Large-Scale Parallel Applications," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM PADS)*, (Montreal, Canada), May 2013.

[78] Spafford, K. L. and Vetter, J. S., "Aspen: A Domain Specific Language for Performance Modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 84:1–84:11, IEEE Computer Society Press, 2012. http://dl.acm.org/citation.cfm?id=2388996.2389110.

[79] Sridharan, S., Gupta, G., and Sohi, G. S., "Holistic Run-time Parallelism Management for Time and Energy Efficiency," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 337–348, ACM, 2013. http://doi.acm.org/10.1145/2464996.2465016.

[80] Stevens, R. and White, A., *Scientific Grand Challenges: Architectures and Technology for Extreme Scale Computing*. 2009. http://extremecomputing.labworks.org/hardware/reports/FINAL_Arch&TechExtremeScale1-28-11.pdf.

[81] Subramaniam, B. and Feng, W.-c., "Statistical Power and Performance Modeling for Optimizing the Energy Efficiency of Scientific Computing," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 139–146, 2010.

[82] Tallent, N. R. and Hoisie, A., "Palm: Easing the Burden of Analytical Performance Modeling," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), pp. 221–230, ACM, 2014. http://doi.acm.org/10.1145/2597652.2597683.

[83] Tiwari, A., Laurenzano, M., Carrington, L., and Snavely, A., "Modeling Power and Energy Usage of HPC Kernels," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2012 IEEE 26th International*, pp. 990–998, 2012.

[84] Ueno, K. and Suzumura, T., "Highly Scalable Graph Search for the Graph500 Benchmark," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 149–160, ACM, 2012. http://doi.acm.org/10.1145/2287076.2287104.

[85] Underwood, K. D., Levenhagen, M., and Rodrigues, A., "Simulating Red Storm: Challenges and Successes in Building a System Simulation," in *Proc. International*

Parallel and Distributed Processing Symposium (IPDPS'07), pp. 1–10, IEEE Computer Society, 2007.

[86] Vydyanathan, N., Krishnamoorthy, S., Sabin, G. M., Catalyurek, U. V., Kurc, T., Sadayappan, P., and Saltz, J. H., "An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications," IEEE Transactions on Parallel Distrib. Syst., vol. 20, pp. 1158–1172, 2009.

[87] Vydyanathan, N., Krishnamoorthy, S., Sabin, G., Catalyurek, U., Kurc, T., Sadayappan, P., and Saltz, J., "An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 1158–1172, Aug. 2009.

[88] Wilke, J. J., Sargsyan, K., Kenny, J. P., Debusschere, B., Najm, H. N., and Hendry, G., "Validation and Uncertainty Assessment of Extreme-scale HPC Simulation Through Bayesian Inference," in *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, (Berlin, Heidelberg), pp. 41–52, Springer-Verlag, 2013. http://dx.doi.org/10.1007/978-3-642-40047-6_7.

[89] Williams, S., Waterman, A., and Patterson, D., "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009. http://doi.acm.org/10.1145/1498765.1498785.

[90] Woo, D. H. and Lee, H.-H., "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer*, vol. 41, pp. 24–31, Dec. 2008.

[91] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J., "Hierarchical DAG Scheduling for Hybrid Distributed Systems," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 156–165, May 2015.

[92] Zhang, H. and Hoffmann, H., "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 545–559, ACM, 2016. http://doi.acm.org/10.1145/2872362.2872375.

[93] Zheng, G. B., Wilmarth, T., Jagadishprasad, P., and Kale, L. V., "Simulation-based performance prediction for large parallel machines," *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 183–207, 2005. Metadata%0020Permalink:%0020http://permalink.lanl.gov/object/view?what=info:lanl-repo/isi/000230652400008.