DISSERTATION

RESOURCE MANAGEMENT FOR EXTREME SCALE HIGH PERFORMANCE COMPUTING SYSTEMS IN THE PRESENCE OF FAILURES

Submitted by

Daniel Dauwe

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2018

Doctoral Committee:

Advisor: Sudeep Pasricha
Co-Advisor: H. J. Siegel

Anthony A. Maciejewski
Patrick J. Burns

ABSTRACT

RESOURCE MANAGEMENT FOR EXTREME SCALE HIGH PERFORMANCE COMPUTING SYSTEMS IN THE PRESENCE OF FAILURES

High performance computing (HPC) systems, such as data centers and supercomputers, coordinate the execution of large-scale computation of applications over tens or hundreds of thousands of multicore processors. Unfortunately, as the size of HPC systems continues to grow towards exascale complexities, these systems experience an exponential growth in the number of failures occurring in the system. These failures reduce performance and increase energy use, reducing the efficiency and effectiveness of emerging extreme-scale HPC systems. Applications executing in parallel on individual multicore processors also suffer from decreased performance and increased energy use as a result of applications being forced to share resources, in particular, the contention from multiple application threads sharing the last-level cache causes performance degradation. These challenges make it increasingly important to characterize and optimize the performance and behavior of applications that execute in these systems.

To address these challenges, in this dissertation we propose a framework for intelligently characterizing and managing extreme-scale HPC system resources. We devise various techniques to mitigate the negative effects of failures and resource contention in HPC systems. In particular, we develop new HPC resource management techniques for intelligently utilizing system resources through the (a) optimal scheduling of applications to HPC nodes and (b) the optimal configuration of fault resilience protocols. These resource management techniques employ information obtained from historical analysis as well as theoretical and machine learning methods for predictions. We use these data to characterize system performance, energy use, and application behavior when operating under the uncertainty of performance degradation from both system failures and resource contention. We investigate how to better characterize and model the negative effects from system

failures as well as application co-location on large-scale HPC computing systems. Our analysis of application and system behavior also investigates: the interrelated effects of network usage of applications and fault resilience protocols; checkpoint interval selection and its sensitivity to system parameters for various checkpoint-based fault resilience protocols; and performance comparisons of various promising strategies for fault resilience in exascale-sized systems.

ACKNOWLEDGEMENTS

I am in debt to my academic committee, Dr. Sudeep Pasricha, Dr. H.J. Siegel, Dr. Tony Maciejewski, and Dr. Patrick Burns. I would like to express my deepest gratitude to my advisors Dr. Sudeep Pasricha and Dr. H.J. Siegel, and my committee member Dr. Tony Maciejewski for their support and guidance throughout my research and time spent at Colorado State University and for providing me with invaluable insight and assistance to help co-author much of the research in this dissertation. I would like to give a special thanks to Dr. Pasricha for allowing me the freedom to broadly explore the areas of HPC in which I am personally interested. He further helped to tailor my interests into research that brought benefit to the field of HPC in addition to my personal, academic, and professional advancement.

I would like to thank my friends and family for their patience and support throughout my time at CSU (and even before that as well) and especially Aspen for her deep-rooted encouragement. I would also like to give a special thanks to my lab mates for the late nights (and, let's be honest, early mornings) spent working, writing, and distracting me from the former. It has been a blast hanging out with all of you, and I never could have finished this dissertation without those distracting conversations! (And regardless of how much all of you have come to regret it, I still really appreciate you introducing me to lentils.)

DEDICATION

I would like to dedicate this thesis to my friends, family, and the progress of HPC.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and Overview

High-performance computing (HPC) systems such as data centers and supercomputers are complex systems that coordinate the execution of large-scale computation of applications over tens or hundreds of thousands of multicore processors. Unfortunately, applications executing in parallel across multicore processors can suffer from decreased performance and increased energy use as a result of the applications being forced to share resources and consequently interfering with each other's execution. This interference occurs at multiple levels of the system from individual components (e.g., threads in a multicore processor forced to share the last-level processor cache) to the system level (e.g., applications sharing network resources). Complicating matters further, as the size of HPC systems continues to grow exponentially the systems also experience an exponential growth in the number of failure-related events that occur and cause interrupts in application execution. These failures cause further decreases in performance and increases in energy use, making HPC progress more challenging. The research discussed here examines ways of intelligently utilizing system resources and addressing distributed failures to mitigate these negative effects in HPC systems.

The progress of HPC systems in terms of both performance and energy efficiency are continually monitored by the Top500 [2] and Green500 [3] lists respectively. Extrapolating the energy use and performance from the top supercomputers on these lists highlights the necessity of designing the next generation HPC systems around energy efficiency. Even if the most energy-efficient HPC system from [3] were scaled to exascale complexities and able to retain its peak performance efficiency of 17 GFlops/W, such a system would still require at least $\$59$ million per year just to operate. Additional overhead from increased system failures add further operational costs. Without suitable resilience techniques in operation the system would experience significantly increased energy costs from failed applications. It is estimated that the significantly smaller 13 petaflop Blue Waters systems would spend over $\$400,000$ in wasted energy without checkpoint/restart resilience

helping to mitigate the wasted energy of system failures. Given the superlinear increase in failure rates that is expected as systems progress toward exascale, the consideration of HPC resilience is critical for the design of an exascale system.

This research investigates both how to better characterize and model the negative effects that resource sharing and system failures have on large-scale computing systems, as well as developing techniques for intelligently utilizing system resources through optimal application scheduling. Relying on scheduling to better utilize system resources for solving these performance and energy problems allows the users of HPC systems the benefit of being able to be unaware of the complex strategies employed by the system on their behalf to provide the most efficient execution possible.

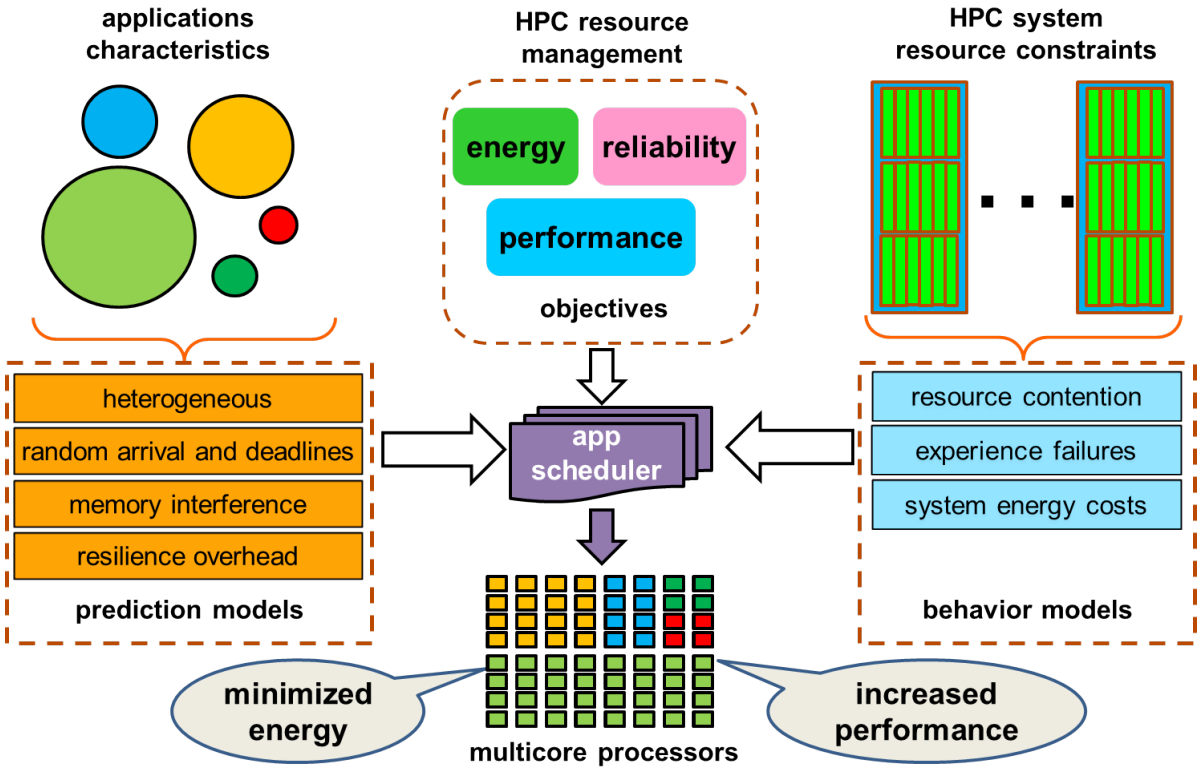

Figure 1.1: Overview of the framework proposed for HPC resource management.

Here we describe our development of a resource management framework used for analyzing and improving these HPC environments. An overview of this framework is depicted in Figure 1.1.

The top center of the figure shows the framework's three resource management objectives of energy, reliability, and performance for HPC systems.

The left-hand side of the figure details examples of the uncertain characteristics associated with application execution (e.g., execution overhead and inter-application interference) that make resource management in HPC systems challenging. To help reduce this uncertainty several of the chapters in this dissertation detail methodologies that allow for the creation of application performance prediction models that can be utilized by HPC resource management techniques to make better scheduling decisions for the applications demanding system resources.

The right-hand side of the figure gives examples of system constraints that limit the demand that can be satisfied by the system scheduler when attempting to accommodate an application's expected (or predicted) execution behavior. Interactions among the demands of applications on system resources, the resource's respective constraints, and the effects of decisions made by the system scheduler are complex. The research discussed throughout this dissertation employs models that describe the behavior of system resources (e.g., resource contention, application resilience behavior, and the system's responses to failures) for these complex situations.

Our proposed framework ensures that the design of the system's application scheduler can be made aware of the system's design objectives as well as the constraints associated with the system's available resources, the application's resource demands, and the application's execution characteristics. The system scheduler then works with the available data to ensure the system's objectives are met by determining for each application:

- when the application should begin execution;
- on which processors the application should execute;
- the type and timing of events associated with the application's resilience to system failures.

The models and design methodologies discussed in this dissertation have been integrated into our simulated analyses of system behavior and are used to assess our resource management technique's ability to accommodate the system objectives of performance, reliability, and minimized energy.

The remainder of this dissertation is organized as follows. The next chapter (Chapter 2) discusses the impact that memory interference has on HPC systems and outlines a methodology that can be used to model application memory interference behavior and make predictions about the effects that co-location will have on the application's execution. We utilize this prediction methodology by creating a memory interference aware resource management framework and analyzing its benefit to an environment that suffers from performance degradation due to application co-location. We analyze the behavior of resilience protocols used for mitigating system failures in HPC systems in Chapter 3 and develop models for simulating the behavior of several resilience protocols being proposed for future extreme scale HPC systems. Chapter 4 discusses how the resilience protocol models from Chapter 3 can be utilized to build a resilience aware resource management framework. Chapters 5 and 6 consider in more detail the optimization of multilevel checkpointing intervals for improving application performance, system energy use, and application execution time prediction accuracy. A closer examination of the interrelated effects of checkpointing and the system's communication network is conducted in Chapter 7. We conclude in Chapter 8 with a discussion of future directions for this research.

Chapter 2

HPC Node Performance and Energy Modeling with the Co-Location of Applications

2.1 Introduction

There is an inherent trade-off in large-scale computing systems between reducing the use of system resources by consolidating applications into as few server processor nodes as possible (to reduce system power), and the performance degradation that occurs to these applications as a result of sharing system resources with other applications (e.g., [7], [8]). Memory interference caused by multiple applications co-located on a multicore processor has been shown to negatively impact application performance (e.g., [9], [10], [11], [6], [12]). Specifically, the sharing of system resources such as DRAM and the last-level cache by co-located applications creates contention and increases the memory intensity of all applications running on the multicore processor [6]. This increase in memory intensity results in a corresponding increase in average memory access time, which ultimately contributes to an increase in the application's overall execution time. This increase in execution time is significant, and in some cases can as much as double or triple the execution time of an application as compared to its baseline execution time, i.e., when unhindered by co-location interference [13].

Multicore processors are pervasive throughout many kinds of computing systems, but the performance degradation effects caused by co-location interference are most likely to be prevalent

This work was done jointly with Ph.D. student Ryan Friese and masters student Eric Jonardi. The full list of co-authors is listed in [4]. This work was supported by the National Science Foundation (NSF) under grant numbers CNS-0905339, CCF-1252500, CCF-1302693, ACI-1339745, and an NSF Graduate Research Fellowship. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The authors thank Hewlett Packard (HP) of Fort Collins for providing us some of the machines used for testing. A preliminary version of portions of this work appeared in [5] and [6].

in large-scale server systems and high-performance computers. This is because in those types of computing systems, executing multiple applications on multicore processors results in high memory interference and therefore causes performance degradation [14]. Having a methodology that is capable of predicting how well a system will run in a particular co-location scenario is very useful for such systems. The results of this work show how the information obtained from accurate predictions of co-location performance degradation can be integrated into intelligent application scheduling, and thus lead to system performance improvement due to better utilization of the hardware. Better utilization of the hardware provides increased opportunities to reduce power and save energy through server consolidation, while still maintaining quality of service constraints in an improved manner over a co-location naïve scheduler. This work provides a methodology that can be used to create co-location aware performance models capable of predicting application execution time and energy use when co-located with other applications on a multicore processor in an HPC node.

The methodology for analyzing system performance that is described in this work is general enough to be applicable to any set of applications running on any multicore processor. Once application performance information for a particular combination of multicore processor and target applications has been collected, the methodology uses machine learning techniques to construct performance models characterizing that information. After they are trained, these models require only a single serial baseline measurement of parameters for each application running alone in the system. The models use this serial baseline measurement to make predictions about the performance degradation from memory interference that will occur when the application is executing with different types of co-located applications. While it has been shown in [10] that the degree to which an application's use of memory resources varies among different phases throughout the application's execution, this work illustrates that having such fine grain information is not always necessary to make accurate predictions.

After describing how the methodology operates, the theory behind the proposed methodology is validated by using real-world data collected on three Intel Xeon server-class machines that were

set to execute a collection of scientific application workloads, with some models providing up to 98% accuracy. In addition to creating and demonstrating a methodology that is capable of being ported across processor architectures, this work also provides insight into what memory-use information is most important to obtain for a set of applications running in a system to predict the impact of co-location on performance and energy use.

The last portion of this work demonstrates the utility of the proposed modeling methodology through the creation and analysis of a consolidating slack-based scheduling heuristic that utilizes the execution time prediction models generated from the proposed modeling methodology to assist in its application co-location decisions. This "co-location aware" scheduling heuristic is shown in simulated studies to provide a significant performance improvement over a similar consolidating slack-based scheduling heuristic that is naïve to the effects of performance degradation caused by co-location.

This chapter makes the following contributions:

(a) identifies factors that can characterize slowdown during application co-location scenarios;

(b) proposes a novel methodology to integrate these factors into multi-granularity and multi-fidelity performance models that can be used to predict application execution time and energy use under co-location scenarios;

(c) shows that a fine level of detail is not always necessary to achieve reasonable prediction accuracy;

(d) validates the methodology with real-world data obtained from running co-located scientific workloads on contemporary Intel Xeon server-scale multicore processors with up to 12 cores per processor;

(e) demonstrates through simulation the utility of the modeling methodology through the creation of a consolidating slack-based scheduling heuristic that utilizes models created from the proposed modeling methodology for its scheduling decisions.

The rest of this chapter is organized as follows. The following section discusses related work in this area. The modeling methodology is presented in Section 2.3. Section 2.4 details the testing environment and data collection used for validating the models. Experimental results that validate the models are examined in Section 2.5. A demonstration of the modeling methodology's utility is shown in Section 2.6. The chapter concludes with a summary of the main contributions in Section 2.7.

2.2 Related Work

2.2.1 Overview

Several works have explored the effect of co-location on application performance and energy use in multicore environments and the use of scheduling heuristics for improving the efficiency of high performance computing. Here the most relevant prior works in these areas are briefly summarized.

2.2.2 The Effect of Application Co-location on Performance and Energy Use

The authors in [9] examine how co-locating multiple applications on a single multicore processor affects performance. However, their work focuses on a general examination of the effects that co-location has on the system as a whole, and does not examine the effects on specific applications as our work does. This work also does not create co-location performance models the way that our work does, nor does it discuss modeling energy use. Our work also discusses a heuristic approach for mapping co-located processes to processor cores, whereas their analysis only examines a single processor node, providing a less precise view than our work where we consider an entire system.

The study in [10] provides an excellent review of how the architecture on which an application is executed can affect the cache use and memory intensity of that application. That work, however, does not attempt to make predictions about performance degradation as we do, but rather it shows the importance of including memory intensity and cache usage information when characterizing performance degradation in the presence of application co-location.

The work in [15] describes the challenges faced by applications sharing resources, and a need for the ability to perform precise predictions of performance degradation. The paper presents its "Bubble-Up" methodology for predicting performance degradation. However, it does not consider the impact of dynamic voltage and frequency scaling on application performance as we do in our work, and their study does not collect experimental data or characterize the memory interference effects of having more than two applications co-located, whereas we examine the performance degradation effects of more than two co-located applications

The authors in [11] present an extension to the "energy roofline" model that explores the effect of memory intensity (from the perspective of arithmetic intensity) on execution time and power use. The study executes a series of constructed microbenchmarks on twelve machine architectures and provides an analysis of the performance of the systems. While that study collects data about performance degradation from memory interference on a set of real machines, it uses small "microbenchmark" tests on machines, as opposed to the scientific workloads we use. Moreover, their work does not create models to predict execution times or energy use based on memory interference.

Similar to our work, the work in [16] examines creating a portable methodology using machine learning techniques for predicting application performance degradation from shared resources. The authors in that paper also incorporate shared resources beyond the last-level cache. However, incorporating those resources causes the resulting model to be complicated, and their model requires the constant monitoring of a large number of processor performance counters, which can cause system-wide slowdown for all running applications. In contrast, our methodology needs to collect performance counter information about each application only a single time, and provides a better prediction of performance. Additionally, our methodology guarantees a uniform selection of training data over the possible co-location space (allowing for more portability) while the work from these authors selects the vast majority of its training data at random.

A methodology for online estimation of an application's execution under co-location is presented as the "Application Slowdown Model" in [12]. Similar to our work, their work makes

predictions of application performance degradation by monitoring processor performance counters. However, their work does not perform any experiments on how the methodology performs for any actual server class processor as our work does, and therefore it is potentially limited in its portability or performance on actual systems. Additionally, their work limits their proposed models to analyzing the effect of performance degradation on application execution time and does not attempt to model energy as our work does.

Our work in [6] measures memory interference from application co-location, and its impact on system performance and energy use for a single Intel i7 machine. However, that work does not create models that predict system performance, and the scope is restricted to only a single consumer class machine.

We acknowledge that work exploring the effect of simultaneous multithreading (SMT) on application performance is an open and active area of research. Papers such as [17], [18], and [19] examine scheduling and resource use of applications executed utilizing SMT. We chose to focus our study on the interference that applications experience at an inter-core granularity, and for this study we have turned off SMT to remove the possibility of application interference in the L1 cache.

2.2.3 Scheduling Heuristics

Examples of prior work in scheduling for large-scale HPC systems have appeared in [20], [21], [22], and [23]. In [20], the authors look at the problem of energy-constrained dynamic allocations of tasks in heterogeneous cluster computing environments, in the presence of individual task deadlines. The work in [21] proposes power and thermal-aware scheduling to optimize individual tasks reaching their deadlines. A "utility" metric is defined in [22] that is used in combination with several energy-aware scheduling heuristics to provide a method of resource allocation that can maximize task performance while operating under a system energy constraint. The work in [23] examines energy-aware static resource allocation of a "bag of tasks" in a heterogeneous computing system. However, all of these existing techniques in scheduling do not focus on co-location effects that can significantly impact the validity of scheduling decisions.

In contrast to the above prior works, some researchers have proposed slack-based heuristics to improve system performance. For example, the work in [24], [25], [26], and any of the other numerous works that rely on the backfilling technique first described in [27], all rely on slack-based heuristics that perform their calculations based on predictions of application execution time. All of these works use application slack to provide better scheduling around uncertainty, but none of them account for the effects of performance degradation from co-location. As we demonstrate later, slack-based heuristics that are co-location aware can outperform co-location naïve slack-based heuristics. Consideration of co-location in the slack calculations of these works could be used to improve upon the work presented in these papers.

2.3 Modeling Methodology

2.3.1 Overview

The proposed modeling methodology uses two types of machine learning techniques, *linear modeling* and *neural networks*, for constructing the predictive models. These techniques have been used in prior related work [16], [15], but were limited in attribute selection and scope. For each machine learning technique, several models of varying levels of complexity and utilization of application features were built.

2.3.2 Model Features

Both the execution time and energy prediction models use up to eight separate features of application execution to predict how the target application performance or energy use is impacted by co-located applications. The eight features were chosen by performing a principal component analysis (PCA) [28] on the data collected from multicore processors considered in this work. PCA allows us to observe which features were most important to include in the models.

The features selected are a general set that are observable in almost all multicore processors. The models we construct use various combinations of the features. These models use different combinations of features, ranging from those that are most-commonly available to a scheduler

to combinations that require detailed information about the application and may be difficult to obtain on some platforms. The eight features that we selected after our PCA analysis are shown in Table 2.1. The table gives the name of the feature in the first column, and the description of that feature in the second column. The "target" application in the table is the one for which slowdown or increased energy use due to co-location is being predicted. The baseline execution time is the execution time of the target application without any co-location present. A task's memory intensity is defined to be the ratio of a task's last-level cache missed to the total number of instructions that the task has executed. Memory intensity is discussed further in Section 2.4.

Table 2.1: Model features

feature name	description of feature
baseExTime	baseline execution time of target application at all P-states
numCoApp	number of co-located applications
coAppMem	sum of co-application memory intensities
targetMem	target application memory intensity
coAppCM/CA	sum of co-application last-level cache misses/cache accesses
coAppCA/NI	sum of co-application last-level cache accesses/instructions
targetCM/CA	target application last-level cache misses/cache accesses
targetCA/NI	target application last-level cache accesses/instructions

Table 2.2: Model feature sets

set name	features within set
A	baseExTime
B	baseExTime, numCoApp
C	baseExTime, numCoApp, coAppMem
D	baseExTime, numCoApp, coAppMem, targetMem
E	baseExTime, numCoApp, coAppMem, targetMem, coAppCM/CA, coAppCA/NI
F	baseExTime, numCoApp, coAppMem, targetMem, coAppCM/CA, coAppCA/NI, targetCM/CA, targetCA/NI

2.3.3 Generality of the Models

The features shown in Table 2.1 can be combined to create models of various complexities. The model feature sets listed in Table 2.2 represent six possible models, one baseline model (model "A") that uses only the *baseExTime* feature for predictions, and five other models. For each of the five other models, the resource manger (scheduler) has a certain amount of baseline information about the system, the target application, and the other applications co-located on the system. The progression from one model to the next represents a realistic process where the resource manager progressively obtains more detailed information about the computing system and its executing applications.

Designing a methodology that provides several models with various levels of complexity allows a system designer greater freedom to make use of the modeling methodology to predict application performance and energy use. Providing a set of models with a range of complexities allows for a prediction model with a basic feature set to be used when more detailed application execution information for applications in the system is not available. For example, a system designer may not know or have the ability to measure the performance counter derived information (explained in Section 2.4) required to create more complicated models, but could have access to adequate application information (such as baseExTime and NumCoApp) to use the methodology to design simpler models that still give reasonable prediction accuracy and benefit from co-location awareness.

2.3.4 Linear Modeling Technique

To predict the impact of application performance degradation and energy use during execution under co-location, twelve linear models were developed: a set of six models for execution time prediction, and a set of six models for energy use prediction. Each of the six models in each set of models were constructed using the six model feature sets listed in Table 2.2. Each linear model is the sum of the products of the utilized features (denoted f_i) and the model coefficients determined during training (denoted for the execution time model as ct_i and denoted for the energy use model

as ce_i), plus a constant $const$. Linear regression is used to calculate the values for the coefficients. A general linear model for predicting co-located execution time using N features (linear execution time prediction $LETP$) takes the form of:

$$LETP = \sum_{i=1}^{N} (ct_i * f_i) + const \ . \tag{2.1}$$

The linear prediction of execution time and energy differ only in the objective that the models are trained to predict (execution time or energy use). A general linear model for predicting co-located energy use using N features (linear energy use prediction $LEUP$) takes the form of:

$$LEUP = \sum_{i=1}^{N} (ce_i * f_i) + const \ . \tag{2.2}$$

2.3.5 Neural Network Modeling Technique

From our observation of application execution in the presence of co-location, we noted that there are a few instances of nonlinearity in some of the features. This observed nonlinearity provided the motivation for creating a prediction model using a neural network that can capture such nonlinearities.

Neural networks are a class of machine learning techniques that can be used for creating predictive models [29]. The approach attempts to mimic the function of the human brain by defining several "layers of neurons." Each neuron layer is composed of some number of individual neurons that take the outputs of the previous layer as each of their inputs, with the inputs to the first layer of neurons being the features of the data available in each model (see Tables 2.1 and 2.2). The final output is the value of the predicted execution time or energy use that the application will experience under co-location.

Each neuron operates by multiplying each of its N inputs, x_i, by N corresponding weight parameters, w_i, summing these results, and finally evaluating the sum with a nonlinear function f. The k^{th} neuron in layer j (denoted y_{jk}) operates according to:

$$y_{jk} = f(\sum_{i=1}^{N} x_i w_i) \ . \tag{2.3}$$

The nonlinear function f in Equation 2.3 is called the activation function of the neuron, and attempts to mimic the biological process that occurs during activation of an actual neuron. Any sigmoidal function will satisfy this activation function, but the hyperbolic tangent function ($\tanh$) was chosen in particular for this work because it allows for faster convergence when using gradient methods for training the weight parameters [30]. It is this activation function that allows neural networks to capture nonlinearities when modeling.

The neural network is trained by adjusting the weight values at each neuron to minimize an objective function that measures the squared error between the neural network's set of predicted values of the training data and the actual values of the data. Optimal weight values were determined using a *conjugate gradient* because it provides fast convergence [31]. Two separate neural networks were created for the execution time and energy predictions, respectively. As with the linear models, the neural network execution time prediction models were trained using measured execution time values of the target application, and the neural network energy use models were trained using measured energy use values of the target application.

2.3.6 Model Accuracy

All models are evaluated using Mean Percent Error (MPE) and Normalized Root Mean Squared Error (NRMSE) to offer two different measures for comparing model predicted values to actual values. Error measurements are only made for the target application's execution time or energy use in each test, not for all applications co-located in the system.

The magnitudes of the actual values within the data vary greatly (e.g., when modeling execution time, actual values could range from as little as 150 seconds to over 1000 seconds based on the application that is being executed and the state of co-location of the applications in the system). Thus, when finding MPE, a calculation of *relative error*, allows the evaluation of prediction accu-

racy independent of these magnitudes for each of the M sample points of data. In the equation, predicted values are denoted p_j and actual values are denoted by a_j. MPE is defined as:

$$MPE = 100 * \frac{1}{M} \sum_{j=1}^{M} |\frac{p_j - a_j}{a_j}| \ . \tag{2.4}$$

NRMSE gives an indication of the variance of the predicted values from the actual values. For M sample points, NRMSE provides a ratio of Root Mean Squared Error (an *absolute error*) and the interval of values that the actual data can take (the largest actual data value a_{max} minus the smallest actual data value a_{min}). Normalized root mean squared error is defined as:

$$NRMSE = \frac{\sqrt{\frac{\sum_{j=1}^{M}(p_j - a_j)^2}{M}}}{a_{max} - a_{min}} \ . \tag{2.5}$$

2.4 Implementation

2.4.1 Testing Environment

This section describes a testing environment that can be used for the modeling methodology's data collection and validation. The testing environment that is described is not only effective and easy to use for collecting the data, but also easy to replicate and can be used on a wide variety of multicore processors.

Operating System

One of the objectives of this research was to design a methodology that can be applied to a wide variety of computing systems. The testing environment was designed to be portable across many multicore processor architectures to allow for simplicity in gathering test data and ease of recreating the testing environment for future users of this work. To ensure accurate data is collected, the testing environment is run from a "lightweight" command line version of the Ubuntu 14.04 operating system [32] installed on a USB drive. This minimizes the effect that the operating system has on application execution. Non-essential OS utilities and kernel daemons were removed so that the applications being monitored suffer as little interference as possible from unpredictable

events in the OS. Such an environment mimics a large-scale computing platform meant to execute multiple applications concurrently.

Processor Performance Counters

Modern multicore processors provide the ability for developers to monitor hardware events that occur inside a multicore processor during the execution of an application [33]. Through the use of specialized "performance counters" present in the processor, it is possible to track the number of occurrences of certain events that take place, such as the number of instructions executed or last-level cache misses. These performance counters are architecture dependent, and due to differences among microarchitectures the number and types of performance counters that are available to the system are not consistent (e.g., differences described in [34], [35], and [36]). Given the design goal of having portability for the methodology, interfacing directly with these hardware performance counters is not a feasible option. Therefore, the testing environment makes use of two tools to facilitate interactions with the hardware.

The first tool is "Performance Application Programming Interface" (PAPI) [37], an API that was made specifically to provide portability when accessing performance counters across different architectures. PAPI has created a general list of more than 100 standard performance counter "presets" that are likely to be present in a modern processor. PAPI has made it more accessible to interface with these counters across architectures.

The second tool the testing environment utilizes is the HPC toolkit [38]. This suite of tools interfaces with PAPI and makes it easier to monitor and collect information from multiple performance counters in the system. Specifically, HPC toolkit's "hpcrun-flat" application profiler is used to collect performance counter information because it is able to run with very low overhead.

Measuring Cache Use

From [6], it is known that applications that need to access data from memory more often tend to experience a larger amount of performance degradation due to co-location. These performance degradations are incorporated into the prediction models by collecting measurements related to

these effects. We have found that three hardware performance counter measurements can be used to collect the information necessary for deriving the metrics used in our methodology's models:

(a) number of last-level cache misses an application experiences (LLCM) that represents the number of times an application must access main memory;

(b) number of instructions the application executes (NI);

(c) total number of last-level cache accesses the application attempts (TCA).

The model features that are derived from these measurements were listed in Section 2.3. It should be noted that last-level cache misses and accesses are dependent on architecture, and can refer to either the L2 or L3 cache depending on the multicore processor that is being used. It is also important to note that when collecting test results for the execution of applications, the values measured in these performance counters can only represent a sum of the total events of that type that have occurred during the time the performance counters are monitored, in this case the duration of the application's execution.

One notable metric derived from this data is application memory intensity. Memory intensity is defined to be a ratio of an application's LLCM to NI. This metric gives an idea of the rate at which an application needs to go to main memory to fetch data. It is useful because it shows whether an application's execution will be more likely to be memory-bound relative to another application, meaning that its performance depends more on memory access speed rather than computational speed. Memory intensity also gives some idea of how much an application tends to access memory. A highly memory-intensive application is more likely to utilize the shared cache resources more, and therefore it will tend to affect, and be more affected by, the effects of memory interference from other applications.

Processor Performance States (P-states)

Processor performance states (P-states [33]) are a set of discrete voltage and frequency values in which a multicore processor can operate. P-states utilize dynamic voltage and frequency scaling (DVFS), supported in all contemporary multicore processors. DVFS techniques can reduce

the dynamic operating power of a multicore processor to consume less power or to temporarily reduce the operating temperature due to the multicore processor having exceeded a thermal threshold. However, these benefits come at the cost of having to throttle the multicore processor speed by decreasing the clock frequency. This effectively always increases the execution time (and thus decreases system performance) of any application running on the multicore processor. The range and number of P-state frequencies that are available in a system are highly dependent on the architecture of the multicore processor. Processor P-states are likely to change in HPC systems based on the system's need to reduce power or temperature. In this work, this effect is taken into account through knowledge of the baseline execution time of each application at a given P-state. The P-state in which each processor is going to be executed is assumed to be known before execution of the co-located applications. Each P-state has a different baseline execution time and the P-state of the processor is implicitly an input to each of the models by virtue of its value of baseline execution time.

Measuring Power and Energy Use

We used a*Watts Up? PRO* power meter [39] to measure application power and energy use. The *Watts Up? PRO* power meter measures instantaneous power use of a target load at the "wall outlet" level with a sampling rate of once per second. Power values were recorded at the "wall outlet" level for the system as a whole. The resulting data then provided a record of the system's power use over the execution of each application. The data could also be summed to give the value of the total energy used by the system during the application's execution, or averaged to give the average power used by the system during the application's execution.

2.4.2 Data Collection and Experimental Setup

Benchmark Applications

The applications run as testing workloads for the model validation were taken from two scientific benchmark suites. The set of eleven applications considered vary in the types of tasks that they perform and are characterized by a wide spread of memory intensity values. Table 2.3

Table 2.3: Memory intensity classification

applications	classification	baseline memory intensity (LLCM / NI)
canneal (P)	Class I	1.84×10^{-2}
cg (N)	Class I	1.56×10^{-2}
ua (N)	Class II	1.63×10^{-3}
sp (N)	Class II	1.50×10^{-3}
lu (N)	Class II	1.11×10^{-3}
fluidanimate (P)	Class II	8.60×10^{-4}
freqmine (P)	Class III	3.47×10^{-5}
blackscholes (P)	Class III	1.88×10^{-5}
bodytrack (P)	Class IV	8.69×10^{-7}
ep (N)	Class V	6.27×10^{-10}
swaptions (P)	Class V	4.22×10^{-10}

shows the applications examined in this study. Applications taken from the PARSEC benchmark suite [40] are denoted with *(P)*, and applications from the NAS benchmark suite [41] are denoted with *(N)*. The table also shows each application's associated baseline memory intensity values, where baseline memory intensity values are measured when the applications are executed on a multicore processor by themselves without interference caused by co-location.

As shown in Table 2.3, these applications have been categorized into five memory intensity classes, denoted "Class I" through "Class V." Class I applications are the most memory-intensive applications, meaning that they have the highest number of last-level cache misses per number of instructions executed and are more memory bound, while Class V applications are the least memory-intensive, meaning that they experience fewer last-level cache misses per number of instructions executed, and their execution is more CPU bound. Categorizing the applications into groups allows applications from particular groups to be referred to in a more general manner. These groupings allow for a broader use of the methodology for performance prediction.

Having application class values allows a developer the possibility to be able to use our modeling methodology even if it is not possible to obtain a detailed measurement of an application's memory use. If an estimate of the memory intensity of a particular application type could be made based on its historical use, or if an application developer had prior experience developing similar

Table 2.4: Multicore processors used for validation

processor	num. cores	L3 cache	frequency range
Intel Xeon E3-1225v3	4	8MB	800 MHz-3.20 GHz
Intel Xeon E5649	6	12MB	1.60-2.53 GHz
Intel Xeon E5-2697v2	12	30MB	1.20-2.70 GHz

applications with known characteristics of memory use, then these applications could be broadly categorized into one of these memory intensity classes. Having this general classification of an application, the developer can still gain some insight as to the expected performance of the system by running the model substituting average memory intensity values for that application's class in place of its unavailable measured values. A system designer can create more classes than those we consider, to improve resolution and classification granularity, especially if a much larger set of applications is considered. However, for the sake of brevity in discussion, we restrict the number of classes to five in this work.

It should be noted that the memory intensity values listed in Table 2.3 are calculated from baseline measurements for one specific system (Xeon E5-2697v2). The memory intensity values do not vary widely among the machines tested, thus the application memory intensity classes are used to represent categories for the Xeon family of multicore processors considered. It is also important to note that the memory intensity values among application classes tend to differ by orders of magnitude. This allows for clearer distinctions to be drawn among application classes.

Multicore Processors Tested

The specifications of the multicore processors tested during the validation of the methodology are shown in Table 2.4. All multicore processors used are from the Intel Xeon family of multicore processors, with a varying number of available cores (ranging from four to twelve), L3 (last-level) cache sizes, and frequency ranges. Detailed information about these processors can be found in [34], [35], and [36].

Model Training

Training data was collected from each multicore processor to construct the models discussed in Section 2.3. The training data for all of the machines was collected in the form of execution time, total energy use, and average power values of various co-location combinations using all eleven applications as target applications co-located with a subset of four of the applications available in the testing environment. Specifically, *cg*, *sp*, *fluidanimate*, and *ep* were used as the applications that were co-located with each "target" application. Because our preliminary results show that more memory intensive applications tend to have a greater interference effect on co-located applications, we biased our selection of three of the co-location applications towards more memory intensive applications. The *ep* application is also included to represent the effects of co-location with a more CPU intensive application. This limitation of four co-location applications was imposed to keep the number of tests that were executed for training tractable.

When measuring application performance, data is collected for only a single "target" application during any given co-location test. Initial baseline tests were run that measured each application's execution without co-location across six P-state frequencies to determine how each application performed without interference from other applications. This baseline test provides a basis of comparison for the effect of co-location interference on each application. The training data was collected for each of the eleven target applications by running tests that co-locate each application with multiple copies of each of the four co-location applications mentioned earlier. Multiple homogeneous copies of each of these co-location application types were heterogeneously executed with the target application, for each of the number of co-locations denoted in the "number of co-locations" column shown in Table 2.5. Each of these sets of tests were performed once for each of the six selected P-states on each multicore processor. The P-state frequencies are shown in Table 2.5. Each of the columns three to six in the table represent nested loops in the data collection code of Algorithm 1.

Thus, each attribute that is included as parameters to the data collection increases the size of the co-location space substantially. For example, the data collected for the third row of Table 2.5

is for six different frequencies, eleven different target applications, a selection of one of four co-located application types, and one of six choices of the number of copies of the selected co-located application to run with the target application for a total of 6x11x4x6 = 1584 co-location scenarios.

Algorithm 1 Training data collection

```
1:  for each multicore processor do
2:      for each P-state frequency do
3:          for each target application do
4:              for co-located application do
5:                  for each number of co-locations of co-located application do
6:                      get_exec_time_of_target()
7:                      get_system_energy_use_during_target_execution()
8:                      get_average_power_use_during_target_execution()
9:                  end for
10:             end for
11:         end for
12:     end for
13: end for
```

The "num. of co-locations" column in Table 2.5 shows the number of additional applications that were homogeneously co-located with the target application (i.e., all co-located applications are of the same type). The applications ranged on each multicore processor from only a single co-located application occupying one additional core, to co-located applications running on all of the multicore processor's available cores (i.e., one target application plus $n-1$ co-located applications for a multicore processor that has n cores). Setting up the training data in this way is an attempt to sample the set of all possible co-locations for a given machine in a uniform way that minimizes the amount of training data that is needed to calculate coefficients for the model. For all of the co-location scenarios represented by the three rows in Table 2.5, we collected experimental data which we used in Section 2.5 to evaluate the accuracy of our prediction methods.

Model Testing

Application testing was performed by partitioning the training data described in Section 2.4.2 (the co-location scenarios from Table 2.5) by repeated random sub-sampling based on the boot-

Table 2.5: Training schedule

processor	num.cores (n)	frequencies (GHz)	target apps.	co-location apps.	num. of co-locations
Intel Xeon E3-1225v3	4	3.20, 2.70, 2.20, 1.80, 1.30, 0.80	all	cg,sp,fluidanimate,ep	1,2,3
Intel Xeon E5649	6	2.53, 2.40, 2.26, 2.00, 1.73, 1.60	all	cg,sp,fluidanimate,ep	1,2,3,4,5
Intel Xeon E5-2697v2	12	2.70, 2.40, 2.10, 1.80, 1.50, 1.20	all	cg,sp,fluidanimate,ep	1,3,5,7,9,11

strapping approach first described in [42]. Thirty percent of the data was randomly selected and withheld from the training process of each model. After training, the withheld data was tested using each of the models and measured for accuracy. In this way, each model was tested using data that had not been seen previously during training. This withheld data is referred to as the testing data. The partitioning process was repeated twenty times, each time with a new random selection of points being withheld from training. The error values from each of these twenty training and testing partitioned groups was then averaged to determine the overall accuracy of each model.

2.5 Experimental Results

2.5.1 Overview

This section discusses the performance results of each of the twenty-four models. Altogether there are two sets (one set for execution time predictions, and the other set for energy use predictions) of 12 models (two classes of modeling techniques - linear, and neural network - with six variants each, based on the six model feature sets in Table 2.2). Each of the feature sets offers a trade-off between prediction accuracy and model sophistication. Figure 2.1 shows the *execution time* prediction accuracy for the 4-core Intel Xeon E3-1225v3, 6-core Intel Xeon E5649, and 12-core Intel Xeon E5-2697v2 multicore processors. Figure 2.2 shows the *energy use* prediction accuracy for the same three processors. Model prediction accuracy for both the training and testing data sets presented in both MPE and NRMSE is included for each of the machine learning techniques.

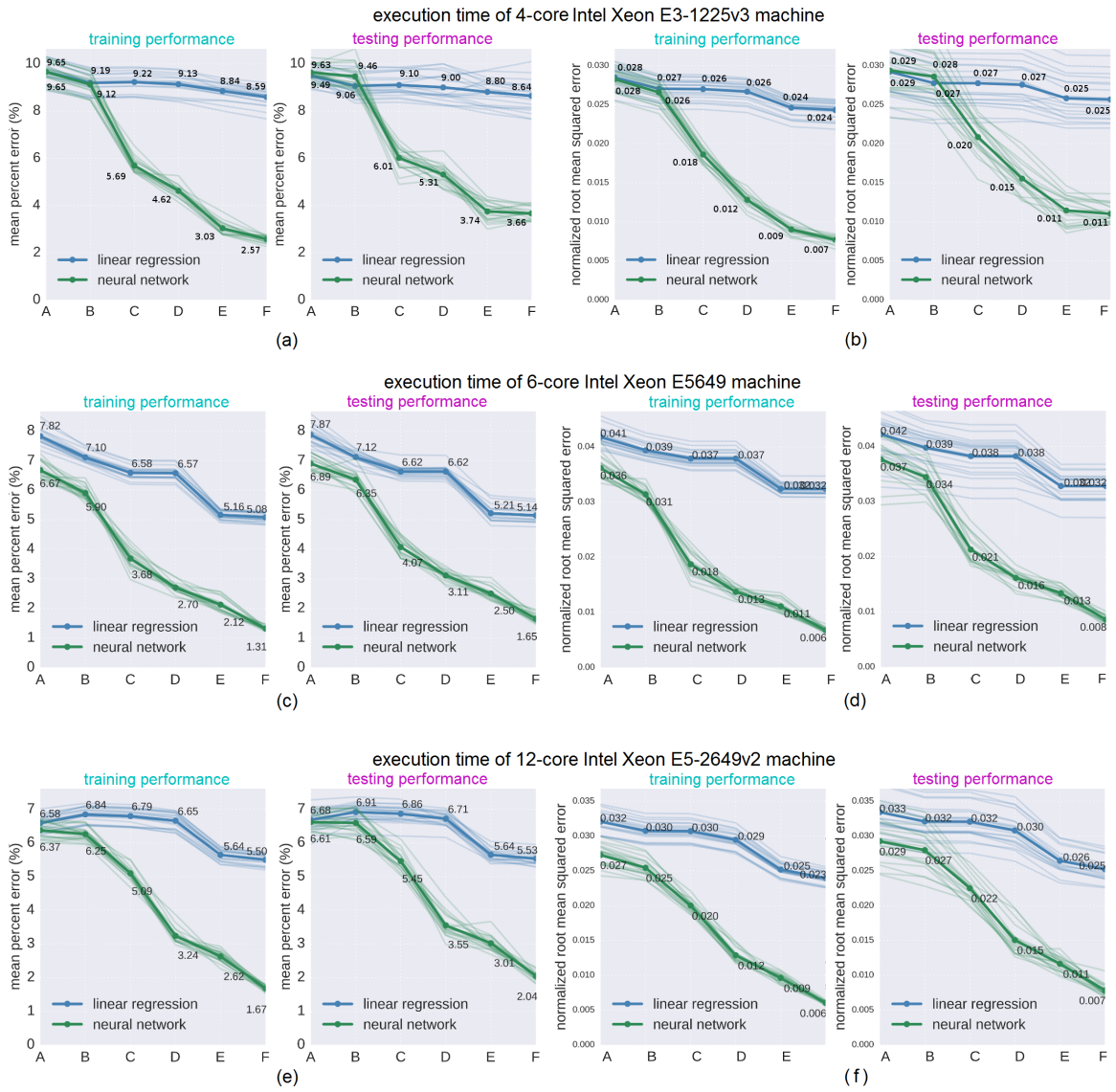

Figure 2.1: Execution time prediction model performance per feature set for each Intel Xeon processor. **(a, c, e)** Show MPE for the performance of training and testing data sets for model feature sets A through F. **(b, d, f)** Show NRMSE for the performance of training and testing data sets for model feature sets A through F. The figure shows results for each of the machine learning techniques: *linear* (blue) and *neural networks* (green). Each point on the figure represents an average of twenty different partitions of the data into training and testing data. Annotations next to points indicate the value of the point. The lighter shaded lines indicate the performance of each model on individual partitions, the darker shaded lines indicate the average value across all twenty partitions.

Each data point in Figure 2.1 and Figure 2.2 represents the average training error or average testing error from twenty partitions of the data (as discussed in Section 2.4.2) for each particular model. Results for each of the twenty individual partitions are shown as the lighter shaded lines

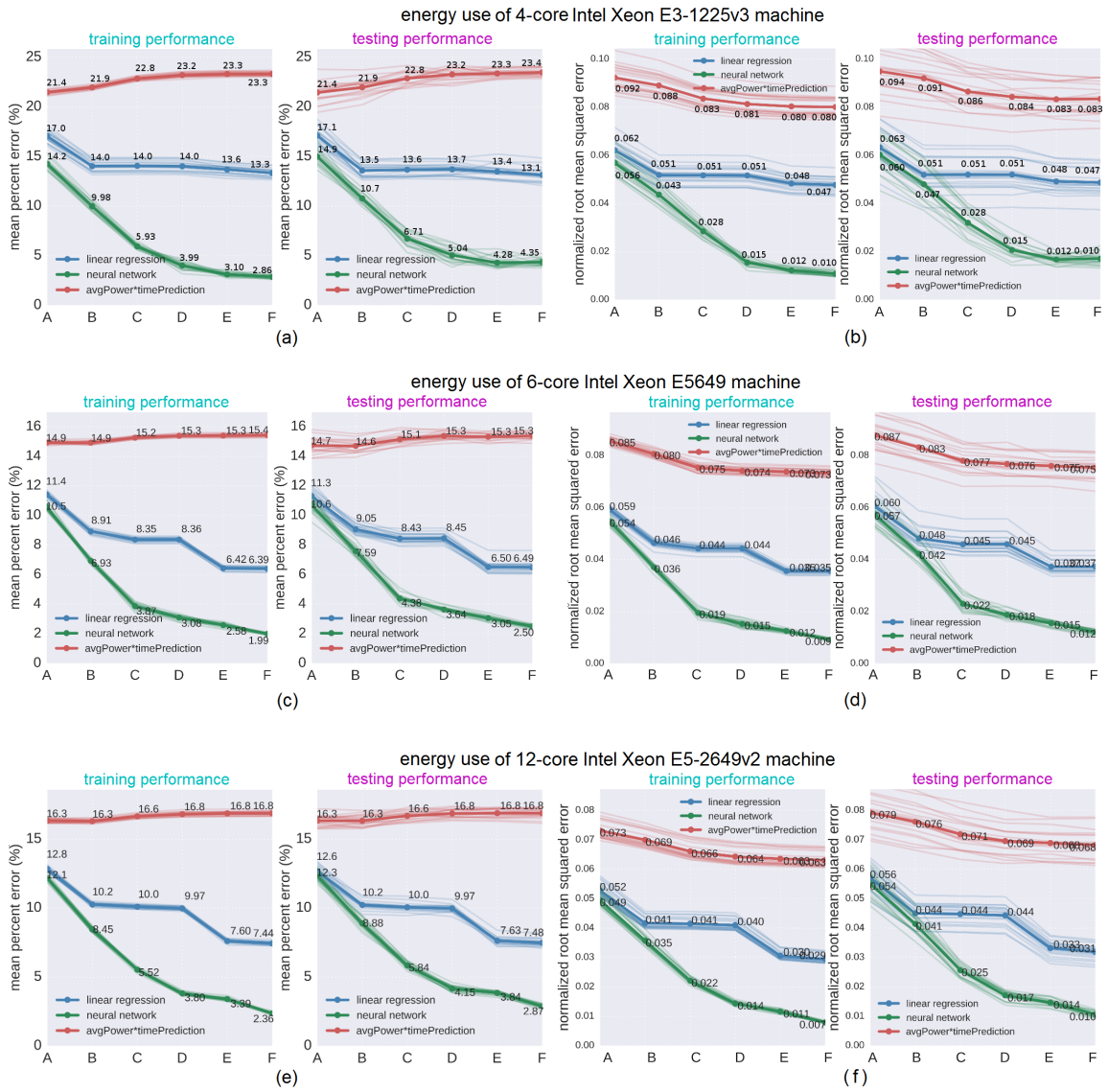

Figure 2.2: Energy prediction model performance per feature set for each Intel Xeon processor. **(a, c, e)** Show MPE for the performance of training and testing data sets for model feature sets A through F. **(b, d, f)** Show NRMSE for the performance of training and testing data sets for model feature sets A through F. The figure shows results for each of the machine learning techniques: *linear* (blue) and *neural networks* (green), as well as a comparison to the results that are obtained by simply multiplying the corresponding feature set of the neural network execution time prediction for each processor (the results shown in Figure 2.1) to each test's measured baseline average power value (red). Each point on the figure represents an average of twenty different partitions of the data into training and testing data. Annotations next to points indicate the value of the point. The lighter shaded lines indicate the performance of each model on individual partitions, the darker shaded lines indicate the average value across all twenty partitions.

for each set of models, showing the range of values for each of the darker shaded average lines. As can be seen in the figures, the range of values among each partition that was tested does not vary

much (at most 1.5% for extreme cases). The annotations next to points represent the average value across the twenty partitions for that model.

In addition to results for each feature set of both machine learning techniques, Figure 2.2 showing energy prediction model results also shows how effective it would be to make power-based predictions about energy use by multiplying an application's measured average power use with its predicted execution time under co-location (i.e., "$avgPower * timePrediction$"). This last set of power-based energy prediction models is presented to serve as a basis of comparison for the models generated by the methodology, and does not serve as part of the overall modeling methodology.

The value representing an application's average power use ($avgPower$) is baseline average power, a measured value collected for each application type and for each processor P-state. This value represents the average amount of power (specified in watts) that the application uses throughout its execution. Similar to the model features listed in Table 2.1, baseline average power use is a value that is measured from an application's execution without co-location, and is obtained during the collection of application energy data specified in Section 2.4.2.

The predicted execution time value ($timePrediction$) used for the calculation of each "$avgPower* timePrediction$" energy prediction model is generated by the corresponding neural network execution time prediction model from the methodology for that particular feature set for that particular processor. For example, if an energy use prediction for an application running on the 4-core Intel Xeon E3-1225v3 processor is made using the "$avgPower * timePrediction$" model feature set "D" then the execution time prediction for the calculation would be made using the neural network model feature set "D" execution time prediction model for the 4-core Intel Xeon E3-1225v3 processor (the neural network model results that are shown in Figure 2.1). The neural network model was used for execution time predictions because it provides better accuracy than the linear model, as discussed below.

2.5.2 Results of Linear Modeling

Overview

As the linear model feature sets become more advanced (i.e., as the models progress from using feature set A to feature set F), both the training and testing errors generally decrease for both the execution time and energy use sets of prediction models for all three processor types, indicating that the models become increasingly accurate when using increasingly advanced feature sets. It can be further seen from the lighter shaded lines showing the performance of individual partitions for both the execution time and energy prediction models of all three processors that as the model feature sets become more advanced the variance seen among individual partition results tends to decrease, indicating that the more complicated models have greater precision in addition to greater accuracy.

The complexity of the sample space makes it challenging for the linear models to perform well and improve significantly beyond the accuracy of the baseline model (relative to the improvement demonstrated by the neural network models). As mentioned earlier, non-linearity in the data makes predictions with these linear models less accurate.

Linear Execution Time Modeling

For all three multicore processors tested, the more advanced linear models provide only a modest improvement in execution time prediction accuracy over the baseline linear model (model A) when feature information is added to the models. Linear execution time model results are shown in Figure 2.1. The 4-core processor results (Figure 2.1 (a, b)) have a training and testing error that reduces by about 1% MPE and 0.004 NRMSE from model A to model F. The 6-core processor (Figure 2.1 (c, d)) shows a reduction of about 2.74% for both training and testing MPE from model A to model F. The 12-core processor (Figure 2.1 (e, f)) shows only about a 1% MPE improvement from the addition of more model features from model A to model F. The linear NRMSE variance results for the 6-core and 12-core processors follow very similar trends, reducing by about 0.008-0.009 NRMSE for both processor's training and testing results from model A to model F.

Linear Energy Use Modeling

In general, the prediction accuracy for the energy use models tends to be lower than the execution time models, indicated by the models having higher MPE and NRMSE values. However, they also show a greater increase in accuracy as the models become more advanced. Linear energy use model results are shown in Figure 2.2. The 4-core processor results (Figure 2.2 (a, b)) have training and testing error that reduces by about 3.7% and 4% MPE, respectively, and a 0.015 NRMSE decrease in training error and 0.016 NRMSE decrease in testing error from model A to model F. The 6-core processor (Figure 2.2 (c, d)) shows a reduction of about 5.01% MPE and 0.024 NRMSE for training data and 4.81% MPE and 0.023 NRMSE for testing data from model A to model F. The 12-core processor results (Figure 2.2 (e, f)) shows a 5.36% MPE and 0.023 NRMSE improvement for training data and a 5.12% MPE and 0.025 NRMSE improvement for testing data from model A to model F.

It should be noted that even though the linear models for energy use are not able to outperform the neural network models, they still significantly outperform the energy calculation made by the "average power multiplied by co-located execution time." This is explained in detail in Section 2.5.4.

2.5.3 Results of Neural Network Modeling

Overview

To allow for a fair comparison, the linear and neural network models use the same training and testing data partitions. The first observation to note regarding the neural network models is their clear improvement in prediction accuracy over the linear models, except for the single case of the execution time prediction model B for the 4-core Intel Xeon E3-1225v3 machine shown in Figure 2.1 (a, b).

The neural network models exhibit similar results to the linear models of the lighter shaded lines converging with the more advanced feature sets, indicating increased model accuracy and precision. In addition, with more advanced feature sets, the neural network model results in a

closer grouping of the lighter shaded lines than with the linear models. This implies that the individual partition results of the neural network models show less deviation from their mean than the predictions made by the linear models with the same partitions of data, indicating that the neural networks typically provide more consistent predictions than the linear models.

Neural Network Execution Time Modeling

Predictably, the complex neural network models, which utilize the most information, perform the best at predicting application execution time.

Neural network execution time prediction model results are shown in Figure 2.1. The 4-core processor results (Figure 2.1 (a, b)) have training and testing error that reduces by about 7.08% MPE for training and 5.97% MPE for testing, and 0.021 NRMSE for training and 0.018 NRMSE for testing from model A to model F. The 6-core processor (Figure 2.1 (c, d)) demonstrates a reduction of about 5.36% for training MPE and 5.24% for testing MPE, and 0.030 NRMSE for training and 0.029 NRMSE for testing from model A to model F. The 12-core processor (Figure 2.1 (e, f)) shows about a 4.7% MPE improvement and a 0.021 NRMSE improvement for training error, a 4.57% MPE improvement and a 0.022 NRMSE improvement for testing error from the addition of more model features from model A to model F.

Neural Network Energy Use Modeling

The general prediction accuracy for the energy use models tends to have higher error values than the execution time models. Neural network energy use model results are shown in Figure 2.2. The 4-core processor results (Figure 2.2 (a, b)) have training and testing error that reduces by about 11.34% and 10.55% MPE respectively, and a 0.046 NRMSE decrease in training error and 0.050 NRMSE decrease in testing error from model A to model F. The 6-core processor (Figure 2.2 (c, d)) shows an MPE reduction of about 8.51% MPE and 0.055 NRMSE for training data and 8.1% MPE and 0.045 NRMSE for testing data from model A to model F. The 12-core processor results (Figure 2.2 (e, f)) shows a 9.74% MPE and 0.046 NRMSE improvement for training data and a 9.43% MPE and 0.044 NRMSE improvement for testing data from model A to model F. It should

be noted that the neural network energy prediction models for all three processors significantly outperform the energy calculation made by "average power multiplied by co-located execution time."

2.5.4 Calculating System Energy Use from Time Predictions and Average Power

The "$avgPower * timePrediction$" results shown in Figure 2.2 (indicated by the red line in each portion of the figure) demonstrate the benefits associated with using our proposed energy prediction models (indicated by the blue and green lines in the figure) as opposed to the more approximate calculation of energy, i.e., using measured average power multiplied by the execution time prediction. The problem with these simpler calculations of energy use lies in the necessity of using an application's baseline average power. Even as the execution time prediction in the calculation becomes more accurate (moving from an execution time model using feature set "A" to one using feature set "F"), the resulting energy use prediction shows little to no improvement. An application's baseline average power use turns out to not be a sufficiently accurate measurement for making energy predictions calculated from power and execution time. Even if it were possible to make perfectly accurate execution time predictions, the "$avgPower * timePrediction$" models could not improve in their energy use predictions, demonstrating the necessity of using a more sophisticated modeling methodology, such as the one we propose, for making predictions about application energy use under the influence of application co-location.

Unfortunately an application's baseline average power is the only fair power measurement that can be used for this energy calculation without making the resulting model at least as complex as using models from our proposed methodology. Any power measurement that could be used to more accurately predict energy use would require either average power measurements directly measuring the effects of co-location on an application's average power, or power measurements that would require a more sophisticated modeling strategy to be used effectively for energy calculation. In

either case, trying to more accurately predict energy use when using power and execution time measurements would not be effective without the creation of a more sophisticated model.

Another interesting effect caused by the inaccuracy of the average power by time calculation is observed when the execution time model increases in accuracy (moving left to right from model A to model F) for each processor type shown in Figure 2.2. Both training and testing MPE actually *increase* when using the "$avgPower * timePrediction$" energy use calculation. This is because the execution time prediction models used for the $timePrediction$ value are making more accurate execution time predictions, and thus causing the energy use calculations to make more precise predictions of an inaccurate energy use value. That is, the model is making more accurate predictions of "$avgPower * timePrediction$," but because the use of baseline average power produces results that are inherently inaccurate, the MPE increases as the execution time predictions incorporate more features.

The reason that all three processors' NRMSE values of "$avgPower * timePrediction$" decrease as the predicted execution time value of the model becomes more accurate is because the calculation of NRMSE starts as a calculation of an absolute error (root mean squared error) that is then normalized to the range of the *aggregate whole* of the actual data (as shown in Equation 2.5), whereas MPE is a relative error that is normalized to the actual energy use value at each data point. Because of this, small changes in these larger energy use predictions produce larger effects in NRMSE than it does in MPE where these larger predictions are each normalized by larger data values.

A more detailed analysis of these model results shows that the predictions made with the "$avgPower * timePrediction$" models tend to have the most variation in prediction accuracy for co-locations that produce higher values of energy use. Consequently, as the execution time predictions improve from models using feature set "A" to models using feature set "F," the energy use data that has higher values experiences the most improvement in its prediction accuracy. Even though the model predictions are inherently inaccurate because of their reliance on baseline average power, these predictions of high valued data were the worst off to begin with, so the im-

provement in the execution time predictions provide the resulting energy use predictions of these higher valued data points a noticeable improvement.

2.5.5 Model Accuracy

The illustrations in Figure 2.3 and Figure 2.4 provide a more detailed view of the accuracy of the predictions made by the most accurate execution time prediction model and the most accurate energy use model created for the 6-core Intel Xeon E5649 machine. Each of the icons in Figures 2.3(a) and 2.4(a) respectively represents the distribution of each application's measured execution time and energy use data. The icons in Figures 2.3(b) and 2.4(b) respectively represent the distribution of error for the predictions of each application's measured execution time and energy use data. The larger width across each icon indicates a higher density of data points.

Figure 2.3(a) shows a detailed view of each application's execution time distribution on a 6-core Intel Xeon E5649 machine. The 120 points inside each application's distribution mark the specific execution time values measured for each test run of the application. Figure 2.3(b) shows a detailed view of the performance of the neural network model using feature set F (the most accurate model) on the execution time data shown in Figure 2.3(a). Distributions of the percent error between the model's execution time predictions and the application's actual execution times are shown for each application. The lines across each distribution represents the distribution's median (dashed line) and upper and lower quartiles (dotted lines). Figure 2.3 demonstrates that the model's predictions are typically accurate (their error is close to zero), that about half of the model's predictions are $\pm 2\%$ from the actual execution time values, and that nearly all of the predictions are within 5% of the actual execution time values.

The energy use prediction model results of the data shown in Figure 2.4 are very similar to those of the execution time prediction model from Figure 2.3. Each of the distributions shown in Figure 2.4(a) shows a detailed view of each application's energy use distribution on a 6-core Intel Xeon E5649 machine, with the 120 points in each distribution marking specific energy use values of each respective application. Figure 2.4(b) shows a detailed view of the performance of the

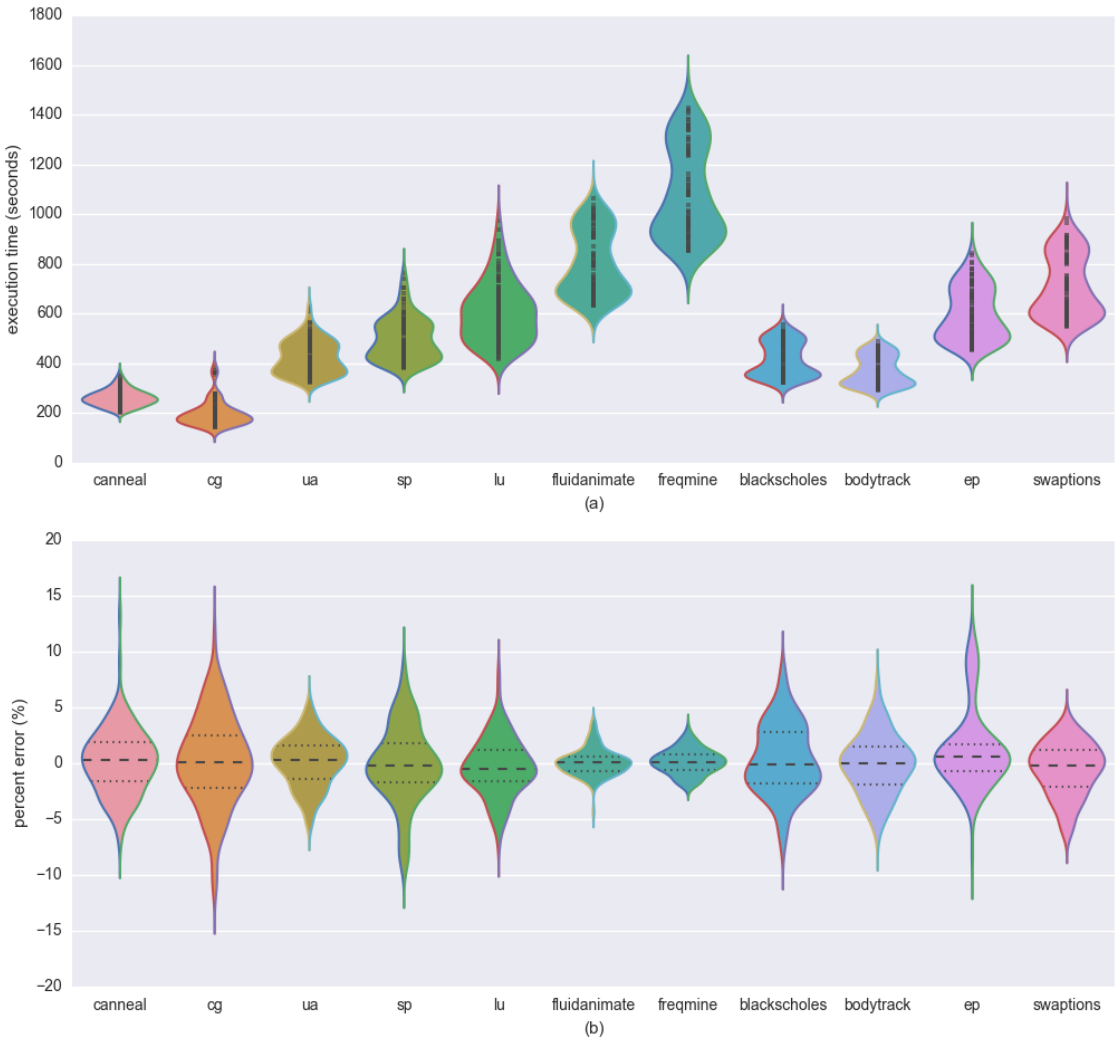

Figure 2.3: **(a)** Distributions of each application's **execution time**. **(b)** The accuracy of the neural network model using feature set F predicting execution time for each application, on the 6-core Intel Xeon E5649 machine.

neural network model using feature set F (the most accurate model) on the energy use data shown in Figure 2.4(a). Distributions of the percent error between the model's predictions of energy use and the actual energy use are shown for each application. The lines across each distribution represents the distribution's median (dashed line) and upper and lower quartiles (dotted lines). Again, the figure shows a low error in model prediction accuracy for the majority of the data points. About half of the model's predictions are approximately $\pm 2.5\%$ from the actual energy use values, and nearly all of the predictions are within about 7% of the actual values.

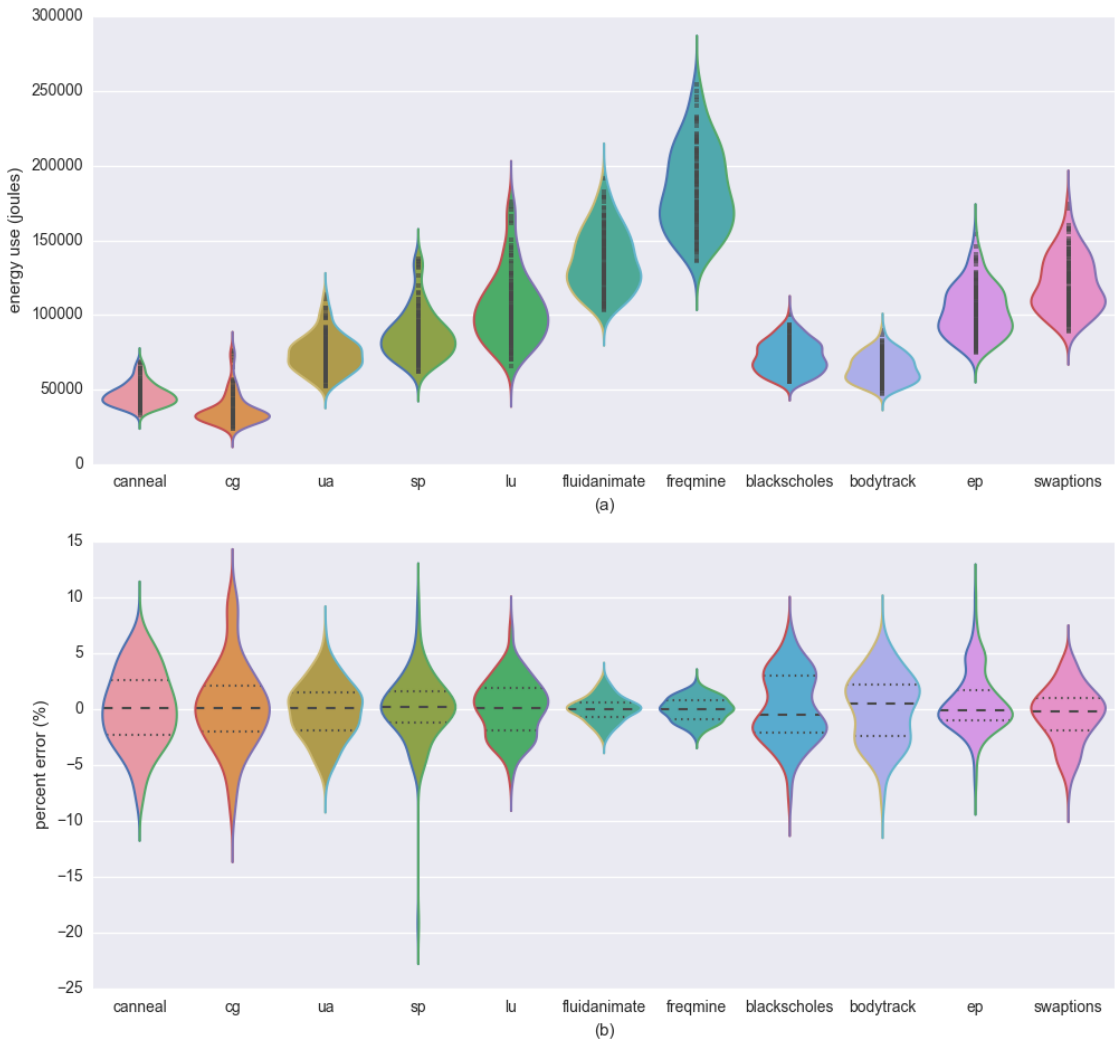

Figure 2.4: **(a)** Distributions of each application's **energy use**. **(b)** The accuracy of the neural network model using feature set F predicting energy use for each application, on the 6-core Intel Xeon E5649 machine.

Not surprisingly, by comparing the distributions of energy use shown in Figure 2.4(a) to the distributions of execution time shown in Figure 2.3(a), it can be seen that the distributions are fairly correlated. However, as discussed in Section 2.5.4, this correlation does not result in accurate energy use predictions when simply multiplying execution time and baseline power.

2.6 Prediction Model Utility

2.6.1 Overview

Predictions made about HPC system node performance are only useful if they can be shown to provide some benefit to the system. This section specifically demonstrates an example of the value of our modeling methodology's execution time prediction models when applied to scheduling compute node resources in a simulated 500-node homogeneous HPC system composed of system nodes with 4-core Intel Xeon E3-1225v3 processors. It is shown that, due to the impact of co-location interference on execution performance, scheduling with our proposed methodology can provide significant improvements in overall system performance.

2.6.2 Simulator

We designed a simulator to provide a comparison between a scheduling heuristic that is naïve to the effects of memory interference from co-location, and a heuristic that is aware of memory interference by utilizing the proposed modeling methodology. The simulator is event-based, with the ability to simulate execution of multiple applications on large-scale HPC platforms. The simulator is also capable of modeling the effects of memory interference due to co-location on individual nodes through the use of "memory interference events." At a memory interference event, applications experience a delay in their execution, resulting in an increase in their execution times that is dependent on their specific co-location scenario (i.e., which application types are running on the other cores of the processor). The amount of execution time increase experienced by a particular application for a given co-location scenario was determined empirically from data collection on the 4-core Intel Xeon E3-1225v3 processor, as discussed later in Section 2.6.3.

2.6.3 Data Collection

To ensure that the simulated HPC environment would be able to simulate memory interference from co-location as accurately as possible, additional data collection beyond what was described in Section 2.4.2 was needed. That is, an exhaustive set of data for all possible co-location combina-

tions of the applications used for simulation was collected on a real machine for these experiments to ensure accurate simulation based analysis. This complete set of data ensured a more accurate estimation of the real-world execution time for all the applications co-located on a multicore processor in the system. Because it would be prohibitively time consuming to collect application execution time information for all possible application co-locations for all of the eleven applications used for validating the methodology (the applications listed in Table 2.3), the work for this simulated study considers only a subset of these applications (*cg, sp, fluidanimate,* and *ep*, described in Section 2.4.2).

All data collection was performed using the same testing environment described in Section 2.4.1. All data was collected at the lowest numbered P-state (highest performance P-state) of the 4-core Intel Xeon E3-1225v3 machine. The data used for the simulator was collected as a series of nested loops specifying the applications to be run on each core, as outlined in Algorithm 2.

Algorithm 2 Simulator data collection

```
1: for each multicore "targetApp" in {cg, sp, fluidanimate, ep} do
2:     for each multicore "coApp1" in {IDLE, cg, sp, fluidanimate, ep} do
3:         for each multicore "coApp2" in {IDLE, cg, sp, fluidanimate, ep} do
4:             for each multicore "coApp3" in {IDLE, cg, sp, fluidanimate, ep} do
5:                 get_exec_time_of_target()
6:             end for
7:         end for
8:     end for
9: end for
```

The three processor cores that execute the co-located applications need to include an "IDLE application" in addition to the four application types *cg, sp, fluidanimate,* and *ep* to allow for application scheduling situations where no applications are scheduled on a particular core. This gave a total of 4x5x5x5 = 500 co-location scenarios for which data was collected. The exhaustive data collection described above was performed ten times. An average of each data point out of the ten sets of collected data was used for the simulation experiments.

2.6.4 Task Simulation

Simulated instances of applications are referred to as tasks. Tasks can be any of the four application types, *cg, sp, fluidanimate,* and *ep*, and upon arrival, each task's type is selected according to a uniform random distribution. This makes the number of each application type arriving into the system for scheduling equally likely.

For a given task i, the arrival time of that task into the system is defined as T_{A_i}, and is determined using a Poisson process. The first task to arrive into the system arrives at time zero ($T_{A_0} = 0$), and all subsequent tasks arrive according to the previous task's arrival time ($T_{A_{i-1}}$) plus an exponential random variable $\mathcal{T}_i \sim Exp(\lambda)$ with an expected arrival rate of $E[\mathcal{T}_i] = \frac{1}{\lambda}$. Tasks arriving in this manner allow for flexibility in adjusting the subscription level of the system by only having to modify a single parameter λ. Task arrival time is given by

$$T_{A_i} = T_{A_{i-1}} + \mathcal{T}_i \ . \tag{2.6}$$

Tasks that do not complete execution by their deadlines are removed from the system. The deadline of any given task i is denoted T_{D_i}, and is generated using the task's arrival time T_{A_i} and *baseline execution time*. Baseline Execution Time is defined in Section 2.3.3 and denoted here as the variable ET_B. Task deadlines take a uniform random value over an interval $[a, b]$ giving $T_{D_i} = \mathcal{U}_i(a, b)$, with a defined as task arrival time plus a parameter β multiplied by baseline execution time, i.e.,

$$a = T_{A_i} + \beta ET_B \ . \tag{2.7}$$

The end of the interval of the uniform random variable, b, is defined as task arrival time plus a parameter γ multiplied by baseline execution time, i.e.,

$$b = T_{A_i} + \gamma ET_B \ . \tag{2.8}$$

For all experiments shown, $\beta = 1.2$ and $\gamma = 2.0$, thus making a task's deadline equal to the task's arrival time plus a random value between $1.2 * ET_B$ and $2.0 * ET_B$. The task deadline for this work is

$$T_{D_i} = T_{A_i} + \mathcal{U}(1.2, 2.0) * ET_B \ . \tag{2.9}$$

2.6.5 Scheduling Heuristics

Overview

The goal of our scheduling heuristics is to maximize the number of tasks that complete by their deadlines. We simulate the behavior of two consolidating slack-based scheduling heuristics, one that is naïve to the effects of memory interference from application co-location (co-location naïve) and one that is aware of the effects of memory interference due to application co-location (co-location aware). These heuristics are utilized during simulated scheduling events that map tasks to processor cores. Scheduling events occur every time tasks arrive to the system, or processor cores become free. Consolidation-based schedulers attempt to maximize the number of cores that are executing tasks in each multicore processor node before powering up additional system nodes. Consolidation-based scheduling has been shown to provide benefits for HPC systems by minimizing the number of processor nodes that need to be active during a system's execution, and therefore reducing the power needs and potentially increasing the energy efficiency of the system (e.g., [7], [8]). We use a consolidation approach for our sample use of co-location in scheduling, and are aware that their are trade-offs between the consolidation approach and an approach where tasks are distributed throughout the system.

Both the co-location naïve and co-location aware heuristics use a measure of a task's slack for making scheduling decisions. Task slack is a prediction of the time a task has remaining between when it is expected to finish and its deadline. The task slack calculation is used to ensure that a task will have enough time to completely execute under a given co-location scenario, and this calculation differs between a co-location naïve heuristic and a co-location aware heuristic. The

task slack is defined in detail in Section 2.6.5. It is the goal of all scheduling heuristics used for these experiments to create schedules that maximize the number of tasks that meet their deadlines, while consolidating tasks as much as possible.

Task Slack

Calculating task slack provides an estimate of the amount of time that is available between when a task is expected to be completed and its deadline. Because predictions of task execution time are necessary for calculations of task slack, it is to be expected that a better prediction of task execution time would produce a better prediction of task slack, and consequently allow any slack-based scheduling heuristics dependent on task execution times to produce better task schedules.

For some task i in the simulated system, co-location naïve slack, S_{CN_i}, is calculated as the task's deadline (T_{D_i} defined in Section 2.6.4) minus the task's co-location naïve predicted time of completion, T_{CN_i}, calculated as the task's *baseline execution time* (defined in Section 2.3.3) plus the simulated current time (CT). Thus, the co-location naïve slack equation is

$$S_{CN_i} = T_{D_i} - T_{CN_i} \ . \tag{2.10}$$

The major difference between the equations for calculating co-location naïve and co-location aware task slack is that, instead of using the value of the task's baseline execution time for predicting a task's execution time, the co-location aware slack calculation uses a prediction of the co-location aware time at which a task will be completed, denoted T_{CA_i}. Before a task begins executing and has been assigned to a processor node the initial value of T_{CA_i} is equal to the current time of the simulator (CT) plus the task's baseline execution time value. After the task's initial completion time has been calculated, the co-location aware slack for each task i, denoted S_{CA_i}, is calculated as the task's deadline (T_{D_i}) minus the task's co-location aware completion time prediction (T_{CA_i}), i.e.,

$$S_{CA_i} = T_{D_i} - T_{CA_i} \ . \tag{2.11}$$

However, after the scheduling heuristic has assigned the task to a processor core, the time at which the task will be completed changes due to its co-location with other applications, and the value of T_{CA_i} must be recalculated. The prediction of a target task's completion time after being co-located with other tasks is challenging because:

(a) the interference a target task will experience during its execution will change over time as either the tasks that it is co-located with finish executing, or as new tasks arriving into the system are co-located with the target task during its execution;

(b) the target task itself will cause interference with the other tasks it is co-located with, increasing their execution time, and thereby changing the duration of each task's interference effects on the target task, making the execution time prediction harder.

The first problem can be solved by having the scheduling heuristic recalculate a task's slack value at every scheduling event (when tasks arrive into the system, or processor cores become free), thus ensuring that the slack value is up-to-date whenever the value needs to be used for scheduling. The second problem of finding an accurate prediction (as far as an underlying execution time prediction model will allow) for all tasks co-located on a multicore processor node despite there being a continuous complex interaction among them is solved through the iterative approximation algorithm shown in the pseudo-code in Algorithm3.

Algorithm 3 takes a set of tasks co-located on a processor, denoted **T**, and calculates the predicted completion time of all tasks in the set. Each task in the set of co-located tasks presented to the algorithm may be in various states of progress through their execution, and will have previous predictions of their completion times recorded from earlier in their execution that will be used in the algorithm.

The algorithm starts by recording the simulator's current time in the variable IS that stores the start time of the current co-location interval. The co-location interval is defined as the largest interval of time that the co-located tasks in the processor is guaranteed not to change. For example, if two tasks, A and B, are co-located on a processor then each of these tasks will have a previously

Algorithm 3 Predicted time to task completion under current co-location (T_{CA_i})

inputs: a set **T** of the tasks co-located on the same processor node
outputs: a set **T**′ of tasks with predicted completion times
let $IS = CT$
while the size of **T** > 0 **do**
    let IE be the earliest task completion time value in **T**
    for each task t in **T** **do**
        calculate PUI of t under co-location **T**
        $T_{CA_i} = T_{CA_i} + PUI * (IE - IS)$
    end for
    for each task t in **T** **do**
        if $IE + 1 \geq T_{CA_i}$ **then**
            add t to **T**′
            remove t from **T**
        end if
    end for
    $IS = IE$
end while
return **T**′

predicted completion time calculated earlier in the simulation. Let the predicted completion time of task A be $CT + 5$ and that of task B be $CT + 10$. Then for this scenario the co-location interval is from CT to $CT + 5$ which is the interval during which these tasks are guaranteed to be executing co-located together. As another scenario, consider a task C that arrives and is co-located with tasks A and B with task C having a completion time of $CT + 4$. In this scenario the co-location interval will be from CT to $CT + 4$.

The calculation of predicted unit interference PUI is where our methodology's execution time prediction models are utilized. The predicted value of a task's execution time under co-location, is denoted ET_C. After the execution time prediction has been made by the prediction model, the task's baseline execution time (ET_B) is subtracted from this value and the result is then dived by the task's baseline execution time to obtain a calculation of the increase in execution time a task experiences during a one second unit of execution for a particular co-location scenario. Thus, PUI is

$$PUI = \frac{ET_C - ET_B}{ET_B} \ . \tag{2.12}$$

The general procedure for calculating each task's completion time occurs in the *while* loop starting on line 4. An iteration of this *while* loop starts by finding the end of the current co-location interval, IE (line 5). Next, the *for* loop on line 6 updates the completion times of each task in **T** by first calculating the task's PUI according to Equation 2.12, then multiplying this value by the value of the current co-location interval ($IE - IS$), and finally adding it to the task's last predicted completion time (lines 6-8). After each task's completion time has been updated for the current iteration of the *while* loop, the *for* loop on line 10 removes tasks from **T** with predicted completion times within 1 second after the end time of the co-location interval (necessary for establishing which tasks will be in the next co-location interval), and puts them in the set $\mathbf{T}'$. The algorithm then updates the co-location interval start time of the next iteration of the *while* loop to be the current interval's end time (line 16). The algorithm continues iterating through the *while* loop until all tasks have been removed from **T**, indicating that completion time predictions have been made for all tasks co-located on the node, and returns the set $\mathbf{T}'$ of tasks with updated completion times. After the predictions of completion time under the current co-location scenario has been completed for each task in the processor node, the co-location aware slack for each task i, is calculated as defined earlier in Equation 2.11.

The neural network model F for the 4-core Intel Xeon E5-1225v3 system (the model used for execution time predictions in this simulated study) is trained just once, using the entire set of data collected as specified in Section 2.4.2. The accuracy of this execution time prediction model has a training performance MPE of 2.50% when trained with the full set of training data.

Co-Location Naïve Scheduling Heuristic

The co-location naïve scheduling heuristic is shown in Algorithm 4, and takes as input the set of "unmapped tasks" **U** that are a set of tasks that have arrived into the system but have not yet been scheduled to a processor core, and the list **N** of processor nodes with available cores. The algorithm returns a mapping of a subset of the tasks in **U** to a subset of the nodes in **N**.

Algorithm 4 starts by sorting tasks in **U** from least slack to most slack according to their slack values calculated with Equation 2.10, then stores these values in the list $\mathbf{U}'$, and initializes the

boolean variable NA, which indicates if a node is available, to false (line 3-4). Next, the algorithm enters the outer *for* loop that iterates over each task u_i in $\mathbf{U}'$ (line 5). Because the list is sorted according to slack, the tasks that are closest to their deadlines and most in need of scheduling get considered for scheduling first. At the beginning of each iteration of the outer *for* loop the set of nodes $\mathbf{N}$ is sorted from the node with the least number of available cores to the node with the most number of available cores, and stored in $\mathbf{N}'$. It is this list $\mathbf{N}'$ that is subsequently iterated over in the inner *for* loop (lines 6-7). Allowing the nodes to be iterated over in this way encourages consolidation. In line 8, the co-location naïve slack S_{CN_i} is calculated for task u_i if it were to be scheduled on node n_j. If the calculated slack prediction is greater than zero then NA is set to true, the available node n_j is recorded to the variable that stores the node with available slack $NWAS$, and the algorithm breaks from the inner loop, otherwise it checks the next processor node (lines 9-12). Once the inner *for* loop is complete, if there was a node available, then task u_i is scheduled onto the node recorded in $NWAS$, otherwise the algorithm moves on the next unmapped task (lines 15-16). After the outer *for* loop has iterated through all of the unmapped tasks, the algorithm is complete.

Co-Location Aware Scheduling Heuristic

The co-location aware scheduling heuristic operates similarly to the co-location naïve scheduling heuristic, and is shown in Algorithm 5. The only difference between the co-location aware heuristic and the co-location naïve heuristic is the scope and calculation of task slack (lines 8-9). In addition to using the co-location aware slack calculation of Equation 2.11 for its task slack predictions, the co-location aware scheduling heuristic is aware of the other tasks already executing on the node n_j and calculates slack for all the tasks on the node, not just the unmapped task u_i. When the algorithm checks the slack for all tasks in the node, it is not only checking to see if the unmapped task u_i will be able to finish executing when subjected to node n_j's co-location, but also checking to make sure that placing u_i on the node will not produce so much interference that it could make the co-located applications already executing on node n_j miss their deadlines.

Algorithm 4 Co-location **naïve** consolidating slack-based scheduling heuristic

inputs: unmapped tasks **U**, nodes with available cores **N**
outputs: a mapping of a subset of **U** to a subset of **N**
let $\mathbf{U}'$ be a sorted list of tasks **U** from least slack to most slack
initialize $NA =$ **False**
for each task u_i in $\mathbf{U}'$ with $i = 1$ to the size of $\mathbf{U}'$ **do**
    let $\mathbf{N}'$ be a sorted list of nodes **N** from least to most number of available cores
    for each node n_j in $\mathbf{N}'$ with $j = 1$ to the size of $\mathbf{N}'$ **do**
        calculate co-location **naïve** slack, S_{CN_i}, for task u_i on node n_j
        if $S_{CN_i} > 0$ **then** *// u_i can be completed on node n_j*
            $NA =$ **True**
            $NWAS = n_j$
            break
        end if
    end for
    if NA **then** *// it is predicted that task u_i can be completed*
        assign u_i to execute on node $NWAS$
    end if
end for

Algorithm 5 Co-location **aware** consolidating slack-based scheduling heuristic

inputs: unmapped tasks **U**, nodes with available cores **N**
outputs: a mapping of a subset of **U** to a subset of **N**
let $\mathbf{U}'$ be a sorted list of tasks **U** from least slack to most slack
initialize $NA =$ **False**
for each task u_i in $\mathbf{U}'$ with $i = 1$ to the size of $\mathbf{U}'$ **do**
    let $\mathbf{N}'$ be a sorted list of nodes **N** from least to most number of available cores
    for each node n_j in $\mathbf{N}'$ with $j = 1$ to the size of $\mathbf{N}'$ **do**
        calculate co-location **aware** slack, S_{CA_k}, if u_i were added to the node
        *// S_{CA_k} is calculated for **all tasks** on node n_j, with $k = 1$ to the number of tasks on the node*
        if all tasks have slack $S_{CA_k} > 0$ **then** *// u_i can be completed and will not cause missed deadlines on node n_j*
            $NA =$ **True**
            $NWAS = n_j$
            break
        end if
    end for
    if NA **then** *// task u_i can mapped without causing missed deadlines*
        assign u_i to execute on node $NWAS$
    end if
end for

Perfect Prediction Scheduling Heuristic

A third scheduling heuristic that we consider in our study demonstrates co-location aware scheduling with perfect prediction (perfect pred) of application interference. As the name suggests, the "perfect prediction" scheduler shows how the co-location aware scheduling heuristic would behave if the modeling methodology was capable of perfectly predicting an application's execution time under any co-location situation. The "perfect prediction" scheduler makes its scheduling decisions using the same scheduling procedure as the co-location aware scheduling heuristic outlined above in Section 2.6.5, except that instead of using our proposed co-location interference modeling methodology for its prediction of a task's increased execution time under co-location, its execution on a simulator allows it to predict the future execution time values under co-location obtained after profiling on a real system. Therefore a task's execution time and slack under co-location are "predicted" perfectly accurately for the purposes of scheduling.

It is not possible for such a scheduler to exist outside of a controlled simulation, but it is shown here for the purposes of providing an upper bound on what is achievable for any consolidation based co-location aware heuristic. The difference in performance between the co-location aware scheduling heuristic using "perfect prediction" and the co-location aware heuristic using the proposed modeling methodology for its predictions gives an indication of how often tasks miss their deadlines due to inaccurate predictions, instead of environmental factors of the computing system.

2.6.6 System Measurements

Overview

We use four different measures for comparing the heuristics in the system: system performance, core utilization, node utilization, and core utilization of active nodes (each defined in the following sections). Each measure gives a different view of how the heuristics behave during system simulation, and allows for a detailed analysis of the system.

System Performance Measure

After a simulation has completed, each task will be in one of three states.

(a) completed: The task was scheduled to a processor core and successfully completed at or before its deadline.

(b) missed deadline: The task was scheduled to a processor core, but was unable to be completed before its deadline. This outcome occurs when a scheduling heuristic makes a mistake in its assumption about a task's execution time (for instance, the task experiences longer execution times due to co-location interference). When a task misses its deadline it is removed from its processor core at its deadline time.

(c) unassigned: When the number of arrived tasks in the system exceeds the number of available cores (either physically available cores, or cores that a scheduling heuristic determines can be used without causing missed deadlines) then the task is put in a queue of arrived tasks waiting to be scheduled. If the task is unable to be scheduled before its deadline, then at the time of its deadline the task is removed from the system and labeled as "unassigned."

The goal of this scheduler is to complete as many tasks by their deadlines as possible, thus reducing the number of tasks that miss their deadlines or are not executed at all. The overall system performance using a particular scheduling heuristic is determined by measuring the number of tasks that meet their deadlines at the end of system simulation.

Core Utilization Measure

A system core is considered active if it is executing a task. The core utilization CU at a particular instant of the system's simulation is calculated from the ratio of the system's active cores (AC) to the system's total cores (TC), i.e.,

$$CU = \frac{AC}{TC} \, . \tag{2.13}$$

Node Utilization Measure

A system node is considered active if it has at least one active core. The system's node utilization NU is a measure of the percentage of active nodes in the system (nodes with at least

one task is running on the node). Node utilization is calculated simply as a ratio of the number of nodes with at least one active core (active nodes denoted AN) to the total number of nodes in the system (denoted TN), i.e.,

$$NU = \frac{AN}{TN} \ . \tag{2.14}$$

Core Utilization of Active Nodes Measure

The core utilization of active nodes (CUAN) is a system measure that examines how consolidated tasks are in the system by examining how "full" active nodes are in the system. If the tasks in the system are consolidated, then the value of CUAN is high, and if the tasks in the system are spread out across processor nodes, then the value of CUAN is low. The CUAN metric is calculated as the sum over all active nodes in the system of the number of active cores in each active node $ACIN_i$ divided by the total number of cores in that active node $TCIN_i$. This sum is then divided by the total number of active nodes (AN), i.e.,

$$CUAN = \frac{\sum_{i=1}^{AN} \frac{ACIN_i}{TCIN_i}}{AN} \ . \tag{2.15}$$

Because the simulated HPC system used in this study is homogeneous with each node consisting of a 4-core processor $TCIN_i$ will always be equal to four in this system. However, this equation can be generalized to a heterogeneous system with any number and distribution of cores per node.

2.6.7 Experimental Setup

Simulations were performed at two different task subscription levels. The total number of tasks that arrive at the simulated system for each subscription level remain a constant value of 8000 tasks in each simulation. This way, the only difference between simulations in one subscription level and another are the $\frac{1}{\lambda}$ values associated with the expected arrival times of the tasks (described in Section 2.6.4). The subscriptions levels are defined as oversubscribed with $\frac{1}{\lambda} = 0.1$ and undersubscribed with $\frac{1}{\lambda} = 0.25$. The oversubscribed system has tasks arriving to the system at a higher rate than the system can execute them.

The subscription levels provide a good range of situations in which an HPC system could be operating. The resulting performance of each of the scheduling heuristics for each of these subscriptions levels provides a good assessment of when the system benefits from a co-location aware scheduling heuristic.

After the arrival pattern and deadlines of the tasks are determined according to Section 2.6.4 for each of the subscription levels, the 500 node system is simulated with each of the three scheduling heuristics described in Section 2.6.5 (co-location naïve, co-location aware, and "perfect prediction") for a total of six simulated system scenarios.

2.6.8 Experimental Results

Simulation results for demonstrating the utility of the modeling methodology are shown in Figures 2.5 and 2.6 for each of the subscription levels discussed in the experimental setup from Section 2.6.7. The most important result from the simulation is its demonstration of the utility of our co-location interference modeling methodology. For both subscription levels, the heuristic performance results shown in Figures 2.5(a) and 2.6(a) indicate that the co-location aware heuristic is able to complete more tasks than the co-location naïve heuristic by avoiding missing task deadlines, while at the same time, performing competitively with the perfect prediction heuristic.

The performance of the "perfect prediction" heuristic in Figures 2.5(a) and 2.6(a) further demonstrates how many of the tasks fail to be completed because of a missed deadline (due to an inaccurate execution time prediction) or fail to be completed because they are not able to be scheduled in the system. As expected, the co-location aware scheduling heuristic with "perfect prediction" does not have a single task that misses its deadline for either of the two task subscription levels. However, even for the "perfect prediction" scheduling heuristic, tasks that are executed in an oversubscribed system (shown in Figure 2.5(a)) are sometimes unassigned, indicating that it would never be possible to complete more than approximately 78% of the tasks that arrive to the system. Given this upper bound on performance, our co-location aware scheduling heuristic performs very well. In both subscription level scenarios where we use our co-location

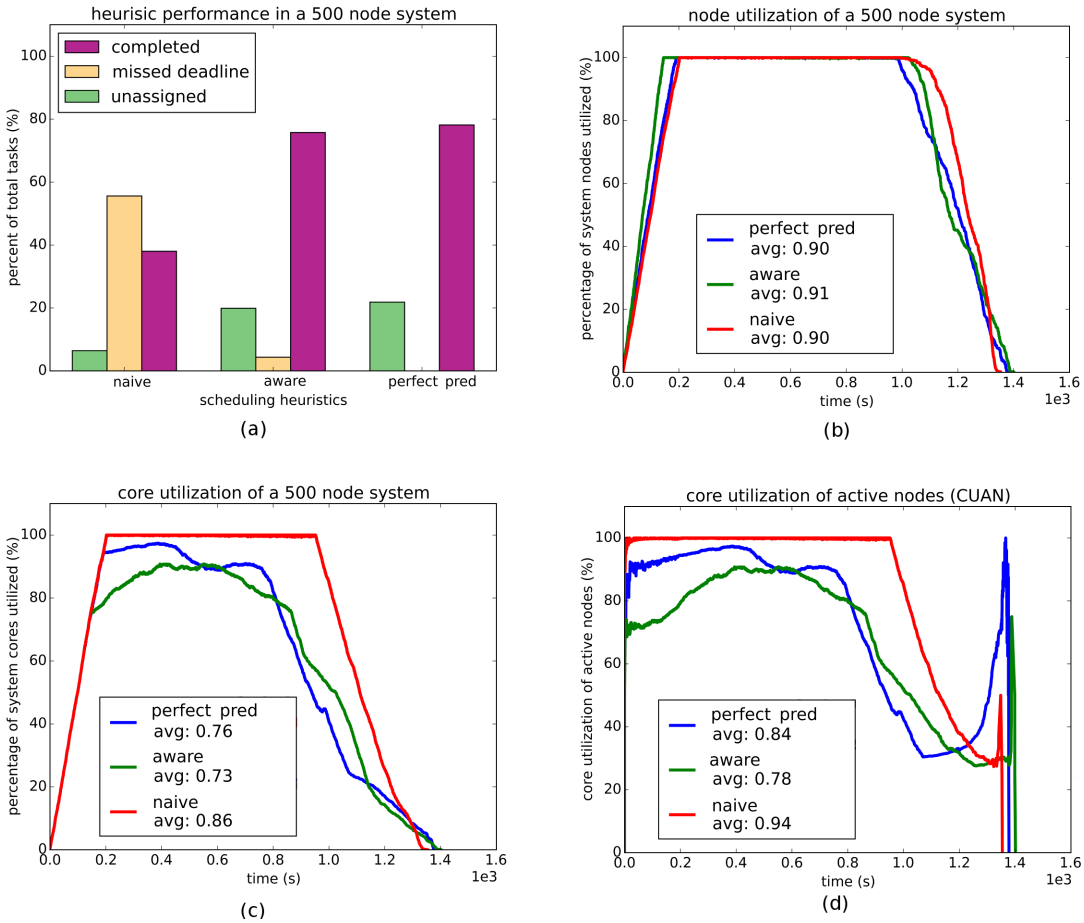

Figure 2.5: Simulation results of an **oversubscribed** 500 node homogeneous system comprised of 4-core Intel Xeon E3-1225v3 processors. **(a)** Shows scheduling heuristic performance. **(b)** Shows node utilization of the simulated system. **(c)** Shows core utilization of the simulated system. **(d)** Shows core utilization of active nodes (CUAN) of the simulated system. In (a), the purple bar shows the percentage of total tasks completed, the brown bar shows the percentage of total tasks that missed their deadlines, and the green bar shows the percentage of tasks that were left unassigned. In (b), (c), and (d) the red line indicates the naïve heuristic utilizations, the green line indicates the co-location aware heuristic utilizations, and the blue line indicates the perfect prediction utilizations.

aware scheduling heuristic, the number of completed tasks for our heuristic comes within 3% of the heuristic with "perfect prediction."

It can be observed from the results for the three utilization metrics of each heuristic shown in Figure 2.5 and Figure 2.6 that there is a trade-off for each heuristic between the subscription level of the tasks in the simulated system and each heuristic's ability to consolidate tasks in the system. For both subscription levels, the co-location aware heuristic provides worse consolidation

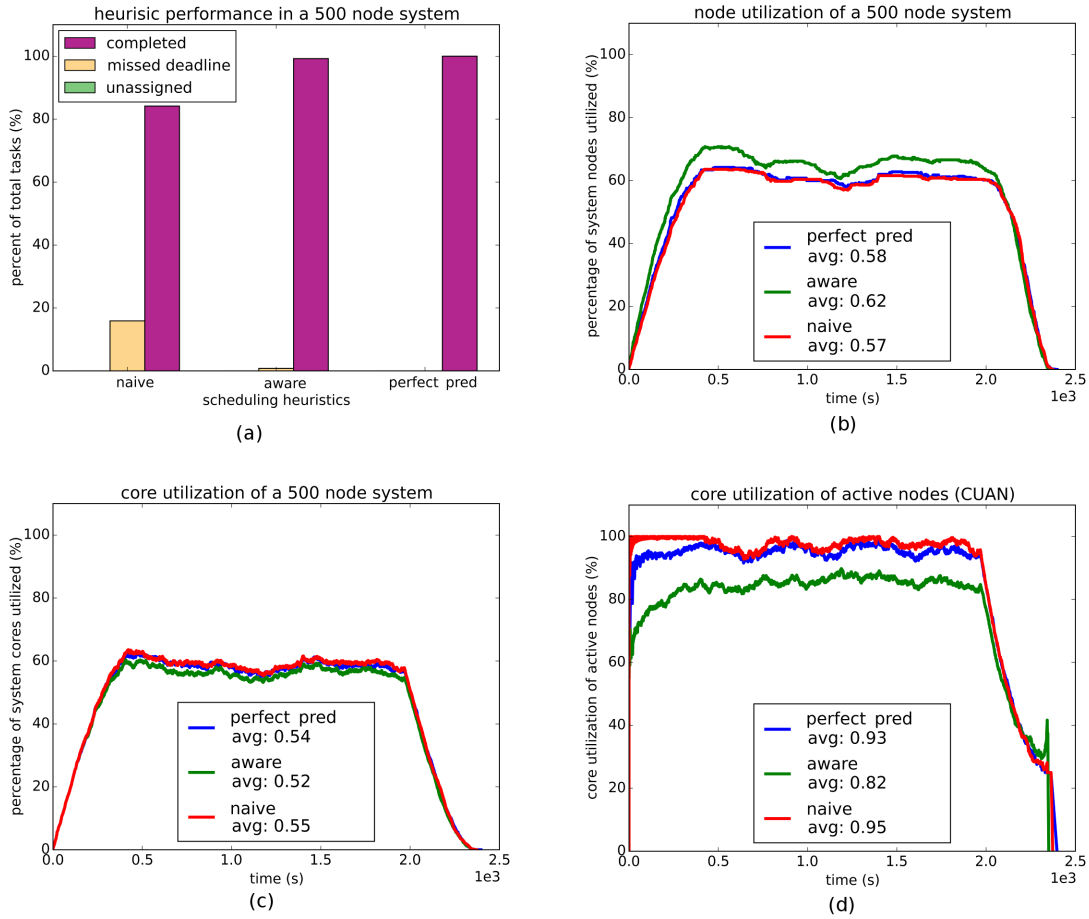

Figure 2.6: Simulation results of an **undersubscribed** 500 node homogeneous system comprised of 4-core Intel Xeon E3-1225v3 processors. **(a)** Shows scheduling heuristic performance. **(b)** Shows node utilization of the simulated system. **(c)** Shows core utilization of the simulated system. **(d)** Shows core utilization of active nodes (CUAN) of the simulated system. In (a), the purple bar shows the percentage of total tasks completed, the brown bar shows the percentage of total tasks that missed their deadlines, and the green bar shows the percentage of tasks that were left unassigned. In (b), (c), and (d) the red line indicates the naïve heuristic utilizations, the green line indicates the co-location aware heuristic utilizations, and the blue line indicates the perfect prediction utilizations.

than the co-location naïve heuristic. This is to be expected because the co-location aware heuristic specifically leaves processor cores idle if scheduling tasks to those cores would cause other tasks in the system to miss their deadlines. By examining the oversubscribed system in Figure 2.5(b),(c), and (d), it can be observed that the co-location naïve heuristic aggressively consolidates tasks in the system because the node utilization, core utilization, and core utilization of active nodes (CUAN) are all close to 100% for the majority of the system simulation. In contrast, the co-location aware

heuristic still takes advantage of all the nodes in the system (node utilization is still at 100%), but is more selective about its use of cores and therefore has core utilization, and CUAN values that peak at about 90% and do not remain constant throughout the simulation. Similar conclusions can be drawn from the results for the undersubscribed system shown in Figure 2.6. In this system, the co-location naïve heuristic still has the highest levels of core utilization and CUAN, but has a *lower* node utilization, indicating that it is more aggressively consolidating tasks in the system due to its inability to foresee the potential problems that arise when tasks are co-located on the same multicore processor.

2.7 Conclusions

We proposed a modeling methodology that predicts application execution time and energy use when under co-location interference effects caused by resource sharing among cores in a multicore processor. The methodology is general enough to be applied to any multicore processor and set of applications. To validate our methodology, its effectiveness was demonstrated by applying it to three server class Intel Xeon multicore processors with up to 12 cores, executing real data workloads from two scientific benchmark suites. After validation, the utility that such prediction models can provide was demonstrated by creating a scheduling heuristic that takes advantage of the proposed modeling methodology for its scheduling decisions.

Specifically, this work used machine learning techniques to make predictions about application performance degradation due to contention in shared cache and main memory resources when multiple applications were co-located on the same multicore processor. While simpler linear models do not provide a significant increase in prediction accuracy from the addition of application memory use information, the results from Figure 2.1 and Figure 2.2 show that neural networks can provide very accurate predictions of application execution time and energy use. For the neural network model, when using only a subset of the application features, it is still able to produce fairly accurate performance predictions. Using all the features, the neural network achieved a very small MPE of 2% and an NRMSE of around 0.01 on all processors tested. Considering that performance

degradation due to co-location can extend an application's execution time quite significantly, even a model with access to only a limited set of model features may be able to provide good enough predictions to improve the performance of schedulers in HPC systems.

Applying our methodology to create models for a large-scale simulated system enabled us to examine the benefit of a co-location aware scheduling heuristic that can provide substantial performance improvement in a simulated homogeneous 500 node system. Although the experiments in Section 2.6 were only performed for a homogeneous system of 4-core nodes, it is reasonable to expect that the benefits demonstrated can be replicated in either a homogeneous or heterogeneous system that includes processor nodes with more cores. Not only is the interference from co-location that applications experience likely to be greater in machines with more cores, but the results shown in Figure 2.1 indicate that for the more advanced feature sets the prediction models tend to perform even better for the 6-core and 12-core systems than they do for the 4-core system used for those experiments.

Chapter 3

A Performance and Energy Comparison of Fault Tolerance Techniques for Exascale Computing Systems

3.1 Introduction

System reliability has recently become a looming and unsolved problem in the field of HPC. Today's HPC systems are nearing the performance capability of one-hundred petaflops [44]. Current HPC systems experience failures on the order of every few days, but models indicate that an exascale-sized system will likely experience system failures several times an hour [45].

Contemporary HPC systems have thus far been able to successfully mitigate the effects of system failures through the use of checkpoint-based rollback recovery and redundant execution of code using additional hardware. However, as the mean time between failures (MTBF) of future HPC systems decreases, these traditional approaches to system resilience either do not scale to meet the system's increased demand for performance or require too much energy to be feasibly implemented [46].

Recent works have proposed several alternative resilience techniques that are potentially better able to handle the increasing numbers of failures associated with these larger scale systems [45] [47] [48]. However, the relative performance and energy use of each of these techniques when compared with one another is unclear.

This work was performed jointly with the full list of co-authors listed in [43]. This work is supported by the National Science Foundation (NSF) under grants CCF-1252500 and CCF-1302693. This research also used the CSU ISTeC Cray System supported by NSF Grant CNS-0923386.

This work provides a simulated comparison of the performance and energy use of three of these new resilience techniques for HPC systems. The simulated system is based on analytical models for the NAS Block Tridiagonal benchmark application [41] executing at extreme scale, as well as measurements taken from the Block Tridiagonal benchmark executed on real-world server-class processors. Each resilience technique's performance and energy use is simulated with varying system sizes and hardware component reliability levels. Analysis is performed detailing the strengths and weaknesses of each resilience technique. We conclude with a comparison between each technique's performance relative to the others, as well as each technique's performance compared to a traditional checkpoint restart resilience approach.

With this chapter we make the following novel contributions:

- we construct a methodology for simulating the execution, power consumption, and energy use of extreme-scale computing systems operating with the uncertainty of hardware failures;
- we provide a comparison of the relative performance and energy consumption of four strategies for extreme-scale high performance and cloud computing resilience.

3.2 Related Work

3.2.1 Overview

We limit the resilience techniques considered to those that are transparent to application programmers and system users. While several other techniques for mitigating the effects of system failures exist, we refer the reader to the summaries provided of such works in [49], [50], and [51]. Specifically this work focuses on examining techniques utilizing rollback recovery and redundancy.

3.2.2 Rollback Recovery

Rollback recovery based techniques rely on periodically saving the system's executing state and rolling back to an earlier state after the occurrence of a failure. Such a technique is referred to as checkpointing [52] [53]. All rollback recovery techniques rely on this notion of checkpointing in

some form. Because of their nature of loading an earlier system state after a failure, all rollback recovery techniques necessarily lose some productivity as the restarted applications must recompute work lost between the time of the failure and the time of the last checkpoint. Our work examines three types of rollback recovery techniques: checkpoint restart, multilevel checkpointing, and message logging.

Checkpointing and Restarting

Checkpointing is by far the most commonly used resilience technique employed by today's HPC systems. The most general implementation of the checkpointing technique operates by stopping the system's execution at regular intervals to save the state of all executing applications to a permanent storage device, typically a parallel file system (PFS). Such a checkpointing technique is called a blocking, coordinated checkpoint [49]. Several variations and improvements on this technique have been made since its initial inception. Attempts have been made to create non-blocking or semi-blocking checkpointing, which allows the system to continue to execute while checkpoints are saved to permanent storage [54] [55]. Attempts have also been made to allow for uncoordinated checkpoints of the system, preventing the need for all processes in the system to restart when a failure occurs [56]. However, the length of time associated with checkpointing, restarting, and recomputing work lost to a system failure, and the frequency that the system needs to take checkpoints for very large scale applications with any of these checkpointing techniques has been shown to provide diminishing returns with increasing system sizes. Traditional checkpointing alone is thus not expected to be capable of providing resilience to systems at exascale sizes.

Multilevel Checkpointing

Multilevel checkpointing involves multiple levels of checkpointing, each level offering a trade-off between the time required by the system to checkpoint or restart, and the level of failure severity from which the checkpoint can recover [47]. In general, checkpoints of different levels correspond to saving data to different levels of the memory subsystem or allow for saving checkpoints across the memory of one or more partner nodes. Checkpoint levels also may employ various encod-

ing techniques (such as RAID or Reed-Solomon coding) to improve the resilience offered by a particular checkpoint level [47] [57]. Attempts also have been made to reduce checkpointing's dependence on the PFS [58] [59] [60]. One challenge associated with using multilevel checkpointing is in determining the optimal number of checkpointing levels and the optimal computation intervals between checkpoints at each level. Various solutions to this problem have been proposed [47] [61] [62].

Message Logging

Message logging attempts to provide resilience to a system by recording messages sent among processes to create snapshots of the system's execution distributed across system memory [63]. When a failure occurs, the failed node is able to use messages stored in the memory of other system nodes to save on the amount of rework that is performed by the system when restarting from a checkpoint on a failure [64].

3.2.3 Redundancy

Redundancy based techniques improve a system's reliability by executing redundant copies of the same piece of code [65]. It is possible to implement redundancy in either hardware or software [50], but in either case the improved reliability comes at a cost of using additional resources.

Because of its necessity for using more resources, redundancy alone is typically not considered to be viable resilience solution for exascale systems. However, recent been made to allow the system to utilize redundancy in less resource-intensive ways. Dynamic redundancy allows for the executing application to choose a subset of processes for redundant execution [66]. Partial redundancy combines redundancy with checkpointing, and allows for applications to redundantly execute a portion of processes in the system, providing improved resilience for part of the system, while using only a portion of the necessary system resources [48].

3.3 HPC System Simulator

3.3.1 Overview

Because no exascale system is currently available, we perform experiments via simulation. We have created an event-based simulator capable of simulating application execution on arbitrarily large systems [22] [4]. Simulated applications execute under the influence of randomly generated system failures, but are able to employ various resilience techniques capable of helping to mitigate the impact that those failures have on the application's execution. Throughout an application's simulated execution, values of execution time and energy use are recorded for the simulated events associated with computation (execution toward the application's completion), checkpoints (saving a backup of the application's current computation progress), restarts (restoring the application progress saved in the last system checkpoint after a failure occurs), recovery (recomputing progress lost to a failure after the system has restarted), and failed checkpoints or restarts (when failures occur during checkpoint or restart events).

With the exception of computation events, the actions taken for each simulated event are determined according to the resilience technique employed by the simulated system, as outlined in Section 3.3.3.

3.3.2 Modeling System Failures

The probability of failures occurring in HPC systems are commonly modeled according to exponential distributions [67]. Our simulator follows this assumption and generates failures according to a Poisson process, with each new failure arriving according to the previous failure's arrival time ($T_{A_{i-1}}$, with $T_{A_0} = 0$) plus a random variate generated from an exponential distribution $\mathcal{T}_i \sim Exp(\lambda_{sys})$ with an expected arrival rate of $E[\mathcal{T}_i] = \frac{1}{\lambda_{sys}}$. The parameter λ_{sys} indicates the average failure rate of the entire system, and is defined as the number of nodes in the simulated system, N_{sys}, divided by the MTBF of the system nodes, $MTBF_{Node}$,

$$\lambda_{sys} = \frac{N_{sys}}{MTBF_{Node}} , \tag{3.1}$$

with N_{sys} and $MTBF_{Node}$ explicitly defined in Sections 3.3.3, 3.5.2, and 3.5.3 for each experiment.

In addition to the time at which failures occur, some resilience technique's behavior depends on having information about which system node has failed (in the case of redundancy) and the severity of the failure (in the case of multilevel checkpointing). Failure generation in our simulator accommodates both of these additional failure attributes. When determining which node has failed the simulator assumes a uniform random distribution over all active nodes in the system and selects one node at random as the failed node. The level of failure severity is determined by the type of system failure and the implementation of the multilevel checkpointing technique. We have assumed the multilevel checkpointing implementation of [47]. During simulation, the probability of having a failure at a severity level j is determined by the ratio of the number of failures that occur at failure severity level j, λ_{L_j}, to the total number of failures, $\lambda_{L_{tot}}$, for failures measured from a system over an extended interval of time. The resulting discrete set of probabilities for each of the j levels is used to create a probability mass function. During simulation, random variates are sampled from this probability mass function to produce integers that define the severity level of each failure.

3.3.3 Resilience Technique Simulation

Four resilience techniques have been implemented in our simulator. A traditional checkpoint restart based technique, *Checkpoint Restart*, as well as three techniques proposed for next-generation HPC systems. The multilevel checkpointing approached described in [47], *Multilevel Checkpoint*, an implementation of message logging outlined in [45], *Parallel Recovery*, and a technique combining traditional checkpointing with partial redundancy of the system hardware in [48], *Redundancy*. We now describe the implementation of these techniques.

Checkpoint Restart

The Checkpoint Restart technique implemented in our simulator is the traditional periodic, blocking, coordinated checkpointing technique, with its checkpoints saved to a PFS. This basic

strategy for checkpointing has been the baseline that is compared against by many contemporary works in HPC resilience. All three of the next-generation resilience techniques explored in this work not only use this traditional checkpointing technique as a baseline for comparing their own work, but all of them, to some extent, base their own improved techniques on this approach.

Multilevel Checkpointing

The approach in [47] implements a three-level checkpointing model. Associated times for checkpointing, $T_{C_{Lj}}$, times for restarting, $T_{R_{Lj}}$, and failure severity ratios, $\frac{\lambda_{L_j}}{\lambda_{L_{tot}}}$, are defined according to [47]. The optimal checkpoint intervals at each level also are determined according to the Markov model in [47].

Parallel Recovery

The technique in [45] adapted in our simulator is an improvement to the message logging resilience technique. Parallel Recovery allows for faster recovery from a system failure by allowing the failed node's work to be temporarily parallelized across several nodes after being restarted, thereby reducing the time needed by the system to recompute the work lost to a failure. As with all message logging techniques, parallel recovery benefits the system by allowing functional nodes being used by the executing parallel job that included the failed node to remain idle while the failed node recomputes work lost due to a failure. This decreases both the system power needed during recovery as well as the chance that a failure will interrupt the recovering system in comparison to other checkpoint based techniques. However, unlike other message logging techniques, parallel recovery improves checkpointing and restart time by utilizing in-memory checkpointing as outlined in [68].

Partial Redundancy

The Partial Redundacy technique in [48] combines traditional checkpointing with varying degrees of hardware redundancy. "Partial" redundancy is achieved by allowing only a fraction of the total system nodes required by the executing application to have redundant hardware during its

execution. For example, a degree of redundancy of $r = 2.5$ dictates that each *virtual process* of an executing application requiring a single node will have at lease two physical nodes performing the same computation, and half of the virtual processes will have three physical nodes performing the same computation. At the same time, checkpoints are taken by the system at regular intervals. When failures occur on system nodes, the system only requires a restart if failures occur on all (possibly redundant) physical nodes associated with one of the application's virtual processes before the next checkpoint.

3.4 Exascale Modeling Methodology

3.4.1 Overview

The foundation of each of our simulated system experiments is either based on measurements taken from a real-world system or extrapolated from analytical equations. This section details the methodology we use to construct various system setups for simulating applications executing at extreme scales.

3.4.2 NAS Block Tridiagonal Benchmark at Extreme Scales

Without access to an exascale system it is not possible to directly measure application performance, energy use, or behavior of the application operating when employing a given resilience technique. However, the work in [69] provides an asymptotic analysis of the execution of the NAS Parallel Benchmark's Block Tridiagonal (BT) application [41] at extreme scales. An instance of the BT application executes with: P MPI ranks (with each rank mapped to a single CPU core), $q = \sqrt{P}$ data blocks per MPI rank, a problem size of N^3 total data points, and $(\frac{N}{q})^3$ data points per data block. Analysis in [69] indicates that, for each executed time step of the application, each MPI rank will have $2994\frac{N^3}{q^2}$ total floating point operations of computation work, occupy $368\frac{N^3}{q^2}$ bytes of data in memory, send a total of $6q$ messages between MPI ranks, and transmit a total of $1320\frac{N^2}{q}$ bytes of messages.

Because the BT application exhibits weak scaling, by definition as the number of MPI ranks P increases to P' the computational work per MPI rank remains constant. With the computational work of each MPI rank constant, it is shown in [69] that the amount of memory used by each MPI rank also remains constant, and that the data volume of communications increases by a factor of $\sqrt[6]{P'/P}$ to give a total scaled communication volume of

$$V = 1320\frac{N^2}{q}\sqrt[6]{P'/P}\,, \tag{3.2}$$

sent in a total of

$$K = 6\sqrt{\frac{P'}{P}} \tag{3.3}$$

messages. For a network latency of L converted to units of seconds and a bandwidth of BW in units of gigabytes per second, using Equations 3.2 and 3.3 we can derive the time spent on communication in each time step as

$$T_{COMM} = KL + \frac{V}{BW}\,. \tag{3.4}$$

For a node with N_{cores} number of cores, executing at R_{FLOPS} floating point operations per second, the time spent on computation in each time step, for each MPI rank is

$$T_{COMP} = \frac{2994N_{cores}}{R_{FLOPS}}\frac{N^3}{q^2}\,. \tag{3.5}$$

3.4.3 Real-World System Measurements

Several parameters required for performing experiments with our simulated system require real-world measurements of execution of a node. Node measurements were taken from the execution of an HP Z820 workstation [70]. The target Z820 machine used has two sockets, with each socket supporting a 12-core Intel Xeon E5-2697v2 processor [36], for a total of 24 cores in the compute node.

System Power measurement

Power measurements of the Z820 system were taken using a "Watts Up? PRO" power meter [39]. The "Watts Up? PRO" monitors the instantaneous power use of the Z820 at the "wall outlet" level, recording samples at one second intervals. Power use of the system was measured for the BT application during application execution, P_{NODE}, application checkpointing, P_{NODE_C}, and application restarting, P_{NODE_R}. The system idle power, P_{idle}, was measured by recording system power use during execution of the Linux sleep command.

Checkpoint and Restart Measurements

Checkpointing and restarting of the BT application was performed on our target system using the Distributed Multithreaded Checkpointing (DMTCP) system [71]. The BT application was executed using 25 MPI ranks (making most efficient use of the Z820 system node), and checkpoint and restart times were measured natively by DMTCP. Checkpointing to RAM was measured by writing to and reading from a system RAM disk. Checkpoint time to and restart time from RAM were measured from an average of ten samples as $T_{C_{RAM}} = 0.34$ seconds and $T_{R_{RAM}} = 0.24$ seconds, respectively.

Node Performance

Measurement of the Z820 system node performance was accomplished using processor performance counters. Performance counters are built into the Intel Xeon E5-2697v2 hardware [36] and allow the user to monitor hardware events during an application's execution, with minimal impact on the application's performance [33]. The system node performance parameter (R_{FLOPS}) was calculated by monitoring the floating point operations executed by the BT application and averaging that total by BT's execution time.

3.4.4 Communication Power Model

The power used for system communication is calculated for each system node. Given a switch power of $P_S = 100$ watts, the number of nodes linked by a single switch (node concentration)

of $NC = 12$ nodes, and given that a network interface controller of a single node consumes $NIC = 10$ watts at full utilization [72], the power spent by a single node for communication during computation is

$$P_{COMM_{Comp}} = \frac{P_S}{NC} + NIC\frac{T_{COMM}}{T_{COMP} + T_{COMM}} \ . \tag{3.6}$$

The power spent by a single node for communication during a time of high network traffic, such as during a checkpoint or restart (CR), is

$$P_{COMM_{CR}} = \frac{P_S}{NC} + NIC \ . \tag{3.7}$$

3.4.5 Resilience Technique Specific Parameters

In addition to the parameters measured for a system node's execution, each of the four resilience techniques implemented in the simulator requires its own set of parameters to govern the technique's execution.

Checkpoint Restart Parameters

Checkpoint Restart reads and writes its checkpoint data to a PFS. For our experiments, we assume an equal checkpoint and restart time using a PFS of $T_{C_{PFS}} = T_{R_{PFS}} = 20$ minutes, as observed in prior work [46]. The optimal checkpoint period, τ, is derived from the system's checkpoint time and failure rate according to [73] as

$$\tau = \sqrt{\frac{2T_{C_{PFS}}}{\lambda_{sys}}} - T_{C_{PFS}} \ . \tag{3.8}$$

Multilevel Checkpointing Parameters

Our experiments use a simulated Multilevel Checkpointing system of three levels. The first level writes checkpoints to the node's local RAM, the second level writes checkpoints to RAM in a partner node, and the third level checkpoint is written to a shared PFS. Additionally, whenever a

higher level checkpoint is taken, all lower level checkpoints are taken simultaneously, with lower level checkpoint time masked by the time of the higher checkpoints, as outlined in [47].

The time used for taking a level one checkpoint, $T_{C_{L1}}$, is $T_{C_{L1}} = T_{C_{RAM}}$ and time for a level one restart, T_{RL1}, is $T_{R_{L1}} = T_{R_{RAM}}$. Checkpointing to and restarting from partner nodes are each expected to take about two minutes [45]. We use this same assumption for our simulation experiments, setting level two checkpoint and restart times to $T_{C_{L2}} = T_{R_{L2}} = 2$ minutes. Level three checkpoints to the PFS are defined to be twenty minutes. The optimal computation interval and number of lower level checkpoints taken before taking a higher level checkpoint is determined using the Markov model in [47].

As outlined in Section 3.3.2, multilevel checkpointing requires knowing the probability of failures at each severity level. We derive these values from data presented in [47] as the level one, two, and three failure ratios $\frac{\lambda_{L1}}{\lambda_{L_{tot}}} = 0.308$, $\frac{\lambda_{L2}}{\lambda_{L_{tot}}} = 0.545$, and $\frac{\lambda_{L3}}{\lambda_{L_{tot}}} = 0.147$, respectively.

Parallel Recovery Parameters

For our simulation study, we assume that the number of system nodes available for parallel recovery after a failure is $\phi = 8$. We assume that this level of parallelism will reduce the time for recovery by a factor of $\sigma = 7.26$, as measured in [45]. The temporary slowdown of the application due to load imbalance after recovery and until the next checkpoint is assumed to be $\Lambda = \frac{\phi+1}{\phi}$. As with multilevel checkpointing, the checkpoint and restart time for parallel recovery from RAM is two minutes.

Partial Redundancy Parameters

All parameters associated with the Partial Redundancy resilience technique remain the same as the Checkpoint Restart technique, except for the inclusion of the parameter r, indicating the system's degree of redundancy. We vary the parameter r from $1.25\times$ to $2\times$ redundancy based on the experiment being performed, as described in Section 7.4.

3.4.6 Simulated System Setup

Studies are performed using three different simulated systems: small scale, sunway, and exascale. Values for the parameters of each simulated system are presented in Table 3.1.

Small Scale System

The *small scale* system is a homogeneous system composed of 10,923 nodes. The small scale system's nodes are based on the same HP Z820 machines that were used in the real-world system measurements in Section 3.4.3. Measurements for network latency and bandwidth were taken from [74]. Parameter values for the small scale system are listed in the second column of Table 3.1. The power and performance values listed are averaged over ten samples, and are rounded to the nearest integer.

Sunway System

The *sunway* system is our recreation of a system capable of performing similar to China's Sunway TaihuLight supercomputer, the world's highest performing system as of November 2017 [44]. The third column of Table 3.1 shows the system parameters we use to simulate the sunway system. While the sunway system has a theoretical peak performance of 125 petaflops, we run our experiments with the simulated system node performance parameter (R_{FLOPS}) calculated from the sunway's maximum measured performance of 93 petaflops. We have assumed a linear increase in power consumption of checkpointing, restarting, and idling from the small scale system to the sunway system. We also assume that the time that it takes for a sunway system node to checkpoint and restart are the same as the values used for our small scale system. All other values for the sunway system's simulation parameters are taken from [75].

Exascale System

Similar to the sunway system the *exascale* system is a homogeneous system. Each system node is similar in architecture to the sunway system, but we assume that at the time of an exascale machine's construction the number of computation processing entities per node will have doubled

Table 3.1: Simulated System Parameters

parameter	small scale	sunway	exascale
P	262,144	10,647,169	134,217,728
N_{sys}	10,923	40,960	260,112
N_{cores}	24	260	516
R_{FLOPS}	27 GFLOPS	2271 GFLOPS	4507 GFLOPS
P_{NODE}	312 watts	375 watts	375 watts
P_{NODE_C}	278 watts	334 watts	334 watts
P_{NODE_R}	278 watts	334 watts	334 watts
P_{idle}	127 watts	153 watts	153 watts
L	1.2μs	1μs	0.8μs
BW	40 Gb/s	96 Gb/s	192 Gb/s
TS	100	100	100
performance	295 TFLOPS	93 PFLOPS	1.2 EFLOPS

P: number of processor cores (MPI ranks) used by BT;
N_{sys}: number of system nodes;
N_{cores}: number of cores per node;
R_{FLOPS}: performance of a node;
P_{NODE}: power consumed by a node during computation;
P_{NODE_C}: power consumed by a node when checkpointing;
P_{NODE_R}: power consumed by a node when restarting;
P_{idle}: power consumed by a node when idle;
L: network communication latency;
BW: network communication bandwidth;
TS: timesteps executed by BT (determines application execution time).
performance: calculated total system performance

from 256 to 512, with each node still having an additional four cores used for both computation and node management. The number of system nodes is then scaled to perform at an exascale level. Parameter values for the exascale system are displayed in the fourth column of Table 3.1. With the exascale system, we assume that node power consumption will continue to decrease, allowing system node power to remain the same as for the sunway system, despite the increase in the number of cores per node. We also assume internode communication will continue to improve, allowing an exascale system to have a network latency $L = 0.8\mu$s and a bandwidth $BW = 192$ Gb/s. Using the scaled value for node performance, an instance of the BT application executing at a problem size requiring 134,217,728 MPI ranks would require 260,112 system nodes, and operate at 1.2 EFLOPS.

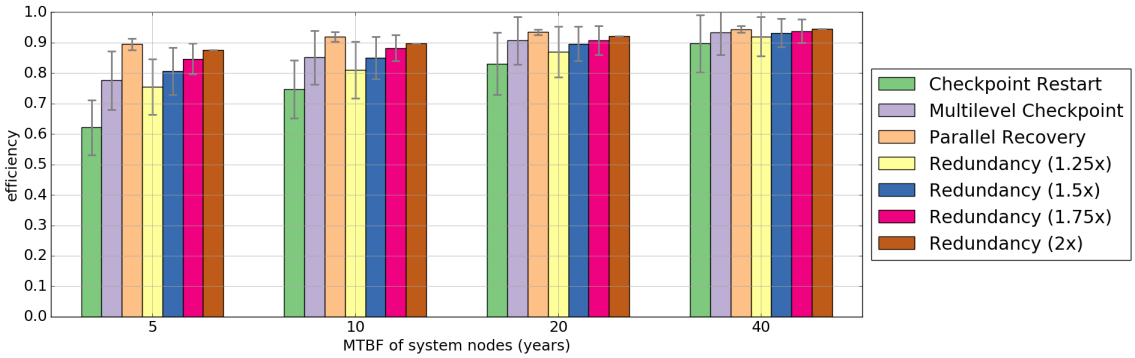

Figure 3.1: Resilience technique efficiency at various levels of system node reliability. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures or checkpointing). Each bar in the figure represents the average of 200 trials. Standard deviations are shown for each bar.

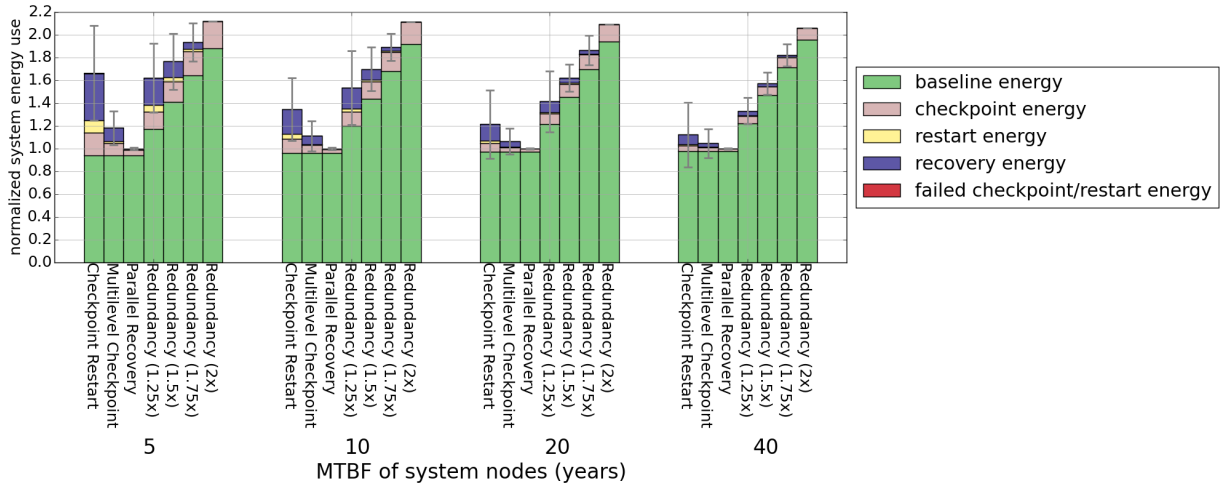

Figure 3.2: Resilience technique energy use for four different levels of system node reliability. The height of each bar in the figure depicts the normalized total energy consumed by each technique for each study. The colors of each bar represent how system energy was consumed. Total energy consumption is normalized to the Parallel Recovery technique at each respective level of system node reliability. Each bar in the figure represents the average of 200 trials. Standard deviations of each technique's normalized total energy consumption are also shown for each bar.

3.5 Simulation Studies

3.5.1 Overview

We utilized our simulation environment to conduct two sets of studies. Each study simulated the BT application according to the methodology presented in Section 3.4. We refer to each independent simulation of a study as a *trial*. The outcomes of each trial vary based on the randomness

associated with modeling failures outlined in Section 3.3.2 For each study we executed 200 independent trails, recording in each trial the amount of simulated time it took the BT application to run to completion, as well as the energy used by the system during computation, checkpointing, restarting, recovering after restarts, and failed checkpoints and recoveries.

3.5.2 System Node Reliability

The first set of studies focuses on investigating how each of the four resilience techniques behaves under varying levels of system node reliability. We ran each simulation on the small scale system, varying the system node reliability by modifying the $MTBF_{Node}$ parameter to values of 5, 10, 20, and 40 years of mean time between failures. We tested each level of system reliability with each of the four resilience techniques. For the Partial Redundancy technique, we explored four different degrees of system redundancy, $1.25\times$, $1.5\times$, $1.75\times$, and $2\times$. The results of the experiments are shown in Figures 3.1 and 3.2. In each figure, groupings of bars along the x-axis indicate the different values of $MTBF_{Node}$.

Figure 3.1 shows the *efficiency* of each resilience technique. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures or checkpointing). The color of each bar indicates the resilience technique in the experiment, and the height of each bar indicates the average efficiency of the trials.

Figure 3.2 shows the normalized system energy consumption of the same experiments shown in Figure 3.1. Each bar in Figure 3.2 has been normalized to the average value of the Parallel Recovery experiment data in each $MTBF_{Node}$ grouping. The colors of each bar indicate the breakdown of energy consumed in each experiment, with the colored portions of each bar representing the portion of total energy used for each type of simulated event defined in Section 3.3. The height of each bar indicates the average of the total normalized energy consumed for the trials in the experiment.

Comparing the results in Figure 3.1 and Figure 3.2 it can be observed that the Parallel Recovery technique provides the most efficiency for every node reliability level except $MTBF_{Node} = 40$,

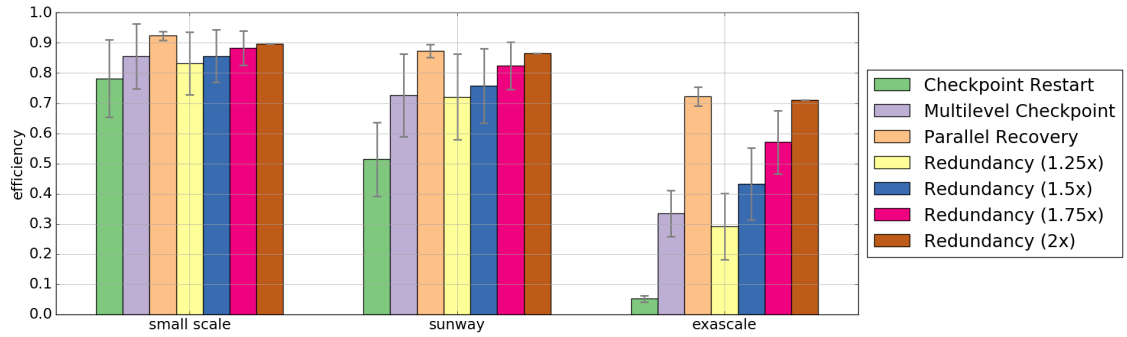

Figure 3.3: Resilience technique efficiency at various system sizes. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures or checkpointing). Each bar in the figure represents the average of 200 trials. Standard deviations are shown for each bar.

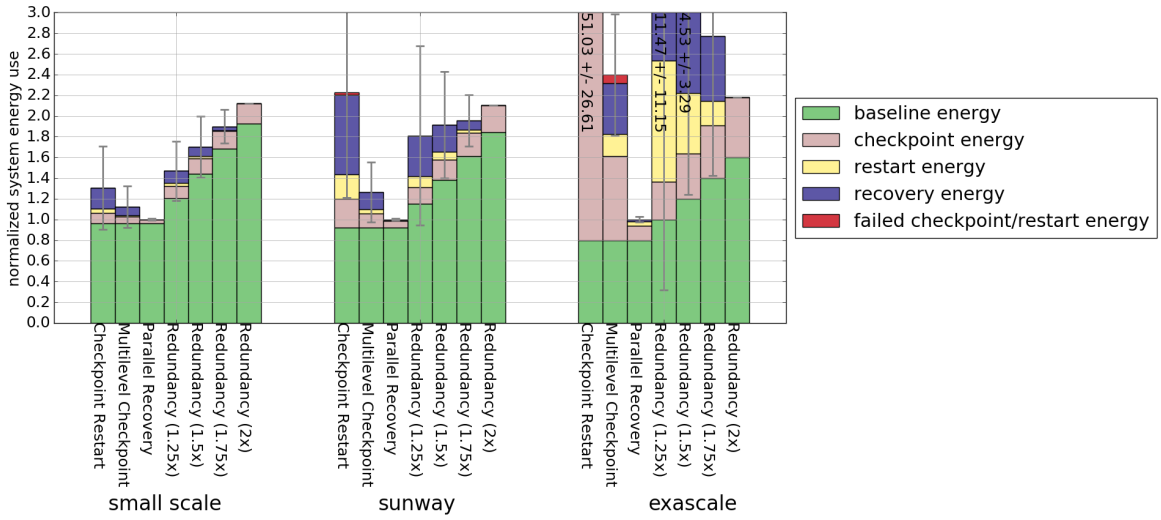

Figure 3.4: Resilience technique energy use at various system sizes. The height of each bar in the figure depicts the normalized total energy consumed by each technique. The colors of each bar represent how system energy was consumed. Total energy consumption for each system size is normalized to the Parallel Recovery technique. Each bar in the figure represents the average of 200 trials. Standard deviations of each technique's normalized total energy are shown for each bar. Annotations in the subset of the exascale results represent each truncated bar's average value and standard deviation.

and uses the least amount of energy for all node reliability levels. However, while the Partial Redundancy technique is more efficient in one case, the system efficiency provided by the Partial Redundancy technique clearly comes at a high energy cost, due to the increase in total system nodes necessary to utilize the technique.

As is to be expected, all resilience techniques result in an increase in efficiency with more reliable system nodes (increasing values of $MTBF_{Node}$). From the standard deviations of each

experiment, it can be observed that Parallel Recovery and Partial Redundancy with a redundancy degree of $2\times$ are much more consistent (have smaller standard deviation values) than the other resilience techniques.

3.5.3 System Size Scalability

The second set of simulations investigates how each of the four resilience techniques behave as the system size increases. We simulated each of the resilience techniques from the studies in Section 3.5.2, but varied the simulated system among the three systems described in Section 3.4.6. A node MTBF of $MTBF_{Node} = 10$ years was used for all simulations. The results from our system size scalability simulations are in Figure 3.3 and Figure 3.4, with each figure's organization similar to Figure 3.1 and Figure 3.2, except that bar groupings indicate system size instead of node reliability. Each of the results in Figure 3.4 has been normalized to the result for the Parallel Recovery technique. The Checkpoint Restart experiments and the Partial Redundancy experiments with a degree of redundancy of $1.25\times$ and $1.5\times$ require an excessive amount of energy and the bars depicting their results have been truncated, with the averages and standard deviations of their respective sets of trials indicated on each respective bar.

Examining the results in Figures 3.3 and 3.4 it is clear that only the Parallel Recovery technique and the Partial Redundancy technique with a degree of redundancy of $2\times$ are capable of scaling to exascale system sizes with any reasonable execution efficiency. However, Partial Redundancy is outperformed by Parallel Recovery for every system size, and use of the Partial Redundancy technique also comes at a high energy cost relative to Parallel Recovery. Examining the results shown for the sunway system, it can be observed that even for today's largest HPC system sizes (i.e., sunway) the energy cost of using traditional checkpointing is larger than for Partial Redundancy, despite the high energy cost of using additional (redundant execution) nodes with Partial Redundancy. Checkpoint Restart also provides far less benefit than the other techniques from an efficiency standpoint, indicating that there exist better alternatives to the traditional Checkpoint Restart technique even for HPC systems being used today.

3.6 Conclusions

We performed simulations comparing the performance and energy use of four HPC and cloud computing resilience techniques. The techniques considered are the traditional Checkpoint Restart technique, as well as Multilevel Checkpointing, Parallel Recovery, and Partial Redundancy, three techniques proposed for next generation large-scale HPC systems. Our results indicate that Parallel Recovery and Partial Redundancy with a degree of redundancy of $2\times$ are the best at maintaining the performance of executing applications, both with decreasing system node reliability, and increasing system sizes, from today's largest computers through to exascale-sized machines. When comparing system energy efficiency, the Parallel Recovery technique was the most energy efficient in all system sizes and node reliabilities.

Chapter 4

Resilience-Aware Resource Management for Exascale Computing Systems

4.1 Introduction

As the computing power of large-scale computing systems increases exponentially over time, the failure rates of these systems have increased exponentially as well. While most current large-scale computing systems experience failures of some type every few days, projection models indicate that the next generation of these machines will experience failures many times an hour [46]. On average, today's 13.3 petaflop Blue Waters system experiences an application failure every 15 minutes. If these failed applications were not able to be recovered through the use of the traditional checkpoint and restart resilience protocol their failed execution would likely cost the system over $\$400$ K in energy wasted due to unrecoverable failed applications [78]. However, the resilience protocols implemented in today's HPC and cloud computing systems are impractical at the exascale level, due to their high overheads [45]. Several new promising resilience protocols have recently been proposed for next generation HPC systems [45], [47], [48].

Unfortunately, little work has been done to assess the performance of these emerging protocols on a common computing environment [79]. This not only makes the relative strengths and weaknesses of these protocols unclear, but also makes assessing the underlying assumptions about each protocol's behavior less consistent. Our work provides a methodology for simulating the execution of applications operating at exascale-like system sizes in the presence of uncertainty due

This work was performed jointly with the full list of co-authors listed in [76]. This work was supported by the NSF under grants CCF-1252500 and CCF-1302693. This work utilized CSU's ISTeC Cray system, which is supported by the NSF under grant number CNS-0923386. The authors thank HP of Fort Collins for providing us some of the machines used for testing. A preliminary version of portions of this work appeared in [77].

to failures across the system. We use our methodology to model an exascale computing environment and utilize this environment to simulate four resilience protocols: one contemporary protocol (checkpointing and restarting) and three protocols proposed for use in future systems (multilevel checkpointing, parallel recovery, and partial redundancy). Using this common environment, we simulate each resilience protocol's performance as greater demands are placed on each one through increasing system utilization for a set of applications with a variety of execution characteristics.

We developed a set of synthetic benchmark applications inspired by an analysis of today's scientific benchmark suites operating at scale [69]. The resulting benchmarks provide the simulated exascale system with a set of applications that have a diverse range of execution characteristics capable of scaling to extreme sizes. We demonstrate how application performance compares when using each resilience protocol and identify the trade-offs present for different combinations of applications and resilience protocols.

We also analyze an exascale-sized system under a typical HPC use-case scenario as it is utilized over a period of days to weeks to service a large number of submissions of applications with a wide variety of execution characteristics. We show the impact that failures and overhead from employing resilience have on this environment and the level of benefit that each resilience protocol would provide to such a system. We conclude by utilizing our analyses of resilience protocol performance and trade-offs to demonstrate how resource management can be used to improve system performance by making "resilience-aware" scheduling decisions. The resulting resource management techniques are able to both select the resilience protocol likely to provide the best efficiency for each application (based on the execution characteristics of the application), as well as providing the resource manager with more accurate predictions of application execution time in the presence of failures and overhead from resilience.

In summary, in this chapter we make the following novel contributions:

- we create a simulation-based methodology capable of modeling and analyzing the execution of applications in an exascale environment;

- we develop a set of exascale synthetic benchmark applications inspired by modern scientific benchmarks;

- we provide a performance comparison of four state-of-the-art HPC resilience protocols (checkpoint restart, multilevel checkpointing, parallel recovery, and partial redundancy) operating over the simulated execution of applications with a diverse range of execution characteristics and sizes;

- we analyze the behavior of a simulated exascale system over an extended period of time when executing many applications under the influence of several strategies for HPC resilience and application scheduling, and demonstrate that overhead from failures and resilience protocols negatively affect system resource management;

- we analyze some of the assumptions made by the parallel recovery protocol [45] and show that, given less optimistic assumptions, our implementation allows for consideration of and alternative response to situations with more severe failures;

- we demonstrate the ability to improve system performance in a large-scale failure-prone system by making the system's resource management techniques "resilience aware."

The remainder of this chapter is organized as follows. Section 4.2 discusses contemporary and proposed resilience protocols, highlights and describes the background of the four protocols compared in this chapter, as well as discusses some prior work related to scheduling in failure-prone systems. In Section 4.3, we describe the modeling methodology we use for our system simulator. Our implementation of HPC resilience protocols is detailed in Section 4.4. Section 4.6 analyzes the behavior of the simulated resilience protocols as applications scale to exascale system sizes. Section 4.7 describes the resource management techniques for our study of a typical system use-case scenario, and the results of these studies are discussed in Sections 4.8 and 4.9. We conclude with a summary of this work in Section 3.6.

4.2 Related Work

4.2.1 Overview

Our work broadly covers topics associated with modeling and evaluating the performance a system can achieve when it is oversubscribed (i.e., there are more applications than the system is capable of executing) as well as the performance of HPC resilience protocols in extreme-scale HPC systems. Many important papers have explored these two topics separately, but very few have examined the effects that failures and overhead from resilience have on the system's performance when managing system resources with multiple applications executing in the system.

4.2.2 HPC Resilience

Background

The work we consider here discusses system-level HPC resilience that allows application programmers and users of the system to be oblivious of the strategies for HPC resilience that are being employed on their behalf. We focus on providing a comparison of several checkpoint-based HPC resilience protocols. Our prior work ([43], [77]) has been among the first efforts to provide an analytical comparison of these protocols in large-scale systems. However, these efforts were less comprehensive than our work here. We have greatly extended our prior work to analyze the impacts of varying workloads and varying application execution characteristics, and to examine trade-offs among resilience strategies in the presence of resource management strategies at exascale system sizes. We acknowledge that other strategies for providing resilience to HPC systems exist and we refer the reader to the surveys of such work, [79], [50]. Our work differs significantly from these high-level surveys by providing simulated comparisons of the performance of the protocols.

All checkpointing-based protocols rely on periodically saving the system's execution state and restarting from an earlier error-free state after the occurrence of a failure [52], [53]. Because recovery from a failure requires these protocols to load a copy of the system state that is not up to date, all checkpointing protocols lose some productivity because of the need to recompute work lost between the time of the failure and the time of the last checkpoint, as well as the time involved with

storing and loading those saved system states. We provide a comparison of four HPC resilience protocols that utilize system checkpointing: checkpoint restart, multilevel checkpointing, message logging, and checkpointing combined with redundancy.

Checkpointing and Restarting

Checkpointing is by far the most commonly used resilience protocol employed by today's large-scale computing systems. The most general implementation of the checkpointing protocol operates by stopping the system's execution at regular intervals to save the state of all executing applications to a permanent storage device, typically a PFS. Such a protocol is referred to as a blocking, coordinated checkpoint [79].

Several variations and improvements on this protocol have been made since its initial inception. Attempts have been made to create non-blocking or semi-blocking checkpointing that allows the system to continue to execute while checkpoints are saved to permanent storage [54], [55]. Attempts also have been made to allow for uncoordinated checkpoints of the system, preventing the need for all processes in the system to restart when a failure occurs [56].

However, the length of time associated with checkpointing, restarting, and recomputing work lost due to a failure, and the frequency that the system needs to take checkpoints for very large-scale applications when implementing any of these checkpointing protocols, has been shown to be ineffective at managing increasing system sizes [45]. Thus, traditional checkpointing alone is not expected to be capable of providing satisfactory resilience at exascale sizes.

Multilevel Checkpointing

Because different types of failures can affect a computing system in different ways, not all failures require restarting the system from a checkpoint to the PFS [80]. Multilevel checkpointing exploits this by providing several levels of checkpointing. A multilevel checkpointing scheme may allow for checkpoints (a) to RAM that are faster but able to recover from fewer types of failures and (b) checkpoints saved to a partner node's RAM that are less frequent but able to recover from more types of failures, in addition to (c) checkpoints saved to the system's PFS. Each level

offers a trade-off between the time required to checkpoint and/or restart, and the level of failure severity from which the checkpoint can recover [47]. Checkpoint levels may also employ various encoding techniques (such as RAID or Reed-Solomon coding) to improve the resilience offered by a particular checkpoint level [47], [57]. Other attempts have been made to reduce checkpointing's dependence on the PFS [58], [60]. One challenge associated with using a multilevel checkpointing protocol is in determining the optimal number of checkpointing levels, and the optimal schedule of checkpoints at each level. Various solutions to this problem have been proposed [47], [61], [62], [81].

Message Logging

Message logging attempts to provide resilience to a system by recording messages sent among processes to create snapshots of the system's execution distributed across system memory [63]. When a failure occurs, the failed node is able to use messages stored in the memory of other system nodes to reduce the amount of rework that is performed by the system when recovering [64]. Using message logging for resilience may save computation time, because the recovering node does not need to wait for the re-computation of other nodes, but rather only for the stored results from the node's computation to be sent. Message logging also saves on the energy used by the system during recovery, because only the failed system node needs to perform re-computation, and the rest of the system can remain idle until the failed node has recovered [82]. The parallel recovery resilience protocol that we consider is an extension to message logging that allows the work that was executing on the failed node to be parallelized across neighboring nodes during recovery [45].

Redundancy

Redundancy improves a system's reliability by executing multiple copies of the same piece of code [65]. It is possible to implement redundancy in either hardware or software [50], but in either case the improved reliability of the system comes at the cost of using additional resources.

Recent attempts allow the system to utilize redundancy in less resource-intensive ways. Dynamic redundancy allows for the executing application to choose a subset of processes for re-

dundant execution [66]. Partial redundancy combines redundancy with checkpointing, and allows for applications to redundantly execute a portion of processes in the system, providing improved resilience for part of the system, using only a portion of the resources [48]. Adaptive process replication attempts to combine partial redundancy with proactive fault tolerance to make better decisions about which processes should be replicated [83].

4.2.3 HPC Resource Management

Scheduling for HPC Systems

A very large body of work exists on scheduling applications in HPC systems to improve performance and resource utilization. We examine how resource management in an exascale-sized system can be improved when considering the use of value-based (or utility) functions. A large amount of work exists discussing how the use of value functions can improve system resource management [84], [85], [86], [87], [88], [22], [89], [90].

These papers do not examine how the behavior of their resource management techniques change when executing applications at extreme scales, where the probability of system failure negatively affects the quality of allocation decisions. Our work does take this extreme-scale behavior into account in our analyses, and further provides methods for allowing some of these techniques to be "resilience aware" and to make better decisions in the unpredictable exascale environment.

Resilience-Aware Resource Management

Most approaches to provide parallel applications with resource management that is aware of uncertainty in system reliability deal either with scheduling that in some form relies on replicated copies of the executed applications (redundancy) or only with application sizes that are substantially smaller than the size of the system [91], [92], [93], [94], [95], [96]. None address the system-level scheduling problem associated with extreme-sized applications arriving to an exascale-sized system or the use of system-level resilience protocols. Furthermore, our results indicate that replication-based resilience protocols are not capable of performing well when scheduling large applications (discussed in Section 4.8).

There have been other of resilience-aware resource management in addition our prior work in [77]. For example, the work in [97] analyzes the occurrence of failures in a system and uses that information to construct a method for analyzing the reliability of any given set of nodes. An arriving application can then be scheduled to the set of nodes that are likely to provide it with the best reliability. The work in [98] proposes a resource management technique that redistributes failed applications in the system to assist in minimizing the application's execution time. Through the use of system-level strategies for resilience, our work has an advantage in that it is not resource management technique specific, and can potentially benefit any resource management technique the system designer chooses to implement.

4.3 Exascale Modeling Methodology

4.3.1 Overview

Given the impossibility of performing experiments on an exascale system, we have designed an event-based simulator used for modeling systems of arbitrary size [43], [77], [4]. The system experiences randomly generated failures that affect the simulated execution of applications in the system. Throughout the system's simulation an application's execution is affected by events associated with each application's:

- *arrival*: time at which an application arrives,
- *mapping*: process by which the resource management heuristic assigns the application to system nodes,
- *computation*: execution toward application completion,
- *failures*: the failure of a system node,
- *checkpoints*: saving a backup of the application's current computation progress,
- *restarts*: restoring the application progress saved in the last system checkpoint after a failure occurs,

- *recovery*: recomputing progress lost to a failure after the system has restarted,
- *failed checkpoints or restarts*: when failures occur during checkpoint or restart events.

Checkpoints, restarts, and recovery are all resilience-protocol specific events that determine how an application behaves with failures. Each of these events affects applications differently based on the type of resilience protocol employed by the application, the application's execution characteristics, and the characteristics of the failures. This is discussed in detail in Section 4.4. The remaining events associated with the simulator's management of application arrival, mapping, computation, and failures are all attributes of the system and behave the same regardless of the resilience protocol being used. In particular, while failure events have a large impact on the behavior of the resilience-protocol related events, failure events themselves are a function of the size of the system and the reliability of the system's nodes, and are not affected by the resilience protocol employed.

4.3.2 Modeling Extreme Scale Applications

To ensure our simulated environment has access to a diverse range of applications that will behave similarly to future exascale applications, we create a set of synthetic benchmarks that vary in their attributes of communication behavior, memory use, and size. We base most of our modeling assumptions for these extreme scale applications on the analysis of the NAS Parallel Benchmark applications [41] performed in [69]. The analysis focused primarily on the Block Tridiagonal (BT) benchmark application, but concluded with a general analysis of the entire NAS Parallel Benchmark suite. The authors determined that, with the exception of the Embarrassingly Parallel (EP) application (which experiences almost no communication), the applications in the benchmark suite all become heavily communication bound at large system sizes. The analysis performed for the BT application indicates that at extreme scales communication began to dominate as much as 80% of the application's execution time depending on which parameter set was used for the application's execution.

Similar to the BT application, our synthetic benchmarks are defined with a discrete set of time steps. The number of time steps needed to execute an application is represented by the variable T_S, with identical execution characteristics in each time step. Each benchmark spends some percentage of each time step communicating, represented by the variable T_C, with the remaining portion of each time step spent working on computation, represented by the variable T_W. We assume time steps are one minute in length with, $T_W, T_C \geq 0$, and $T_W + T_C = 1$ minute, thus allowing application execution times to be of arbitrary length (equal to a chosen number of time steps) and unaffected by the application's size. For all simulated studies performed here, applications have between 180 to 1440 time steps giving every executing application an execution time between three hours to a full day when executed without delays from failures or events related to resilience (such as time spent checkpointing). This delay-free execution time is the application's *baseline execution time* and is represented by the variable T_B.

In keeping with the results in [69], our synthetic benchmarks have two levels of communication, $T_C = 0$ (representing a type of application similar to the NAS EP application with little to no communication) and $T_C = 0.75$)representing a similar level of communication seen by the communication dominant applications from the analysis in [69]). Each of the two levels of communication to have two sizes of memory requirements represented by the variable N_m. Applications can have values of $N_m = 32$ GB of memory per node or $N_m = 64$ GB of memory per node. Defining the synthetic benchmarks in this way allows the system access to four application types with a diverse range of communication and memory characteristics. Each of the application types are defined in Table 4.1.

We assume that all of our synthetic application types exhibit weak scaling so that as the number of nodes used by the application increases with application size. The application's attributes of computation time, communication time, and memory used per node remain constant. Details about the sizes of applications in each simulated study are discussed further in Sections 4.6, 4.8, and 4.9.

For the simulated studies in Sections 4.8 and 4.9, each application arriving has an individual *deadline*. Applications that are removed from the oversubscribed system because they could not

Table 4.1: Characteristics of Application Types

	memory per node	
communication intensity	32 GB	64 GB
low communication: 0% ($T_C = 0.0$)	L_{32}	L_{64}
high communication: 75% ($T_C = 0.75$)	H_{32}	H_{64}

meet their deadlines are called *dropped applications* and the percentage of total applications that are dropped is one performance metric that we use. Deadline values for each application are selected to be the application's arrival time, T_A, plus the application's baseline execution time multiplied by a random value, $\mathcal{U}$, uniformly selected between 1.5 and 2.5, giving a deadline of

$$T_D = T_A + \mathcal{U}(1.5, 2.5) * T_B \ . \tag{4.1}$$

4.3.3 Simulated System Setup

The simulated exascale system is a homogeneous system inspired by the architecture used to develop China's Sunway TaihuLight supercomputer [75], the world's highest performing system since June of 2016 and still the world's top system as of June 2017 [99]. Each Sunway TaihuLight system node has a multicore architecture composed of four clusters of 64 computational processing elements (CPEs), with each cluster managed by a single management processing element (MPE) that also performs computational work allowing for a total of 65 cores of computation in each core cluster. The four core clusters in a node provide a total of about 3.1 TFLOPs over 260 cores. Our exascale system assumes that the number of CPEs on a node will roughly increase by a factor of four by the time an exascale machine is developed, allowing for a total of $1,028$ cores per node providing approximately 12 TFLOPs for each system node. A system composed of 120,000 of these nodes would perform at an exascale level.

The Sunway TaihuLight system has 8 GB of DDR3 RAM at each of its four core clusters, giving each node a total of 32 GB of RAM. We again assume that future systems are likely to have memory increases of about a factor of four in comparison to today's systems, giving our simulated

system a total memory capacity of 128 GB per node. In addition to an increase in capacity, we also assume that future memory is likely to utilize newer architectures, allowing for increased aggregate memory bandwidth, B_M. Today's best memories perform at a rate of up to 25 GB/s [100], and we conservatively estimate that future memories will perform at about $B_M = 40$ GB/s.

4.3.4 System Failure Model

We assume that failures can be characterized by three attributes: the time of the failure's occurrence, the node on which the failure occurs (the location of the failure), and the *severity class* of the failure. We model the uncertainty associated with each attribute using random variables and assume independence between both the individual failure occurrences as well as the attributes of each failure.

The time between system failures is modeled by a Poisson process, a common assumption in failure modeling [67]. Every failure occurs according to the previous failure time ($T_{F_{i-1}}$, with $T_{F_0} = 0$) plus a random variate generated from an exponential distribution $\mathcal{T}_i \sim Exp(\lambda_s)$ with an expected rate of $E[\mathcal{T}_i] = \frac{1}{\lambda_s}$. The parameter λ_s indicates the average failure rate of the entire system, and is defined as the number of nodes in the simulated system that are active, N_a, divided by the MTBF of the system nodes, M_n, i.e.,

$$\lambda_s = \frac{N_s}{M_n} \, . \tag{4.2}$$

The location of the failure's occurrence represents which system node failed and consequently which application is impacted by the failure. When determining which node has failed, the simulator assumes a uniform random distribution over all active nodes (nodes that are not idle) in the system, and selects one node at random as the failed node.

The severity class of failure corresponds to the type of failure that has occurred in the system. This attribute is used by multilevel checkpointing protocols to determine which level of saved checkpoint is necessary to enable recovery from a specific type of failure and is also used when determining the optimal duration of intervals between checkpoints of different levels. For the

implementation of the message logging rollback recovery protocol that we consider, the severity of the failure that occurs is used to determine if it will be possible to use the system's logged messages during recovery, or if (beyond a certain failure class) the system will need to rework from a saved checkpoint without the assistance of message logging. These assumptions and the effect of a failure's class on each resilience protocol's behavior is discussed further in Section 4.4.

We define the specific mapping of types of failures to classes of failure severities based on the analyses of types of failures present in HPC systems presented in [78] and [101]. We define the probability of experiencing a class j severity failure according to the ratio of the number of failures that occur at each failure severity class, λ_{C_j}, to the total number of failures, λ_{C_t}, measured over an extended interval of time. The resulting discrete set of ratios for each class is used to create a probability mass function from which random variates are sampled to define the severity attribute of each failure. We assume that failure types in a future exascale-sized system will occur in similar relative frequencies to those experienced by today's system, but with failures occurring more frequently.

We define three severity classes. The first class corresponds to failures that could reasonably be recovered from using a checkpoint stored in a node's local RAM (events such as software-related network and file system interrupts). The second failure severity class relates to failures that require a system node to be restarted or replaced and consequently require restarting the application from a checkpoint stored outside of the node, in this case a checkpoint stored in a partner node's RAM. Class two severity failures include hardware failures affecting only a single node or software interrupts that require the node to restart. The third class of failure severity encompasses all other types of failures that cannot be handled by failure classes one and two, including hardware failures affecting multiple nodes and hardware or software events that cause a system-wide outage. The data presented in [78] and [101] details the frequencies of 33 types of failures in the Blue Waters system that caused several hundreds of system interrupts over a timespan of hundreds of days of system execution. Using this information we calculated the probabilities of failure class severities for our three-class failure model to be approximately $\frac{\lambda_{C_1}}{\lambda_{C_t}} = 0.138$, $\frac{\lambda_{C_2}}{\lambda_{C_t}} = 0.784$, and $\frac{\lambda_{C_3}}{\lambda_{C_t}} = 0.078$.

Because the Blue Waters system is heterogeneous (with CPUs and GPUs running together) and our simulated system is homogeneous, our calculations of failure severity class distribution excludes failures related to GPUs. Additionally, our failure severity class distribution excludes failures that do not cause system interrupts.

4.3.5 Communication Model

System communication plays a large role in the behavior and performance of some of the resilience protocols we examine. We assume that future exascale systems are likely to have improved communication over today's systems, and base our communication model on the "NDR InfiniBand" network in [102]. Our communication network assumes a latency value of $L = 0.5$ μs, a bandwidth value of $B_N = 600$ GB/s, and a maximum number of simultaneous connections at each switch $N_S = 12$. Further details about the role of communication is discussed for each resilience protocol in Section 4.4.

4.4 Resilience Protocol Modeling

4.4.1 Overview

Four styles of HPC resilience protocols have been implemented in our system simulator. A traditional checkpoint restart based protocol, *checkpoint restart*, as well as three protocols proposed for next-generation computing systems: a multilevel checkpointing approach described in [47], *multilevel checkpoint*; an implementation of message logging based on the work presented in [45], *parallel recovery*; and a protocol combining traditional checkpointing with partial or full redundancy of the executing application from [48], *redundancy*. The following subsections present details of how each resilience protocol was modeled, with all relevant parameters summarized in Table 4.2. Optimal checkpoint interval determination for each resilience protocol is discussed in Section 4.5.

Table 4.2: Resilience Protocol Parameters

parameter	use in modeling
T_S	number of time steps
T_B	application baseline execution time
T_C	portion of each time step spent on communication
T_W	portion of each time step spent on computation work
N_m	memory used by the application
N_a	number of system nodes used by the applications
L	network latency
B_N	communication bandwidth
B_M	memory bandwidth
N_S	number of network switch connections in router
λ_a	application failure rate
M_n	system component MTBF
τ	optimal checkpoint period
$T_{C_{PFS}}$	time required to checkpoint to a PFS
$T_{C_{L1}}$	time required for a level one checkpoint
$T_{C_{L2}}$	time required for a level two checkpoint
μ	message logging slowdown
r	degree of redundancy

4.4.2 Checkpoint Restart

Our implementation of the checkpoint restart resilience protocol performs periodic blocking uncoordinated checkpointing, with its checkpoints saved to a PFS. This checkpointing strategy allows simultaneously executing applications to be checkpointed or restarted independently from one another. This protocol also allows for optimal checkpoint intervals to be defined for individual applications rather than for the system as a whole, which benefits smaller applications that would otherwise experience suboptimal performance if checkpointed at exascale failure frequencies.

The time that the checkpoint restart protocol requires to read and write its checkpoint data to a PFS, $T_{C_{PFS}}$, is dependent on application size, memory use, and system parameters for communication. We assume that at any given time the maximum number of nodes from which the PFS

is capable of receiving data is equal to the maximum number of switch connections that enter the memory system (N_S). Application checkpoints to the PFS are typically slowed down by network congestion due to the large number of nodes needing access the PFS during a checkpoint. Because the latency associated with checkpoints to the PFS is negligible in comparison to the total checkpoint time, we do not consider latency in our equation for checkpoints to the PFS. The time for checkpoints to a PFS is defined as

$$T_{C_{PFS}} = \frac{N_m}{B_N} * \frac{N_a}{N_S} \ . \tag{4.3}$$

Parameters for the applications and environment in this study impose a checkpoint and restart time of between 8-17 minutes for an exascale-sized application depending on the application's type.

The optimal checkpoint period is dependent on the application's checkpoint time and the observed failure rate. Each application's failure rate is given by $\lambda_a = \frac{N_a}{M_n}$.

4.4.3 Multilevel Checkpointing

The multilevel checkpointing approach from [47] that we implement in our simulator is a three-level checkpointing model. Each checkpointing level offers a trade-off between the time required to save or restore a checkpoint and the severity class of the failure from which it can recover.

The first checkpoint level writes to the node's local RAM, with the time required for taking a level one checkpoint being simply the amount of memory per node required by the application divided by the node's memory transfer rate

$$T_{C_{L1}} = \frac{N_m}{B_M} \ . \tag{4.4}$$

The second checkpoint level stores its checkpoints to RAM in a partner node. Application nodes are assumed to be contiguous allowing for minimum latency between checkpoints sent between nodes. The nodes used by the application are partitioned into two groups, with each node in one group having a corresponding partner node in the second group. During a level two check-

point, the nodes in one group first perform a level one checkpoint (saving the checkpoint data to their own local node) and then send that checkpoint data to their respective partner nodes in the second group. The two groups then switch roles to allow for checkpoint data from the second group to be saved. The time to perform a level two checkpoint is equal to the time required to perform a level one checkpoint, plus the time required to send the data to the partner node, plus the time required to write the data to memory in the partner node. This time is then multiplied by two to account for the time required for both groups of partner nodes to store checkpoint data making the total time for a level two checkpoint equal to

$$T_{C_{L2}} = 2(T_{C_{L1}} + L + \frac{N_M}{B_M}) \ . \tag{4.5}$$

The third level checkpoint is written to a PFS, and the time required is the same as presented in Eqn. 4.3. During a level three checkpoint, a level two checkpoint (and consequently also a level one checkpoint) is also saved by the system. These lower-level checkpoints can be performed in parallel with the level three checkpoint. Additionally, we also assume checkpoint and restart times are symmetric. Failure severity classes are defined according to the outline in Section 4.3.4.

4.4.4 Parallel Recovery

The parallel recovery protocol in [45] is an improvement to the message logging resilience protocol, and we base most assumptions about the behavior of a message logging protocol on the behavior of parallel recovery. Parallel recovery allows for faster recovery from a system failure by allowing the failed node's work to be temporarily parallelized across several nodes after being restarted, thereby reducing the time needed by the system to recompute the work lost to a failure. As with all message logging protocols, parallel recovery benefits the system by allowing most of the system to remain idle while only the failed node is recovered. This decreases both the system power needed during recovery as well as the chance that a failure will interrupt the recovering system. However, unlike other message logging protocols, parallel recovery improves checkpointing and restart time by utilizing the in-memory checkpointing protocol outlined in [68].

The in-memory checkpointing protocol behaves similarly to the level two checkpoint to a partner node described in Section 4.4.3. We therefore used Eqn. 4.5 to represent the time required for an in-memory checkpoint or restart.

Utilization of the parallel recovery protocol imposes additional overhead involved with message logging, because the system must spend time storing every message that is sent. The amount of overhead an application experiences from message logging, μ, is therefore directly proportional to the amount of communication required by the application. Here we assume this value for our synthetic applications is equal to $\mu = 1 + \frac{T_C}{10}$ which gives a range of values for message logging slowdown that are very close to those listed in [45]. The increase in execution time from message logging increases the application baseline execution time when using parallel recovery to

$$T_B^* = \mu T_S(T_W + T_C) = \mu T_S \ . \tag{4.6}$$

The parallelization allowed for a recovering system node in [45] is assumed to be limited to parallelizing across a maximum of eight other system nodes for an estimated $7.26\times$ reduction in recovery time. Through experimentation, we found that for our simulated exascale system the improved performance achievable though node parallelization during recovery starts to produce diminishing returns if parallelized across more than 32 nodes. Based on [45], we allow a recovering application parallelized across 32 nodes to produce an approximate $29.04\times$ estimated reduction in recovery time. The remainder of the parallel recovery specific parameter values are from [45].

The parallel recovery protocol from [45] assumes that all system failures can use in-memory checkpointing, restarting, and stored messages required for parallel recovery. However, because some types of failures can cause multiple simultaneous node failures (i.e., failures of the type belonging to failure severity class three defined in Section 4.3.4), this assumption is likely to be optimistic if parallel recovery were employed in a real exascale system. In contrast, the implementation of parallel recovery employed by our simulated system allows for applications to utilize the recovery advantages offered by parallel recovery and in-memory checkpointing for failures of severity class one and class two, but requires a restart from the PFS for class three severity failures,

similar to the behavior of multilevel checkpointing. Aside from the impact that this has on the resilience protocol's performance, this also means that the parallel recovery protocol will operate with two checkpoint levels (one level checkpointing to a partner node's RAM and a second level checkpointing to the PFS).

4.4.5 Partial Redundancy

The partial redundancy protocol in [48] combines the traditional checkpointing protocol with varying degrees of hardware redundancy. "Partial" redundancy is achieved by allowing only a fraction of the total system nodes required by the executing application to have redundant hardware during its execution. For example, a degree of redundancy of $r = 1.5$ dictates that each *virtual process* of an executing application requiring a single node will have at least one physical node performing the application's required computation but half of the virtual processes will have a second physical node performing the same computation. Checkpoints are still taken by the system at regular intervals. When failures occur on nodes in the system, the system only requires a restart from the last checkpoint if failures occur on all (possibly redundant) physical nodes associated with the application's virtual processes before the next system checkpoint.

Apart from the application baseline execution time, all parameters associated with the partial redundancy resilience protocol remain the same as for the checkpoint restart protocol. To account for the increase in application execution time for duplicated communication used by redundancy, based on [48], the communication term in the equation for baseline execution time is scaled by the degree of system redundancy, r,

$$T_B^* = T_S(T_W + rT_C) \ . \tag{4.7}$$

4.5 Resilience Protocol Execution Time Prediction

4.5.1 Overview

This work utilizes several execution time prediction equations that were developed for use in resilience-aware resource management. Sections 4.5.2, 4.5.3, 4.5.4, and 4.5.5 discuss equations we

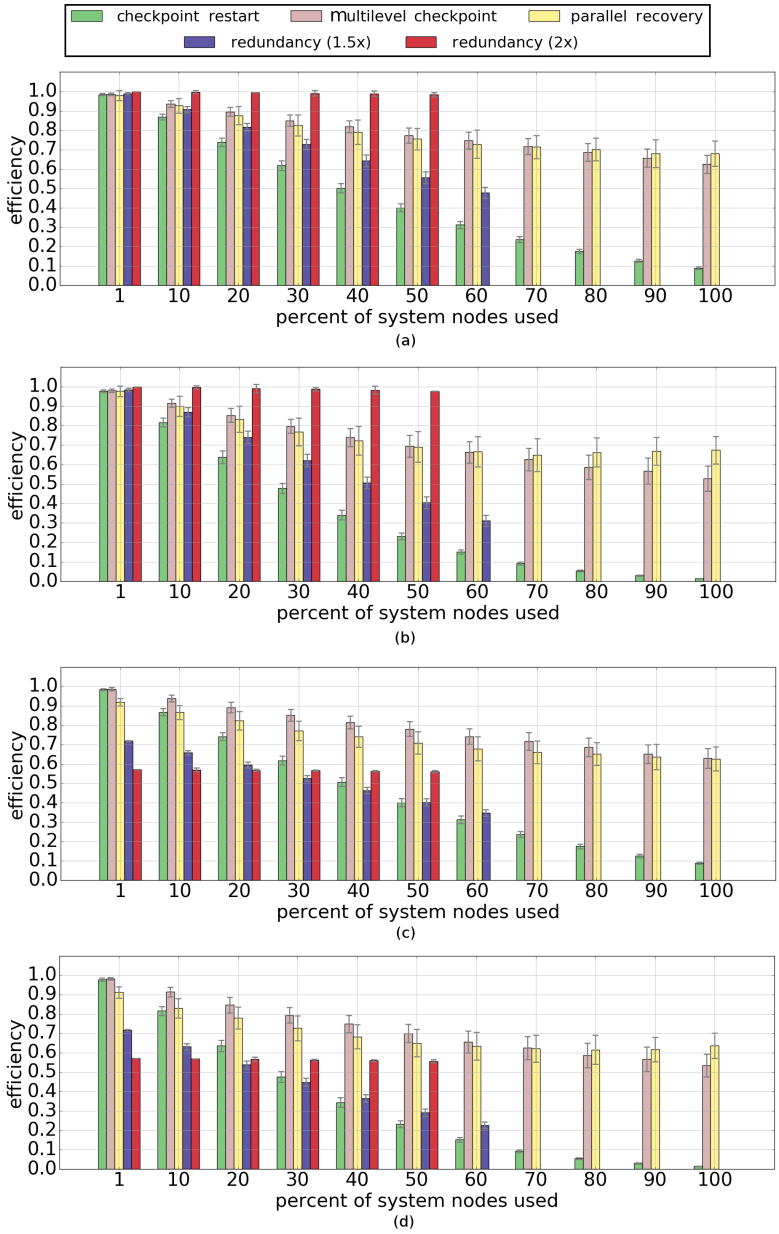

Figure 4.1: Resilience protocol efficiency at increasing percentages of total system use, **(a)** by the low memory use and low communication application defined in Table 4.1 as L_{32}, **(b)** by the high memory low communication application (L_{64}), **(c)** by the low memory high communication application (H_{32}), **(d)** by the high memory high communication application (H_{64}). Efficiency is defined to be the ratio of an application's execution time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures and checkpointing). Processors in the system experience a 2.5 year MTBF. Each bar in the figure represents the average of 200 trials. Standard deviations are shown for each bar.

have derived and used to both predict the expected execution time of applications executing in the presence of failures and to determine the optimal checkpointing intervals that should be selected for any applications that are utilizing the resilience protocols in Section 4.4 of this chapter. These equations are also used for some of the work in Chapter 5. In Section 4.5.6, we discuss how the execution time prediction models are used for determining optimal checkpointing intervals.

The equations we derive are inspired by work from [73] and [45], but have been greatly extended to accommodate more detailed modeling of certain aspects of application execution as well as enabling the modeling of multiple levels for the Multilevel Checkpointing and Parallel Recovery protocols. All equations are organized to estimate the expected values of each type of resilience, failure, and execution-related events during application execution as described in Section 4.3.1. The sum of these values is the total expected execution time of the application's execution. The application's expected execution time for all occurrences of each event type is generally calculated by multiplying an estimator for the expected number of occurrences of the event by the expected execution time of the event.

4.5.2 Execution Time Model with Checkpoint/Restart

The expected application execution time when using Checkpoint Restart, T_{CR}, is equal to the sum of the application's time for all:

- computation of the application without overhead from resilience or failures (baseline execution time), T_B;
- successful checkpoints, T_δ;
- failed checkpoints, $T_{\delta'}$;
- successful restarts, T_R;
- failed restarts, $T_{R'}$;
- re-computation of work lost to a failure occurring during a computation interval, T_{W_τ};

- re-computation of work lost to a failure occurring during a checkpoint, T_{W_δ}.

This gives a total expected execution time of

$$T_{CR} = T_B + T_\delta + T_{\delta'} + T_R + T_{R'} + T_{W_\tau} + T_{W_\delta} \ . \tag{4.8}$$

For the Checkpoint Restart protocol, an estimate of the application's baseline execution time, T_B, is assumed to be known by the system designer. The time required for a single checkpoint, δ, is also known (from profiling) so the total time for successful checkpoints, T_δ, can be precisely calculated as the total number of successful checkpoints multiplied by δ, with the number of successful checkpoints equal to the application's baseline execution time divided by the computation interval, τ, giving

$$T_\delta = \left(\frac{T_B}{\tau} - 1\right)\delta \ . \tag{4.9}$$

All remaining terms in Eqn. 4.8 are dependent on the number of failures that occur during the application's execution and must therefore be estimated. The estimated total number of failures throughout the application's execution is the product of the failure rate, λ, and the application's total execution time, giving an estimated total number of failures equal to (λT_{CR}). This estimator is used for all terms in Eqn. 4.8 that are dependent on failures.

Each term's expected value is estimated as the expected number of occurrences of the event multiplied by the expected time of the event if a failure occurs. For a chosen probability density function (PDF) used to model the probability of a failure occurring, we calculate the expected execution time for any event in which a failure has occurred as the expected value of the PDF with its domain truncated to the duration of that event and normalized to the probability of a failure occurring during the event's duration (a truncated distribution). The use of this approach in our equation-based model is flexible and allows for any integrable PDF to be used to represent the distribution of system failures.

As stated in Section 3.3.2, we are assuming that failures follow an exponential distribution, making the probability of a failure occurring during any given interval of time t for a failure rate λ

equal to

$$P(t, \lambda) = 1 - e^{-\lambda t} \ . \tag{4.10}$$

As opposed to the expected value of the general PDF, which is calculated over the entire domain, $[0, \infty)$, the truncated domain is calculated over $[0, t]$ and makes the expected value of the truncated PDF for the event when using an exponential distribution equal to

$$E(t, \lambda) = \frac{\frac{1}{\lambda} - e^{-\lambda t}(\frac{1}{\lambda} + t)}{P(t, \lambda)} \ . \tag{4.11}$$

The estimator for the expected number of failures that occur during a checkpoint, α', can be shown to be a function of the number of successful checkpoints and the probability of a failure occurring during a checkpoint to give

$$\alpha' = \frac{P(\delta, \lambda)(\frac{T_B}{\tau} - 1)}{1 - P(\delta, \lambda)} \ . \tag{4.12}$$

Using Eqns. 4.11 and 4.12 we can calculate the expected time that the application wastes due to failed checkpoints as

$$T_{\delta'} = \alpha' E(\delta, \lambda) \ . \tag{4.13}$$

In addition to the time incurred for the failed checkpoint itself, each failed checkpoint also incurs overhead associated with the lost computation interval when the checkpoint fails. This value, T_{W_δ}, is calculated as

$$T_{W_\delta} = \alpha' \tau \ . \tag{4.14}$$

Note that α' can also be divided by the value of the total number of failures during execution (λT_{CR}) to give the percentage of total failures that occur during a checkpoint, denoted as α.

For the Checkpoint Restart protocol, the expected percentage of failures that occur during a restart, ζ, can be shown to be simply equal to the probability of a failure occurring during a restart

$$\zeta = P(R, \lambda) \ . \tag{4.15}$$

This makes the total expected time that the application spends for successful restarts equal to the total number of failures that do not occur during a restart multiplied by the time for a successful restart

$$T_R = (\lambda T_{CR})(1-\zeta)R \; . \tag{4.16}$$

Similarly, the total time that the application spends for failed restarts is equal to

$$T_{R'} = (\lambda T_{CR})(\zeta)E(R,\lambda) \; . \tag{4.17}$$

The calculation of T_{W_τ} in Eqn. 4.8 uses the values of α and ζ to estimate the total number of failures that occur during computation and estimates the total expected time that the application loses due to failures during computation as

$$T_{W_\tau} = (\lambda T_{CR})(1-\zeta-\alpha)E(\tau,\lambda) \; . \tag{4.18}$$

Once all values are defined, the expected execution time of the application can then be calculated by solving Eqn. 4.8 for T_{CR}.

4.5.3 Execution Time Model with Multilevel Checkpoint

The work in Chapters 4 and 5 utilize the following equations for making execution time predictions for the multilevel checkpointing protocol, however an improved version of these equations is discussed in Chapter 6 and the reader is encouraged to utilize the improved equations when modeling or optimizing the multilevel checkpointing protocol. The application execution time model when using Multilevel Checkpoint discussed here is similar to that of Eqn. 4.8, however it is defined hierarchically to account for each level of the Multilevel Checkpoint protocol. For this protocol, failure rates of different levels must be differentiated. For a failure severity, $i = 1, ..., L$, the corresponding failure rate, λ_i, is defined as the product of the system failure rate, λ, and the probability of a failure at that severity, S_i, making the failure rate $\lambda_i = S_i\lambda$. Each of the other vari-

ables defined for the Checkpoint Restart protocol (i.e., T_δ, $T_{\delta'}$, ζ, R, etc.) are now L-dimensional vectors.

The Multilevel Checkpoint protocol from [47] that we are modeling defines each higher-level checkpoint to occur after some number of occurrences of the previous level of checkpoint (e.g., an L_2 checkpoint to a partner node's RAM occurs after some number of instances of L_1 checkpoints to the node's local RAM). These values defining the number of L_{i-1} checkpoints that must occur before each L_i checkpoint is taken are the set of $L-1$ integer decision variables $N_1, ..., N_{L-1}$ used for optimizing the equation. The last decision variable is the computation interval, a real number that we define as τ_0. This set of decision variables hierarchically defines the amount of computational progress made by the application once each level i checkpoint has been completed. The variable N_L, while not a decision variable, represents the number of level L checkpoints that will occur during the execution of the entire application and is defined based on the amount of computational progress that is made for each level L checkpoint interval.

The amount of total time spent between each level i checkpoint is referred to as the level $i+1$ computation interval. Each higher level computation interval, τ_{i+1}, is calculated as

$$\tau_{i+1} = T_{B_i} + T_{\delta_i} + T_{\delta'_i} + T_{R_i} + T_{R'_i} + T_{W_{\tau_i}} + T_{W_{\delta_i}} \tag{4.19}$$

with the application's total expected execution time when using Multilevel Checkpoint $T_{ML} = \tau_{L+1}$.

The time spent checkpointing at each level is now defined as

$$T_{\delta_i} = N_i \delta_i \ . \tag{4.20}$$

The estimator for the expected number of failures that occur during each level i checkpoint, α'_i, is now defined as

$$\alpha'_i = \frac{P(\delta_i, \sum_{j=1}^{i} \lambda_j) N_i}{1 - P(\delta_i, \sum_{j=1}^{i} \lambda_j)} \ , \tag{4.21}$$

making the expected time that is wasted due to failed checkpoints

$$T_{\delta_i'} = \alpha_i' E\left(\delta_i, \sum_{j=1}^{i} \lambda_j\right) . \tag{4.22}$$

The overhead associated with the lost computation interval caused by the failed checkpoint is calculated as

$$T_{W_{\delta_i}} = \alpha_i' \sum_{k=1}^{i} S_k \tau_k . \tag{4.23}$$

The expected percentage of failures that occur during a restart of level i, ζ_i, is now

$$\zeta_i = S_i P\left(R_i, \sum_{j=1}^{i} \lambda_j\right) . \tag{4.24}$$

Making the total expected time that the application spends for successful restarts equal to

$$T_{R_i} = (\lambda T_{ML})(S_i - \zeta_i) R_i . \tag{4.25}$$

The total time that the application spends for failed restarts is now

$$T_{R_i'} = (\lambda T_{ML})(\zeta_i) E\left(R_i, \sum_{j=1}^{i} \lambda_j\right) . \tag{4.26}$$

The calculation of T_{W_τ} becomes

$$T_{W_{\tau_i}} = (\lambda T_{ML})(S_i - \zeta_i - \alpha_i) E(\tau_i, S_i \lambda) . \tag{4.27}$$

4.5.4 Execution Time Model with Parallel Recovery

The parallel recovery protocol behaves differently for failures of different severity classes. We assume that above some severity class limit the Parallel Recovery protocol is no longer capable of utilizing message logging or recovery in parallel and that failures from successively higher severity classes behave according to the Multilevel Checkpoint equations from Section 4.5.3.

Parallel Recovery must account for time lost during the same computation events as Multilevel Checkpoint, however failures from lower severity classes that are able to utilize message logging and recovery in parallel no longer need to account for computation time lost when a failure occurs but instead account for the time needed for a single machine to *recover* the lost computation performed by the node that suffered the failure as well as the additional time lost if those recovery events fail. The time for recovering computation from a failure occurring during a computation interval is denoted as $T_{C_{\tau_i}}$, the time for recovering computation from a failure occurring during a checkpoint is denoted as $T_{C_{\delta_i}}$, the additional time lost to recoveries that fail while attempting to recover from a failure during computation is denoted as $T_{C'_{\tau_i}}$, and time lost to recoveries that fail when attempting to recover from a failure during a checkpoint is denoted as $T_{C'_{\delta_i}}$. For a lower severity failure the expected time to compute interval i is equal to

$$\tau_{i+1} = T_{B_i} + T_{\delta_i} + T_{\delta'_i} + T_{R_i} + T_{R'_i} + T_{C_{\tau_i}} + T_{C_{\delta_i}} + T_{C'_{\tau_i}} + T_{C'_{\delta_i}} \ . \tag{4.28}$$

Higher level intervals are defined by Eqn. 4.19.

Parallel Recovery's baseline execution time, T_{B_i}, is defined accorded Eqn. 4.6. The terms in Eqn. 4.28 representing the time for checkpoints, restarts, failed checkpoints, and failed restarts remain the same as their definitions from Section 4.5.3.

The expected percentage of failures that occur during a recovery for a level i interval is defined by the parameter β_i. Given ζ_i defined by Eqn. 4.24 and a parameter, σ, representing the system's expected recovery speedup when recovery is parallelized, the value of β_i is calculated as

$$\beta_i = (S_i - \zeta_i) P\left(\frac{\tau_i}{\sigma}, \sum_{j=1}^{i} \lambda_i\right) \ . \tag{4.29}$$

Given α_i defined according to Eqn. 4.21, the value of β_i is used when calculating the amount of time spent recovering from a failure during computation to give

$$T_{C_{\tau_i}} = (\lambda T_{PR})(S_i - \zeta_i - \alpha_i - \beta_i)\frac{E\left(\tau_i, \sum_{j=1}^{i} \lambda_i\right)}{\sigma} \ . \tag{4.30}$$

The corresponding time spent for failures that occur during these recoveries is equal to

$$T_{C'_{\tau_i}} = (\lambda T_{PR})(\beta_i)E\left(\frac{E\left(\tau_i, \sum_{j=1}^{i} \lambda_i\right)}{\sigma}, \sum_{j=1}^{i} \lambda_i\right) \ . \tag{4.31}$$

The amount of time spent recovering from a failure that occurs during checkpointing is calculated as

$$T_{C_{\delta_i}} = (\lambda T_{PR})(\alpha_i)\frac{\tau_i}{\sigma} \ . \tag{4.32}$$

The corresponding time spent for failures that occur during these recoveries is equal to

$$T_{C'_{\delta_i}} = (\lambda T_{PR})(\alpha_i\beta_i)E\left(\frac{\tau_i}{\sigma}, \sum_{j=1}^{i} \lambda_i\right) \ . \tag{4.33}$$

4.5.5 Execution Time Model with Redundancy

When employing Redundancy, not every failure that occurs requires the system to restart. Consequently, we simplify some of the Checkpoint Restart assumptions from 4.5.2 for the Redundancy protocol. Specifically, the execution time when using Redundancy, T_{Red}, does not consider the effect of failures during restarts (as they far less likely to occur), and also consolidates the cost of rework for failed computation intervals and failed checkpoints into a single term. The execution time of an application utilizing Redundancy is equal to the sum of the application's time for:

- computation of the application without overhead from resilience or failures (baseline execution time), T_B;
- successful checkpoints, T_δ;
- successful restarts, T_R;

- work lost to a failure, including failures during computation and failures during checkpointing, $T_{W_{Red}}$.

This gives a total expected execution time of

$$T_{Red} = T_B + T_\delta + T_R + T_{W_{Red}} \ . \tag{4.34}$$

For Redundancy, T_B is now affected by the increased communication required for all redundant copies of the executing application to execute synchronously. To account for this overhead, we scale the baseline execution time of the application by a factor, μ_{Red}. Similar to [48], we defined μ_{Red} based on the fraction of time the application spends on communication without any redundancy (T_C from Section 4.3.2) and the amount of redundancy permitted in the system, r, we define this scaling factor as

$$\mu_{Red} = 1 - T_C + rT_C \ . \tag{4.35}$$

The application baseline execution time when using Redundancy is then defined as the product of μ_{Red} and the number of time steps (T_S from Section 4.3.2) required for the application's execution

$$T_B = \mu_{Red} T_S \ . \tag{4.36}$$

The total amount of time that the application spends checkpointing, T_δ, is then defined as the same as in Eqn. 4.9.

As with traditional Checkpoint Restart, the amount of time spent restarting is a function of the total number of failures that occur throughout the application's execution (λT_{Red}). However, unlike traditional Checkpoint Restart only a fraction of the failures experienced by the system require the application to restart. To account for this in our model we define the fraction of failures that require a restart, P_i, for an application requiring N_B system nodes without redundancy and a system utilizing a redundancy of r as

$$P_i = 1 - \prod_{j=1}^{i} \left(r - 1 - \frac{j-1}{N_B} \right), \tag{4.37}$$

with $P_0 \equiv 0$. Eqn. 4.37 assumes that the length of a computation interval and its corresponding checkpoint is greater than the application's MTBF (i.e., $\tau + \delta > \frac{1}{\lambda}$), indicating that there will be multiple failures during this interval, but only some of them will require restarts. This assumption allows Redundancy-based protocols to have a larger computation interval between checkpoints. Eqn. 4.37 also assumes that $1 \leq r \leq 2$. Utilizing Eqn. 4.37 gives a value for T_R of

$$T_R = (\lambda T_{Red})(P_{\lfloor \lambda(\tau+\delta) \rfloor - 1})R\,. \tag{4.38}$$

Because an application's susceptibility to a failure increases for longer computation intervals, when calculating the expected amount of rework caused by a failure, we must sum the probability of a failure that causes a restart occurring as failures occur through the $(\tau + \delta)$ interval. This is accomplished by utilizing Eqn. 4.37 to give

$$T_{W_{Red}} = (\lambda T_{Red}) \left[\sum_{i=1}^{\lfloor \lambda(\tau+\delta) \rfloor - 1} \left(\frac{(\tau+\delta)i}{\lfloor \lambda(\tau+\delta) \rfloor} \right) (P_i - P_{i-1}) \right]. \tag{4.39}$$

4.5.6 Checkpoint Interval Optimization

To optimize Eqns. 4.8 and 4.34 for maximum performance efficiency we sweep through the set of values over the interval $(0, T_B)$ for the decision variable τ and evaluate each resilience protocol's respective equation to find the interval value that results in the minimum execution time. Similarly, selecting decision variables that maximize performance efficiency for Eqns. 4.19 and 4.28 for Multilevel Checkpoint and Parallel Recovery is accomplished by evaluating each respective protocol's execution time at every point in a bounded region of the solution space and determining which decision variable values provide the shortest execution time. This sweep of decision variable values is bounded for τ_0 by the interval $(0, T_B)$, and bounded for $N_1, ..., N_{L-1}$ such that the product of these values and τ_0 is greater than zero and less than the application's baseline execution time, i.e.,

$0 < \tau_0 \left(\prod_{i=1}^{L} N_i \right) < T_B$. For both Checkpoint Restart and Multilevel Checkpoint, we can guarantee a global optimum is found when bounding the solution space in this way because decision variable values outside of this region produce infinitely large execution times when the system's MTBF is less than the application's baseline execution time, as is the case here.

4.6 Resilience Protocol Performance with Application Scaling

We utilized our simulation environment to conduct several sets of experiments examining the performance of each resilience protocol. We evaluated the performance of each of the four application types defined in Table 4.1, with each application type being scaled in size from one percent of the exascale system (about 1.2 million CPU cores, similar in size to some of today's largest applications) through to an exascale-sized application requiring 123 million CPU cores. For these experiments, the baseline execution time for each application is defined as $T_B = 1440$ minutes (one full day).

As systems trend towards manycore architectures, with hundreds or thousands of CPU cores on a single socket, component failure rates are likely to increase [103], [104]. Our simulated exascale system assumes an approximate $4\times$ increase in the number of CPU cores per processor over the Sunway TaihuLight system, which is likely to decrease processor reliability and increase the likelihood of failures in the system. Most of the works we consider assume a node MTBF of ten years for current HPC systems. We assume component failure rates will increase linearly with the increased size of system nodes. For our experiments, we assume an MTBF of $M_n = 2.5$ years.

Figure 4.1 highlights the results from this set of experiments analyzing execution efficiency for varying application sizes. Efficiency is defined to be the ratio of an application's baseline execution time over the application's execution time with slowdowns from failures or resilience protocol overhead delay, e.g., due to checkpointing. Each bar in the figure represents the average of 200 trials simulated with the uncertainty of randomly occurring failures. The standard deviation of these trials are shown around each bar.

The data in Figures 4.1a-d depicts the efficiency of each of the applications in Table 4.1 as the size of each application increases. This is indicated on the x-axis of each figure as an increase in the percentage of system nodes occupied.

Analyzing trends when comparing the two high-communication applications (Figures 4.1c and 4.1d) to the two low-communication applications (Figures 4.1a and 4.1b), it can be seen that both the parallel recovery protocol and the two redundancy protocols suffer a larger decrease in efficiency for all application sizes when the application has a higher amount of communication than the checkpoint restart or multilevel checkpoint protocols suffer. This decrease is due to the parallel recovery and redundancy protocol's higher reliance on communication.

An increase in application memory use increases the time required to perform checkpoints. Because the checkpoint restart, multilevel checkpointing, and $1.5\times$ redundancy protocols rely more heavily on checkpoints (particularly to the PFS) than the parallel recovery and $2\times$ redundancy protocols, results observed when comparing the high-memory applications (Figures 4.1b and 4.1d) to the low-memory applications (Figures 4.1a and 4.1c) indicate that the checkpoint restart, multilevel checkpointing, and $1.5\times$ redundancy protocols are more negatively impacted by an application's memory use.

One trend seen throughout Figure 4.1 is a trade-off among which resilience protocol provides the best performance for any size of the given application type. For low communication applications (Figure 4.1a and 4.1b) occupying at most half of the system, the $2\times$ redundancy protocol allows for the best application performance. However, because of their costly need for additional system resources, the redundancy protocols provide zero efficiency when the application is scaled above certain applications sizes because there are not enough nodes available in the system to employ either redundancy protocol. The multilevel checkpointing and parallel recovery protocols also change between which protocol is optimal as application sizes increase in all but the low-memory high-communication application (Figure 4.1c). Results in the figure indicate that the multilevel checkpointing protocol tends to dominate over the parallel recovery protocol for application occupying between 1% to 50% of the system (depending on the application type), but the parallel

recovery protocol tends to dominate multilevel checkpointing for larger application sizes. There are no situations in which checkpoint restart or $1.5\times$ redundancy resilience protocols are optimal. These results motivate the need for an adaptive strategy to achieve low-overhead resilience.

4.7 System Resource Management

4.7.1 Overview

We explore the behavior of several techniques for resource management operating in a system with failures and considering overhead from employing resilience techniques. Each resource management technique takes as input the set of unmapped applications and idle system nodes (nodes that are not currently executing an application) at a mapping event and outputs a mapping of applications to system nodes. System mapping events occur immediately after an application arrives to the system as well as immediately after an application finishes its execution. If not enough idle system nodes are available during the mapping event to accommodate all unmapped applications, then the remaining applications stay in the set of unmapped applications until they are either scheduled during a future mapping event or reach their deadlines and are removed from the system.

4.7.2 FCFS Technique

First come first served (FCFS) is the most commonly employed resource management technique in HPC systems and it is therefore both important to assess its behavior in an exascale system and also an important point of comparison for other resource management techniques. This technique operates by scheduling applications from the set of unmapped applications in the order that they arrive to the system until there are not enough nodes left for the most recently arrived application. Applications not assigned to nodes are scheduled in a future mapping event.

4.7.3 Random Technique

The *random* resource scheduling technique randomly selects an application from the set of unmapped applications and assigns it to execute on any available set of nodes able to accommodate

the application's size. If not enough nodes are available, then the application is returned to the set of unmapped applications for consideration in the next mapping event. This process is repeated until the set of mappable applications is empty.

4.7.4 Slack-Based Technique

An application's slack is calculated as the application's deadline minus the sum of its baseline execution time and its time of arrival to the system. The slack-based resource management technique allows the system a means to prioritize applications based on the application's execution time and deadline. The set of unmapped applications is sorted based on each application's slack value. A negative slack value indicates that an application will not be able to complete execution before its deadline, and it is "dropped" from the system. The technique schedules applications to nodes in the system ordered based on increasing slack. Applications that cannot begin execution immediately are returned to the set of unmapped applications. The slack-based scheduler continues evaluating applications until all applications are either executing in the system or have been returned to the set of unmapped applications to be considered in future mapping events.

4.7.5 Value-Based Techniques

Typically, not all applications that arrive to a system have the same importance. Furthermore, because the results of executing applications are typically more valuable if they are processed quickly, the benefit that an HPC system can provide to a set of users can typically be improved if these aspects are kept in consideration when making resource management decisions. To more accurately analyze system performance when considering the time-varying nature and importance of applications, we use *value functions*. A value function is a monotonically decreasing function defined for each application that arrives to the system and is commonly used to quantify an application's importance based on information associated with the application's expected execution time and resource requirements [84].

For the work performed here, each application's time-varying value is a function of the application's execution time, size, and the time since the application's arrival. Each application has a

starting value, V_S, that is proportional to both an increase in the application's size and execution time:

$$V_S = (100\frac{N_a}{N_S}) * (\frac{T_S}{360}) \ . \tag{4.40}$$

An application that arrives to the system is awarded its starting value if it completes execution by a specified "soft" deadline, T_{D_S}, defined by its arrival time and execution time

$$T_{D_S} = T_A + 1.5T_B \ . \tag{4.41}$$

Applications unable to complete execution by their soft deadlines experience a linear depreciation in value defined by the current time of the simulation, C_T, the application's hard deadline defined in Eqn. 4.1, its soft deadline defined in Eqn. 4.41, and its starting value from Eqn. 4.40. The value earned by the application if it completes execution after its soft deadline, V_F, is equal to

$$V_F = V_S + -\frac{\frac{3}{5}V_S}{T_D - T_{D_S}}(C_T - T_{D_S}) \ . \tag{4.42}$$

Defining the value function in this way implies that an application's final value at the time it reaches its hard deadline will have depreciated down to a fraction of its starting value, which is consistent with other work utilizing value-based heuristics. Applications that reach their hard deadlines without completing execution are removed from the system and earn the system zero value.

Our resource management techniques consider four scheduling heuristics based on [89] to analyze the value that will be earned by the application for successfully completing execution. We consider heuristics that prioritize scheduling applications by:

- *Max Value*: the total value that will be earned by the application upon its completion
- *Max VPT*: the value able to be earned by each application per unit time
- *Max VPN*: the value able to be earned by each application per node

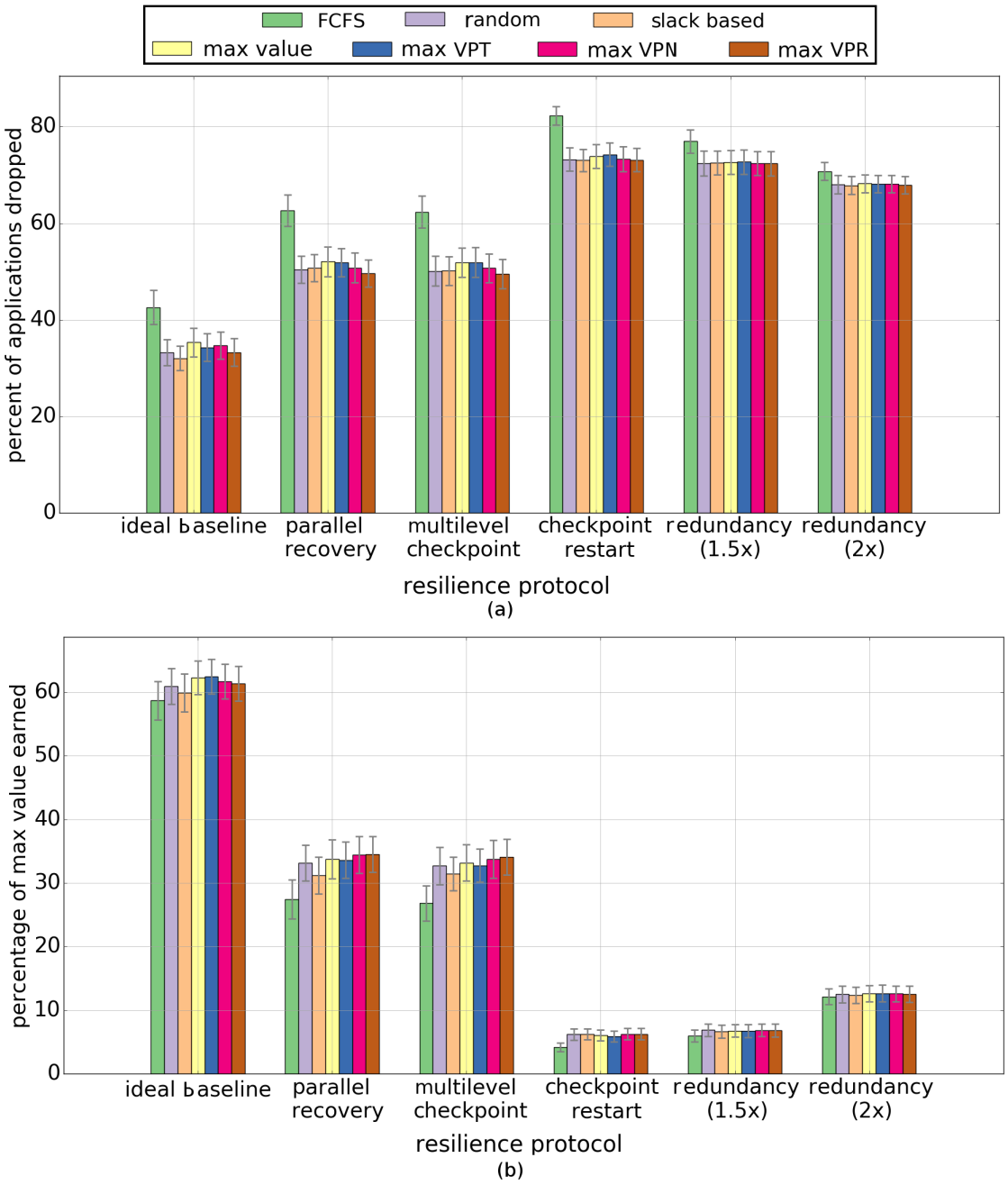

Figure 4.2: Performance of the system by: **(a)** percentage of applications dropped from the system, and **(b)** percentage of the maximum value earned by the system, for each resilience protocol and resource management technique combination. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

- *Max VPR*: the value able to be earned by each application per resource required by the application (a product of the application's execution time and number of nodes)

4.8 Resilience Protocol Effects on Resource Management

In practice, it is unlikely that exascale systems will always be used for executing a single exascale-sized application. Instead, in many cases such systems will spend the majority of their time executing a larger number of smaller applications. We explore the behavior of an exascale-sized system under this more typical use-case scenario as the system is utilized over a period of several days to service a large number of petascale sized applications with a wide variety of execution characteristics. We show the impact that failures have on this environment and the level of benefit that is provided to the system by each resilience protocol.

We assume the exascale environment is oversubscribed, meaning there are always more applications submitted to the system needing to be executed than the system has the capacity to execute. Oversubscription is typical in most HPC environments. Our experience working with Oak Ridge National Labs and the DoD has been that their large-scale systems are almost always oversubscribed [22], [90]. Furthermore, because an undersubcribed system is never at risk of having applications that are unable to execute to completion, the impact of failures and resilience on the performance of the system at any given time will simply be a function of the system's utilization and can be inferred from our analyses in Section 4.6. We therefore provide an analysis on performance in an oversubscribed system that is constrained by requiring individual applications to meet deadlines as defined in Section 4.3.3.

Each simulation begins by filling the entire exascale system with applications, forcing the system to begin operation at full utilization. Applications then arrive to the system randomly according to a Poisson process with a mean arrival time of six hours until a total of 500 applications have arrived to the system. Each application that arrives to the system is uniformly randomly selected from the set of four application types discussed in Table 4.1. Baseline execution times for each arriving application are uniformly randomly selected to be either three, six, twelve, or twenty-four

hours in length. The number of system nodes required by each arriving application is uniformly randomly selected to use between one to one-hundred percent of the exascale system, based on the eleven application sizes experimented with in Section 4.6. The processor MTBF for these studies is 2.5 years.

Each set of 500 applications that arrives to the simulated system is referred to as an *arrival pattern*. Fifty different such arrival patterns were created. The behavior of each resilience protocol and resource management technique was examined using the same set of arrival patterns for fair comparisons.

We compare the five resilience protocols by averaging the results of 50 arrival patterns for each experiment and comparing those values to the average results of an *Ideal Baseline* that executes without delays from failures or delays associated with overhead from resilience protocols. Each resource management technique and resilience protocol combination is assessed in terms of both its ability to minimize the percentage of applications that are dropped from the system, shown in Figure 4.2a, as well as their ability to maximize value earned by the system, shown in Figure 4.2b.

In comparison to the performance of the Ideal Baseline, the results from these simulated studies in Figure 4.2 show how the presence of failures and overhead from resilience protocols negatively impacts system performance by both increasing the percentage of dropped applications in the system and earning the system less value regardless of which resilience protocol is utilized. The results also demonstrate the superiority of the multilevel checkpointing and parallel recovery resilience protocols over checkpoint restart and both redundancy-based protocols in an oversubscribed system. However, the performance of multilevel checkpointing and parallel recovery are very similar in their ability to both maximize value and minimize the number of dropped applications. Although the results from Section 4.6 indicate that the $2\times$ redundancy protocol is capable of providing the best efficiency to some applications, the results in Figures 4.2a and 4.2b demonstrate that while a particular resilience protocol may allow certain application types the best performance in some situations, it may not be suitable for more typical system use cases of oversubscribed systems where resources are in demand by many applications. Because a large number of the pro-

posed resilience protocols discussed in Section 4.2 rely on some form of redundancy, these results imply that there is a substantial constraint on the utility that can be expected from these proposed works.

Another observation that can be made from these results is that, with the exception of FCFS which always performs worse than the other resource management techniques, when using the checkpoint restart or redundancy-based resilience protocols the set of applications that are able to be successfully executed by the system is so low that it reduces the decision making ability of the resource management techniques, making their performance very similar. The multilevel checkpoint and parallel recovery protocols see a greater variability between the performance of each resource management technique. However, the performance of the multilevel checkpoint and parallel recovery protocols themselves are consistent with the results in Section 4.6.

4.9 Resilience-Aware Resource Management

The implication from the results of our studies in Section 4.6 and Section 4.8 is that there is a potential for improving system performance by providing the system a means of making scheduling decisions that are aware of the negative impact that system failures and overhead associated with resilience protocols has on application execution. In addition to deciding when and on what nodes an application will execute, the system resource manager intelligently selects the resilience protocol that is most likely to provide the best performance for each application. The optimal application-specific resilience protocol choice can be made by comparing the efficiencies each application is expected to achieve based on predictions made for each resilience protocol by the execution time prediction equations described in Section 4.5. We call this resilience-awareness feature *resilience selection*, and it allows the system to take advantage of the trade-offs between resilience protocol optimality described in Section 4.6 regardless of the resource management technique being employed.

If applications are being scheduled in the system using either the slack-based resource management technique or any of the value-based resource management techniques, then the system has

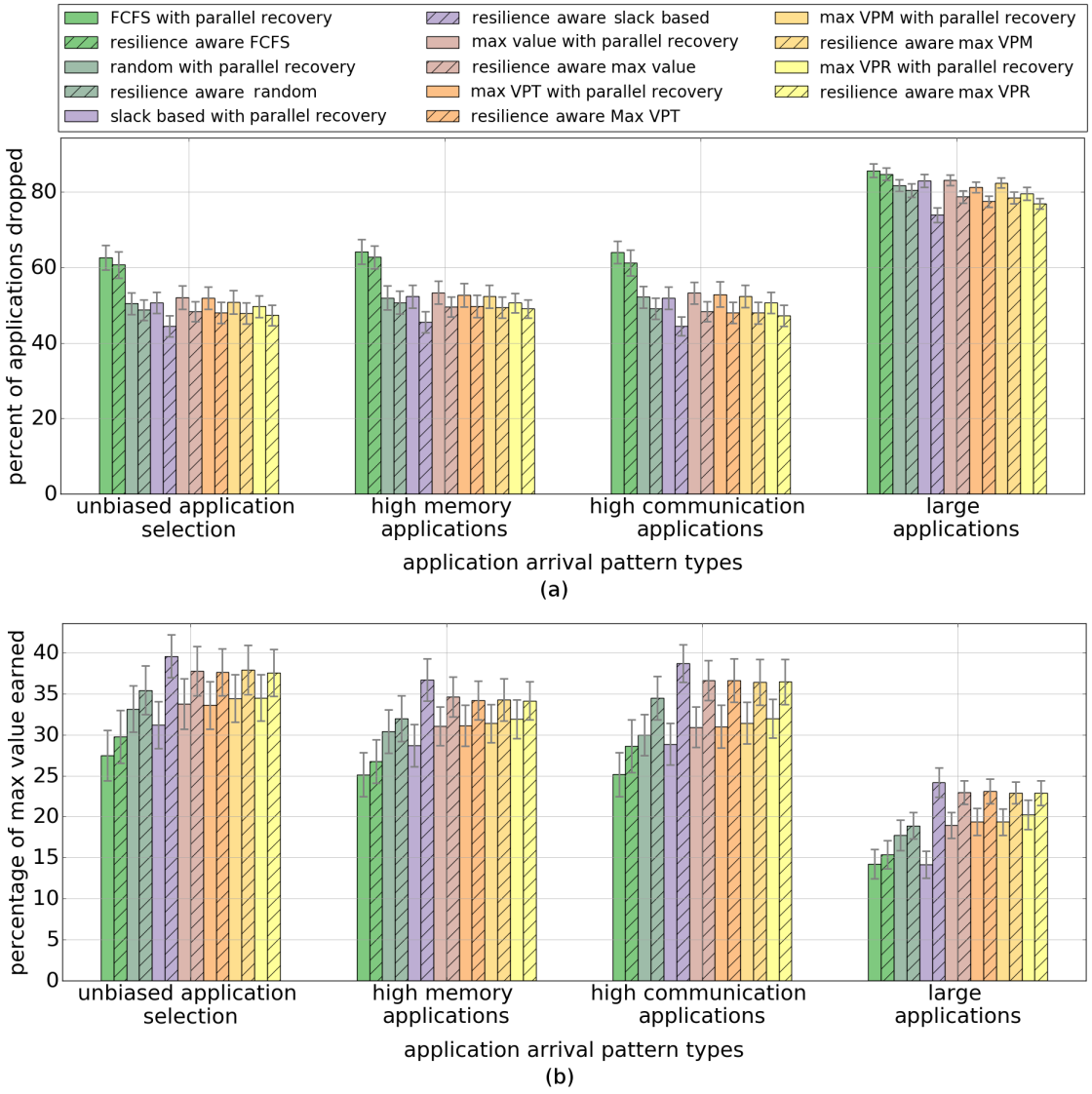

Figure 4.3: Performance of the system by **(a)** percentage of applications dropped from the system, and **(b)** percentage of the maximum value earned by the system, for each resource management technique when resilience-naïve and using the parallel recovery resilience protocol, and each resource management technique when resilience-aware and employing resilience-selection. Groupings of bars show four different types of application arrival patterns. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

the additional advantage of being able to use these same equation-based execution time predictions to make scheduling decisions that are also aware of the decreased performance that the application will experience in a system with failures. While the results shown in Figure 4.1 indicate that some application types perform most efficiently when utilizing $2\times$ redundancy, the results from Section 4.8 demonstrate that in oversubscribed systems, such as the one considered here, use of redundancy has a negative impact on the overall system performance. Use of redundancy is therefore excluded from consideration when resource managers are utilizing resilience selection.

Simulated studies exploring resilience-aware resource management have a similar setup to Section 4.8. However, in addition to the unbiased application arrival pattern seen in Section 4.8 that allows for a uniformly random selection of applications of different sizes and types, these studies also experiment with arrival patterns that are biased toward:

- high memory applications requiring $N_m = 64$ GB;
- high communication applications having communication values of $T_C = 0.75$;
- large applications that occupy at least fifty percent of the exascale system.

These application arrival pattern types were chosen because they are likely exist in exascale environments [69]. Our results depict the average percentage of applications dropped in each executed arrival pattern in Figure 4.3a and the average percentage of maximum value earned by the system for each arrival pattern in Figure 4.3b. Results are shown for each resource management technique from Section 4.7 when utilizing the parallel recovery resilience protocol (indicated by each of the bars without hash marks in the figures) because it is most consistently the best performing resilience protocol. Each resource management technique utilizing parallel recovery is also compared to execution of the same set of arrival patterns when each technique is resilience-aware both in its prediction of execution times and in the resilience protocol that is selected to provide the application the best performance possible. Resilience-aware results are indicated by the hashed bars in the figures.

In general, the results shown in the figures demonstrate that in all cases the resilience-aware resource management techniques improve upon their resilience-naïve counterparts by several percent, and in some cases the improvement is substantially more. Because scheduling decisions made by the FCFS and Random resource management techniques do not rely on execution time predictions, the relatively small improvements they gain when resilience-aware in comparison to the slack-based and value-based resource management techniques demonstrate that the improvement achievable using only resilience protocol selection is not as great as that achievable when also allowing for execution time predictions.

Even though the high-memory and high-communication sets of application arrival patterns were expected to prove more challenging for the resource management techniques, they both perform similarly to the unbiased set of application arrival patterns. Unsurprisingly, arrival patterns biased toward large applications perform notably worse than the other arrival pattern types because they require more system resources. But the large application arrival patterns still benefit by a similar amount from being resilience-aware as the other three arrival pattern types.

Results also indicate that, for all metrics and for all application arrival patterns, the resilience-naïve slack-based and four resilience-naïve value-based resource management techniques using parallel recovery generally only perform within 1% to 2% of the resilience-naïve Random resource management technique using parallel recovery due to inaccurate execution time estimates. However, allowing these techniques to perform "resilience-aware" on the same application arrival patterns enables every technique to earn the system much more value (in some cases as much as 10% more). In the case of the value-based techniques, this improvement comes by allowing each technique to more accurately assess the relative values achievable when scheduling different applications. However, even though there are differences in performance between the resilience-naïve value-based techniques the resilience-aware value-based techniques, all resilience-aware techniques tend to perform similarly within each arrival pattern type, indicating that if resilience-aware predictions are used then the type of value-based heuristic used by the system does not matter much. Improvement in value for the slack-based technique occurs as a bi-product of the

resilience-aware technique being able to more accurately assess which applications are closest to their deadlines, and respond by successfully preventing more of them from being dropped.

The performance of the slack-based resource management techniques in particular demonstrates the benefit that can be obtained when providing certain resource management techniques with resilience information during scheduling. For all application arrival pattern types, the slack-based resource management techniques improve from being one of the worst-performing techniques when resilience-naïve (not even able to out-perform the Random technique), to clearly being among the best performing technique in all metrics by a significant margin when resilience-aware. The findings from our study motivate the design of a new class of resource management techniques that consider resilience protocol selection as an integral part of their decision making.

4.10 Conclusions

HPC resilience has become an increasingly important topic as we approach exascale system sizes. It has also become increasingly important that the resilience protocols that are proposed for use in these systems are analyzed in a common computing environment. This work is one of a few studies that tests a variety of new HPC resilience protocols in such a manner. We describe a methodology that can be used to simulate exascale HPC system sizes with a diverse set of applications able to scale to arbitrary sizes.

We utilize our simulation models to evaluate four protocols for HPC resilience, i.e., the traditionally employed checkpoint restart protocol, as well as three techniques proposed for next generation large-scale systems: multilevel checkpointing, parallel recovery, and partial redundancy. Our analysis indicates that each resilience protocol has performance trade-offs that vary based on application execution characteristics.

Because a production exascale system is unlikely to execute only exascale sized applications, we study the effects that HPC resilience and system failures have on resource management for a workload consisting of several petascale applications. Our results indicate that while multilevel checkpointing and parallel recovery are likely to be the best performing resilience protocols, for all

of the resource management techniques we consider there is still a significant decrease in system performance due to failures and overhead from resilience protocols. However, we also show that the system performance can be improved by using resilience-aware resource management techniques that schedule applications to nodes, select the resilience protocol used for each application, and use execution time predictions to address both the uncertainties introduced by system failures and the overhead introduced by resilience.

Chapter 5

Optimizing Checkpoint Intervals for Reduced Energy Use in Exascale Systems

5.1 Introduction

With the number of CPU cores in large-scale computing systems increasing exponentially over time, the failure rates in these systems have increased exponentially as well. It is expected that the next generation of HPC machines will experience failures up to several times an hour, making the need for effective fault resilience important for building tomorrow's HPC systems [46].

Another important consideration for the development of extreme-scale HPC systems is the increasingly high energy costs associated with using the system once it is operational. Today's largest HPC system consumes approximately 15.371 MW at full capacity (not including the costs for cooling) [75]. Assuming even very low electricity costs of $0.06 per kWh [106] today's systems cost over $8M per year to operate. Based on the conservative exascale system assumptions we outline in Section 5.3, extrapolating these costs to exascale indicates that system operation alone will cost a minimum of $47M per year. Reducing an exascale system's energy requirements has been credited as being one of the most important and challenging roadblocks associated with developing such a system [107]. While some work has been done to explore the role that fault resilience plays in the energy use of large scale systems, little work has been performed that specifically attempts to reduce the impact that resilience techniques have on system energy use.

Performance efficiency is defined to be the ratio of an application's baseline execution time over the application's execution time with slowdowns from failures or resilience technique overheads

This work was performed jointly with masters student Rohan Jhaveri the full list of co-authors listed in [105]. This work was supported by the NSF under grants CCF-1252500 and CCF-1302693. The authors thank Hewlett Packard (HP) of Fort Collins for providing us some of the machines used for testing.

(e.g., delays when taking a checkpoint). There is a critical need to allow a system designer the opportunity to trade-off application performance efficiency (corresponding to minimizing application execution time) with minimizing system energy use in extreme-scale systems, to optimally balance performance goals with energy overheads.

Because the costs of operating an exascale system are very high, even for scientific applications with an execution time as short as one day without overheads from failure and resilience related delays, optimizing checkpoint intervals for energy use at the cost of a slight increase in application execution time would likely allow for over a million dollars of energy savings by the end of the application's execution. Our results demonstrate that for a few percent reduction in performance, the energy required to execution an application can be reduced by as much as 10%.

In summary, this chapter makes the following novel contributions:

- we develop a set of execution time and energy use prediction equations that can be utilized to determine optimal checkpoint intervals for both traditional checkpointing and multilevel checkpointing-based HPC fault resilience techniques;
- we provide a methodology for simulating the execution of applications operating at exascale system sizes in the presence of uncertainty due to failures;
- we use our methodology to model an exascale computing environment and utilize this environment to simulate the execution and energy use of both a traditional checkpointing technique as well as a multilevel checkpointing technique proposed for future exascale-sized HPC systems;
- we simulate each resilience technique's performance and energy use when optimizing checkpoint intervals for either application performance efficiency or lower energy use and demonstrate that a trade-off exists between optimizing for either metric;
- we perform a sensitivity analysis of the parameters associated with this trade-off.

The remainder of this chapter is organized as follows. Section 5.2 discusses the resilience techniques we examine and discusses other work that has considered exascale resilience and energy

use. In Section 5.3, we describe the modeling methodology we use for our system simulator. Our implementation of HPC resilience is presented in Section 5.4. Section 5.5 details some of the equations we derive and use for determining optimal checkpoint intervals. Section 5.6 describes the simulated studies we perform to demonstrate the trade-off between performance efficiency and energy use and provides a sensitivity analysis on these results. We conclude with a summary of this work in Section 5.7.

5.2 Related Work

The prior work we discuss here focuses on system-level checkpointing-based HPC resilience that allows application programmers and users of the system to be oblivious to the strategies for HPC resilience that are being employed on their behalf. All checkpointing-based techniques rely on the notion of periodically saving the system's executing state and restarting from an earlier error-free state after the occurrence of a system failure [52]. We focus on two of the most popular checkpointing-based techniques, a traditional checkpoint/restart based technique and the more recently proposed multilevel checkpointing technique.

Traditional checkpoint/restart is by far the most commonly used resilience technique employed by today's large-scale HPC systems. However, the length of time associated with checkpointing, restarting, and recomputing work lost to a system failure for a large-scale application can be quite large. Moreover, the frequency at which the system needs to take checkpoints for very large-scale applications when implementing traditional checkpointing techniques has been shown to significantly degrade performance with increasing system sizes [45]. Traditional checkpointing alone is not expected to be capable of providing satisfactory resilience to exascale-sized systems.

Because different types of failures can affect a computing system by different amounts, not all failures require restarting the system from a checkpoint to the parallel file system [80]. Multilevel checkpointing exploits this by providing the system with several levels of checkpointing. A system employing a multilevel checkpointing scheme may allow for levels that trade-off between checkpoints to RAM (that are faster but able to recover from fewer types of failures), to checkpoints

saved to a partner node's RAM (that are less frequent and slower, but able to recover from more types of failures), to checkpoints saved to the system's parallel file system (that are the slowest but allow recovery from almost all failures). Each level offers a trade-off between the time required by the system to checkpoint or restart, and the level of failure severity from which the checkpoint can recover [47].

One challenge associated with using a multilevel checkpointing technique is in determining the optimal number of checkpointing levels to support in the system, and the optimal computation intervals between checkpoints at each level. Various solutions to this problem have been proposed [47] [61] [62] [81]. We also provide our own solution to the problem of determining optimal multilevel checkpoint intervals that we described in Chapter 4 Section 4.5 and Section 5.5 of this chapter.

Assessing the energy use associated with fault resilience techniques has been considered in several works [108] [109] [82] [110] [43]. However, none of these works specifically optimize checkpoint intervals to attempt to minimize energy use as our work does.

The work in [111] does specifically attempt to minimize system energy use for the checkpoint/restart resilience technique. However, the authors achieve this through power capping and not through checkpoint interval optimization as our work does. The authors in [112] do specifically reduce energy use in a checkpoint restart-based technique by optimizing checkpoint intervals, but their work is primarily focused on checkpointing in mobile devices that store checkpoints over the internet and is of limited use for the exascale-size HPC systems that we consider in our work. Both of these works also do not consider energy reduction for multilevel checkpointing.

An approach to predict checkpoint/restart execution time and energy use was explored in [45], [82], and [73]. We have greatly extended this prior work to allow for much more accurate predictions and further extended the initial ideas to allow for modeling multilevel checkpointing with an arbitrary number of checkpoint levels.

5.3 Exascale Modeling Methodology

5.3.1 Overview

We have designed an event-based simulator used for modeling HPC systems of arbitrary size [43] [77] [4]. The system experiences randomly generated failures that affect the simulated execution of applications in the system. Throughout the system's simulation an application's execution is affected by events associated with:

- *computation*: execution toward the application's completion,
- *failures*: the simulated failure of a system node,
- *checkpoints*: saving a backup of computation progress,
- *restarts*: restoring the application progress saved in the last system checkpoint after a failure occurs,
- *recovery*: recomputing progress lost to a failure,
- *failed checkpoints or restarts*: behavior of the system when failures occur during checkpoint or restart events.

Checkpoints, restarts, and recovery are all resilience-technique specific events that determine how an application behaves in a system with failures. This is discussed in detail in Section 5.4. The remaining events associated with computation and failures are all attributes of the system and behave the same regardless of the resilience technique being used by the system. In particular, while failure events have a large impact on the behavior of the resilience-technique related events, failure events themselves are a function of the size of the system and the reliability of the system's nodes, and are not affected by the resilience technique employed by the system.

5.3.2 Applications Model

We consider synthetic applications that exhibit weak scaling so that as the number of nodes used by the application increases with application size, the application's attributes of computation

and memory used per node remain constant. For all simulated studies performed here, applications are defined to execute for a full day of execution time when executed without delays from failures or events related to resilience (such as time spent checkpointing). This delay-free execution time is the application's *baseline execution time* and is represented by the variable T_B.

Memory needed for the application is represented by the variable N_m. Most of the simulations performed in this work have a value of $N_m = 32$GB of memory required per node. Details about the size and memory use of applications in each simulated study are discussed further in Sections 5.6.

5.3.3 Simulated System Setup

The simulated exascale system is a homogeneous system inspired by the architecture used in China's Sunway TaihuLight 125 petaflop supercomputer [75], the world's highest performing system since June 2016 and still the world's top system as of June 2017 [99]. Each Sunway TaihuLight system node has a multicore architecture composed of four clusters of 64 computational processing elements (CPEs) with each cluster managed by a single management processing element (MPE) that also performs computational work allowing for a total of 65 cores of computation in each core cluster. The four core clusters in a system node provide a total of about 3.1 TFLOPs over 260 cores. As systems trend towards manycore architectures, with hundreds or thousands of CPU cores on a single socket, component failure rates are likely to increase [103] [104]. Our simulated exascale system assumes an approximate $4\times$ increase in the number of CPU cores per processor over the Sunway TaihuLight system by the time an exascale machine is developed, allowing for a total of 1028 cores per node providing approximately 12 TFLOPs of compute power for each system node. A system composed of 120,000 of these high performing nodes would perform at an exascale level.

The Sunway TaihuLight system has 8 GB of DDR3 RAM at each of its four core clusters, giving each node a total of 32 GB of RAM. We again assume that future systems are likely to have memory increases of about a factor of four in comparison to today's systems giving our simulated

system a total memory capacity of 128GB per system node, enough to allow for the multilevel checkpointing resilience technique to operate with a working memory size of up to 40GB and still be able to accommodate the additional memory required for lower-level checkpoints (discussed in Sections 5.3.4 and 5.4). In addition to an increase in volume, we also assume that future memory is likely to utilize newer architectures, allowing for increased aggregate memory bandwidth, B_M. Today's best memories perform at a rate of up to 25 GB/s [100], we conservatively estimate that future memories will be able to perform at about $B_M = 40$ GB/s.

Reports on the Sunway system [75] [113] mention that a system node requires around 375 watts to operate during computation. We assume that the total power required for computation will increase because of the large increase in the number of cores per system node, however, it can also be expected that the energy efficiency of future processors will improve. Given that we are assuming a $4\times$ increase in the number of cores per system node in our simulated system, we conservatively assume the power required for computation work, P_W, will double from the power requirements of the Sunway system.

Because the system halts all computation during checkpoints and restarts, the power required by the system during these events is much lower than the power required for computation. Based on power measurements we have made on several server-class Xeon processor based servers, we assume that the power required during a checkpoint or restart, P_δ, will be equal to the node's idle power (we assume this to be 150 watts) plus the power required for network communication (discussed in Section 4.3.5).

5.3.4 System Failure Model

Failures are characterized by two attributes: the time that the failure occurs and the "severity class" of the failure. The uncertainty associated with each attribute is modeled using random variables. We assume independence between individual failure occurrences as well as between both the attributes of each failure.

A common assumption when modeling failures in an HPC system is that failures follow an exponential distribution and can be modeled by a Poisson process [67]. Every failure occurs according to the previous failure's arrival time ($T_{F_{i-1}}$, with $T_{F_0} = 0$) plus a random variate generated from an exponential distribution $\mathcal{T}_i \sim Exp(\lambda)$ with an expected arrival rate of $E[\mathcal{T}_i] = \frac{1}{\lambda}$. The parameter λ indicates the average failure rate of the application, and is defined as the number of nodes required by the application, N_A, divided by the mean time between failures of the system nodes, M_n, i.e.,

$$\lambda = \frac{N_A}{M_n} \, . \tag{5.1}$$

The severity class of failure corresponds to the type of failure that has occurred in the system. This attribute is used by multilevel checkpointing to determine which level of saved checkpoint is necessary to enable the system to recover from a specific type of failure and is also used when determining the optimal duration of intervals between checkpoints of different levels. These assumptions and the effect of a failure's class on multilevel checkpointing's behavior is discussed further in Section 5.4.

The mapping of failure types to failure severity classes is based on the analyses of types of failures present in modern day HPC systems presented in [101] and [78]. We define the probability of experiencing a class i severity failure according to the ratio of the number of failures that occur at each failure severity class, to the total number of failures, measured over an extended interval of time. The resulting discrete set of ratios for the set of classes is used to create a probability mass function from which random variates are sampled to define the severity attribute of each failure. We denote the probability of each of the L severity classes as $S_1, ..., S_L$. We assume that types of failures in a future exascale-sized system will occur in similar relative amounts to those experienced by today's system, but with the total number of failures occurring more frequently.

For this work we define three severity classes. The first class (S_1) assumes that failures can be recovered from using a checkpoint stored in a node's local RAM, the second class (S_2) requires the restarting application from a checkpoint stored in a partner node's RAM, and the third and highest class (S_3) requires the system to checkpoint and restart from a parallel file system. The

data presented in [101] and [78] details the frequencies of thirty-three types of failures in the Blue Waters system. Using this information we determined the probability of failure severities for our three-class failure model to be approximately $S_1 = 0.138$, $S_2 = 0.784$, and $S_3 = 0.078$.

5.3.5 Communication Model

System communication plays a key role in the performance of checkpoints written to both a partner node and the parallel file system. We account for the effects of communication on application checkpoints taken in the system. We assume that future exascale systems are likely to have improved communication over today's systems, and base the communication model for the studies performed here on the "NDR InfiniBand" network described in [102]. For most simulations in Section 5.6, our communication network assumes a latency value of $L_N = 0.5\mu$s, a bandwidth value of $B_N = 600$GB/s, and a maximum number of simultaneous connections at each switch $N_S = 12$. The effects of these values on the corresponding time required to checkpoint the system is discussed further in Section 5.4.

The power used for system communication is calculated for each system node. Given a switch power of $P_S = 200$ watts and a network interface controller of a single node that consumes $NIC = 15$ watts at full utilization [72], the power spent by a single node for communication during a time of high network traffic (such as during a checkpoint or a restart), P_N, and defined as

$$P_N = \frac{P_S}{N_S} + NIC \ , \tag{5.2}$$

making P_N equal to about 28.33 watts. Power associated with communication during application computation is assumed to be accounted for in the power values taken from [75].

5.4 Fault Resilience Techniques

5.4.1 Overview

Two HPC fault resilience techniques are considered in our work and were implemented in our system simulator: a traditional checkpoint restart based technique, *checkpoint/restart*, as well as the implementation of multilevel checkpointing proposed for next-generation HPC systems in [47]. The following subsections present details of how each resilience technique was modeled.

5.4.2 Checkpoint Restart

Our implementation of the checkpoint/restart resilience technique performs periodic, blocking checkpointing, with checkpoints saved to a parallel file system. The time that the checkpoint/restart technique requires to read and write its checkpoint data to a parallel file system, $T_{C_{PFS}}$, is dependent on number of nodes required by the application, N_A, memory use N_m (defined in Section 5.3.2), and communication bandwidth B_N (defined in Section 5.3.5) to give

$$T_{C_{PFS}} = \frac{N_m}{B_N} * N_A \ . \tag{5.3}$$

The optimal checkpoint period is dependent on the application's checkpoint time and failure rate. As defined in Section 5.3.4, the value for an application's failure rate is dependent on the application's size. We describe our calculation of the optimal interval between application checkpoints (τ) when optimizing for either maximum performance efficiency or minimum energy use in Chapter 4 Section 4.5 and Section 5.5 of this chapter.

5.4.3 Multilevel Checkpointing

We implement the three-level multilevel checkpointing technique from [47] in our simulator. Each checkpointing level offers a trade-off between the time required to save or restore a checkpoint and the severity class of the failure from which it can recover.

The first checkpoint level writes to the node's local RAM. All applications save the checkpoint concurrently with the time required for taking a level one checkpoint being simply the amount of memory per node required by the application divided by the node's memory transfer rate

$$T_{C_{L1}} = \frac{N_M}{B_M} \ . \tag{5.4}$$

The second checkpoint level stores its checkpoints to RAM in a partner node. The time for a level two checkpoint is equal to the time required to send the data to the partner node (calculated using variables defined in Section 5.3.5) plus the time required to write the data to memory

$$T_{C_{L2}} = 2(T_{C_{L1}} + L_N + \frac{N_M}{B_M}) \ . \tag{5.5}$$

The equation is multiplied by two to account for both the time required for half of the system to concurrently send checkpoint data to their respective partner nodes as well as the time required for those nodes to receive their partner's checkpoint data.

The third level checkpoint is written to a parallel file system, and the time required is the same as presented in Eqn. 5.3. We assume checkpoint and restart times are symmetric. Failure severity classes are defined according to Section 5.3.4 and determination of optimal checkpoint intervals for each level is describe in Chapter 4 Section 4.5 and Section 5.5 of this chapter.

5.5 Energy Use Prediction and Checkpoint Interval Optimization

The equations we derived that predict the expected execution time of applications executing in the presence of failures when employing either the traditional checkpoint/restart or the multilevel checkpointing techniques were described in Chapter 4 Section 4.5. This section discusses how these execution time prediction equations are then extended for use in predicting the system's expected energy use when executing the application. These equations are general and able to

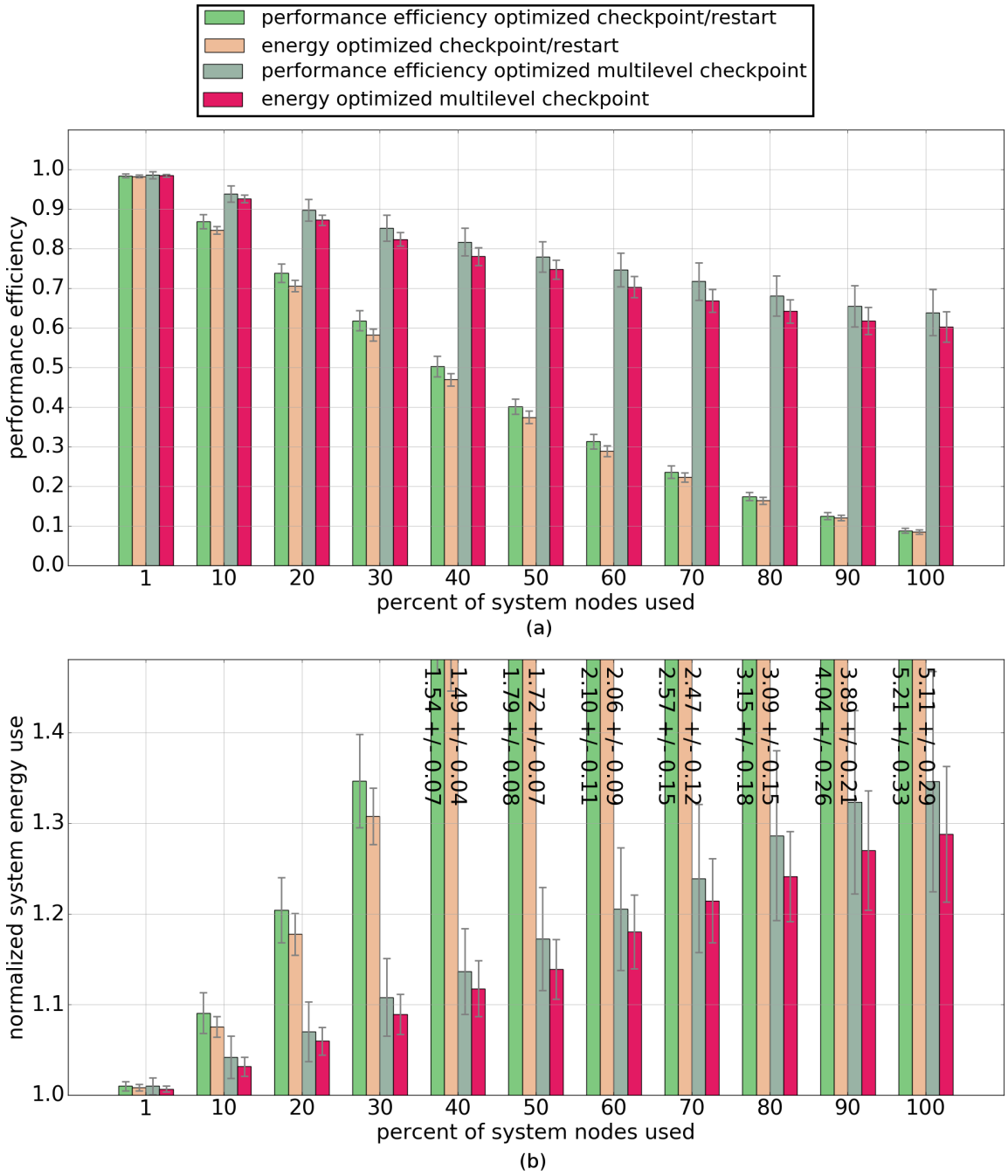

Figure 5.1: Application **(a)** performance efficiency and **(b)** normalized energy use, when resilience technique checkpoint intervals are optimized for either performance efficiency or energy use and the percentage of system nodes used by the application is increased. Bars in the figure represent the average of 200 simulated trials. Standard deviations are shown for each bar. Annotations in the figure indicate values for the average and standard deviation of bars that have been truncated.

be used on systems and applications of any type provided the relevant system and application parameters can be estimated.

The energy use models for both checkpoint/restart and multilevel checkpointing are calculated by multiplying the expected execution time of each event type with the power used to perform that event. For computation events, system power use is equal to the power used by a node during computation (P_W, defined in Section 5.3.3) multiplied by the number of nodes used by the application (N_A from Section 5.3.4). For checkpoint or restart related events, power use is defined by the number of nodes used by the application multiplied by the power used by a node while it is checkpointing or restarting (P_δ, defined in Section 5.3.3). Once the values for the total execution time have been calculated the expected execution time for each event type can be calculated. This makes the expected total energy for checkpoint/restart, E_{CR}, equal to

$$\begin{aligned} E_{CR} &= P_W N_A (T_B + T_{W_\tau} + T_{W_\delta}) \\ &+ P_\delta N_A (T_\delta + T_{\delta'} + T_R + T_{R'}) \ , \end{aligned} \tag{5.6}$$

and E_{ML}, the expected energy for multilevel checkpointing

$$\begin{aligned} E_{ML} &= P_W N_A (T_B + T_{W_{\tau_L}} + T_{W_{\delta_L}}) \\ &+ P_\delta N_A (T_{\delta_L} + T_{\delta'_L} + T_{R_L} + T_{R'_L}) \ . \end{aligned} \tag{5.7}$$

We use the same brute force sweeping technique to optimize decision variables for minimal energy use as we used in Chapter 4 Section 4.5.6 for determining intervals for optimal performance efficiency except that instead of evaluating for execution time we evaluate Eqns. 5.6 and 5.7 to find decision variable values that produce the minimum expected energy use. When optimizing for energy use, the solution space for decision variable values has the same bounds as when optimizing for performance efficiency.

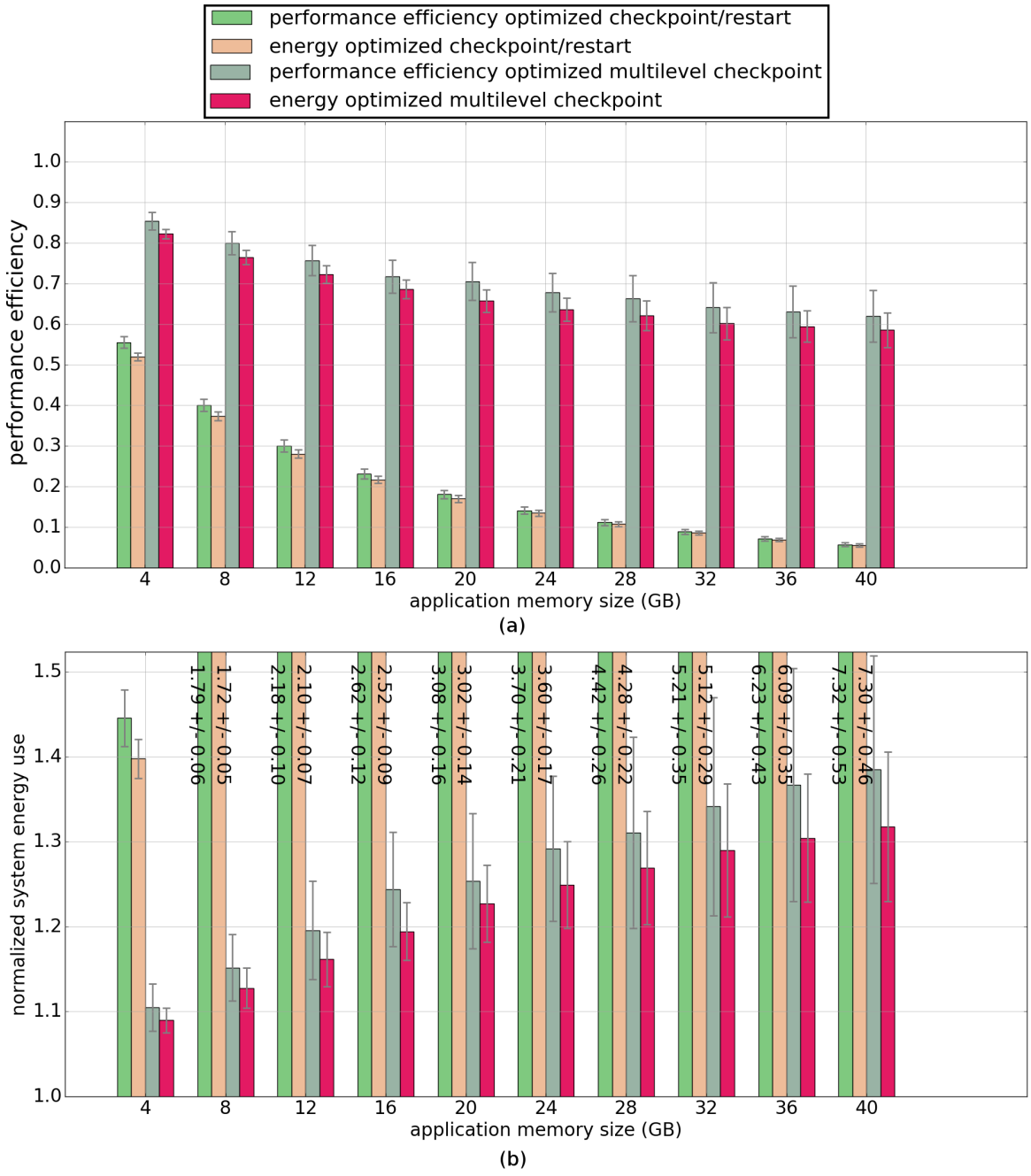

Figure 5.2: Application **(a)** performance efficiency and **(b)** normalized energy use, when resilience technique checkpoint intervals are optimized for either performance efficiency or energy use and the amount of memory used by the application is increased. Bars in the figure represent the average of 200 simulated trials. Standard deviations are shown for each bar. Annotations in the figure indicate values for the average and standard deviation of bars that have been truncated.

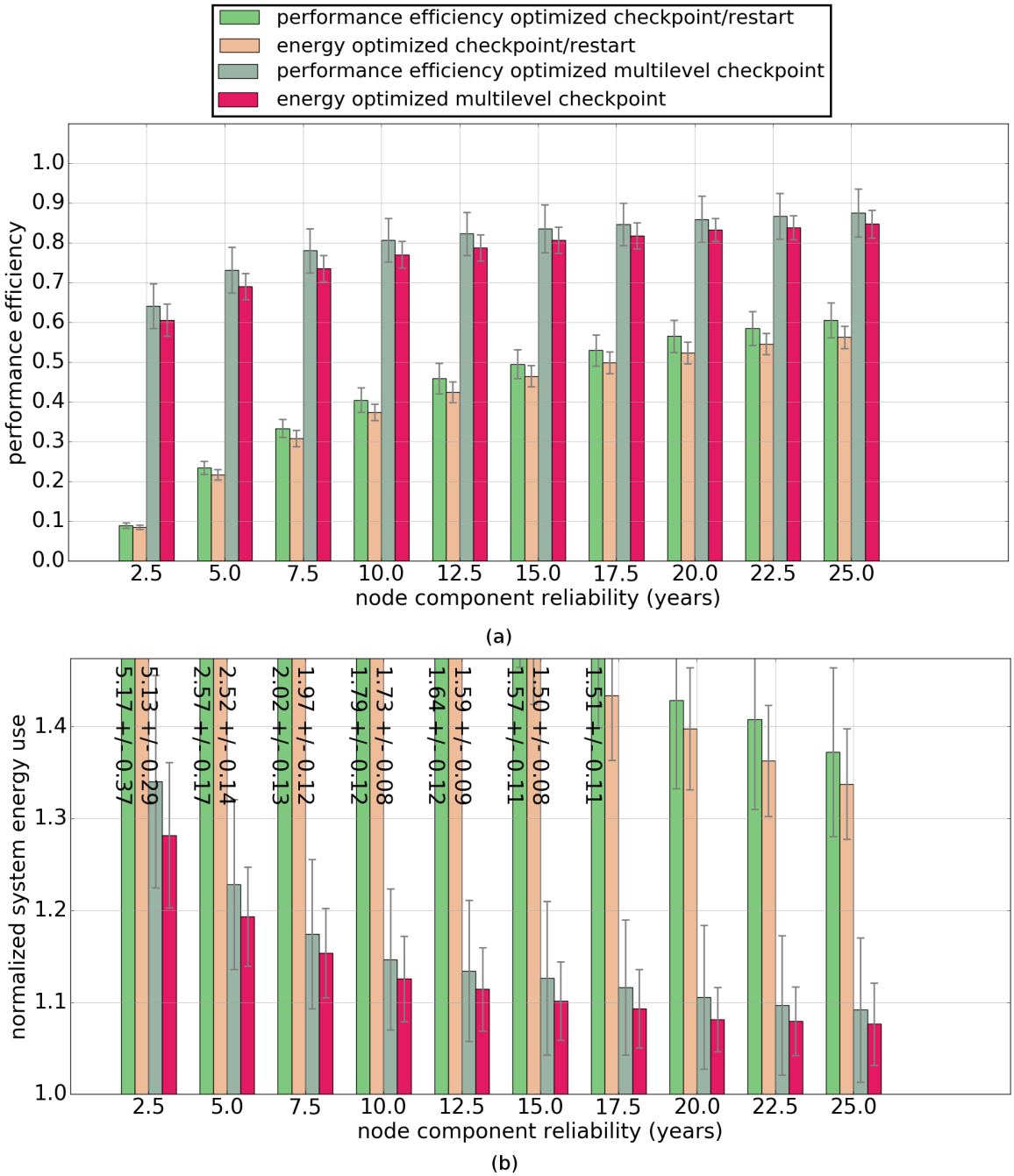

Figure 5.3: Application **(a)** performance efficiency and **(b)** normalized energy use, when resilience technique checkpoint intervals are optimized for either performance efficiency or energy use and the reliability of system nodes is increased. Bars in the figure represent the average of 200 simulated trials. Standard deviations are shown for each bar. Annotations in the figure indicate values for the average and standard deviation of bars that have been truncated.

5.6 Simulation Experiments

5.6.1 Overview

We use the equations defined in Section 5.5 to calculate checkpoint intervals that either maximize performance efficiency or minimize energy use and use the resulting interval values with the simulation environment discussed in Section 5.3 to conduct several simulated experiments in Section 5.6.2 that examine the trade-off between performance and energy. In Section 5.6.3, we perform a sensitivity analysis that explores how this trade-off is affected when various application characteristics and system parameters are scaled through a range of values.

5.6.2 Optimization Trade Off

In Figure 5.1, we demonstrate the performance of and energy use of the system as the simulated application is scaled in size from one percent of the exascale system (about 1.2 million CPU cores, similar in size to some of today's largest applications) through to an exascale-sized application requiring 123 million CPU cores. For these experiments, the baseline execution time for each application is defined as $T_B = 86400$ seconds, or one full day of execution. The energy use values are normalized to the calculated value of the application's baseline energy use, $E_B = P_W N_A T_B$, for each application size. Most of the prior work we consider assume a node MTBF of ten years for current HPC systems. We assume component failure rates will increase linearly with the increased size of system nodes, and consequently for our experiments we assume an MTBF of $M_n = 2.5$ years. We assume that the application uses $N_m = 32$GB of memory per node. Simulated trials vary because of uncertainty associated with randomly occurring failures. Bars in the figure represent the average of 200 simulated trials and the error bars indicate standard deviations.

Figure 5.1 shows that for both checkpoint/restart and multilevel checkpointing there exists a distinct trade-off between optimizing checkpoint intervals for performance efficiency and optimizing checkpoint intervals for energy use. Checkpoint/restart is 0.5-3% more efficient when optimizing for performance efficiency than when optimizing for energy use but consumes as much as 15% more energy. Multilevel checkpointing has as much as 4% higher performance efficiency

when optimizing for performance efficiency but can consume as much as 7% less energy when optimizing for minimum energy use. This difference in optimality arises because of the difference in power requirements when the system is checkpointing or restarting as opposed to when it is performing computation. If a system designer desires to use less energy, then the results in Figure 5.1 indicate that it is more beneficial to slightly increase the execution time of the application by taking more frequent checkpoints that require less power but require less time be spent performing power costly computation when recovering from a failure.

As the application's size increases it can be observed that when optimizing for energy use with the multilevel checkpointing technique the loss in performance efficiency increases at a slower rate than the decrease seen in energy use, allowing the same burden on efficiency to provide larger decrease in energy use for larger application sizes. This effect is less prevalent in the results for checkpoint/restart.

Another trend that is seen with both checkpoint/restart and multilevel checkpointing is that while the variance in results for both efficiency and energy use increases with application size, the variance of results when optimizing for minimum energy use increases more slowly than the variance in efficiency results. This indicates that in terms of both performance efficiency and system energy use execution results are more consistent if checkpoint intervals are optimized for minimal energy use rather than higher performance efficiency. If a system designer desires more predictability of application execution then it is better to optimize for minimum energy use.

5.6.3 Sensitivity Analysis

We analyzed the sensitivity of the results from Section 5.6.2 to a variety of application characteristics and system parameters including: application memory use, system component reliability (expected time between component failures), communication network bandwidth, communication network latency, and system node memory transaction speed. All simulations analyzing sensitivity were performed by simulating an exascale-sized application with all system parameters and execution characteristics that were not being scaled for the sensitivity analysis (i.e., the parameters that

are held constant) remaining as described in Section 5.3. All experiments we performed supported the results discussed in Section 5.6.2 indicating the presence of a trade-off when optimizing for efficiency or energy use as well as all analyzed parameters indicating that there is less variance present in application execution when optimizing checkpoint intervals to minimize system energy use.

A subset of the sensitivity results are shown in Figures 5.2 and 5.3. The figures show the efficiency and energy use when scaling values of application memory use and system component reliability, respectively. Bars in the figures represent the average of 200 simulated trials with standard deviations shown for each bar.

Increase in application memory use (Figure 5.2) has a particularly large effect on both the efficiency and energy use results as well as the trade-off in improvements that can be gained by optimizing for either efficiency or energy use. Though not all results are shown, the results for sensitivity to changes in application memory size had the largest impact on performance efficiency and system energy use for an exascale-sized application. Checkpoint/restart shows a 5-14% decrease in energy use for at most a 3% decrease in performance efficiency when optimizing for energy use over performance efficiency. Multilevel checkpointing achieves between a 2-6% decrease in energy use for a 2-3% decrease in performance efficiency when optimizing for energy use over performance efficiency.

The results in Figure 5.3 show that as component reliability (MTBF) increases, the performance of the system increases and the energy improvement gained when optimizing energy use over performance efficiency decreases. However, even in a highly reliable system where failures are less common, the trade-off as well as the increased predictability of performance when optimizing for energy use are still present and can be taken advantage of by a system designer.

5.7 Conclusions

HPC resilience has become an increasingly important topic as we approach exascale system sizes and failures become more frequent. Similarly, as systems begin to be developed that re-

quire hundreds of thousands of system nodes, the energy requirements of these systems becomes extremely costly and reducing this burden continues to be an important consideration for system designers.

We described a methodology that can be used to simulate exascale HPC systems (with the ability to scale to arbitrary system sizes) and model the effects that extreme-scale systems have on performance efficiency and energy use in the presence of node failures.

We utilize our simulation models to evaluate two techniques for HPC resilience, the traditionally employed checkpoint/restart technique, as well as the multilevel checkpointing technique proposed for next generation large-scale systems and use a set of equation-based models we have developed to optimize the checkpointing intervals of these techniques to either maximize performance efficiency or minimize system energy use. Our analyses indicate that a performance trade-off exists between optimizing these techniques for either metric. Given the presence of this trade-off, we performed a sensitivity analysis on several parameters associated with application and system behavior and conclude that this trade-off exists in all circumstances we test. Our results also indicate that optimizing for minimal energy use provides the system with less variable ranges of execution times.

Chapter 6

An Analysis of Multilevel Checkpoint Performance Models

6.1 Introduction

As applications have demanded more computing power over time, HPC systems have required exponentially increasing numbers of system CPU cores to provide this performance . With HPC systems approaching exascale computing complexities, it is expected that they will require several million CPU cores. Simultaneously, the drive to improve performance and energy-efficiency by fabricating at smaller transistor technologies has decreased component reliability [49].

Because of these trends, as HPC systems approach extreme scales, system failure rates have increased rapidly. A recent study of the Blue Waters system in [78] indicates that a $2.2\times$ increase in application size (from $10,000$ system nodes to $22,000$ system nodes) resulted in a $20\times$ increase in the probability of application failure. Given that [46] suggests that an exascale application is likely to require at least $100,000$ system nodes, an exascale system can be expected to experience significantly higher failure rates. Moreover, the study in [78] concludes that for the 13.1 petaflop Blue Waters system, on average an application failure caused by a system-related issue occurs every 15 minutes. An exascale machine is therefore likely to experience failures much more frequently and has been estimated to have a system MTBF of as little as three minutes in extreme cases [47].

In such an environment, it is imperative that protocols are in place that allow HPC systems to respond to failures when they occur and mitigate their impact on application performance. However, analysis has shown that current HPC resilience protocols such as traditional checkpoint/restart

This work was performed jointly with the full list of co-authors listed in [114]. This work was supported by the NSF under grants CCF-1252500 and CCF-1302693. The authors thank Hewlett Packard (HP) of Fort Collins for providing us some of the machines used for testing.

and redundancy-based resilience are not suitable for scaling to exascale system sizes [49] [79] [45] [43] [77]. One solution for future systems that has been researched over the last decade in anticipation of these extreme-size HPC systems is multilevel checkpointing. Multilevel checkpointing protocols exploit the fact that not all failures require costly restarts of the application from a PFS, and that less severe failures can be restarted in significantly less time from higher levels of memory (e.g., local or remote DRAM).

As with traditional checkpoint/restart protocols, when performing a checkpoint or restart operation the system must temporarily halt application execution. While this is necessary for successful computation in failure-prone systems, every checkpoint incurs overhead that slows the progress of application execution. Just as application progress is impeded by failures if the duration of computation between checkpoints is too large, checkpoints taken too frequently also incur overhead that prevents application progress. This interaction between a given execution environment and the duration of the interval of computation between checkpoints becomes significantly complex with a multilevel checkpointing system. There is an optimal set of successive intervals between levels of a multilevel checkpointing protocol. Determining these optimal intervals is an open problem associated with multilevel checkpointing and is one of the major challenges with the successful implementation of the protocol.

We have already developed and utilized the equations discussed in Chapters 4 and 5 for the purposes of execution time prediction for resilience-aware resource management as well as checkpoint interval optimization. However, this chapter discusses additional equations used for execution time prediction and checkpoint interval optimization for the traditional checkpoint/restart and multilevel checkpointing resilience protocols. The equations presented in this chapter provide a great improvement over the previous set of equations. This chapter also provides a much more detailed discussion and comparison of our equations to other contemporary equations for multilevel checkpointing including examples comparing utilization of our equations under a variety of HPC system execution scenarios. In this chapter we make the following contributions:

- we provide a detailed comparison among several state-of-the-art techniques for determining multilevel checkpointing intervals;
- we demonstrate the necessity of accounting for failures during checkpoint and restart events when modeling extreme-scale systems;
- we derive an application execution time prediction model in the presence of multilevel checkpointing that can be used for determining the performance of checkpoint intervals;
- we show some of the limits under which multilevel checkpointing ceases to be a viable option for providing resilience to extreme-scale systems;
- we demonstrate the superior accuracy of execution time predictions made with our model, as well as situations in which our model outperforms other state-of-the-art techniques for determining multilevel checkpoint intervals.

The remainder of this chapter is organized as follows. We discuss the historical development and recent progress of multilevel checkpointing, and discuss the challenge of constructing accurate models for both application execution time prediction and checkpoint interval optimization in Section 6.2. We present our own approach to modeling a multilevel checkpointing protocol with an arbitrary number of levels in Section 6.3. We provide a set of simulation studies comparing several state-of-the-art multilevel checkpointing models in Section 6.4. We end the chapter with some concluding remarks in Section 7.5.

6.2 Related work

6.2.1 Overview

Traditional checkpoint/restart techniques have been used for decades to mitigate the effects of system failures in HPC systems. The first attempt to optimize checkpoint intervals was Young's first-order approximation in [53]. This execution time model was substantially improved by Daly's higher order execution time approximation in [73]. Daly's work remains the most common ap-

proach for optimizing traditional checkpointing. However, traditional checkpointing has been shown to not provide adequate resilience for extreme-sized systems.

The first notion of utilizing different levels of checkpoints for recovering from different types of failures was presented in [115] and this was extended to a more practical (two-level) multilevel checkpointing protocol in the Markov model presented in [80]. Whereas traditional checkpoint/restart performs a checkpoint or restarts from one level (typically the PFS), multilevel checkpointing relies on checkpoints and restarts from multiple levels of memory (e.g., local DRAM, remote DRAM, PFS). The benefits of a multilevel checkpointing model is that time-consuming higher severity failures that typically occur less frequently can restart from a checkpoint to a slower and more reliable (lower) level of memory such as the PFS; whereas the more frequently occurring lower severity failures can restart much more quickly from higher (faster) levels of memory such as local/remote DRAM. Checkpoints that are a "higher" level help restore the system from "higher" severity failures but typically store checkpoint data in correspondingly lower levels of memory. It is usually the case that for a multilevel checkpointing protocol with L checkpoint and restart levels (with durations denoted δ_i and R_i, respectively) and L levels of failure severity (with rates denoted λ_i) we have $\lambda_1 > \lambda_2 > ... > \lambda_L$ while $\delta_1 < \delta_2 ... < \delta_L$ and $R_1 < R_2 ... < R_L$. This relationship benefits multilevel checkpointing by a direct reduction in the number of level L checkpoints/restarts taken and from the fact that lower-level checkpoints to higher levels of the memory hierarchy are able to utilize resources across the system as a whole more effectively, allowing for better scalability and a probable reduction in network overhead. High-level checkpoints/restarts to or from a single PFS tend to be much more dependent on the number of nodes used by the application, i.e., the number of nodes needing to store or retrieve data.

6.2.2 Multilevel Checkpointing Techniques Considered

Our work, and that of the other multilevel checkpointing models presented here, consider HPC systems that employ two types of multilevel checkpointing protocols. We discuss the assumptions and common implementation of each protocol below.

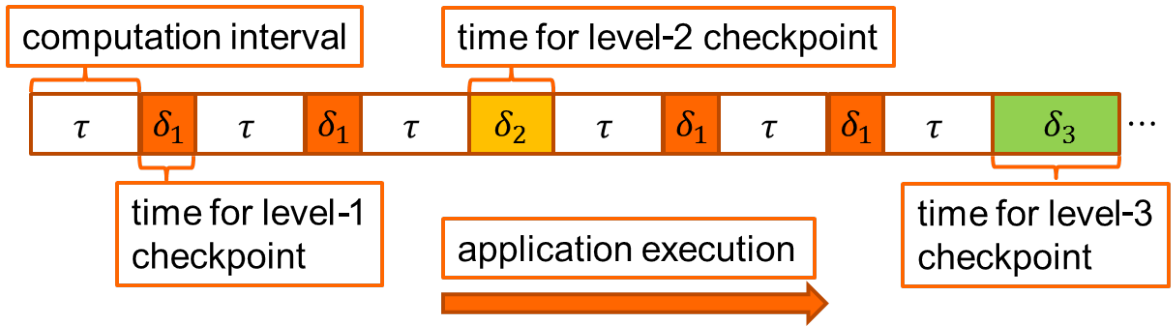

Figure 6.1: A checkpoint interval pattern for a three-level checkpointing protocol with its computation interval denoted τ, its checkpoint lengths of each level i denoted δ_i, and a pattern that performs two level-1 checkpoints before a level-2 checkpoint and a single level-2 checkpoint before each level-3 checkpoint.

1) *Scalable Checkpoint/Restart* (SCR) [47] is an extension of the ideas from [80]. The authors (Moody et al. [47]) develop their own Markov model that is capable of modeling application execution under a multilevel checkpointing protocol with an arbitrary number of checkpoint levels. SCR is designed as a pattern-based multilevel checkpointing model. This assumes that the duration of computation between successive checkpoints is a fixed amount of time, and that the duration of time between higher checkpoint levels is determined by the discrete number of lower level checkpoints (e.g., each level-2 checkpoint will occur after some number of level-1 checkpoints and each level-1 checkpoint will occur after a fixed interval of computation). These checkpoint interval "patterns" define the frequency of checkpoints at each level. Figure 6.1 provides an example of a checkpoint interval pattern for a three-level checkpointing protocol with its computation interval denoted τ, its checkpoint lengths of each level i denoted δ_i, and a pattern that performs two level-1 checkpoints before a level-2 checkpoint and a single level-2 checkpoint before each level-3 checkpoint. Though it is not shown in the figure, when a higher-level checkpoint is performed the SCR protocol first performs all lower-level checkpoints (e.g., the length of a level-2 checkpoint, δ_2, would include the time required to first perform a level-1 checkpoint).

SCR is somewhat limiting in its use of patterns because it both restricts checkpoints to be taken at discrete intervals and mandates that patterns be identical throughout an application's execution. It is not known (and hard to prove) if under these assumptions it is possible to produce checkpoint intervals that are truly optimal for multilevel checkpointing protocols with an arbitrary number of

levels. Nevertheless, these assumptions are important from a practical standpoint when considering the model's implementation in production HPC systems. SCR has been highly influential in the development of multilevel checkpointing and most multilevel checkpointing models follow these assumptions.

This model was developed as part of the SCR library and implemented as a three-level checkpointing protocol on a BlueGene/L system. The protocol stores its lowest-level checkpoints in the node's local RAM, second level checkpoints are stored across partner nodes using XOR encoding, and last level checkpoints are stored in the PFS. The authors present the effectiveness of their model by analyzing its effect on application *efficiency*, which they define as the ratio between the minimum run time required to complete a portion of work (with no overhead from checkpointing or failures) and the expected run time to complete that same portion accounting for checkpoint and recovery overheads as predicted by the model. We use this same performance metric for the analyses we perform in this chapter.

2) *The Fault Tolerance Interface* (FTI) [57] extends the three-level SCR checkpointing protocol defined in [47] with work from [60] and [116]. The FTI protocol incorporates Reed-Solomon encoding to provide an additional checkpointing level that is more reliable (and more computationally costly) than SCR's level-two XOR-encoded checkpoint between partner nodes, but less reliable than a checkpoint to the PFS, and is therefore categorized as the third checkpoint level out of four. FTI uses the scalable SCR Markov model from [47] for estimating application efficiency and determining optimal checkpoint intervals.

6.2.3 Multilevel Checkpoint Interval Optimization

Progress has been made in recent years toward optimizing checkpoint intervals in multilevel checkpointing systems. One key requirement for all techniques when determining optimal checkpoint intervals is having an accurate model of the application's execution behavior under the influence of overhead from failures and resilience.

The work that we consider by Moody et al. in [47] utilizes their Markov model to perform a brute-force search of all possible checkpoint intervals to determine the best efficiency when optimizing SCR. Because this Markov model is frequently used in other work (e.g., the FTI protocol), this same model is used for optimizing those implementations of multilevel checkpointing. While an optimal checkpoint pattern for an L-level checkpoint protocol is not guaranteed to have identical sub-patterns, optimizing a multilevel checkpoint protocol with identical sub-patterns is of practical importance to HPC system design, as mentioned earlier.

There have been two recently proposed optimization techniques in [62] and [81] that use novel approaches for determining optimal checkpoint intervals and we consider them in this work. An optimization of pattern-based multilevel checkpointing is considered by Benoit et al. in [81]. Work by Di et al. in [62] includes both pattern-based and *interval-based* optimization techniques. An interval-based multilevel checkpointing protocol allows the interval of time between checkpoints at each level to be independent of the inter-checkpoint time at other levels (unlike pattern-based protocols where higher-level checkpoints have intervals that are multiples of lower-level checkpoint intervals). Their work indicates that the interval-based optimization can perform better than pattern-based optimization. However, as noted in [81], challenges exist with practical implementations of interval-based optimization techniques that might limit their use in real systems. In particular, determining how the system should behave if checkpoints of different levels are scheduled to occur simultaneously. Furthermore, to the best of our knowledge no multilevel checkpointing protocols exist that have been designed to accommodate anything other than pattern-based multilevel checkpointing. We therefore only consider the offline pattern-based optimization technique from [62].

In our work we do not assume that checkpoint and restart events are free from failures, as is common in several other state-of-the-art models (including [62] and [81]). Indeed, as we will show in Section 6.4, the assumption that these events are free from failures negatively impacts the prediction accuracy of their models. Our work also considers the effect that application execution time has on interval optimization, which is not considered by the work in [47] and [81].

6.3 Multilevel Checkpointing Model

6.3.1 Overview

In this section, we discuss our proposed multilevel checkpointing model. We first present the set of equations used to estimate the execution time of an application employing multilevel checkpointing and we end the section with a brief description of the model's use for determining optimal checkpoint intervals.

6.3.2 Execution Time Prediction Model

Our model is a set of continuous equations that estimate the execution of an application that employs a pattern-based multilevel checkpointing protocol following the behavior of SCR described in [47]. We model the equation's prediction of application execution time hierarchically, which allows for the expected execution time of each lower level checkpoint interval (including application computation as well as all overhead associated with checkpointing and failures) to be utilized in the computation of higher level checkpoint intervals.

We define the baseline execution time of the application, T_B, as the time to execute the application without overhead from resilience or failures. In addition to T_B, the expected execution time of the application when using multilevel checkpointing, T_{ML}, is equal to the sum of the application's time spent executing each type of event associated with checkpointing and failures (each variable is an L-dimensional vector):

- successful level i checkpoints, T_{δ_i};
- level i checkpoints that have failed (failed checkpoints), $T_{\delta_i'}$;
- successful level i restarts, T_{R_i};
- level i restarts that have failed (failed restarts), $T_{R_i'}$;
- re-computation of work lost to a failure occurring during a level i computation interval, $T_{W_{\tau_i}}$;
- re-computation of work lost to a failure occurring during a level i checkpoint, $T_{W_{\delta_i}}$.

Each term's expected value is estimated as the expected number of occurrences of the event multiplied by the expected time of the event. For a chosen probability density function (PDF) used to model the probability of a failure occurring, we calculate the expected execution time for any event in which a failure has occurred as the expected value of the PDF with its domain truncated to the duration of that event and normalized to the probability of a failure occurring during the event's duration (a truncated distribution).

We assume that failures follow an exponential distribution as assumed in most prior work in the area [47] [73] [62] [81]. This makes the probability of a failure occurring during any given interval of time t for some failure rate X equal to

$$P(t, X) = 1 - e^{-Xt} \ . \tag{6.1}$$

In contrast to the expected value of the general PDF, which is calculated over the entire domain ($[0, \infty)$ in this case), the truncated domain is calculated over $[0, t]$ and makes the expected value of the truncated PDF for the event when using an exponential distribution equal to

$$E(t, X) = \frac{\frac{1}{X} - e^{-Xt}(\frac{1}{X} + t)}{P(t, X)} \ . \tag{6.2}$$

We define the failure rates associated with each checkpoint level i as λ_i. The system failure rate, λ, is equal to the sum of each λ_i and this value is also equal to the inverse of the system's MTBF. We define a failure's *severity class* to indicate the level of checkpoint required to restart the system after the failure occurs. Each failure severity class variable, S_i, indicates the probability of experiencing a failure of severity i, and is equal to the ratio of λ_i to λ. This also means that for a failure severity, $i = 1, ..., L$, the corresponding failure rate, λ_i, is the product of the system failure rate, λ, and the probability of a failure at that severity, S_i, making the failure rate $\lambda_i = S_i\lambda$.

The multilevel checkpoint protocol from [47] that we are modeling defines each higher-level checkpoint to occur after some number of occurrences of the previous level of checkpoint (e.g., an L_2 checkpoint to a partner node's RAM occurs after some number of instances of L_1 checkpoints

to the node's local RAM). These values define the number of L_{i-1} checkpoints that must occur before each L_i checkpoint is taken, and are the set of $L-1$ integer decision variables $N_1, ..., N_{L-1}$ used for optimizing the equation. Thus, the variable N_i is the number of level i checkpoints before a level $i+1$ checkpoint. The last decision variable is the computation interval, a real number that we define as τ_0. This set of decision variables defines the amount of computational progress made by the application once each level i checkpoint has been completed. The variable N_L, while not a decision variable, represents the number of level L checkpoints that will occur during the execution of the entire application and is defined based on the amount of computational progress that is made for each level L checkpoint interval, i.e.,

$$N_L = \frac{T_B}{\tau_0 \prod_{i=1}^{L-1}(N_i+1)} \ . \tag{6.3}$$

An advantage to estimating total execution time hierarchically is that execution time predictions for lower level computation intervals do not need to account for the occurrence of higher severity failures when predicting the duration of each application execution event. Level i events only need to account for failures of levels less than or equal to i making the failure rate in most of the terms equal to $\sum_{j=1}^{i} \lambda_j$ and we denote this value as λ_c.

The amount of total time spent between each level $i+1$ checkpoint (including overhead from resilience and failures) is referred to as the level i execution interval. Each higher level execution interval, τ_{i+1}, is calculated as

$$\tau_{i+1} = \tau_i(N_i+1) + T_{\delta_i} + T_{\delta_i'} + T_{R_i} + T_{R_i'} + T_{W_{\tau_i}} + T_{W_{\delta_i}} \tag{6.4}$$

with the application's total expected execution time when using multilevel checkpointing $T_{ML} = \tau_{L+1}$. The remainder of this section discusses how the terms in Eqn. 6.4 are obtained.

For each level $i+1$, the term $\tau_i(N_i+1)$ represents the total time of lower level intervals, τ_i, occurring in τ_{i+1}. We define the total number of failures during τ_i as γ_i and estimate this value using a negative binomial distribution to obtain

$$\gamma_i = \frac{P(\tau_i, \lambda_i)}{1 - P(\tau_i, \lambda_i)} \ . \tag{6.5}$$

Note that because lower level intervals have already accounted for lower severity failures during computation, the failure rate used to calculate both γ_i and the expected value no longer needs to be summed and becomes just λ_i. This makes $T_{W_{\tau_i}}$ equal to the expected number of failures multiplied by the expected time of those failures multiplied by the number of τ_i intervals occurring during τ_{i+1}, i.e.,

$$T_{W_{\tau_i}} = \gamma_i E(\tau_i, \lambda_i)(N_i + 1) \ . \tag{6.6}$$

The total time for successful checkpoints at each level is defined as

$$T_{\delta_i} = N_i \delta_i \ . \tag{6.7}$$

The estimator for the expected number of failures that occur during each level i checkpoint, α_i, can be modeled using a negative binomial distribution and is calculated using

$$\alpha_i = \frac{P(\delta_i, \lambda_c) N_i}{1 - P(\delta_i, \lambda_c)} \ , \tag{6.8}$$

so that the expected time that is wasted due to failed checkpoints is given by

$$T_{\delta_i'} = \alpha_i E(\delta_i, \lambda_c) \ . \tag{6.9}$$

The additional overhead associated with the execution progress that is lost due to the failed checkpoint is equal to the number of failed checkpoints at this level (α_i) multiplied by the sum of the entire failed interval plus the expected value of the overhead associated with that failed interval level multiplied by the percent of checkpoints of that level (S_k) for each level up to and including i. This is expressed as

$$T_{W_{\delta_i}} = \alpha_i \sum_{k=1}^{i} (\tau_k + \gamma_k E(\tau_k, \lambda_k)) S_k \ . \tag{6.10}$$

The estimator for the expected number of successful restarts is calculated from the total number of level i severity failures that occur in τ_{i+1} during computation and checkpointing. We call this value β_i and it is summed using α_i and γ_i as

$$\beta_i = S_i\alpha_i + \gamma_i(S_i\alpha_i + N_i + 1) \ . \tag{6.11}$$

The expected number of failures that occur during restarts of level i, ζ_i, is once again modeled with a negative binomial distribution as

$$\zeta_i = \frac{P(R_i, \lambda_c)\beta_i}{1 - P(R_i, \lambda_c)} \ . \tag{6.12}$$

The total expected time that the application spends for successful restarts is equal to

$$T_{R_i} = \beta_i R_i \ , \tag{6.13}$$

and the total time that the application spends for failed restarts is equal to

$$T_{R_i'} = \zeta_i E(R_i, \lambda_c) \ . \tag{6.14}$$

6.3.3 Checkpoint Interval Optimization

Optimizing Eqn. 6.4 by selecting decision variables that minimizing execution time is accomplished by evaluating the equation's execution time at every point in a bounded region of the solution space and determining which decision variable values provide the shortest execution time. This sweep of decision variable values is bounded by the interval $(0, T_B)$ for τ_0, and also bounded such that the product of $N_1, ..., N_{L-1}$ with N_L and τ_0 is greater than zero and less than the application's baseline execution time, i.e., $0 < \tau_0\left(\prod_{i=1}^{L}(N_i + 1)\right) \leq T_B$. We can guarantee a global optimum is found when bounding the solution space in this way because decision variable values outside of this region produce infinitely large execution times when the system's MTBF is less

Table 6.1: Test Systems Examined in Prior Work

system	*paper*	*num. levels*	*MTBF (min.)*	*failure distribution (probability per level)*	*c/r time (min. per level)*	*base. exec. time (min.)*
M	[47]	3	6944.45	$(0.083, 0.75, 0.167)$	$(0.008, 0.075, 17.53)$	1440.0
B	[117]	4	333.33	$(0.56, 0.28, 0.14, 0.03)$	$(0.17, 0.5, 0.83, 2.5)$	1440.0
D1	[62]	2	51.42	$(0.857, 0.143)$	$(0.333, 0.833)$	1440.0
D2	[62]	2	24.0	$(0.833, 0.167)$	$(0.333, 0.833)$	1440.0
D3	[62]	2	12.0	$(0.833, 0.167)$	$(0.167, 0.667)$	1440.0
D4	[62]	2	6.0	$(0.833, 0.167)$	$(0.167, 0.667)$	1440.0
D5	[62]	2	12.0	$(0.833, 0.167)$	$(0.333, 1.67)$	1440.0
D6	[62]	2	6.0	$(0.833, 0.167)$	$(0.167, 1.67)$	720.0
D7	[62]	2	4.0	$(0.833, 0.167)$	$(0.667, 3.33)$	360.0
D8	[62]	2	3.13	$(0.870, 0.130)$	$(0.833, 5.0)$	360.0
D9	[62]	2	3.13	$(0.870, 0.130)$	$(0.833, 5.0)$	180.0

than the application's baseline execution time as is the case here. Efficiency is then calculated by dividing the application's baseline execution time by the calculated expected execution time.

6.4 Simulation Studies

6.4.1 Overview

We performed a set of simulation studies to both validate the behavior of our equation-based multilevel checkpointing execution time prediction model and to provide a comparison of its performance with the performance of prior work in the field when executing applications at extreme scales. In addition, we also demonstrate the effects that both higher failure rates and longer checkpoint/restart times have on application efficiency and model prediction accuracy. We identify the importance of modeling failures during checkpoint and restart events in systems with high failure rates. We also show the advantage that consideration of application execution time has for the interval optimization of short-running applications.

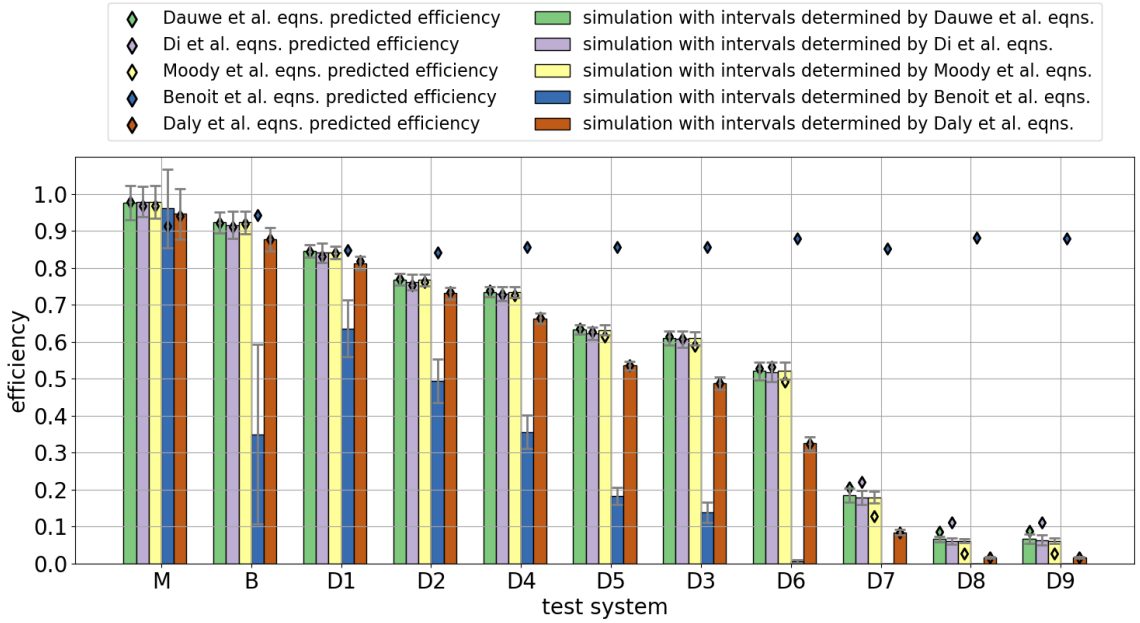

Figure 6.2: Performance of the *multilevel checkpoint* and *traditional checkpoint/restart* checkpoint interval optimization techniques executing on the test systems from Table 6.1. Bars in the figure indicate the average of 200 simulation trials with randomly occurring failures. Diamonds in the figure indicate each technique's prediction of the simulated performance. Standard deviations are shown for each bar.

6.4.2 HPC System Simulator

Because exascale systems do not exist, we turn to simulation to analyze the performance of each multilevel checkpointing model. The simulations performed in this section use an event-based simulator that models all events that occur throughout an application's execution in a system operating with the uncertainty of system failures. We provide a more detailed explanation of our simulator's operation in our prior work [43].

6.4.3 Performance on Prior Work Test Systems

This section provides a comparison between our technique for multilevel checkpoint interval optimization that we described in Section 6.3 and some of the techniques by others discussed in Section 6.2.3. Specifically, in addition to our own technique we consider the work by Di et al. from [62], the work by Moody et al. from [47], the work by Benoit et al. from [81], and the classic optimization of single-level checkpointing described by Daly in [73]. We simulate the performance of each of these techniques on systems with varying characteristics that have been defined in prior

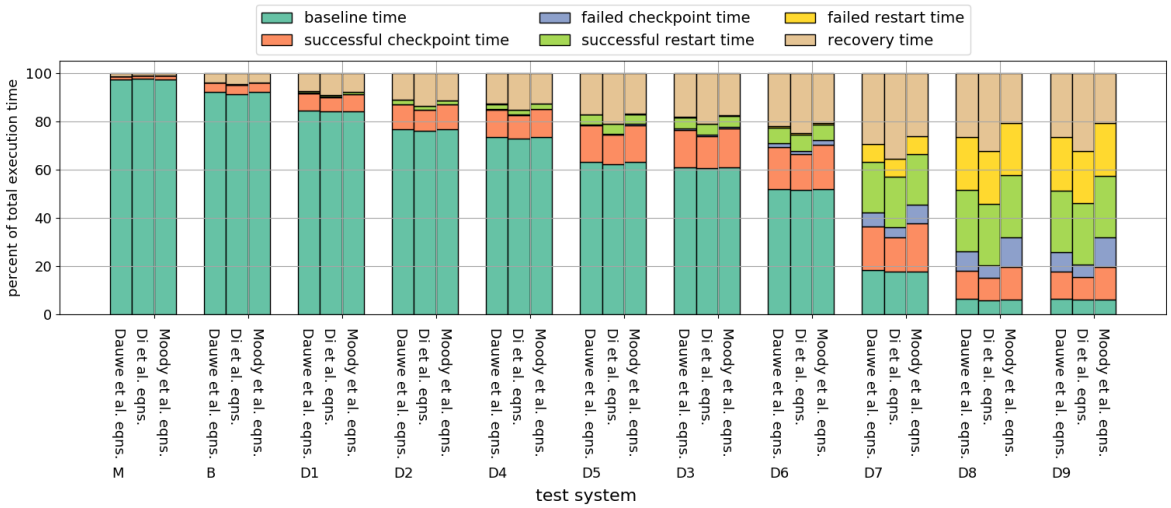

Figure 6.3: Percentage of application execution spent on baseline execution of the application as well as all resilience and failure event related overhead during the application's execution. Each test scenario shown represents the average of 200 trials with randomly occurring failures.

work. Each unique combination is referred to as a *test system* and the details of each is outlined in Table 6.1.

The test systems are organized in order of monotonically increasing difficulty of providing fault resilience to the system. The challenge of providing fault resilience primarily increases through either a decrease in system MTBF (due to higher failure rates) or increasing checkpoint/restart times (particularly to the PFS). However, in addition to differences in failure rates and checkpoint costs, the systems also differ in the number of checkpointing levels supported by each system as well as the distribution of failures for each failure severity class. All values in Table 6.1 are functionally identical to the information provided by each paper indicated in column two, however the values have been converted in format to allow for consistency so that all time values are now in minutes and failures for each severity are expressed as probability distributions. Checkpoint times are assumed to be equal to restart times for each system, as assumed in prior work [47] [62] [81]. In the case of Daly's traditional single-level checkpoint/restart model and the two-level checkpointing model from Di et al., when considering systems that have more available checkpointing levels than the model is able to accommodate, only the highest levels are considered (i.e., traditional

checkpoint/restart only ever uses the highest level-L checkpoints to a PFS and Di et al. uses only level-L and level-$L-1$ checkpoints).

Each of our following experiments simulates the execution of a single large application that employs multilevel checkpointing (with the exception of Daly's equation which uses traditional checkpoint/restart) to mitigate the effects of system-wide failures on application execution. In all cases, the checkpoint intervals of the simulated test case are optimized using the multilevel checkpointing (or checkpoint restart) modeling technique indicated in each figure.

Figure 6.2 shows the performance (efficiency) of the five checkpointing techniques considered (our technique is shown in green) for each of the test systems described in Table 6.1. Bars in the figure represent the average of 200 simulation trials with random failures for each specific setup, with the standard deviations of those trials shown around each bar. The diamonds that are color-coded to each bar in the figure indicate the prediction of the system's efficiency by each technique. Predictions are made based on the system's execution characteristics and the checkpointing intervals determined by each technique with accurate predictions being those located closer to the tops of each bar.

The first trend highlighted in Figure 6.2 is the improved efficiency that a multilevel checkpointing approach can have over traditional checkpoint/restart (Daly). The figure shows how even though Daly's equations for traditional checkpoint/restart are highly accurate at predicting application efficiency (and consequently good at selecting appropriate checkpoint intervals) the traditional checkpoint/restart protocol does not perform as well as the multilevel checkpointing protocol when optimized by either Dauwe et al., Di et al., or Moody et al. Daly's checkpoint/restart's efficiency is 50% less than that of multilevel checkpointing in the worst case. In addition to reaffirming the conclusions from prior work, the data for traditional checkpoint/restart in Figure 6.2 also helps to highlight the importance of having an accurate multilevel checkpointing model that is capable of making an appropriate selection of checkpoint intervals. In particular, while the multilevel checkpoint technique by Benoit et al. performs well on test system M, because some of the approximations made by the model have large effects on prediction accuracy, its efficiency on more

challenging test systems is worse than for a well-optimized implementation of traditional checkpointing.

The efficiency predictions of the equations modeled by Benoit et al. in [81] are optimistic because they do not consider the effect of failures during checkpoints or restarts and only consider failures during computation. Consequently, the corresponding computation intervals determined by these equations are excessively long. For all test systems, the length of the computation interval chosen by these techniques is at least $2.5\times$ greater than that of the other multilevel checkpointing techniques. This disparity also increases as the challenge of providing resilience to the system increases and can be seen in Benoit et al.'s model's faster decrease in efficiency in comparison to the other techniques displayed in Figure 6.2. At the same time, the execution time prediction model results (blue diamonds for Benoit et al.) indicates that the chosen intervals have optimistically low execution time predictions resulting in optimistically high efficiency predictions. The sharp drop in efficiency of Benoit et al.'s equations on test system B is due to the decreasing accuracy of their equations as the number of checkpoint levels increase. While the decrease in performance of the other multilevel checkpointing techniques is monotonic and follows the increase in difficulty of providing resilience to each system, the Benoit equations drop sharply from system M (with three checkpoint levels) to system B (with four checkpoint levels) and subsequently increase in efficiency in system D1 (with two checkpoint levels).

Simulated performance of multilevel checkpointing when optimized by either Dauwe et al. (our work), Di et al., or Moody et al. is similar across all test systems, however the prediction accuracy for each of these techniques can be seen to decrease slightly as test system difficulty increases. The causes of this will be discussed in detail in the upcoming sections.

6.4.4 Failures During Checkpoints and Restarts

Figure 6.3 shows the breakdown of how application time is spent when executing an application in a failure-prone environment and employing a resilience protocol to mitigate the effects of failures. Each test scenario shown represents the average of 200 trials with randomly occurring

failures. The figure shows the percentage of time spent on baseline execution time as well as time lost to overhead from each of the resilience and failure-related events discussed in Section 6.3. Data is shown for the same test systems presented in Table 6.1, but we limit our analysis to the three best-performing techniques from Figure 6.2.

It is evident from the data in Figure 6.3 that as the difficulty in providing resilience to systems increases, applications lose increasing amounts of time to failed checkpoints and restarts with at least 30% of application time spent in these areas in the most extreme cases. Furthermore, this increase is non-linear (in fact the increase follows the α_i and ζ_i variables of Eqns. 6.8 and 6.12) and affects the more extreme *D7, D8,* and *D9* systems to a greater degree than the other systems shown in the figure. Results for test systems *D8* and *D9* look almost identical because these test systems are identical in all respects except their baseline execution time.

The rapid increase in the number of failed checkpoint and restart events is caused by the system MTBF approaching (or even becoming less than) the length of time required to checkpoint or restart to the PFS. While optimal intervals between checkpoints can be adjusted to compensate for decreasing MTBF, checkpoint and restart times cannot, and this can force the system to retry checkpoint and restart events several times before they can complete successfully. Because extreme-scale systems experience increased amounts of failures during checkpoint and restart events, consideration of these events is necessary for accurate execution time modeling.

6.4.5 Performance at Extreme-Scale System Difficulty

For the studies discussed in this section we analyze the four-level checkpointing system from [117] (the system defined in Table 6.1 from Section 6.4.3 as system B) under a variety of exascale-like execution scenarios. Specifically, we scale both system MTBF and the length of time required by the system for checkpointing to or restarting from the PFS. It has been noted in [47] that exascale systems are likely to experience failures with an MTBF between $3 - 26$ minutes and therefore we explore five system MTBF values in this range.

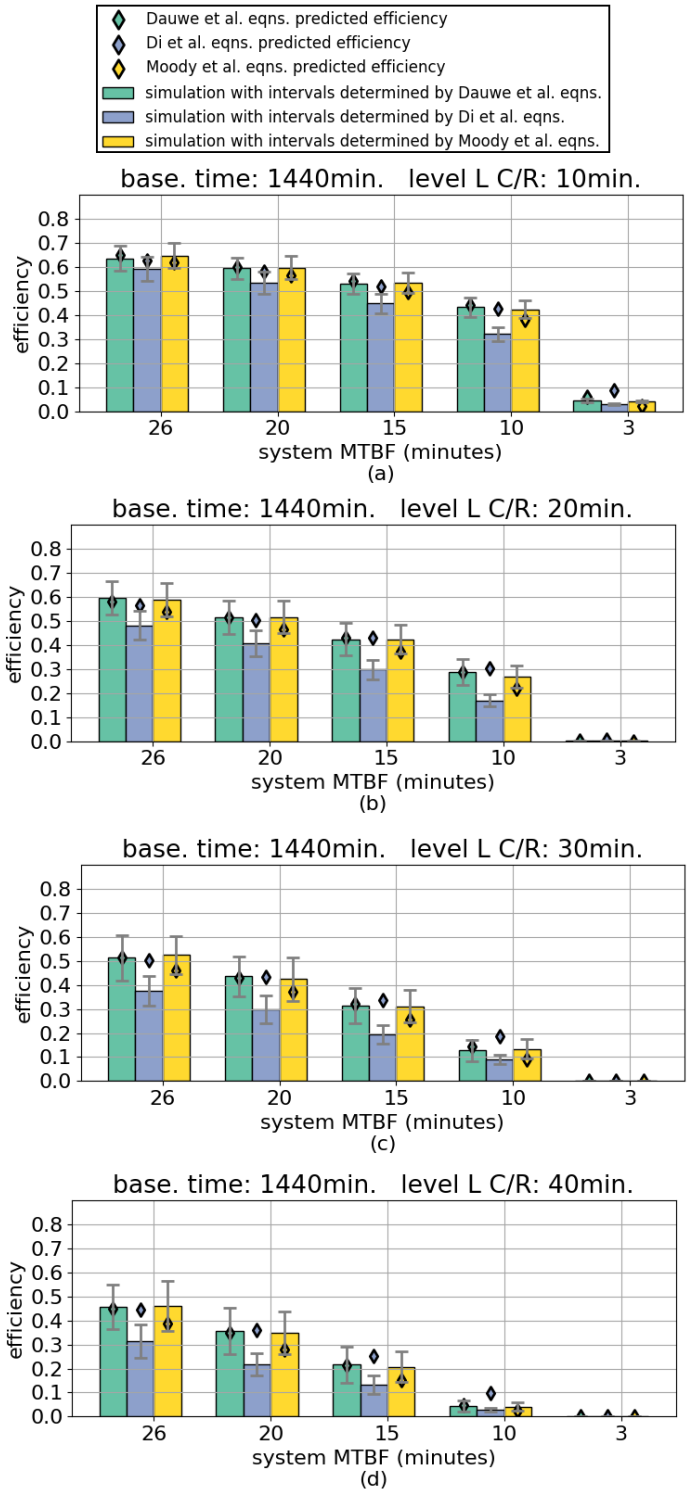

Figure 6.4: The execution of a 1440 minute application under a variety of execution scenarios with level-L checkpoint and restart times of **(a)** 10 minutes, **(b)** 20 minutes, **(c)** 30 minutes and **(d)** 40 minutes. Bars in the figure indicate the average of 200 simulation trials with randomly occurring failures. Diamonds in the figure indicate each technique's prediction of the simulated performance. Standard deviations are shown for each bar.

Data from [46] suggests that the improvements to network speed will increase at a similar rate as the data required to checkpoint larger applications and consequently checkpoint times to a PFS will likely remain constant as system sizes increase. From [46], we assume that checkpoint times to a PFS for an exascale-sized application will likely be between $20 - 40$ minutes. We examine four values for the time costs associated with checkpointing and restarting to the PFS. These values range from 10 minutes, likely to be a conservative estimate, to 40 minutes. We only consider scaling of the level L checkpoint/restart time because (as noted in Section 6.3) checkpoint/restart levels less than L spread checkpoint data across system resources. Lower level checkpoints are therefore less affected by application size. Checkpoint and restart times for lower level checkpoints remain the same as those values listed for test system B in Table 6.1.

The results of this study are shown in Figure 6.4. The difficulty of providing resilience to each test scenario increases both across each x-axis (as system MTBF decreases) and across sections (a)-(d) of the figure (as the time penalty for checkpoints/restarts to a PFS increases). Bars in the figure indicate the average of 200 simulation trials with randomly occurring failures. Diamonds in the figure indicate each technique's prediction of the simulated performance. Standard deviations are shown for each bar. It is evident that multilevel checkpointing's ability to provide resilience to an exascale HPC system will be more impacted by system MTBF than increased checkpoint/restart times. Decreasing system MTBF from 26 minutes to 3 minutes can decrease efficiency from over 60% to less than 1% in some cases but increasing checkpoint/restart times from 10 minutes to 40 minutes produces a maximum decrease of about 40% efficiency. It is also clear that some of these possible exascale applications push the limit of the resilience that multilevel checkpointing can provide. The most extreme case of a 3 minute MTBF produces less than 1% efficiency for checkpoint/restart lengths greater than 10 minutes. Even a system with a 15 minute MTBF produces less than 50% efficiency for checkpoint/restart lengths greater than 10 minutes.

The results in Figure 6.4 also clearly show the negative effect of Di et al.'s constraint of only considering two checkpoint levels. For all execution scenarios that produce more than 1% efficiency, the performance of checkpoint interval optimizations from Di et al. is noticeably worse

than that of Dauwe et al. and Moody et al., which utilize all four checkpoint levels. It should be noted that the poorer performance of Di et al.'s equations is primarily due to its constraint of two checkpoint levels. We deduce this by noting that the simulated performance of all three techniques are close to equal for the two-level checkpointing test systems from Figure 6.2 indicating that the decreased performance of Di et al.'s equations in Figure 6.4 is caused by its restriction to only using two checkpoint levels.

Figure 6.4 also indicates that the prediction accuracy of all optimization techniques decrease with both decreasing system MTBF and increasing checkpoint/restart times. Prediction accuracy of each technique is further discussed in Section 6.4.7.

6.4.6 Consideration of Application Execution Time

One advantage that both our equations and those of Di et al. have over those of Moody et al. is our consideration of application execution time. Because even for the most pessimistic MTBF values the highest severity system failures are still infrequent, under execution scenarios with high level-L checkpoint/restart times it is more efficient (on average) for shorter applications not to take time-consuming level-L checkpoints and instead risk a total application restart. As our equations calculate the expected execution time, this effect is identified by our equations and those of Di et al. Consequently, when selecting checkpoint intervals for those scenarios our equations (Dauwe et al.) and those of Di et al. correctly select intervals that are optimized not to include level-L checkpoints, while the equations from Moody et al. select interval values that are appropriate only for longer running applications.

We demonstrate this in Figure 6.5, which shows the same set of execution scenarios discussed in Figures 6.4a and 6.4b but with a shorter application that executes for only 30 minutes. Bars in the figure now indicate the average of 400 simulation trials with randomly occurring failures. Diamonds in the figure indicate each technique's prediction of the simulated performance, with standard deviations shown for each bar. Because the application's execution time is less than the mean time between level-L severity failures, our equations and those of Di et al. do not take

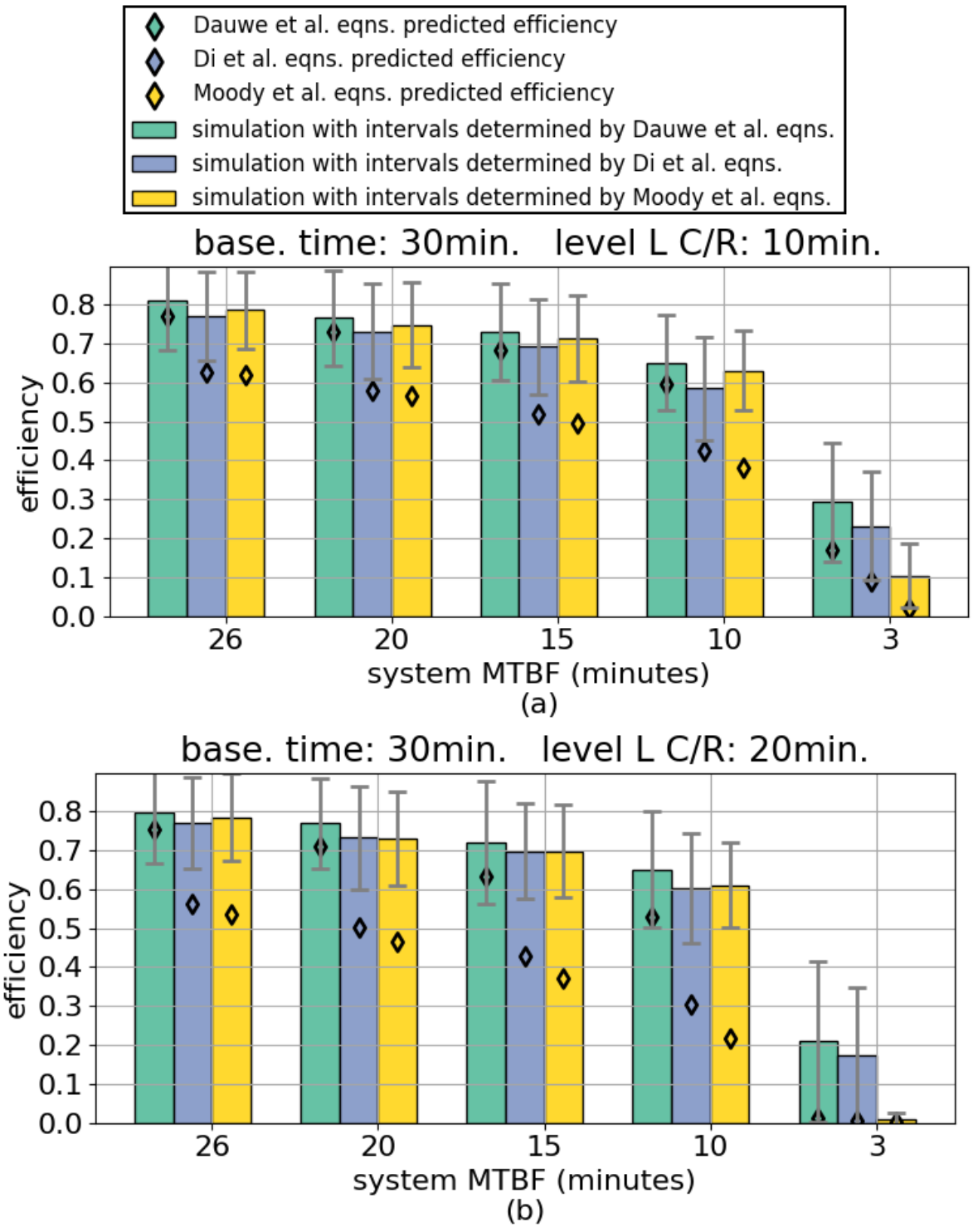

Figure 6.5: The execution of a 30 minute application under a variety of execution scenarios with level-L checkpoint and restart times of **(a)** 10 minutes and **(b)** 20 minutes. Bars in the figure indicate the average of 400 trials with randomly occurring failures. Diamonds in the figure indicate each technique's prediction of the simulated performance. Standard deviations are shown for each bar.

level-L checkpoints in any of the experiments in Figure 6.5. Here we have shown the execution of a 30 minute application as an extreme example but we have found these same results occur to a lesser extent for an application that is 120 minutes in length. We expect that this result is present for all extreme-size applications that have a baseline execution time that is shorter than the mean time between the highest severity failures. Benefits to short applications from not including level L checkpoints increase with both the increase in checkpoint/restart lengths and decreasing MTBF values, and provide as much as a 20% efficiency improvement in some cases. While the advantage gained in other cases may be lower, we have determined with 95% confidence that all improvements in the figure are statistically significant.

Because this effect is only beneficial on average, one difference between the results in Figure 6.5 and those of the longer application in Figure 6.4 can be seen when comparing standard deviations between techniques. While standard deviations are nearly identical between techniques for every execution scenario tested in Figure 6.4, the standard deviations shown in Figure 6.5 for the execution scenarios that have had their level-L checkpoints excluded by our equations now have a slightly greater variation in execution time than the results for the equations from Moody et al. that still perform a level-L checkpoint.

6.4.7 Model Prediction Accuracy

Figure 6.6 shows the prediction accuracy of the 20 system scenarios shown in Figure 6.4 in terms of each model's prediction of application efficiency minus the efficiency value determined through simulation. The system scenarios in the figure have been sorted according to increasing magnitude of error of the results for Moody et al. and show each optimization technique's deviation from the ideal error of zero. Here, we have only shown results for the long duration applications discussed in Section 6.4.5 because they provide the clearest depiction of model prediction performance. However, for shorter applications, our equations still has the best prediction accuracy in most cases.

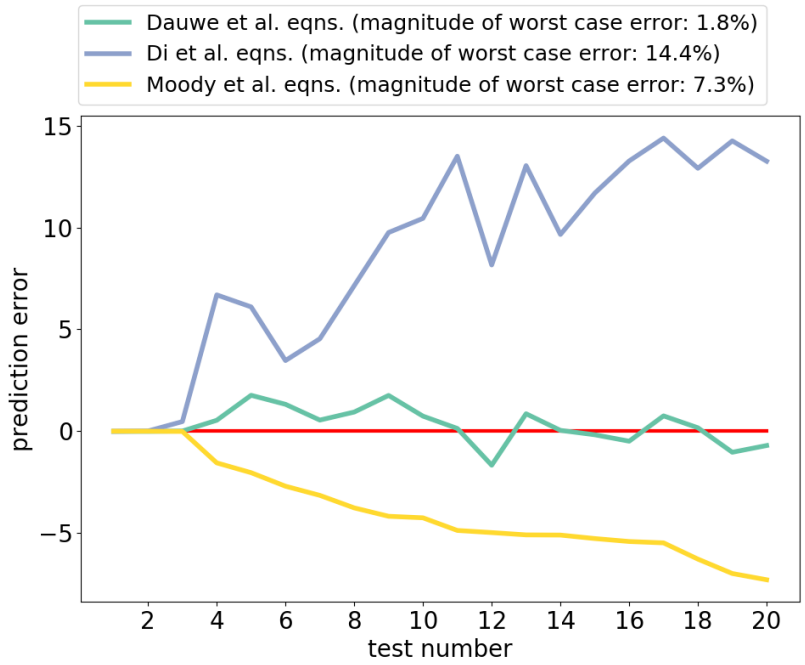

Figure 6.6: Prediction error for the 20 application execution scenarios shown in Figure 6.4. The prediction error shown is simply the difference between each multilevel checkpoint model's prediction of efficiency and the corresponding efficiency determined through simulation. The tests are ordered by increasing magnitude of error of the Moody et al. model. The red line indicates a value of zero (the target error).

The results in Figure 6.6 demonstrate the benefit that our multilevel checkpointing model provides in terms of prediction accuracy over both Di et al.'s and Moody et al.'s models. As the test numbers on the x-axis increase, the difficulty in prediction also increases and Moody et al.'s model tends to underestimate application efficiency (by as much as 7.3%) while Di et al.'s model tends to overestimate application efficiency (by as much as 14.4%).

The difference in prediction accuracy between our work and that of Moody et al. [47] and Di et al. [62] are the assumptions made by each set of equations about the system's behavior during failed restarts. Our equations assume that if the system is restarting from a class i severity failure and experiences a second failure of severity less than or equal to i then the system can still be restarted from a subsequent level i checkpoint. The simulations make this assumption for all techniques.

Di overestimates efficiency because it neglects considering the effects of failures during restarts entirely. Specifically, it does not account for the increasing impact of repeated failed restarts discussed in Section 6.4.4 that occur more frequently with both decreased MTBF and increased check-

point/restart times. Di et al. are aware of this limitation in their model, and made a note of its effect on prediction accuracy in [62].

The model by Moody et al. underestimates efficiency because of its pessimistic assumption about failures during restarts. Specifically, if the system is restarting from a level-i severity failure and experiences a second failure of level-i severity then the system needs to subsequently restart from a level $i + 1$ stored checkpoint. This causes an unrealistic escalation of failure levels for extreme-sized systems that experience significant lower severity failures. For example, if one of the 100,000 nodes in the test system B considered here required a restart from local RAM it is unreasonable to assume that a second event of that type occurring on any other node in the system would necessitate any response other than attempting to load the same checkpoint from RAM a second time. This assumption causing escalating failures in conjunction with the presence of rapidly increasing numbers of failures (as discussed in Section 6.4.4) causes Moody et al.'s model to have increased prediction error. Although the effect on system behavior implied by this model assumption would have been present in several of the more extreme test cases that Moody et al. explore with their Markov model in [47], they do not perform any simulations of their model demonstrating the necessity of this assumption in an actual HPC system nor do they discuss this effect in their results.

6.5 Conclusions

With this work we have developed a hierarchical continuous equation-based model for a multilevel checkpointing resilience technique operating with an arbitrary number of checkpoint levels. Through simulation of exascale HPC systems, we have shown the model's ability to accurately predict application execution time in failure-prone environments as well as for determining optimal checkpoint intervals. We have implemented several multilevel checkpointing optimization techniques from the literature and shown the benefit that our technique can provide over these techniques in both model prediction accuracy and determining checkpoint intervals for short duration

applications. In all circumstances, our model either outperforms the models from others' work or it is capable of performing within 1% of their model.

Using our simulation data we have shown that extreme-scale systems experience increasing numbers of failures during checkpoint and restart events. We determined that the increase is caused by decreasing MTBF values approaching the values of increasing checkpoint/restart length times. The increased probability of failure during checkpoint/restart events causes an extremely rapid (non-linear) increase in the amount of time lost to these events at extreme scales, and makes the consideration of these events a necessity for accurate execution time modeling - something that is frequently ignored.

We have also more generally shown some of the limitations of multilevel checkpointing. Simulations of single level checkpoint/restart (using Daly's equation) show that the usefulness of a single-level resilience protocol is limited to petascale-sized systems with MTBF on the order of hours, and indicate that larger systems require more advanced resilience protocols such as multilevel checkpointing. Similarly, our data also suggests the limits of multilevel checkpointing. When utilizing multilevel checkpointing, a system with even a 15 minute MTBF will drop below 50% efficiency for checkpoint/restart lengths greater than 10 minutes. In such cases, regardless of the checkpoint interval optimization technique used, the system will spend less than half its time on useful computation. Given that operating exascale systems are likely to cost tens of millions of dollars a year, this level of resilience is likely to be unacceptable. As system sizes increase further, other strategies may need to be employed to complement (or possibly replace) the multilevel checkpointing protocol.

Chapter 7

Modeling Application Fragmentation and Network Congestion in the presence of HPC Resilience

7.1 Introduction

Studies suggest that energy consumption at exascale may be completely dominated by the costs of data movement between communicating nodes of an application and could become the main performance constraint [118]. In fact, several sources suggest that exascale applications are not only likely to continue to be communication-bound as is frequently the case with today's applications, but are likely to have an increased dependence on communication over today's applications, [69], [119].

Similarly, high performance computing (HPC) resilience also has been predicted to be a serious performance bottleneck for future large-scale systems [43], [77], and [45]. Given that traditional checkpoint and restart forms of resilience rely on highly communication-intensive events, as the mean time between application failures decreases and the duration of checkpoint intervals increases, contemporary protocols for providing HPC systems with resilience to failures will put an ever-increasing burden on HPC communication networks.

Application performance degradation due to congestion in the communication network has been shown to be further exacerbated by sub-optimal placement of applications in the system. Executing with a heterogeneous collection of varying-sized applications in the system has been shown to lead to larger applications being "fragmented" into several smaller disjoint groups that force application traffic to be routed through the congested paths of neighboring applications and further increasing the application's exposure to network congestion [120].

With these interrelated issues for HPC systems in mind, this work makes two contributions toward developing a solution. First, we construct a methodology based on the multi-commodity

maximum flow (MCMF) problem that can be used not only to model HPC network use but also to model:

- network congestion effects from multiple applications executing while co-located in the system;
- multi-granularity fragmentation effects from large applications being split into several smaller groups that are distributed across the system;
- highly communication-intensive checkpoint-based HPC resilience effects.

Second, we discuss some of the effects that resource management has on both application susceptibility and resilience-event susceptibility to performance degradation from network congestion.

The remainder of this work is outlined as follows. Section 7.2 briefly discusses some of the related work in HPC network modeling using MCMF. Section 7.3 presents our modeling methodology and outlines some of our model assumptions. Section 7.4 describes our set of simulation experiments and gives an analysis of their results. We conclude in Section 7.5 and discuss some directions for future work.

7.2 Related Work

There has been a substantial amount of prior work that utilizes MCMF to model communication networks [119], [121], [122], and [123]. Our work differs from these in several ways.

First, we are not just stress-testing each network topology as has been the focus of previous work. Our methodology allows the simulated applications to have communication needs that place specific demands on the network and will delay application execution by a quantifiable amount if they are not met. Specifically, we do not assume infinite demand from each source to sink node in our MCMF problem formulation. If the network is operating at or above the capacity required by the applications currently present in the system then application communication needs may be satisfied without incurring delay to the application.

Second, we are measuring the effects of both intra- and inter-application network contention by allowing more than a single application to exist in the HPC system at the same time. Additionally, we are also acknowledging and modeling the existence of application fragmentation and the effects that this has on application network contention.

Last, we are modeling (and indeed focus on highlighting) the presence and effects of the highly communication-intensive HPC resilience events (e.g., application checkpoints) that will necessarily occur quite frequently in future extreme-scale HPC systems. We show how application communication both affects and is affected by these resilience events utilizing HPC communication networks.

7.3 Modeling Methodology

7.3.1 Overview

We employ simulation to investigate the behavior of multiple applications executing co-located in the same HPC network. We define a "system state" as the communication demand required between all nodes in the HPC network. A "communication demand" is defined as the average rate of data transfer (in GB/s) needing to be sent from some particular node to another and is assumed to remain constant throughout the system's computation in any given system state. The ability for the network to meet application communication demands is dependent on the communication requirements of all applications executing in the system and consequently changes with the system state as either the set of applications executing in the system changes or the communication requirements of any particular application changes. A particular application's communication requirements may change over time based on the application's execution requirements or because of the occurrence of resilience-related checkpoint or restart events. Note that any given system node is likely to have many communication demands.

Our work models the system's instantaneous network performance for a given system state (i.e., the network performance that each application is expect to achieve for the duration of time that the system remains in that state). It is assumed that in order for the application as a whole to

progress toward completion all communication dependencies need to be satisfied for each node. If some executing node is only able to receive a portion of the data that it needs to continue execution it necessarily experiences a delay. As a simplification, we assume that because the execution of all nodes are interrelated a delay in one node's execution will result in a delay for the application as a whole. Because we also assume that applications are likely to be communication-bound, our definition of "application performance" implies that the percentage of an application's performance as a whole is equal to the maximum percentage of communication demand that can be achieved by minimum-performing node.

As an example, take an application that executes on three nodes, *node a*, *node b*, and *node c*. Say the application's execution is communication-bound and that ideally *node b* receives data from *node a* at a rate of 1 GB/s and that *node b* receive data from *node c* at a rate of 2 GB/s. If the system state is such that *node c* is able to send data at its full demand requirement of 2 GB/s but that *node a* is only able to send data at half the rate required by the node's communication demand (i.e. its performance is 0.5) then the application's performance as a whole will necessarily be 0.5. Even though the network has enough bandwidth available for *node c* to send data at a rate of 2 GB/s, *node b* cannot make use of the data without receiving all of the data that it requires from *node a*. Further, the expected delay to the application while executing in this non-ideal system state can be estimated by dividing the time that the application spends in this system state by its calculated performance value while in this system state.

7.3.2 Network Model

Network Topology

Prior work suggests that the dragonfly network topology is the most likely candidate for future exascale systems [124]. We therefore use a variant of this network topology for testing the effects of network congestion on application and resilience protocol performance. Specifically, the Cray XC30 dragonfly topology utilizing the Cray Aries network interconnect is used for simulations [125].

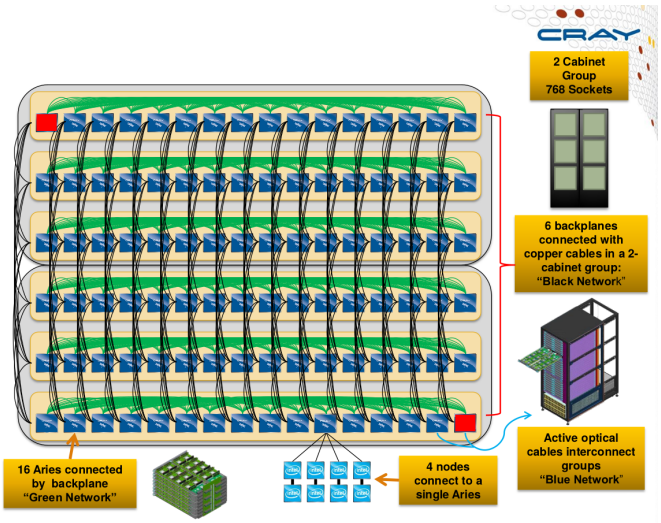

Figure 7.1: Overview of the interconnection of the Cray XC30 network. Dark blue tiles indicate system blades composed of eight compute nodes (compute nodes indicated by the light-blue tiles at the bottom of the figure). Red tiles indicate blades used as cabinet service nodes with each cabinet having two service nodes. Green links indicate all-to-all backplane link connections present between blades in a chassis (with chassis indicated by the tan boxes). Black links indicate the connections existing between chassis. This figure has been adapted from [1].

Figure 7.1 shows the interconnection of Cray XC30 dragonfly topology. The topology is organized so that each Aries interconnect services eight multicore processors defined as a "blade" (depicted in the figure as dark blue squares for compute blades and red squares for blades designated as "service nodes" that forward data to the PFS). Groups of sixteen blades are combined to form a "chassis," (indicated by the tan-colored bounding boxes in the figure) and six chassis compose a "two-cabinet group." Each blade in a chassis is fully connected to every other blade through 14 GB/s backplane links (fifteen links per blade, shown in green in the figure). Blades in each chassis are then also fully connected to their equivalent blade in every other chassis in the cabinet through 14 GB/s copper links (i.e., blade number one of chassis zero is fully connected to blade number one of every other chassis in the cabinet, this is depicted by the black connections in the figure). Cabinets are then connected to each other through 12.5 GB/s optical links in a similar fashion to cabinet chassis (i.e., equivalent blades are fully connected between cabinets).

Network Traffic Flow

We model the HPC system's network use as a multi-commodity maximum-flow (MCMF) problem [126]. When modeling the network as a MCMF problem the HPC network architecture is represented as a bidirectional graph with network interconnects defined as nodes of the graph and links between interconnects defined as the edges between graph nodes. Bandwidth limits for each link in the HPC network are defined as capacity constraints for edges of the graph. The resulting graph represents the physical constraints of the network being modeled.

Once the HPC network architecture has been expressed as a graph, applications are scheduled to nodes in the HPC system. Each node-to-node communication demand required by the application's communication pattern is defined as a separate commodity (and subsequently as separate sets of constraints) in a multi-commodity flow linear program. In accordance with the assumptions we defined in Section for our network model, for our problem formulation we also define the an additional constraint that for each application executing in the system the percentage of communication demand satisfied must be equal for each demand in the application, i.e., the maximum percentage of communication demand that can be achieved by minimum-performing node. These commodity constraints are combined with the network graph's physical constraints to define the entire set of constraints for the linear programming problem. The objective function of MCMF problem is then to maximize the percentage of demand that can be satisfied for all applications in the system.

Variations of MCMF have been used to model network traffic for nearly two decades [121]. Modeling the HPC system's communication network in this way provides the best case traffic flow, i.e., least possible congestion for the application. If the system modeled with MCMF experiences intra- and inter-application congestion that both affects and is affected by HPC resilience events for this scenario of best-case traffic flow then it can be expected that this effect must exist to an even greater degree in a real-world network traffic pattern. We therefore give each of our test cases the best possible opportunity for performance and show the magnitude of these negative effects of network congestion and highly communication-dependent resilience events.

For all simulation experiments, we also assume that the application's ability to communicate is the only constrained resource and that applications execute with an unlimited amount of memory and are continuously producing communication messages at a constant rate. This assumption allows our model to directly examine the effects of network congestion in isolation of other system factors that might further decrease application performance.

7.3.3 Application Communication Pattern

Because our simulation studies need to model an application with an arbitrarily scalable communication pattern, we turn to designing a synthetic benchmark with parameterizable communication characteristics. Our synthetic application is based closely on the "BigSort" application from CORAL's HPC benchmark suite [127]. The application operates by having each executing thread sort a large number of uniformly random integer values into buckets to be distributed to other executing threads throughout the system. This application consequently exhibits an "all-to-all" communication pattern with a parameterizable amount of communication intensity (adjustable based on the number of integers needing to be passed between nodes of the system).

Modeling applications in this way allows for the simultaneous execution of an arbitrary number of variously-sized applications to be modeled in the system (provided there are enough compute nodes available). However, for this study we examine only two large applications each requiring one full cabinet's worth of compute blades.

7.3.4 Parallel File System

Access to the parallel file system (PFS) for each compute node is handled by blades specially designated as "service nodes" (indicated as red squares in Figure 7.1). Despite being denoted as a service "node" by [125], each service node actually occupies a full blade of the system. Each cabinet has two blades dedicated to acting as service nodes allowing access to or from the PFS for the surrounding compute nodes of the system. For each simulation, service nodes are positioned at opposite cabinet corners of the system.

We assume that the PFS is buffered well enough that it is capable of receiving and storing (or accessing and sending) data at a rate equal to that of the maximum that is allowed by the service node's connection to the rest of the network, i.e., the bottleneck for using the PFS will be in sending data through the communication network and not in the ability of the PFS to store the data. This assumption allows the focus of the simulations to be on the limitations of the communication network without additional overhead present in the PFS. Therefore the system PFS is not modeled directly, but instead modeled as bandwidth demand requirements on the system's service nodes.

7.3.5 System Resilience

One focus of this work is to examine both the impact that resilience events have on network congestion as well as the impact that network congestion has on the performance of each resilience event. This analysis focuses on examining the aspects of the multilevel checkpoint protocol discussed in [47] that utilize an HPC communication network. Specifically, the analysis focuses on checkpoints and restarts that are saved to either a partner node's RAM or to a PFS (respectively denoted as a "level 2" and "level 3" checkpoints in [47]).

Because "level 2" checkpoints need to be capable of recovering from multiple node failures in a cabinet, we assume that checkpoints and restarts that are stored in a partner node's RAM will need to choose a partner that is located in a separate cabinet. Level three checkpoints are stored in the PFS and therefore require that computation nodes performing a checkpoint interact with system service nodes to store data.

The quantity of data that needs to be sent during a checkpoint can easily consume more bandwidth than is instantaneously available to the node. This effectively means that during checkpoint and restart events applications will need to send (or receive) significantly more data than there is bandwidth available to any given blade's interconnect. Checkpoint and restart communication demands for each application are therefore defined to be equal to the smallest link bandwidth that will be used by the node. This utilization of interconnect bandwidth is also considered to be the best

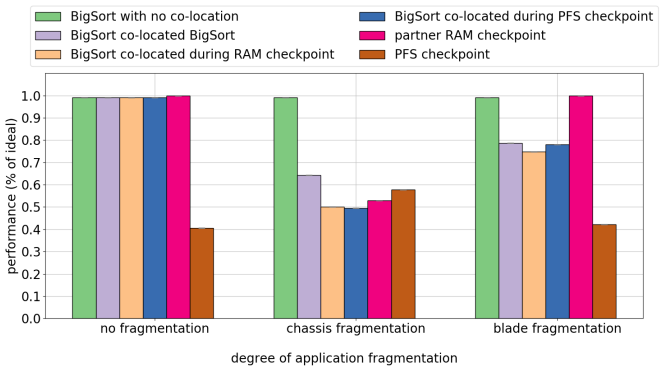

Figure 7.2: Percentage of ideal performance achieved by the "BigSort" application under various execution scenarios.

possible value that can be achieved during the checkpointing event. If network congestion causes a reduction in this utilization then the checkpoint event is experiencing performance degradation.

7.4 Simulation Experiments

We model application execution in the presence of network congestion, application fragmentation, and multiple levels of checkpoint resilience events. For the studies performed here, we examine two large applications each occupying half of the system's nodes and sharing system network resources. Each application's required communication is susceptible to performance degradation from network congestion when co-located in a system with other executing applications.

While the Cray Aries network is capable of scaling to over 500,000 cores, our analysis requires only that there be enough cabinets to allow for multiple applications to be fragmented between cabinets across the system. We therefore analyze a system with two cabinets utilizing the Cray XC30 Aries dragonfly network. This allows for a system with sixteen blades per chassis $\times$ six chassis per cabinet $\times$ two cabinets for a total of 1504 Xeon 12-core processors. A system of this size is both sufficient for the purposes of our study and allows for tractable simulations times.

Application sizes are defined to occupy the number of nodes available in a single cabinet. This results in a total of 752 multicore processors per application (equivalent to 9024 MPI ranks). Application communication demands are defined for each application such that if the application were executing alone over a contiguous set of nodes in a cabinet it will be capable of executing with

a performance value of at least 0.99 (i.e., with minimal performance degradation from network congestion). This is a very conservative assumption because it implies that communication is a negligible bottleneck if the application is not experiencing additional communication overhead from network congestion. The study in [118] suggests that future exascale-sized applications will be significantly more communication-dominant than today's applications, which implies that any performance degradation applications experience from network congestion at the system size we are analyzing is going to exist (and will likely be much worse) when scaled to exascale.

Figure 7.2 shows the percentage of ideal performance achieved by applications executing when susceptible to performance degradation due to network congestion. We examine two levels of fragmentation: coarse-grain fragmentation at the system chassis level (the fragmented application(s) have groups of compute nodes scheduled on alternating tan boxes from Figure 7.1) and fine-grain fragmentation at the blade level (the fragmented application(s) have groups of compute nodes alternating between the blue blades shown in Figure 7.1). For each granularity, we examine the effects that fragmentation has on both application performance under standard execution as well as when performing checkpoints. Bar groupings are shown across the x-axis of Figure 7.2.

Individual bars in each grouping show a different execution scenario:

- BigSort with no co-location: BigSort's performance achievable when executed without any other applications present (this value is defined so that it is within 99% of ideal);
- BigSort co-located with BigSort: BigSort's performance when co-located with another executing copy of BigSort;
- BigSort co-located during RAM checkpoint: BigSort's performance when co-located with an application that is performing a checkpoint to a partner node's RAM;
- BigSort co-located during PFS checkpoint: BigSort's performance when co-located with an application that is performing a checkpoint to the PFS;

- partner RAM checkpoint: denotes the performance of a level two checkpoint of BigSort to a partner node's RAM when executing co-located with a second copy of BigSort that is not in the process of checkpointing;

- PFS checkpoint: denotes the performance of a level three checkpoint of BigSort to the PFS when executing co-located with a second copy of BigSort that is not in the process of checkpointing.

When observing the execution performance of BigSort under various co-location scenarios, the application performance remains unaffected when it is not fragmented. However, the application is affected by both co-location with other applications as well as checkpoints once the application is fragmented. This effect is because of the increased network congestion experienced by the application when it is fragmented across system nodes. BigSort actually experiences worse performance degradation from the coarser fragmentation across system chassis than it does from the finer fragmentation across system blades. Because BigSort has an all-to-all communication pattern the finer granularity actually provides the application with more possible paths that meet the network's "minimum latency" routing requirement. BigSort's performance decreases by as much as 50% when it is experiencing the coarser chassis fragmentation but only decreases by a worst case of 25% when it is fragmented across blades. The figure also indicates that the checkpoint events (both to the PFS and to a partner node's RAM) cause greater performance degradation than if BigSort is executed co-located with another application.

The checkpoint destination (RAM or PFS) does have variations in its effect on performance degradation based on the application's degree of fragmentation. When the BigSort application experiences chassis fragmentation and is co-located with a checkpointing application, checkpoints to the PFS impact BigSort slightly more than checkpoints to a partner node's RAM, however, this ordering reverses at finer fragmentation levels. This effect is caused because at finer granularities, a checkpointing application is spread out over the system and consequently have better access to the PFS because they are able to access more service nodes than when it is isolated in a single

cabinet. This effect is also why the performance of checkpoints to a PFS increase in performance by as much as 18% with chassis fragmentation.

Partner node checkpoints to RAM exhibit interesting behavior in that they only decrease in performance with coarse fragmentation. This is again caused by the layout of the application across the system. Without fragmentation, checkpoints to a partner node's RAM do not experience any congestion because partner nodes are stored across cabinets. Because there is no fragmentation, applications performing a checkpoint to a partner node's RAM are able to utilize the entire link between cabinets. Similarly, when the checkpoint to a partner node is performed at finer fragmentation, the placement of applications across blades happens to allow for utilization of the entire network link as is seen without fragmentation. This demonstrates how highly dependent these checkpoints are to how the application is placed in the system.

In addition to examining the behavior of checkpoint events, we also examined the impact that restart events (both to a partner node's RAM and to a PFS) have on application execution. We found that restart events had nearly identical effects on network congestion as checkpoint events and we have therefore not included these data in the results.

7.5 Conclusions

The preliminary results from our study demonstrate that applications are negatively impacted by network congestion and more specifically that the amount of network congestion that the application experiences is affected by its degree of fragmentation. Results in this work also demonstrate that scheduling has a significant effect on both an application's susceptibility to performance degradation from network congestion when co-located with other applications as well as an application's ability to perform checkpoints with the least possible overhead. Prior work has shown the benefit of attempting to reduce fragmentation to increase application performance. However, our work also demonstrates the benefit that optimal scheduling can have on the resilience protocols employed by those applications and that the placement of applications to system nodes can have a large impact on the resilience protocol's performance. Specifically, we have shown that for both

types of checkpoints examined the placement of the checkpointing application in the system can either benefit checkpoint performance or cause severe performance degradation.

While the focus of this chapter is on measuring the magnitude of the effects that system-level network resource contention has on applications and resilience events, we have designed the network modeling methodology in a way that allows a system designer to use the network model to estimate the expected performance degradation that both the application and resilience events will experience for any specific mapping of applications to system nodes. This can allow an HPC system designer access to better predictions of both application execution times and checkpoint and restart event durations. These predictions can then be integrated into the system designer's framework for HPC resource management so as to allow for both better placement of applications in the system (locations that minimize network congestions) and the selection of more optimal checkpoint intervals.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The challenge of mapping application needs to system resources in HPC environments is complex and involves consideration of uncertainty in application execution characteristics as well constraints of system resources. The research discussed over the course of this dissertation explored the elements of a framework for providing resource management to HPC systems.

Chapter 2 discussed the fine-grained resource contention that exists in an HPC system when application threads share multicore processor resources. Specifically, the negative impact that memory interference caused by application co-location has on HPC systems. We outlined a novel methodology that can be used by a system designer to model application memory interference behavior and make predictions about the effects that co-location will have on the applications executing in a large-scale HPC system. We then demonstrated for the first time how this prediction methodology can be used to create a memory interference aware system scheduler and analyzed its benefit to an environment that suffers from performance degradation due to application co-location.

The results from Chapter 2 are very helpful in our subsequent studies of HPC resilience protocols to provide a more accurate performance estimation in HPC platforms. Most of the remaining chapters of the dissertation focused on examining the system-level effects of HPC resilience in the presence of system failures that result in interrupts in application execution. In Chapter 3, we analyzed the behavior of several resilience protocols being proposed for mitigating system failures in future extreme scale HPC systems and we developed models for simulating their behavior on a common platform for the first time since the protocol's proposal for use in large-scale systems. The performance analysis from Chapter 3 provided insights that allowed for the work in Chapter 4 discussing how those resilience technique models can be utilized to build a resilience aware resource management framework.

While the work of Chapters 2-4 developed resource management techniques that focused on scheduling applications to nodes of the HPC system Chapters 5 and 6 focus more on the optimal scheduling of checkpointing events. Specifically, these chapters consider in more detail the optimization of multilevel checkpointing intervals for improving application performance, system energy use, and application execution time prediction accuracy.

A closer examination of the interrelated effects of checkpointing and the system's communication network is taken in Chapter 7. While the focus of this chapter was on measuring the magnitude of the effects that system-level network resource contention has on applications and resilience events, the network modeling methodology has been designed in a way that compliments this dissertation's focus on resource management so as to allow the system designer both better placement of applications in the system and the selection of more optimal checkpoint intervals.

8.2 Future Work

In addition to the suggestions for future work discussed in some of the prior chapters, there are several other avenues for future research that could be taken to continue developing our work:

- The framework discussed here develops resource management strategies for fine-grain resource contention separately from coarse-grained resource management. The logical next steps would be to integrate the memory interference aware resource management framework from Chapter 2 into the resilience aware resource management framework in Chapter 4. This unified resource management framework could further benefit from incorporating the predictions of performance degradation on application execution and resilience caused by the network congestion that is provided by the work in Chapter 7.

- Applying our methodology from Chapter 2 to create execution time prediction models for a large-scale simulated system enabled us to examine the benefit of a co-location aware scheduling heuristic that can provide substantial performance improvement in a simulated homogeneous 500 node system. The experiments were performed for a homogeneous system of 4-core nodes. It would be therefore beneficial to extend this work to heterogeneous

systems utilizing multicore processors with a greater number of cores per node. The interference from co-location that applications experience is both likely to be greater in machines with more cores and also the execution time prediction models tend to perform even better for the 6-core and 12-core systems than they do for the 4-core system.

- The work in Chapter 2 assumes that applications executing co-located in multicore processors are independent and serial. It is likely that the prediction techniques we developed will translate well to parallel applications if the threads that compose parallel applications are treated as if they are a large number of serial tasks. It would be beneficial for our prediction methodology to be extended to parallel applications in this way and for it to be tested further.

- The unpredictability that memory interference causes to applications executing in HPC systems has the potential to reduce the effectiveness not only of resource management strategies but also the system's resilience. However, integration of these two portions of the framework offers potential beyond optimizations of resource management strategies aimed at mitigating the effects of these two sources of application performance degradation. Because applications tend to execute in memory intensive phases and because the overhead associated with checkpointing affects and is affected by communication, integration of these two frameworks allows for the possibility of improving resilience technique performance through the online optimization of checkpoint intervals that update the schedule of applications checkpoints as the state of the system changes over time.

- Similarly, results from Chapter 6 demonstrate that the optimal set of checkpoint intervals change as a function of checkpoint time. Our results from Chapter 7 indicate that application checkpoint and restart times are highly dependent on the state of the system needing to be accommodated by the network (i.e., the communication patterns of applications in the system and how those applications are scheduled in the system). Given that the state of the system may change several times over the duration of an application's execution, it follows that the optimal set of checkpoint intervals for each application in the system also changes

as a function of the system's state. It would therefore be beneficial to have application checkpoint intervals adapt to the changing system state by developing an online checkpoint interval optimization technique (or possibly an approximation heuristic). Unifying the work in Chapters 6 and 7 would allow for a prediction to be made of the expected duration for performing a checkpoint or restart and for that prediction to subsequently be utilized for the optimization of each application's checkpoint intervals.

- The work discussed here would also benefit from the development and integration of a thermal aware resource management framework. Few works have assessed the effects of system temperature changes on the behavior of failure prone systems and this would provide an interesting avenue for future research. In particular, while future exascale systems are likely to be homogeneous systems, thermal management using dynamic voltage and frequency scaling techniques will indirectly impose heterogeneity across system nodes when CPU frequencies are changed to enable system cooling. This "imposed heterogeneity" will effect the execution times of applications in the system, causing further system unpredictability and making effective system resource management even more challenging.

- Given the conclusions of our results in Chapter 7, future work would benefit from utilizing our findings to design an HPC resource manager with the ability to intelligently co-locate applications so as to minimize the performance impact of resilience events in the system. In addition to the work we presented, such a resource manager would benefit from further examining network congestion effects. Specifically, the effects that network congestion and fragmentation have on other network topologies, a wider variety of application communication patterns, and examination of additional HPC resilience protocols.

- One of the main assumptions about emerging HPC resilience techniques present in our work and common in most of the literature is that exascale HPC systems will have enough memory to easily accommodate the additional memory overhead required for resilience. Given that some of the emerging resilience techniques require the system to store multiple addi-

tional copies of the application's working memory, an important next step of the comparison work we present in Chapter 3 is to assess how the burden each resilience protocol places on memory changes on difference systems and with different applications. It would be of further benefit to investigate strategies each resilience protocol could take for reducing its burden on memory (e.g. using compression, application level checkpoints that denote and store only what it considers important, or alternate resilience strategies that are more aware of their memory use).

Bibliography

[1] Cray XC30 architecture overview, accessed Feb. 2018.

[2] Top500 June 2018, accessed June 2018.

[3] Green500 June 2018, accessed June 2018.

[4] Daniel Dauwe, Eric Jonardi, Ryan D Friese, Sudeep Pasricha, Anthony A Maciejewski, David A Bader, and Howard Jay Siegel. HPC node performance and energy modeling with the co-location of applications. *The Journal of Supercomputing*, 72(12):4771–4809, Dec. 2016.

[5] Daniel Dauwe, Eric Jonardi, Ryan Friese, Sudeep Pasricha, Anthony A Maciejewski, David A Bader, and Howard Jay Siegel. A methodology for co-location aware application performance modeling in multicore computing. In *17th Workshop on Advances on Parallel and Distributed Computing Models (APDCM '15)*, pages 434–443, in the proceedings of 2015 Int'l Par. and Dist. Processing Sympo. Workshops (IPDPSW 2015), May 2015.

[6] Daniel Dauwe, Ryan Friese, Sudeep Pasricha, Anthony A. Maciejewski, Gregory A. Koenig, and Howard Jay Siegel. Modeling the effects on power and performance from memory interference of co-located applications in multicore systems. In *The 2014 Int'l Conf. on Par. and Dist. Processing Techniques and Applications (PDPTA '14)*, pages 3–9, July 2014.

[7] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of HPC applications. In *22nd Annual International Conference on Supercomputing (ICS '08)*, pages 175–184, June 2008.

[8] Qian Zhu, Jiedan Zhu, and Gagan Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–12, Nov. 2010.

[9] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *38th Annual International Symposium on Computer Architecture (ISCA '11)*, pages 283–294, June 2011.

[10] Andreas Sandberg, Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. Modeling performance variation due to cache sharing. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA '13)*, pages 155–166, May 2013.

[11] Jee Choi, Marat Dukhan, Xing Liu, and Richard Vuduc. Algorithmic time, energy, and power on candidate HPC compute building blocks. In *IEEE 28th Int'l Par. and Dist. Processing Sympo. (IPDPS '14)*, pages 447–457, May 2014.

[12] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *48th International Symposium on Microarchitecture (MICRO-48 '15)*, pages 62–75, Oct. 2015.

[13] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *5th European Conference on Computer Systems (EuroSys '10)*, pages 153–166, Apr. 2010.

[14] C. Luque, M. Moreto, F.J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero. CPU accounting for multicore processors. *IEEE Transactions on Computers*, 61(2):251–264, Feb 2012.

[15] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM 44th International Symposium on Microarchitecture (MICRO '11)*, pages 248–259, Dec. 2011.

[16] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A practical method for estimating performance degradation on multicore processors,

and its application to HPC workloads. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 83:1–83:11, Nov. 2012.

[17] F.J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically controlled resource allocation in SMT processors. In *37th International Symposium on Microarchitecture (MICRO-37 '04)*, pages 171–182, Dec. 2004.

[18] Matthew De Vuyst, Rakesh Kumar, and Dean M Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *IEEE 20th Int'l Par. and Dist. Processing Sympo. (IPDPS '06)*, pages 10–20, Apr. 2006.

[19] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing fairness in SMT multicores with a progress-aware scheduler. In *IEEE 29th Int'l Par. and Dist. Processing Sympo. (IPDPS '15)*, pages 187–196, May 2015.

[20] B Dalton Young, Jonathan Apodaca, Luis Diego Briceño, Jay Smith, Sudeep Pasricha, Anthony A Maciejewski, Howard Jay Siegel, Bhavesh Khemka, Shirish Bahirat, Adrian Ramirez, and Yong Zou. Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment. *The Journal of Supercomputing*, 63(2):326–347, Feb. 2013.

[21] Abdulla M Al-Qawasmeh, Sudeep Pasricha, Anthony A Maciejewski, and Howard Jay Siegel. Power and thermal-aware workload allocation in heterogeneous data centers. *IEEE Transactions on Computers*, 64(2):477–491, Feb. 2015.

[22] Bhavesh Khemka, Ryan Friese, Sudeep Pasricha, Anthony A Maciejewski, Howard Jay Siegel, Gregory A Koenig, Sarah Powers, Marcia Hilton, Rajendra Rambharos, and Steve Poole. Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system. *Sustainable Computing: Informatics and Systems*, 5:14–30, Mar. 2015.

[23] Mark Oxley, Sudeep Pasricha, Anthony A Maciejewski, Howard Jay Siegel, Jonathan Apodaca, Dalton Young, Luis Briceño, Johan Smith, Shirish Bahirat, Bhavesh Khemka, Adrian Ramirez, and Yong Zou. Makespan and energy robust stochastic static resource allocation of bags-of-tasks to a heterogeneous computing system. *IEEE Tran. on Par. and Dist. Systems*, pages 2791–2805, Oct. 2015.

[24] D. Talby and D.G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *13th International Parallel Processing Symposium (IPPS '99)*, pages 513–517, Apr. 1999.

[25] S. Sadhasivam, N. Nagaveni, R. Jayarani, and R.V. Ram. Design and implementation of an efficient two-level scheduler for cloud computing environment. In *International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom '09)*, pages 884–886, Oct. 2009.

[26] Gladys Utrera, Julita Corbalan, and Jesús Labarta. Scheduling parallel jobs on multicore clusters using CPU oversubscription. *The Journal of Supercomputing*, 68(3):1113–1140, June 2014.

[27] David A Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303, Apr. 1995.

[28] Ian Jolliffe. *Principal Component Analysis*. John Wiley & Songs, Hoboken, NJ, first edition, 2002.

[29] Edwin KP Chong and Stanislaw H Zak. *An Introduction to Optimization*. John Wiley & Sons, Hoboken, NJ, fourth edition, 2013.

[30] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[31] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, first edition, 2006.

[32] Canonical. Ubuntu 14 release notes, 2012.

[33] Intel 64 and IA-32 architectures software developer's manual combined volumes 1,2a,2b,2c,3a,3b,3c and 3d. Technical report, Dec. 2015.

[34] Intel Xeon E3-1225v3 processor, accessed Jan. 2016.

[35] Intel Xeon E5649 processor, accessed Jan. 2016.

[36] Intel Xeon E5-2697v2 processor, accessed Jan. 2016.

[37] Performance application programming interface, accessed Jan. 2016.

[38] HPCToolkit, accessed Jan. 2016.

[39] Watts Up? plug load meters, accessed Jan. 2016.

[40] PARSEC benchmark suite, accessed Jan. 2016.

[41] NAS parallel benchmarks, accessed Jan. 2016.

[42] Bradley Efron and Robert J Tibshirani. *An Introduction to The Bootstrap*. CRC press, New York, NY, first edition, 1994.

[43] Daniel Dauwe, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. A performance and energy comparison of fault tolerance techniques for exascale computing systems. In *The 6th IEEE Int'l Symp. on Cloud and Service Computing*, pages 436–443, Dec. 2016.

[44] Top500 November 2017, https://www.top500.org/lists/2017/11/, accessed Apr. 2018.

[45] Esteban Meneses, Xiang Ni, Gengbin Zheng, CL Mendes, and LV Kalé. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Trans. Par. and Dist. Systems*, 26(7), July 2015.

[46] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *Int'l Journal of HPC Applications*, 23(3):212–226, Aug. 2009.

[47] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Int'l Conf. for HPC, Networking, Storage and Analysis*, 11 pp., Nov. 2010.

[48] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for HPC. In *Int'l Conf. on Dist. Computing Systems*, pages 615–626, June 2012.

[49] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kalé, Bill Kramer, and Mark Snir. Toward exascale resilience. *Int'l Journal of HPC Applications*, 2009.

[50] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, Sep. 2013.

[51] Said Limam and Ghalem Belalem. A migration approach for fault tolerance in cloud computing. *Int'l Journal of Grid and HPC*, 6(2):24–37, 2014.

[52] David P Jasper. A discussion of checkpoint restart. *Software Age*, 3(10):9–14, Oct. 1969.

[53] John W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, Sep. 1974.

[54] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *Conf. on Supercomputing*, 13 pp., Nov. 2006.

[55] Xiang Ni, Esteban Meneses, and Laxmikant V Kalé. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *Int'l Conf. on Cluster Computing*, pages 364–372, Sep. 2012.

[56] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. In *Int'l Par. Dist. Proc. Symp.*, pages 989–1000, May 2011.

[57] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Int'l Conf. for HPC, Networking, Storage and Analysis*, pages 32:1–32:32, Nov. 2011.

[58] Kathryn Mohror, Adam Moody, Greg Bronevetsky, and B de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Trans. Par. and Dist. Systems*, 25(9):2255–2263, Nov. 2014.

[59] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Low-overhead diskless checkpoint for hybrid computing systems. In *Int'l Conf. on HPC*, 10 pp., Dec. 2010.

[60] Leonardo Arturo Bautista-Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed diskless checkpoint for large scale systems. In *Int'l Conf. on Cluster, Cloud and Grid Computing*, pages 63–72, May 2010.

[61] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *Int'l Par. and Dist. Proc. Symp.*, pages 1181–1190, May 2014.

[62] S. Di, Y. Robert, F. Vivien, and F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model. *IEEE Trans. Par. and Dist. Systems*, 28(1):244–259, Jan. 2017.

[63] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, Feb. 1985.

[64] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Symp. on Principles of Dist. Computing*, pages 171–181, Jan. 1988.

[65] John F Wakerly. Microcomputer reliability improvement using triple-modular redundancy. *Proceedings of the IEEE*, 64(6):889–895, June 1976.

[66] Saurabh Hukerikar, Pedro C Diniz, and Robert F Lucas. A case for adaptive redundancy for HPC resilience. In *Euro-Par 2014: Par. Proc. Workshops*, pages 690–697, Aug. 2014.

[67] Guang Yang. *Life Cycle Reliability Engineering*. John Wiley & Sons, Hoboken, NJ, second edition, 2007.

[68] Gengbin Zheng, Lixia Shi, and L. V. Kalé. FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Int'l Conf. on Cluster Computing*, pages 93–103, Sep. 2004.

[69] Rob F. Van der Wijngaart, Srinivas Sridharan, and Victor W. Lee. Extending the BT NAS parallel benchmark to exascale computing. In *Int'l Conf. on HPC, Networking, Storage and Analysis*, pages 94:1–94:9, Nov. 2012.

[70] HP Z820 workstation. Technical Report 4AA4-0130ENUC, 2015.

[71] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Int'l Par. and Dist. Proc. Symp.*, 12pp. 2009.

[72] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. Energy proportional datacenter networks. In *Int'l Symp. on Computer Architecture*, pages 338–347, June 2010.

[73] J.T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, Feb. 2006.

[74] Infiniband performance, accessed Aug. 2016.

[75] Jack Dongarra. Report on the Sunway Taihulight System. Technical Report UT-EECS-16-742, http://www.netlib.org/utk/people/JackDongarra/PAPERS /sunway-report-2016.pdf, June 2016.

[76] D. W. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Resilience-aware resource management for exascale computing systems. *IEEE Transactions on Sustainable Computing*, 14 pp., accepted 2018, to appear.

[77] Daniel Dauwe, Sudeep Pasricha, Anthony A. Maciejewski, and Howard Jay Siegel. An analysis of resilience techniques for exascale computing platforms. In *19th Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, 2017 Int'l Par. and Dist. Processing Sympo. Workshops (IPDPSW 2017), pp. 914-923, May 2017.

[78] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *IEEE Int'l Conf. on Dependable Systems and Networks*, pages 25–36, June 2015.

[79] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1(1):5–28, June 2014.

[80] Nitin H. Vaidya. A case for two-level distributed recovery schemes. *SIGMETRICS*, 23(1):64–73, May 1995.

[81] A. Benoit, A. Cavelan, V. Le FÃÍvre, Y. Robert, and H. Sun. Towards optimal multi-level checkpointing. *IEEE Trans. Computers*, 66(7):1212–1226, July 2017.

[82] Esteban Meneses, Osman Sarood, and Laxmikant V. KalÃl'. Energy profile of rollback-recovery strategies in high performance computing. *Par. Computing*, 40(9):536–547, Oct. 2014.

[83] C. George and S. Vadhiyar. Fault tolerance on large scale systems using adaptive process replication. *IEEE Trans. on Computers*, 64(8):2213–2225, Aug. 2015.

[84] E Douglas Jensen, C Douglas Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *Real-Time Syst. symp.*, volume 85, pages 112–122, Dec. 1985.

[85] Binoy Ravindran, E Douglas Jensen, and Peng Li. On recent advances in time/utility function real-time scheduling and resource management. In *Int'l Sympo. on Object-Oriented Real-Time Distributed Computing*, pages 55–60, May 2005.

[86] Ken Chen and Paul Muhlethaler. A scheduling algorithm for tasks described by time value function. *Real-Time Systems*, 10(3):293–312, May 1996.

[87] Mehdi Kargahi and Ali Movaghar. Performance optimization based on analytical modeling in a real-time system with constrained time/utility functions. *IEEE Trans. on Computers*, 60(8):1169–1181, Aug. 2011.

[88] Cynthia B Lee and Allan E Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *16th Int'l Sympo. on High Performance Distributed Computing*, pages 107–116, June 2007.

[89] Dylan Machovec, Bhavesh Khemka, Nirmal Kumbhare, Sudeep Pasricha, Anthony A. Maciejewski, Howard Jay Siegel, Ali Akoglu, Gregory A. Koenig, Salim Hariri, Cihan Tunc, Michael Wright, Marcia Hilton, Rajendra Rambharos, Christopher Blandin, Farah Fargo, Ahmed Louri, and Neena Imam. Utility-based resource management in an oversubscribede energy-constrained heterogeneous environment executing parallel applications. *Parallel Computing*, 26 pp., accepted 2017, to appear.

[90] Bhavesh Khemka, Ryan Friese, Luis D Briceno, Howard Jay Siegel, Anthony A Maciejewski, Gregory A Koenig, Chris Groer, Gene Okonski, Marcia M Hilton, Rajendra Rambharos, and Steve Poole. Utility functions and resource management in an oversubscribed heterogeneous computing environment. *IEEE Trans. on Computers*, 64(8):2394–2407, Aug. 2015.

[91] Xiaoyong Tang, Kenli Li, Renfa Li, and Bharadwaj Veeravalli. Reliability-aware scheduling strategy for heterogeneous distributed computing systems. *Journal of Par. and Dist. Computing*, 70(9):941 – 952, Sep. 2010.

[92] Anne Benoit, Mourad Hakem, and Yves Robert. Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Computing*, 35(2):83 – 108, Feb. 2009.

[93] Alain Girault, Christophe Lavarenne, Mihaela Sighireanu, and Yves Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. In *Int'l Par. and Dist. Processing Sympo.*, 8 pp., Apr. 2001.

[94] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Trans. on Par. and Dist. Systems*, 9(11):1137–1152, Nov. 1998.

[95] Xiao Qin and Hong Jiang. A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5):331 – 356, June 2006.

[96] Qin Zheng and Bharadwaj Veeravalli. On the design of communication-aware fault-tolerant scheduling algorithms for precedence constrained tasks in grid computing systems with dedicated communication devices. *Journal of Par. and Dist. Computing*, 69(3):282 – 294, Mar. 2009.

[97] N. R. Gottumukkala, C. B. Leangsuksun, N. Taerat, R. Nassar, and S. L. Scott. Reliability-aware resource allocation in HPC systems. In *Int'l Conf. on Cluster Computing*, pages 312–321, Sep. 2007.

[98] A. Benoit, L. Pottier, and Y. Robert. Resilient application co-scheduling with processor redistribution. In *2016 45th Int'l Conf. on Parallel Processing*, pages 123–132, Aug. 2016.

[99] Top500 June 2017, https://www.top500.org/lists/2017/06/, accessed Aug. 2017.

[100] JEDEC Solid State Technology Association. DDR4 SDRAM standard. Technical Report JESD79-4B, https://www.jedec.org/system/files/docs/JESD79-4B.pdf, June 2017.

[101] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *Int'l Conf. on Dependable Systems and Networks*, pages 610–621, June 2014.

[102] N. R. Tallent, K. J. Barker, D. Chavarria-Miranda, A. Tumeo, M. Halappanavar, A. Marquez, D. J. Kerbyson, and A. Hoisie. Modeling the impact of silicon photonics on graph analytics. In *Int'l Conf. on Networking, Architecture and Storage (NAS)*, 11 pp., Aug. 2016.

[103] A. Marowka. Back to thin-core massively parallel processors. *Computer*, 44(12):49–54, Dec. 2011.

[104] Tajana Simunic, Kresimir Mihic, and Giovanni De Micheli. Optimization of reliability and power consumption in systems on a chip. In *Int'l Workshop on Integrated Circuit and System Design Power and Timing Modeling, Optimization, and Simulation*, pages 237–246, Sep. 2005.

[105] D. Dauwe, R. Jhaveri, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Optimizing checkpoint intervals for reduced energy use in exascale systems. In *2017 Eighth Int'l Green and Sustainable Computing Conf. (IGSC)*, 8 pp., Oct 2017.

[106] Pacific Gas and Electric Company. Electric schedule e-19. Technical Report ELEC - SCHEDS-E-19, Apr. 2015.

[107] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, Sep.

[108] Esteban Meneses, Osman Sarood, and Laxmikant V Kalé. Assessing energy efficiency of fault tolerance protocols for hpc systems. In *IEEE 24th Int'l Sympo. on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 35–42, Oct.

[109] Osman Sarood, Esteban Meneses, and Laxmikant V Kalé. A 'cool' way of improving the reliability of HPC machines. In *2013 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, Nov.

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

[111] R. R. Chandrasekar, A. Venkatesh, K. Hamidouche, and D. K. Panda. Power-check: An energy-efficient checkpointing framework for HPC clusters. In *Int'l Sympo. on Cluster, Cloud and Grid Computing*, pages 261–270, May 2015.

[112] S. H. Lim, S. W. Lee, B. H. Lee, and S. Lee. Power-aware optimal checkpoint intervals for mobile consumer devices. *IEEE Trans. on Consumer Electronics*, 57(4):1637–1645, Nov. 2011.

[113] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen

Yang. The Sunway Taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7), 24 pp., June 2016.

[114] Daniel Dauwe, Sudeep Pasricha, Anthony A Maciejewski, and Howard Jay Siegel. An analysis of multilevel checkpoint performance models. In *19th Workshop on Advances on Parallel and Distributed Computing Models (APDCM '18)*, in the proceedings of 2015 Int'l Par. and Dist. Processing Sympo. Workshops (IPDPSW 2018), May 10 pp., accepted 2018, to appear.

[115] E. Gelenbe and D. Derochette. Performance of rollback recovery systems under intermittent failures. *Commun. ACM*, 21(6):493–499, June 1978.

[116] L. B. Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Low-overhead diskless checkpoint for hybrid computing systems. In *2010 International Conference on High Performance Computing*, Dec. pp. 10, 2010.

[117] Prasanna Balaprakash, Leonardo A. Bautista-Gomez, Mohamed-Slim Bouguerra, Stefan M. Wild, Franck Cappello, and Paul D. Hovland. *Analysis of the Tradeoffs Between Energy and Run Time for Multilevel Checkpointing*, pages 249–263. Nov. 2015.

[118] Sébastien Rumley, Dessislava Nikolova, Robert Hendry, Qi Li, David Calhoun, and Keren Bergman. Silicon photonics for exascale systems. *Journal of Lightwave Technology*, 33(3):547–562, 2015.

[119] S. A. Jyothi, A. Singla, P. B. Godfrey, and A. Kolla. Measuring and understanding throughput of network topologies. In *SC16: Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 761–772, Nov. 2016.

[120] Christopher Zimmer, Saurabh Gupta, Scott Atchley, Sudharshan S Vazhkudai, and Carl Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 11 pp., Nov. 2016.

[121] Anupam Gupta, Jon Kleinberg, Amit Kumar, Rajeev Rastogi, and Bulent Yener. Provisioning a virtual private network: A network design problem for multicommodity flow. In *Proceedings of the Thirty-third Annual ACM Symp. on Theory of Computing STOC*, STOC '01, pages 389–398, July 2001.

[122] Murali Kodialam, T. V. Lakshman, and Sudipta Sengupta. Traffic-oblivious routing in the hose model. *IEEE/ACM Trans. Netw.*, 19(3):774–787, Jun. 2011.

[123] Ankit Singla, P Godfrey, and Alexandra Kolla. High throughput data center topology design. In *Proceedings of the 11th USENIX Conf. on Networked Systems Design and Implementation*, pages 29–41. USENIX Association, 2014.

[124] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 Int'l Sympo. on Computer Architecture*, pages 77–88, June 2008.

[125] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J. Wright, and Laxmikant V. Kale. Maximizing throughput on a dragonfly network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 336–347, 12 pp., Nov. 2014.

[126] T. C. Hu. Multi-commodity network flows. *Operations Research*, 11(3):344–360, 1963.

[127] Coral benchmarks, accessed Apr. 2018.