Exploiting Field Data Analysis to Improve the Reliability and Energy Efficiency of HPC Systems

by

Nosayba El-Sayed

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2016 by Nosayba El-Sayed
Abstract

Exploiting Field Data Analysis to Improve the Reliability and Energy Efficiency of HPC Systems

Nosayba El-Sayed
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

As the scale of High-Performance Computing (HPC) clusters continues to grow, their increasing failure rates and energy consumption levels are emerging as two serious design concerns that are expected to become more challenging in future Exascale systems. The efficient design and operation of such large-scale installations critically relies on developing an in-depth understanding of their failure behaviour as well as their energy consumption profiles.

Among the main obstacles facing the study of HPC reliability and energy efficiency issues, however, is the difficulty of replicating HPC problems inside a lab environment or obtaining access to operational field data from HPC organizations. Examples of such field data include node failure logs, hardware replacement logs, system event logs, workload traces, data from environmental sensors, and more. Fortunately, the recent decade has witnessed an increasing number of HPC organizations willing to share their operational data with researchers or even make them publicly available.

In this work, we exploit field data analysis in improving our understanding of HPC failures in real world systems, and in optimizing HPC fault-tolerance protocols while analyzing their respective performance and energy overheads. Throughout our analyses, we investigate various HPC design tradeoffs between system performance, system reliability, and energy efficiency.

Our results in the first part of this thesis provide critical insights into how and why failures happen in HPC installations as well as which types of failures are correlated in the field. We study the impact of various factors on system reliability, including environmental factors such as data center temperature and power quality. We find that the effect of
temperature, for example, on hardware reliability in large-scale systems is smaller than often assumed. This finding implies that the operators of these facilities can achieve high energy savings by raising their operating temperatures, without making significant sacrifices in system reliability. Our analysis of power problems in large HPC facilities, on the other hand, revealed strong correlations between different power issues (e.g., power outages, voltage spikes, etc.), and increased failure rates in various hardware and software components. Based on our observations, we derive learned lessons and practical recommendations for the efficient design and operation of large-scale systems.

The second part of this thesis utilizes the knowledge obtained from our HPC failure analysis in improving HPC fault-tolerance techniques. We focus on the most widely used fault-tolerance mechanism in modern HPC systems: ‘checkpoint/restart’. We study how to optimize checkpoint-scheduling in parallel applications for both performance and energy efficiency purposes. Our results show that exploiting certain failure characteristics of HPC systems in designing checkpoint-scheduling policies can reduce the energy/performance overheads that are associated with faults and fault-tolerance in HPC systems significantly.
To my wonderful parents Zeinab and Abubakr.
Acknowledgements

“The essence of all beautiful art, all great art, is gratitude.”
–Friedrich Nietzsche

Every now and then, when I’m not at the lab running simulations or at home watching ‘The Office’, I like to write poetry. And I usually conclude every poem I write with a verse that contains a short supplication or an expression of gratitude and praise to those worthy of it, or both. Only when I write this last verse is when I feel that my poem is complete. It is now the time to write the final verse of my PhD dissertation; it is time to give thanks where thanks are due.

My first expression of gratitude goes to my advisor Professor Bianca Schroeder, whose outstanding support and guidance have made my PhD journey truly rewarding, educational, and quite enjoyable. I have been very lucky to be working under the supervision of a great scholar who is as immensely knowledgeable yet approachable as Bianca. Her guidance has helped shape my approach to computer systems research over the years. I learned from Bianca, among many things, the value of asking the right research questions whose answers offer meaningful insights to the systems community, both theoretically as well as practically. Her constant mentoring and advice on how to ask the right questions, how to communicate ideas and results effectively, how to deliver high-quality presentations, and how to seek and offer good feedback, have all been very enriching to my graduate school experience.

Secondly, I would like to thank my committee members, Professor Angela Demke-Brown and Professor Cristiana Amza for the valuable and interesting feedback they have given me during our committee meetings and more recently on my thesis document. Their comments have helped me in different occasions become aware of my own blind spots in ways that widened my research horizons. Special thanks goes to Professor Sotirios Damouras from the department of Statistics for his insightful answers to my statistics questions over the years. Sotirios’s input has strengthened the statistical rigor of the work conducted and presented in this dissertation greatly. I would also like to thank my PhD external examiner Professor Kenneth Salem from Waterloo University for his valuable feedback and comments on my PhD dissertation.

I would like to thank all the organizations whose datasets I’ve utilized in my thesis, including Google, Los Alamos National Labs, the Parallel Data Lab at Carnegie Mellon University, Canada’s SciNet, and the US National Oceanic and Atmospheric Administration. I would also like to thank the Government of Ontario for the financial support I received to conduct my research, mainly through the Ontario Graduate Scholarship (OGS) and the Queen Elizabeth II Graduate Scholarship in Science and Technology (QEII–GSST), as well as the Department of Computer Science (DCS) at the University of Toronto for providing me with financial support and a great working environment during graduate school.
I am very grateful for the labmates I had in the Systems and Networks (SysNet) lab and for all the intellectually-stimulating discussions I’ve had the pleasure of participating in—sometimes unwillingly—, with this group of incredibly smart, curious, and friendly people. I would like to particularly thank Andy Hwang, George Amvrosiadis, Ioan Stefanovici, Daniel Fryer, Bogdan Simion and Suprio Ray, for being there. Literally. All the time. Thank you for making it a lot easier for me to work late in the lab or show up on weekends. I could not think of a better group of people to work with—except of course for the other, more quiet folks in the lab: Sahil Suneja, Tianzheng Wang, Junji Zhi, and Kang-Nyeon Kim, to name a few. Thank you all for being such wonderful labmates. Other colleagues outside of SysNet that I would like to thank are Mazen Al-Borno, Michael Glueck, Orion Buske (Cookiemaster!), Aditya Bhargava, Elizabeth Patitsas, and Patricia Thaine. Special thanks goes to Shahan Khatchadourian for generously reviewing parts of my thesis and offering me great feedback. Thanks are also due to our amazing DCS staff members: Joseph Raghubar, Lynda Barnes, Sara Burns, Lisa DeCaro, Marina Haloulous, Celeste Francis Esteves, and Sarah Lavoie. Thank you for your helpfulness and assistance with various administrative matters during my PhD studies.

I’d also like to express my sincere gratitude to all the wonderful friends I have here who were more like a second family to me during my stay in Toronto. Special thanks goes to my good old friend Khadige Abboud, whose exceptional support and thoughtfulness helped me survive a lot of stressful moments over the years. Thank you for being you. My wonderful Toronto friends: Eman Hammad, Toka Sabry, Aya Aboudina, Sara Anis, Sarah Salem, Nourhan Safwat, Lina El-Shamy, Dalia Hashim, Somaia Ali, Bailsan Khashan, Rana Mohamed, Mona El-Mosallamy, and Nagwa El-Ashmawy. Your friendships are among the most precious things I’ve gained from my graduate school years. I’m sincerely grateful for your support, your generosity, and for helping me step outside of my comfort zone in numerous occasions. Thank you for filling the past five years with many priceless memories.

My deepest gratitude goes to my wonderful family who live far away from here for their continuous, unconditional support throughout this journey. I am particularly indebted to my lovely mother and role model, Zeinab. Without your support, your invaluable advice, and your beautiful prayers I would not be where I am today. I also could not have made it this far without the support of my beloved father, Dr. Abu-Bakr, who instilled in me a deep appreciation for science and encouraged me to pursue my dreams. Thank you both for always being there for me. I could not be more proud or more grateful to be your daughter. My wonderful siblings Asma, Aisha, Somayya, Abdul-Rahman, and Abdullah: thank you all for your constant encouragement. It brightens my day to share a laugh with you or to look at the photos and messages you share in our family group to keep myself updated with your news. Finally, all praise goes to God, Lord of the worlds. The Most Gracious, the Most Merciful. May the work in this thesis bring some benefit to the world.
Contents

1 Introduction ................................................................. 1
  1.1 Motivation .................................................................. 1
  1.2 Thesis Framework ...................................................... 3
  1.3 Thesis Contributions .................................................. 4

2 Failure Analysis in HPC Systems ........................................ 6
  2.1 Introduction ............................................................... 6
    2.1.1 Motivation ............................................................. 6
    2.1.2 Background .......................................................... 7
  2.2 Related Work ............................................................. 9
    2.2.1 Failure Characterization .......................................... 9
    2.2.2 Failure Prediction .................................................. 11
    2.2.3 Temporal and Spatial Correlations ........................... 13
    2.2.4 Impact of Environmental Factors .............................. 14
  2.3 Understanding HPC Failure Correlations ........................ 15
    2.3.1 Failure-Type Correlations ....................................... 16
    2.3.2 Failure-Prone HPC Nodes ....................................... 23
  2.4 Impact of Usage and Utilization .................................... 25
    2.4.1 Effect of Usage on Node Reliability ......................... 25
    2.4.2 User Proneness to Failures .................................... 27
  2.5 Impact of Environmental Factors ................................... 28
    2.5.1 Power quality and Reliability ................................ 28
    2.5.2 Temperature and Reliability .................................. 32
    2.5.3 External Factors: Cosmic Radiation ......................... 48
  2.6 Combining Multiple Factors: Regression Analysis ............. 50
  2.7 Large-Scale Job Failure Analysis .................................... 52
    2.7.1 Description of workload traces ............................... 52
    2.7.2 What characterizes unsuccessful jobs in large clusters? 54
    2.7.3 What are the root-causes behind job failures? ............ 64
List of Tables

2.1 Summary of the LANL failure data. ........................................... 17
2.2 Overview of Google LSE dataset. ................................................. 34
2.3 Parameters from fitting linear and exponential models to monthly LSE probabilities as a function of avg. temperature. ............................................. 36
2.4 Overview of Google Disk Failure Logs. ......................................... 40
2.5 Parameters from fitting a linear and an exponential model to monthly disk failures as a function of avg. temperature. ............................................. 41
2.6 Summary of Regression variables .................................................. 51
2.7 Poisson regression coefficients ..................................................... 51
2.8 NB Regression Coefficients ......................................................... 52
2.9 Description of exit status meaning in the LANL job traces. .................. 54
2.10 Overview of the workload traces and basic job statistics for the clusters in our dataset. ................................................................. 54
2.11 Breakdown of Google’s batch task events that were correlated with machine removals. ................................................................. 73
2.12 Summary of job status classification and prediction variables for the Google cluster ................................................................. 79
2.13 Overview of the different datasets used in Chapter 2. ....................... 88
2.14 Summary of key observations from our analysis of failures in large-scale systems. 88

3.1 Overview of the LANL clusters in our dataset. ................................. 96
3.2 Comparison of the fraction of time wasted in a system when using different checkpointing methods, for all the LANL systems in our dataset. (The lowest observed overhead in terms of wasted work is marked in bold font.) .... 105
3.3 Optimizing the checkpoint interval for energy with a bound on runtime, assuming (P_{\text{comp}}/P_{\text{checkpt}})=3X. ........................................ 116
3.4 Comparison of the performance/energy tradeoffs introduced by different checkpointing methods under (P_{\text{comp}}/P_{\text{checkpt}})=3X, C=5 minutes. (Note: a negative sign in the time/energy overhead columns indicates savings). ............ 118
3.5 Optimizing the checkpoint interval for energy with a bound on I/O for all LANL systems (simulation results under C=10 minutes, $P_{\text{comp}}/P_{\text{ckpt}}=3X$).
# List of Figures

1.1 Improving the reliability and energy efficiency of HPC systems: A research framework. ................................................................. 3

2.1 Building units of a High-Performance Computing cluster. ................ 7
2.2 Sample records in LANL's failure data. ........................................... 16
2.3 Sample HPC machine layout file at LANL. ..................................... 17
2.4 Correlations between failures in the same node: The bars show the probability that any node-failure follows a failure of type X. .................. 18
2.5 Correlations between failures in the same node: The bars show the probability that a failure of type X follows other node-failures. ................. 19
2.6 Correlations between failures in the same rack ............................... 21
2.7 Correlations between failures in the same system. Each bar corresponds to the probability that a node-failure of type X is followed by any failure in another node in the same system. ......................................... 22
2.8 Total number of failures as a function of Node-ID ............................ 23
2.9 Root cause breakdown in failure prone nodes vs other nodes ............. 23
2.10 The probability of different failure types in failure prone nodes compared to the rest of the nodes in a system. ................................. 24
2.11 The impact of usage on node reliability. ......................................... 26
2.12 Distribution of failed jobs over different LANL users. ..................... 27
2.13 Breakdown of environmental failures in LANL systems .................... 28
2.14 Impact of power problems on hardware failures ............................ 29
2.15 Impact of power problems on software failures ............................. 31
2.16 Distribution of power-related failures across nodes over time (LANL System 2) 32
2.17 The monthly probability of LSEs as a function of temperature. ........... 34
2.18 The monthly probability of LSEs as a function of variability in temperature, measured by the coefficient of variation. .......................... 37
2.19 The quartiles of number of LSEs for drives with LSEs as a function of temperature. ................................................................. 38
2.20 The monthly probability of LSEs as a function of temperature by drive age.

2.21 The monthly probability of LSEs as a function of temperature for drives with high and low read loads (left) and write loads (right).

2.22 The monthly probability of a disk failure as a function of temperature separated by disk model.

2.23 Node temperature as a function of its position inside the rack in LANL System 20.

2.24 Probability of node outages at LANL due to DRAM problems as a function of temperature (left) and rack positions as a proxy for temperature (middle, right).

2.25 DIMM replacements at SciNet.

2.26 Probability of uncorrectable DRAM errors at Google as a function of temperature.

2.27 Probability of node outages at LANL as a function of temperature (left) and rack positions as a proxy for temperature (middle, right).

2.28 Probability of hardware replacements at SciNet.

2.29 Box plots for per node downtime as a function of temperature.

2.30 Probability of node outages by temperature (left) and by coefficient of variation (right).

2.31 Impact of temperature related problems on hardware failures.

2.32 Probability of CPU and memory failures as a function of average monthly neutron counts.

2.33 Comparison of the distribution of job durations among jobs that finish, get killed, or fail, in all clusters included in our traces.

2.34 Breakdown of number of mapper tasks and reducer tasks per job in CMU’s OpenCloud traces.

2.35 The rate of job interruptions per task-week in Google and CMU Hadoop clusters.

2.36 Comparison of job status breakdown (left) and job interruption rates (right) between single-node jobs and parallel jobs in LANL’s HPC cluster.

2.37 Effect of different scheduling constraints on a job’s final status in Google’s cluster.

2.38 Requested resources by tasks in a job versus the exit status of a job in the Google cluster and CMU’s OpenCloud cluster.

2.39 The average resource utilization in Google’s jobs, for jobs that finish successfully, fail, or get killed.

2.40 Comparison of I/O usage in CMU Hadoop jobs that complete successfully, fail, or get killed.
2.41 Comparison of I/O usage in CMU Hadoop jobs that complete successfully, fail, or get killed. 63
2.42 Accuracy of CPU and memory estimation in Google’s Batch tasks. 66
2.43 Breakdown of the parameter values for the maximum number of task retries configured in the CMU Hadoop jobs. 68
2.44 The probability that a task succeeds in the next attempt as a function of the number of past failed attempts. 69
2.45 The distribution of the number of actual task retries in tasks that failed. 71
2.46 The relationship between task exit status and job exit status. 71
2.47 Distribution of the variability in resource consumption over a task’s lifetime in Google’s batch tasks. 74
2.48 Ratio of resource usage by failed tasks to completed tasks in the same job. 75
2.49 User job interruption rates and cluster resource usage in Google (left) and CMU (right) clusters. 76
2.50 Comparison of resource usage and execution time between failed jobs and completed jobs that belong to the same application in the Google cluster. 77
2.51 Evaluation of job interruption predictions in the Google cluster using different input data as predictors. 80
2.52 Distribution of the remaining time (left) and the remaining portion (right) of a Google job, after the first task attempt failure. 81
2.53 Google task failure prediction results under different input variables and data sampling ratios. 83
2.54 Per-task distributions of true positive and false positive sliding window predictions in Google’s cluster. 84

3.1 Example of a compute cycle in a checkpoint/restart system. 91
3.2 Wasted time under Young compared to wasted time under the optimal fixed checkpoint interval for all LANL systems in our dataset, and under different checkpoint cost assumptions. 97
3.3 Wasted time assuming an error in the MTTF estimation. 98
3.4 Checkpointing using different techniques that exploit system MTTF variability over time, in the four largest LANL systems in our dataset. 100
3.5 Failure rates as a function of system age for two LANL systems. 101
3.6 Expected remaining time to fail given time since last failure. 103
3.7 The Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) of failure inter-arrival times for LANL system 20. 104
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>Errors in estimation of wasted work using equations with respect to the wasted work under Young when using trace-based simulations, for different checkpointing cost assumptions. (Note that a positive sign indicates overestimation while a negative sign indicates underestimation.)</td>
</tr>
<tr>
<td>3.9</td>
<td>Errors in estimation of time spent writing checkpoints using equations plotted against each LANL system’s best Weibull shape fit.</td>
</tr>
<tr>
<td>3.10</td>
<td>Errors in estimation of time spent recovering lost work using equations plotted against each LANL system’s best Weibull shape fit.</td>
</tr>
<tr>
<td>3.11</td>
<td>Time and energy overheads as a function of the static checkpoint interval under different checkpoint costs ($C$) and power-ratios ($P_{\text{comp}}/P_{\text{checkpt}}$).</td>
</tr>
<tr>
<td>3.12</td>
<td>Energy/performance tradeoffs for all checkpointing policies under different power-ratio assumptions ($P_{\text{comp}}/P_{\text{checkpt}}$) and assuming $C=5$ minutes, in LANL System 20. (Hollow markers represent runtime-optimized methods; filled markers represent energy-optimized methods).</td>
</tr>
<tr>
<td>3.13</td>
<td>Evaluation of different approaches to estimating energy-optimized Hazard in practice for four LANL systems, under ($P_{\text{comp}}/P_{\text{checkpt}}$)=3X.</td>
</tr>
<tr>
<td>3.14</td>
<td>Breakdown of wasted work across a wide range of static checkpoint intervals for two LANL systems (Y-axis is logscale).</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

As the component count in high-performance computing (HPC) installations continues to increase to keep up with the growing demand for compute power and storage capacity, system reliability and energy efficiency are emerging as two main challenges for the efficient design and operation of such large-scale systems. These issues will become even more challenging in future exascale systems, which are expected to combine the compute power of millions of CPU cores.

The first challenge, system reliability, has attracted a significant body of research in the past decade. In large-scale platforms, failures are becoming the norm rather than the exception, leading to major problems such as data loss or corruption and the expensive cost of system downtime and repair. Despite having relatively reliable individual components, the sheer number of components in future Exascale platforms will increase failure rates to unprecedented levels. For example, recent studies predict that the mean time to failure in an exascale machine will be in the order of minutes [12]. Efficiently operating future HPC systems at such large scales therefore requires an in-depth understanding of their failure behaviour.

The second main challenge facing HPC systems is energy efficiency. The energy consumed by data centers nowadays is starting to make up a significant fraction of the world’s energy consumption and carbon emissions. In fact, world’s data centers are estimated to consume power equivalent to seventeen 1,000 MW power plants, equaling more than 1% of total world electricity consumption, and to emit as much carbon dioxide as all of Argentina [59].

Designing energy-efficient data centers, however, is quite challenging as it requires a good understanding of how to balance different tradeoffs between potential energy savings
and system performance or system reliability issues. For example, while different energy efficiency mechanisms are being proposed and discussed in the literature, their impact on system reliability and system performance in practice remains to a large extent not well understood. Additionally, the energy profile of many resilience techniques in HPC systems remains unexplored, unlike their performance impact, making it difficult to assess their energy impact when they are deployed in the field.

Among the main obstacles that make it challenging to understand these reliability and energy efficiency issues in HPC platforms is the difficulty of replicating HPC problems in a lab environment, or obtaining access to sensitive failure logs and operational trace data from HPC organizations. The analysis of field data from production HPC systems provides researchers with insights into how and why failures happen in these systems under real life circumstances. Additionally, real world data can be used to drive trace-based simulations, which are helpful for replicating real life scenarios and investigating research questions of practical nature. Fortunately, the past decade has witnessed a gradual increase in the number of HPC organizations willing to share their field data with researchers, or even make them publicly available [6, 60, 83, 91].

Our main goal in this work is to exploit field data analysis in improving the reliability and energy efficiency of large-scale systems. We focus mostly on HPC systems, which are large-scale computing systems designed to run parallel, tightly-coupled HPC applications that solve complex computational problems (e.g. scientific simulations, climate modeling, bioinformatics, etc.). Understanding how failures are correlated in these systems is critical due to the heavy communication that takes place between the nodes collaborating on a job. In such systems, even a single failure can cause a system-wide outage. Besides HPC systems, in some parts of our work we also examine and compare other types of large-scale systems, such as multi-purpose clusters that run diverse workloads with high degrees of heterogeneity.

We first analyze failure logs from different HPC organizations to study the impact of a diverse set of factors on HPC reliability, and to understand how HPC failures of various types are correlated in the field. Next, using the knowledge obtained from our failure log analysis, we propose and evaluate different methods for optimizing the fault-tolerance of “checkpoint/restart” systems, where the system (or the application) writes periodic checkpoints of critical data to stable storage to recover from in the case of a failure. In particular, we study how checkpoint-scheduling policies can be optimized either for performance or for energy gains, while exploiting knowledge of different system failure properties.

Throughout our HPC failure behaviour study and fault-tolerance optimizations, we explore various HPC design tradeoffs between system reliability, performance, and energy efficiency, while utilizing real world traces to drive our simulations and analyses. In the remainder of this section we present our thesis research framework and the main contributions our work in this thesis has achieved so far.
1.2 Thesis Framework

In this section we provide a high-level overview of our research objectives in this thesis, which we develop under the main goal of improving the reliability and energy efficiency of HPC systems. In Figure 1.1 we outline our general research framework, while highlighting some of the primary research questions that we explore in each part. Our framework consists mainly of three components: (a) HPC Failure Analysis, (b) HPC Fault-Tolerance Optimization, and (c) Exploring Energy, Reliability and Performance Trade-offs.

(a) Failure Analysis in HPC Systems. Our first goal is to understand the failure behaviour of HPC systems by analyzing real world failure logs collected at HPC production installations. In particular, we investigate several questions that help us improve our understanding of why, when, and how failures happen in HPC systems, including:
• What are the root-causes of failures in HPC systems, and how are they correlated in time and space?
• What is the impact of usage and utilization on node reliability and job reliability?
• What is the impact of environmental factors in particular, such as temperature or power quality, on HPC reliability?

We present the results of the studies we conducted to tackle these questions in Chapter 2.

(b) HPC Fault-tolerance Optimization. Using the results obtained from our HPC failure analyses in the first part of this thesis, we study ways of utilizing this knowledge for the design of efficient and practical checkpoint/restart policies in HPC platforms. We first study the limitations of existing checkpointing solutions proposed in the literature from a performance perspective. Next, we design and evaluate new checkpointing policies that exploit different HPC failure properties, while taking different practical considerations into account. We then study how checkpoint-scheduling policies can be optimized for energy savings rather than for performance purposes. Chapter 3 presents our comprehensive analysis and evaluation of a wide array of checkpoint-scheduling solutions when optimized for runtime or for energy savings.

(c) Exploring Performance, Reliability and Energy Tradeoffs. Throughout our failure analyses and checkpoint-scheduling optimizations, we identify different tradeoffs between the key HPC system design factors: performance, reliability, and energy efficiency. As HPC architectures grow in size and complexity, capturing the interplay between these factors in practice is becoming increasingly challenging.

For example, while various energy efficiency mechanisms are proposed in the literature, their impact on system reliability remains to a large extent not well understood in practice, and often difficult to predict or quantify. We utilize field data analysis to tackle this problem in Section 2.5, for instance, where we use several years worth of data to study the reliability cost of raising the operating temperatures in HPC facilities – a technique that reduces the energy consumed by the cooling system significantly. Another design tradeoff we investigate in our work is the tradeoff between energy savings and performance overheads when optimizing HPC checkpoint-scheduling for energy purposes, in Section 3.4.

1.3 Thesis Contributions

Below, we summarize some of the main contributions made by our work in this thesis:

• Our study of failure correlations in HPC systems is the first large-scale research study that considers temporal and spatial correlations between different types of failures in HPC production systems. For example, based on the analysis of a decade worth of failure data, we identified various statistically-significant correlations between failures related to environ-
mental factors, such as power outages or chiller system failures, and node hardware failures. Our results provide HPC researchers and operators with intuitive insights into which factors are more likely to cause follow-up failures in HPC systems, which components are more likely to be affected, and what the magnitude of increase in node failure rates is expected to become [33].

• Understanding the effect of temperature on system reliability in large-scale systems is important as a large fraction of a data center’s energy bill goes into cooling. Our trace-based analysis of the impact of temperature on different hardware components indicates that the impact of temperature on hardware components might be much weaker than often assumed; for some types of errors, such as DRAM errors, we report no correlation at all [38]. Our findings, which are based on a large collection of field data from different production environments, show significant potential for saving energy in large-scale systems.

• Checkpoint/Restart is the most commonly used fault-tolerance technique in HPC systems. In this thesis, we evaluate the performance of a wide array of checkpoint-scheduling policies, some previously known in the literature as well as new ones that we propose, using real world failure logs. Our results show that a simple, closed-form solution can easily be adapted for use in practice and achieves near-optimal performance. We also show that back-of-the-envelope formulas can be used to accurately estimate the wasted work in HPC systems, and make projections of future HPC systems and requirements for their efficient use [34, 35].

• Checkpointing solutions have been traditionally optimized and analyzed from a performance perspective. With energy efficiency emerging as a key driver in the design of HPC architectures, optimizing checkpoints needs to be revisited from an energy perspective. Our study of energy-optimized checkpointing [36,37] is the first to provide a comprehensive analysis of the different tradeoffs introduced by energy-optimized checkpoint-scheduling in HPC platforms. We provide insights into the energy overhead as well as the performance impact associated with different checkpointing policies that vary in their design and complexity, using real world HPC traces.
Chapter 2

Failure Analysis in HPC Systems

2.1 Introduction

2.1.1 Motivation

System reliability is one of the major challenges in running and designing high-performance computing (HPC) systems. As architectural constraints limit the speed of individual devices, the component count in HPC systems is continuously growing. For example, future Exascale systems are expected to combine the compute power of millions of CPU cores, and to have a mean time to fail (MTTF) of one hour or less [12]. Efficiently running systems at such large scale requires developing a good understanding of their failure behaviour.

The analysis of HPC logs provides researchers with critical insights into why and how failures happen in real world scenarios, and in identifying factors or circumstances that are predictive of future failures. Understanding what those factors are can help HPC operators mitigate them, or take proactive measures against impending failures in cases where they cannot be avoided.

The first goal of this thesis is to provide a comprehensive study of how production HPC systems fail, using field data analysis. We apply rigorous statistical and analytical methods to different HPC datasets in order to provide intuitive insights into whether and how failures are correlated in the field. We also investigate the impact of various factors, such as temperature, power quality, node utilization, and external factors such as cosmic radiation, on HPC system reliability.

The rest of this section will provide some background on HPC reliability definitions and a summary of the different types of HPC logs that we consider in our analyses.
Chapter 2. Failure Analysis in HPC Systems

2.1.2 Background

a) Definitions. Throughout this chapter, we assume the following definitions for HPC related terminology and system dependability concepts:

Node: A single computer server (host) that represents the building block of an HPC cluster (see Figure 2.1-(a)). Nodes are typically configured into the following types, according to their function: login nodes, compute nodes, storage (I/O) nodes, and network nodes.

Rack: In HPC platforms, multiple nodes are stacked one above the other in a single physical rack (as illustrated in Figure 2.1-(b)). Racks are configured to consolidate network resources and equipment cooling mechanisms.

System: A cluster of loosely or tightly connected nodes that are designed to cooperate on running HPC applications and are typically connected to each other over high-performance local area networks (LAN). An example of such a system is shown in Figure 2.1-(c).

Fault: A fault is a defect in the system that may or may not cause an error. Examples include software bugs or defects in hardware components.

Error: An error occurs when there is a discrepancy between the actual behaviour and the expected behaviour of a system or a subsystem. Errors can be the result of the activation of a fault in the system.

Failure: A failure occurs when the system fails to perform its required function or behaves contrarily to its design specifications. Failures can occur -but not necessarily- due to unhandled errors.

Node Failure: The event of a node outage where the node stops operating entirely. This could happen due to different root-causes: hardware component failures, software failures, network issues, etc.

Mean Time To Fail (MTTF): The average time to a failure event in a node, a rack or a system (depending on the context). Note that the system MTTF in tightly-coupled HPC platforms is the average time to any node failure in the system, since a failure of any one node involved in the parallel computation requires all nodes to stop running.
Job: The term job in HPC systems refers to a request of resources submitted by a user to the cluster’s job scheduler and usually consists of one or more tasks that execute a user program (application).

b) Description of Different HPC Logs. Different types of datasets are logged and collected at HPC installations: failure records, hardware replacements, error logs, workload logs, and more. We next elaborate on different types of HPC logs and provide some examples for each type.

Node Failure Logs. These are records of all node outages that occur in a HPC system. Each record typically indicates the time when the failure happened and identifiers of the node and the system affected. Additional data fields can include the type of the node and the root-cause of the failure as diagnosed by the hardware engineers or system operators.

Hardware Replacements. Part replacements in HPC clusters are typically tracked and logged by system administrators whenever broken hardware is replaced. These logs can include replacements of faulty memory DIMMs, inoperable or damaged hard-disk drives, server fans, power-supplies, etc.

Hardware Error Logs. In addition to replacement logs, data on errors occurring in different hardware components can be collected as well, regardless of whether these errors led to failures. For example, hard-disk drives (HDDs) are equipped with a self-monitoring facility commonly referred to as SMART (Self-Monitoring, Analysis and Reporting Technology). SMART counters monitor various drive reliability indicators such as read/write errors and latent sector errors (LSEs). Other examples of hardware error logs include records of correctable or uncorrectable errors on DRAM chips.

Workload Logs. Traces of job submissions to HPC clusters can be collected and maintained as well. Such job traces typically include timestamps for each job’s submission time, scheduling time, and exit time; the ID of the user that submitted the job; and the IDs of the nodes (hosts) on which the job was running. Additional information can be collected about the resources requested and consumed by each job (e.g. CPU and memory utilization), and the final status of the job (e.g. completed successfully, failed, terminated by the user, etc).

Machine Layout Files. These files contain information on the physical layout of nodes in a HPC machine, including the positions of the nodes inside the physical racks, and the locations of the racks in the machine room.

Environmental Data. Health-monitoring tools in HPC facilities can periodically poll environmental sensors to log things like ambient temperature, humidity, power quality, or even node internal temperatures using motherboard sensors or HDD SMART tools (if enabled). Monitoring and collecting environmental data is helpful for understanding the impact of environmental factors on system reliability.
Chapter 2. Failure Analysis in HPC Systems

2.2 Related Work

A large body of work in the literature has focused on analyzing the failure behaviour of HPC systems and deriving statistical models that capture the observed failure process [41, 42, 44, 48, 48, 62, 78, 90, 99, 105, 108]. Some of these studies focused primarily on using the derived models to develop failure prediction frameworks [42, 62, 105], while others employed the models to enhance fault-tolerance techniques in HPC systems, particularly checkpoint/restart [44, 48].

Another series of studies focused primarily on the analysis and filtering of system logs collected from supercomputing installations. For instance, examining raw logs and developing log-filtering algorithms to improve the extraction, categorization and diagnosis of failures has been studied by [46, 63, 73, 74, 78, 84, 97, 107]. Understanding correlations between failures in the time and space domains have also been the focus of several papers [41–43, 62, 86, 99]. We next examine each group of related studies more closely.

2.2.1 Failure Characterization

In the past decade, large-scale computing systems emerged as complex hardware and software platforms that enabled the advancement of various high-performance computing applications in vital areas like climate science, computational biology, nuclear energy, and many more. A large body of research has therefore been devoted to understanding, characterizing and modelling the failure behaviour of large-scale systems, which is crucial for maintaining their reliability and dependability.

Many studies viewed the sequence of failure events in large-scale systems as a stochastic process, and studied the statistical properties of the interarrival times of failures (i.e. the elapsed times between failures), to better understand the temporal characteristics of failures [42, 43, 62, 87, 90].

In 2004, Sahoo et al. [87] studied the empirical and statistical properties of failures in a parallel distributed system at IBM T.J. Watson research center, using error logs collected over a year from about 400 machines. Sahoo et al. studied the empirical distribution of the interarrival times of failures from a system-wide point of view and observed a long-tailed probability distribution function, with small inter-arrival intervals being more popular than larger ones, pointing out the bursty nature of failure arrivals in that system. They also studied how the elapsed time since the occurrence of a failure affects the expected time until the next failure – a notion captured by what is known as the hazard rate function of a given distribution. An increasing hazard rate function predicts that if the time since a failure is long then the next failure is expected to occur soon, whereas a decreasing hazard rate function predicts the opposite. Sahoo et al. reported an increasing hazard rate function both when looking at failure arrivals from a system-wide view or for individual nodes [87]. A subsequent study by the same authors [62] analyzed failure logs collected from a production
IBM BlueGene/L cluster. In this work they focused on studying the temporal characteristics of specific types of failures: memory, network, and application I/O failures. They found that network and application I/O failures occurred in bursts over time – a property that memory failures did not exhibit as strongly. They then derived failure prediction models based on their failure analysis.

In 2006, Schroeder and Gibson [90] published a large-scale study of failures in HPC systems that was based on failure data collected over a decade from 22 HPC clusters at Los Alamos National Laboratory (LANL), including 18 Symmetric multiprocessing (SMP) clusters and 4 Non-uniform Memory Access (NUMA) machines. Unlike existing studies at that time, which were often based on a few months of data and covering typically few hundred failures, Schroeder and Gibson’s work was based on what is still considered, to date, the largest failure data set studied in the literature and made publicly available to researchers [6]. The results presented in Schroeder and Gibson’s study were therefore of high significance to the research area of HPC reliability, due to both the long time-period the data spans and the large number of nodes and processors it covers.

The statistical properties of failures in LANL’s systems were thoroughly studied in Schroeder and Gibson’s work [90], starting with how failure rates varied across systems and across individual nodes within systems. Understanding variabilities in failure rates in HPC systems can be utilized for different purposes including job scheduling, for instance, where critical jobs can be assigned to more reliable nodes. From a system-wide view, this study found that failure rates varied across LANL’s systems, even those sharing the same hardware infrastructure. However, when normalizing failure rates by the number of processors a system comprised, the variability reduced significantly, indicating that failure rates did not grow significantly faster than linearly with system size [90].

Schroeder and Gibson [90] also looked at how failure rates varied across individual nodes within a system, and found that some nodes exhibited much higher failure rates than others. This observation agrees with what Sahoo et al. [87] reported when analyzing node-failures at an IBM cluster; they found that less than 4% of the nodes had encountered 70% of the failures. When further investigating the reasons behind this, both Sahoo et al. [87] and Schroeder and Gibson [90] found that the type of workload a node is running plays an important role in a node’s proneness to failures and errors. Sahoo et al. [87] found that the nodes most prone to failures were serving as file servers or database servers. Schroeder and Gibson [90] found that the nodes with highest failure rates in LANL’s systems typically ran different types of workloads than the rest of nodes, such as visualization or graphics workloads; interestingly, when looking at nodes that shared the same workload type (e.g. compute nodes only), they reported finding varying failure rates across these nodes. Both studies [87, 90], however, did not provide further insights into the effect of other possible factors that could contribute to varying failure rates across nodes in HPC systems, besides
workload-type (e.g. location of a node inside the physical rack, ambient temperature, etc).

Schroeder and Gibson [90] also performed a detailed statistical analysis of times between failures at individual nodes, as well as entire systems, in LANL. Using distribution fitting, they found that inter-arrivals of failures both at the node level and system level can be modeled well by a gamma distribution or a Weibull distribution. The exponential distribution was a very poor fit both at the node level and the system level. This finding had an interesting implication. An important property of the exponential distribution is that it is “memoryless”. This means that if failure inter-arrivals followed an exponential distribution, the probability of a failure would always be independent from the elapsed time since the last failure. Schroeder and Gibson found that failure arrivals in HPC systems were not independent and that the exponential distribution was consistently a poor fit to their data across different systems and nodes in LANL [90]. When studying the nature of the hazard rate function, unlike Sahoo et al. [87] who reported increasing hazard rates in their analysis of failure inter-arrivals, Schroeder and Gibson [90] found decreasing hazard rates in LANL’s clusters, with Weibull shape parameter in the range of 0.7–0.8 (i.e. not seeing a failure for a long time meant it is less likely to encounter one in the near future).

### 2.2.2 Failure Prediction

One useful application of studying the failure behaviour of HPC systems is the enhancement of failure prediction models and techniques. The growing size, complexity and dynamics of large scale computing systems has made it very challenging to design proactive fault-management frameworks that are able to predict failures accurately, and take necessary actions, accordingly. Examples of such actions are scheduling proactive checkpoints to avoid or minimize loss of computation, for instance, or migrating jobs away from nodes that are predicted to experience failures in the near future.

A series of studies have looked into using Reliability, Availability, and Serviceability (RAS) logs that are collected at HPC systems to improve failure prediction techniques [44, 62, 86, 105]. Some of the work that studied failure prediction in HPC systems used event-driven approaches, such as [62, 86]. For example, Sahoo et al. [86] used association rules and time-series analysis to predict failures in a 350-node IBM cluster, and Liang et al. [62] used different statistical techniques to capture the event causal correlations and improve failure forecasting in a BlueGene/L system. Alternatively, a period-based approach was studied in [106] where different classifiers are periodically explored and evaluated using BlueGene/L RAS logs. We will elaborate more next on some of these approaches and the effectiveness of the failure predictors presented in these studies.

Sahoo et al. [86] collected and analyzed event logs containing Reliability, Availability and Serviceability (RAS) data from a 350-node IBM cluster, over one year. The input (predictive) variables they fed their prediction models were either system-activity related variables
(e.g. CPU utilization, network utilization), or variables derived from RAS event-logs (e.g. event type, event source, subsystem affected, error severity). The output (response) variables that they were attempting to predict also fell into those two categories: system activity and system errors. They ran different prediction algorithms including time-series models and association-rule algorithms. For system activity prediction, they used time-series analysis techniques. In particular, they ran some common linear time-series models such as Autoregression (AR), Moving Average (MA), and Autoregressive Moving Average (ARMA) models. And to predict rare, severe events in the system, they resorted to association-rule algorithms. By identifying event types that frequently precede such target events within a fixed time window, they derived a probabilistic rule-based prediction model, and tested it for different window sizes. They reported that their algorithms could predict critical, rare events with up to 70% accuracy.

In another study by Liang et al. [62], RAS event logs collected over few months at a BlueGene/L production system are used to develop failure prediction models. The authors studied the effectiveness of predicting failures by observing the sequence of times between failures, for two particular failure types: network and application I/O failures. They reported being able to predict 37% of network failures and 48% of I/O failures in the BlueGene/L system. They also considered the spatial distribution of failures of different types (e.g. memory, network, I/O) over physical racks and midplanes. When observing that some failure types exhibited high skewness in their distribution over physical midplanes, such as network failures, they suggested that failure predictors be ‘location-aware’ in order to anticipate and handle these recurring bursts of failures effectively.

In an alternative approach to event-driven failure prediction, Zhang et al. [106] studied a period-based approach, based on event logs collected from an IBM BlueGene/L cluster. Zhang proposed a moving ‘observation period’ where they observe different parameters extracted from event logs then apply different rule-based classifiers to predict ‘fatal’ events in a given prediction window. A more recent paper by Yu et al. [105] presented a comparison of event-driven and period-based failure prediction approaches for HPC systems. They analyzed and evaluated the two approaches under a variety of testing parameters using RAS logs collected from production HPC systems. Their results suggest that event-driven approaches are more suitable for proactive fault management in large scale systems [105].

Some recent studies focused on utilizing failure predictors in improving checkpointing strategies in HPC systems [44, 48]. For instance, Heien et al. [48] analyzed five years of system logs collected at a production HPC system to model and predict hardware failures on a per-component level, and proposed that their component-aware model be utilized in tuning checkpoint strategies. Heien modeled the distribution of failure inter-arrivals for specific hardware components first (e.g. memory, disk, CPU, or network), then derived a

---

1Each rack in this cluster consists of two panels called ‘midplanes’ which provide slots for connecting the different electronic devices in the rack.
holistic failure model that takes into account the component-usage of the running application and predicts future failures, accordingly. By exploiting how the statistical properties of failures varied across different hardware parts, and assuming knowledge of an application’s component-usage, they hypothesized that their checkpoint model would be more effective than previous approaches [48]. However, they did not run simulations to verify this hypothesis or measure the effectiveness of their model – in terms of application execution time or total wasted work, for instance.

In a more recent study (2012), Gianaru et al. [44] introduced a hybrid approach for predicting failures in HPC systems that is based on both signal-processing concepts and data-mining techniques. The authors used signal analysis to characterize the behaviour of system events and detect outliers in a BlueGene/L system, and data-mining algorithms to extract patterns and correlations among failures. They analytically studied the benefits of deploying their failure prediction module in current checkpointing strategies in HPC systems, but did not run any realistic simulations to demonstrate these benefits.

### 2.2.3 Temporal and Spatial Correlations

As noted earlier, times between failures in HPC systems were found to be modelled well by the Weibull distribution [90]. The exponential distribution, on the other hand, was often found to be a poor-fit. This finding had several important implications for HPC reliability design and analysis as it challenged an existing assumption that HPC nodes failed independently and according to a random Poisson process.

Contrary to that assumption, recent studies that analyzed field data collected from HPC production systems indicated that failures are highly unlikely to be independent, and that both temporal and spatial correlations exist between failures [41–43, 84, 90, 99]. Developing an in-depth understanding of how HPC failures are correlated is therefore crucial as it serves two main purposes. First, it helps create a deeper understanding of their underlying root causes. Second, it helps in the prediction of failures, which is useful, for example, for scheduling application checkpoints or for designing job migration strategies.

Fu and Xu [42, 43] studied temporal and spatial correlations in HPC systems in several papers. Their analysis was based on failure events and system performance variables collected at two sources: a 256-node HPC cluster at LANL [6], and 40 HPC compute servers in the Wayne State University Computational Grid (WSU Grid) [10]. The authors used different clustering algorithms to quantify correlations among failure instances in time and space. They reported a high temporal locality of failure events and attributed that to two main root causes: a) hardware or software faults that can lead to multiple nodes failing within short time intervals of each other, or b) repeat failures that appear multiple times on a single node, before the problem is solved. For spatial correlations, they used what they referred to as an aggregate stochastic model that quantifies the probabilistic dependency of
failures in different nodes, and feeds the resulting statistics to their failure predictor. They report that a small fraction of nodes in HPC systems can encounter the majority of the failures, and that multiple nodes may fail simultaneously. To explain why this tends to happen, they examined job-scheduling logs for the two clusters in their study. They found that in many cases simultaneous failures on multiple nodes can be attributed to bugs or faults triggered by a common application these nodes were running.

Another, more recent study from Google by Ford et al. [41] characterizes the availability of data in Google’s main storage infrastructure observed over the course of a year. They derive statistical models that capture their distributed storage system’s behaviour, while accounting for correlated failure events. They analyze both temporal and spatial correlations using heuristics that they propose which aim at capturing the size of failure bursts in the time and space domains [41]. They report interesting results on how likely failure bursts tend to be associated with a certain domain, such as a physical rack. They found that failure bursts that affect more than 20 nodes typically exhibited a high association with the physical rack. They speculate that these bursts could be attributed to power outages.

2.2.4 Impact of Environmental Factors

Investigating how different environmental factors affect system reliability in HPC installations is one of the main objectives of this thesis. Understanding the effect of temperature on system reliability, for instance, is important as a large fraction of a data center’s energy bill goes into cooling. The quality of power equipment in data centers is another critical aspect as power issues are known to have a negative impact on hardware components (e.g., voltage spikes can cause damage to memory DIMMs or hard disk drives). While many papers have studied failure characterization, failure prediction and failure correlation in HPC platforms, less work has focused on investigating the impact of environmental issues in particular. We summarize here several studies that briefly discussed environmental issues as part of a more general analysis of HPC failure behaviour [80, 90, 94].

Schroeder and Gibson’s study [90] examined the root causes of node failures in 22 HPC clusters. The root causes logged belonged to five high-level categories: environment failures, including power-outages or A/C failures for instance; hardware failures; failures resulting from human-errors; software failures; and network failures. The process of assigning failures to categories in LANL was done by system administrators according to classification rules developed jointly by hardware engineers, administrators and operations staff. The paper mentions examples of failures that were categorized as being ‘environmental’, which included: power outages, voltage spikes, UPS (Uninterruptable Power Supply) equipment failures, and chiller system failures. They did not provide further analysis or insights, however, into whether and how these environmental issues affected the reliability of hardware and software components in LANL’s machines.
Follow-up work by Schroeder et al. [94] conducted a large-scale field study of DRAM failures in particular, using field data from Google. They studied the effect of different factors on both correctable and uncorrectable DRAM errors, including DIMM capacity, utilization, age, and temperature. Their temperature measurements were taken from motherboard sensors on the machines. Since high temperatures are expected to increase leakage current in DRAM chips therefore increasing the probability of flipped bits in the memory array [11,94], quantifying this effect in the field is important. Schroeder et al. [94] reported that out of all the factors that have an impact on DIMM error rates in the field, temperature had a surprisingly small effect—especially when controlling for DIMM utilization.

In another Google study by Pinheiro et al. [80], the authors investigated failure trends in hard disk drives (HDDs) within a large production Internet services deployment at Google, while focusing on evaluating the predictive power of SMART [4] signals in disks (e.g. reallocation counts, scan errors, etc.) on estimating disk failure probability. Their work includes an analysis of the effect of internal disk temperatures on disk failures using SMART logs. Surprisingly, they report a strong drop in disk failure rates with increasing temperature, except for very high temperatures (above 45°C). Overall, they observe little correlation between disk failure rates and both high temperature or high activity levels. This is in contrast with common reliability models, which estimate disk failure rates to increase exponentially with temperature.

Developing a more comprehensive understanding of the impact of environmental factors on HPC system reliability requires conducting more large-scale empirical studies that quantify the magnitude and strength of these factors in the field. (We discuss our contributions to this problem in Section 2.5 where we study various environmental factors using HPC log analysis.)

So far we have reviewed and summarized related research to this thesis in the areas of failure characterization, failure prediction, failure correlation, and the impact of environmental factors on large-scale systems. In the remaining sections of this chapter, we present our own analyses, results and contributions to these different topics.

## 2.3 Understanding HPC Failure Correlations

Our first goal in this thesis is to improve our understanding of failure correlations in HPC platforms by providing intuitive insights into how and why failures of different types are correlated in the field, rather than focusing on deriving statistical failure models.

We next discuss our analysis of how HPC failures are correlated in time and space (Subsection 2.3.1), followed by our study of failure correlations with specific nodes in HPC platforms (Subsection 2.3.2).
2.3.1 Failure-Type Correlations

The first question we address is how HPC failures of different types are correlated with each other in time and space. Discovering correlations between failures in HPC systems serves two main purposes. First, it helps create a deeper understanding of their underlying root causes. Second, it helps in the prediction of failures, which is useful, for example, for scheduling application checkpoints or for designing job migration strategies.

We first describe the data set we used in our study, then present our results of analyzing correlations between failures on the node level, rack level, and system level.

2.3.1.1 Description of HPC field data

Our correlation analysis is based on failure data collected at 10 different high-performance computing (HPC) clusters at Los Alamos National Lab (LANL) over a period of 9 years, and is publicly available at [6]. We divided the 10 clusters into two different groups, based on their hardware architecture. Group-1 includes seven systems that are based on 4-way SMP (Symmetric Multi-Processing) nodes with one or two network interfaces (NICs) and a varying amount of main memory per node. In total these systems have 2848 nodes and 11392 processors. On the LANL web page, where the data is available, these systems correspond to the systems with IDs 3, 4, 5, 6, 18, 19 and 20. Group-2 includes 3 systems that are based on NUMA (Non-Uniform Memory Access) technology and contain a smaller number of nodes, but a larger number (typically 128) of processors per node. In total the systems in group-2 contain 70 nodes and 8960 processors, and correspond to the systems with IDs 2, 16, and 23 on the LANL web page.

For each of the systems the data contains records of all node outages that occurred during the measurement period, including information on the root cause of the node outage, the time when the outage happened and the ID of the node that was affected. The root cause of each failure falls into one of six high-level categories: environment failures, including power-outages for instance; hardware failures; failures resulting from human-errors; software failures; network failures, whenever a node outage is attributed to a problem with the network to which the node is connected; and undetermined, whenever the root cause of the failure is unknown. As mentioned earlier, the process of assigning failures to categories in LANL over the 9 years that the data spans was done by system administrators according to classification rules developed jointly by hardware engineers, administrators and operations staff [90]. Besides the high-level categorization of root causes, for many failures more detailed information is available, such as the hardware component responsible for a hardware failure.

Figure 2.2 shows an example of entries in LANL’s data:

<table>
<thead>
<tr>
<th>Start Timestamp</th>
<th>End Timestamp</th>
<th>System ID</th>
<th>Node ID</th>
<th>Type of Node</th>
<th>Failure Root-Cause</th>
<th>Additional Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1225497600</td>
<td>1225670400</td>
<td>1</td>
<td>1</td>
<td>graphics.fe</td>
<td>Hardware</td>
<td>Memory DIMM Failure</td>
</tr>
<tr>
<td>1225597900</td>
<td>1225660400</td>
<td>6</td>
<td>1</td>
<td>compute</td>
<td>Software</td>
<td>Parallel File System</td>
</tr>
</tbody>
</table>
The LANL logs we include in our analysis contain a total of 20,445 node failures. Table 2.1 below presents a summary of the traces, including the breakdown of the reported root causes behind the failures in LANL’s systems.

<table>
<thead>
<tr>
<th>LANL Group-1</th>
<th>LANL Group-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Nodes</td>
<td>2,848</td>
</tr>
<tr>
<td>Number of Processors</td>
<td>11,392</td>
</tr>
<tr>
<td>Total Processor-Days</td>
<td>13,182,256</td>
</tr>
<tr>
<td>Number of Node Failures</td>
<td>10,707</td>
</tr>
<tr>
<td>Environmental Failures (%)</td>
<td>0.5%</td>
</tr>
<tr>
<td>Hardware Failures (%)</td>
<td>74.5%</td>
</tr>
<tr>
<td>Human Errors (%)</td>
<td>0.46%</td>
</tr>
<tr>
<td>Software Failures (%)</td>
<td>19.5%</td>
</tr>
<tr>
<td>Network Failures (%)</td>
<td>0.48%</td>
</tr>
<tr>
<td>Undetermined (%)</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of the LANL failure data.

In addition to logs of node outages, data on the physical layout of nodes in the machine room is made available for a subset of the systems. In particular, LANL group-1 systems have “machine layout” files that describe the position of each node inside a rack, and the location of a rack inside the server room. For example, Figure 2.3 shows a part of the machine layout file for LANL System with ID 3:

<table>
<thead>
<tr>
<th>Machine 3, bldg 3, room1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ID, RackPos1, RackPos2, PositionInRack, Row facing</td>
</tr>
<tr>
<td>0, 11.06, 2.01, 1, rear to west</td>
</tr>
<tr>
<td>1, 11.06, 2.01, 2, rear to west</td>
</tr>
<tr>
<td>... ... ... ... ... ... ...</td>
</tr>
</tbody>
</table>

Figure 2.3: Sample HPC machine layout file at LANL.

2.3.1.2 Correlations between failures within a node.

In the first part of our correlation study we only focus on correlations between failures in the same node, i.e. we are asking the question whether current failure behaviour of a node is predictive of its future failure behaviour, by asking a specific set of questions:

a) How does a node failure affect the likelihood of later failures?

As a starting point, we calculate the daily and weekly probability of a node failure for group-1 and group-2 LANL systems, i.e. the probability that a random node will fail in a random day/week. We then compare those probabilities against the probability of a node failing during a day or week following another failure (of any kind).

We find that the unconditional probability of a node failure on a random day is 0.31% and 4.6% for group-1 and group-2 systems, respectively. We observe that the daily failure probability is markedly higher during the 24 hours following a prior failure: 7.2% and 21.45% for group-1 and group-2 systems, respectively, which corresponds to roughly a 20X increase and 5X increase for groups 1 and 2, respectively. We observe similar, albeit somewhat
Chapter 2. Failure Analysis in HPC Systems

Figure 2.4: Correlations between failures in the same node: The bars show the probability that any node-failure follows a failure of type X.

weaker trends, for the entire week following a failure: the failure probability of a node in a given week increases from 2.04% to 15.64% in group-1 and from 22.5% to 60.4% in group-2.

b) Does the type of a failure affect the chance of follow-up failures?

Since we have information on the root cause of failures, an interesting question is whether some types of failures increase the probability of follow-up failures more than others. To answer this question Figure 2.4 shows the probability that a given node will fail within the one-week period following a failure of a particular type. The failure type is any of the six different categories of root causes that are distinguished in LANL: Environment, hardware, human error, network, software or undetermined failures. Each bar in the figure corresponds to one of those failure types. To provide a baseline, the right-most bar shows the probability for a node failing on a random week (not necessarily preceded by a failure). Note that the error bars in this figure (and in all other figures in this section) are computed using a 95% confidence level.

Based on Figure 2.4, we make several interesting observations. First, all types of failures increase the probability of failure in the following week, most commonly by factors of 7-10X in group-1 systems and factors of 2-3X in group-2 systems. For some cases, such as network or environmental failures in group-1 systems, the increase in failure probability is more than 10X compared to a random week. We also note that prior failures increase the likelihood of later failures to significant levels. For example, while the probability of failure in a random week is only 2.04% in group-1 systems, chances are 30-50% that a node will experience a failure in the week following a network or environmental failure.

The second interesting observation is that the overall trends are very similar for group-1 and group-2 systems. In both cases the increase in failure probabilities is highest following a network or environmental failure. For group-1 systems a network or environmental failure increases the probability that a node will fail in a given week by a factor of 14-23X, and for group-2 systems it increases the failure probability by a factor of 3-4X.
Figure 2.5: Correlations between failures in the same node: The bars show the probability that a failure of type X follows other node-failures.

We note that the factor increases are in general smaller for group-2 systems, since their baseline probability is higher. The probability for a node to experience a failure in a given week is 22.5% for a group-2 node (compared to only 2.04% for a group-1 node), which means the failure probability can not increase by more than a factor of 5X. One possible reason for the higher failure rates in group-2 systems is that the nodes in those systems are of a different type: they are NUMA nodes with 128 processors per node, compared to SMPs with 4 processors per node for group-1 systems, and the larger component count can lead to higher failure rates. For example, by looking at the failure logs, we find that a node in group-2 systems was 10X more likely to experience failures related to the ‘node board’, than a node in group-1 systems.

c) Does the type of a failure predict the type of a follow-up failure?

Often it might be useful to know what type of failure to expect in the future. For example, are failures of type X usually followed by failures of type Y? To answer this question we computed all pairwise probabilities $p(x, y)$, where $p(x, y)$ is the probability of a failure of type Y in a week following a failure of type X, and compare this to the probability of a type Y failure in a random week.

Our first observation is that a failure always significantly increases the probability of a follow-up failure of the same type, and more so than a random failure. Figure 2.5 shows the probability of a failure of type X in the week following a failure of type X, compared to the week following any type of failure, and compared to a random week. We observe that the increase in the failure likelihood can be dramatic. For example for group-1 systems, the probability of an environmental or a network failure in a given week increases by a factor of around 700X (to absolute values above 7%) if a failure of the same type was observed previously.

Besides correlations between failures of the same type, we notice significant correlations between network, environmental and software problems, i.e. each of these three types in-
creases the follow-up probability of a failure of one of the other two types. We have been in discussions with operators at LANL and have not been able to come up with a clear explanation for these correlations. A closer analysis of the correlations between these three error types revealed that there are a few nodes who happen to have a relatively large number of network, environmental and software problems. It is possible that the correlation is biased by a few nodes that coincidentally had a large number of these three types of failures and does not imply a causal relationship.

\textit{d) How are hardware failures correlated?}

We pay special attention to hardware failures since these are the single most common failure category: 60\% of all failures are attributed to hardware problems. Our data set contains more detailed information on the root cause of hardware failures. The data shows that by far the most common types of hardware failures are due to problems with CPU (40\% of H/W failures) or memory (20\% of H/W failures).

When repeating a correlation analysis similar to the one performed for the high-level failure categories, we find that past failures significantly increase the future probability of memory and CPU failures. In the week following a memory failure the probability of experiencing an additional memory failure is 20.23\% for group-1 systems, a factor of nearly 100X increase over the probability of 0.21\% in a random week. For group-2 systems, the weekly probability of a memory failure increases from 4.2\% to 12.6\%. All increases are statistically significant based on the two-sample hypothesis test.

The strong correlations between hardware-related failures allow us to draw some conclusions about the nature of these failures. Based on discussions with people at LANL, node failures that are attributed to memory or CPU problems are usually due to bit corruption events that go beyond what the built-in ECC can correct. This type of data corruption could either be due to soft errors, which are caused by random events, such as cosmic rays or random noise, or it could be due to hard errors, i.e. problems with the underlying hardware. The strong correlation between those errors points to hard errors as the more likely source of the problem, as one would not expect correlation between random events, such as cosmic rays. We study the impact of cosmic rays on hardware failures in Section 2.5.3.

2.3.1.3 Correlations between failures within a rack

The data for group-1 systems also includes information on the machine room layout, including the rack layout, which allows us to study how failures in different nodes in the same rack are correlated. We begin with the probability of a node failing (with a failure of any type) within a week following a failure (of any type) of another node in the same rack. We find that this probability is 4.6\%, which is more than double the probability of a node failing in a random week (which is 2.04\%). The increase in the daily probability is higher: the failure probability on a day following the failure of another node in the rack is 1.2\%, which is nearly a factor of 3X higher than the baseline probability of 0.31\%.
As we did in the case of correlations within the same node, we also looked at which failure types have the biggest effect on the probability of another node failing later on in the same rack. The results are shown in Figure 2.6 (left). We observe some increase in the failure probability for all types of failures, although with factors of 1.4–3X these are markedly lower than the increase of failures in the same node. Statistical testing with the two-sample hypothesis test allows us to conclude only for software failures that the probability of follow-up failures is significantly increased.

When looking at pairwise correlations, i.e. the probability of a failure of type y within a week of a failure of type x, we find that a failure of a particular type always increases the probability of the same type of failure within the following week. Moreover, this increase is much larger than the increase for the same type of failure following a random failure (i.e. not necessarily the same type). Figure 2.6 (right) summarizes our results. We observe an increase in failure probability as high as 170X for environmental failures and nearly 10X for software failures. All increases are statistically significant based on the two-sample hypothesis test.

Finally, we take a look specifically at hardware failures as these are the most common type of failure. We find that both memory and CPU failures experience a significant increase in probability in the day or week following another failure of the same type. This observation provides some room for hypotheses explaining the cause of such errors. One possible explanation might be that nodes in the same rack share similar environmental factors, such as the quality of the supplied power. This observation, combined with the strong effect of environmental failures on the frequency of follow-up failures, motivates us to study power related failures in more detail in Section 2.5.1.
2.3.1.4 Correlations between failures in the same system

In this section we ask the question of whether and how failures between different nodes in the same system (not necessarily in the same rack) are correlated. We find that the weekly probability of a node experiencing a failure does increase after another node in the same system had a failure, however the increase is significantly smaller than for nodes in the same rack: in group-1 systems the weekly probability of a node experiencing a failure increases from 2.04% to 2.68% and for group-2 systems it increases from 22.5% to 35.3%. Both increases are not significant enough to allow the rejection of the hypothesis that a node failure does not increase the likelihood of follow-up failures in nodes within same system, based on the two-sample hypothesis test.

The results are more interesting when breaking them down as a function of the failure type. Figure 2.7 shows the probability that a node in a system will fail within a week following a failure of type X (where X can be: environment, hardware, human-errors, network, software, memory, CPU failures, or undetermined). We observe that software, hardware and human failures in a node in group-1 systems increase the probability that also other nodes in the system will see failures. The increase following software failures (a factor of 1.27X) is statistically significant based on the two-sample hypothesis test. For group-2 systems, all types of failures show an increase in Figure 2.7, but by far the biggest increase, with a factor of 3.69X, is observed following a network failure. The two-sample hypothesis test allows us to show that all failure types, except hardware and human, increase the chance of follow-up failures in other nodes significantly.
2.3.2 Failure-Prone HPC Nodes

As discussed in our review of failure characterization studies back in Section 2.2.1, several studies reported evidence of failures being correlated with specific nodes in HPC clusters [87, 90]. These studies found that failure-prone HPC nodes typically ran different types of workloads—workload levels—than the average node in the cluster. Both studies however did not further discuss the types of failures exhibited in these nodes or how they fail differently than the rest of the nodes. In this section we ask the question of whether all nodes in LANL’s systems are equally likely to fail, or whether some nodes are more prone to failures than others. We further examine how such failure-prone nodes fail differently and which types of failures are more likely to affect them.

2.3.2.1 Do some nodes fail more frequently than others?

Figure 2.8 shows the total number of failures for each node in systems 18, 19 and 20 (the three largest systems of all LANL systems in terms of number of nodes: 1024, 1024 and 512 nodes, respectively). The graphs show that in all systems a single node (the node with ID 0) had significantly more failures than rest of nodes. For example, for system 20 node 0 reported 19 times more failures than the average node and for system 19 node 0 reported more than 30 times higher failure rates than the average node. To test the significance of differences between failure rates in nodes, we performed chi-square tests for differences between proportions: with 99% confidence level we are able to reject the null hypothesis that all nodes in each system had equal failure rates (p-value < 2.2e-16). Interestingly, even when repeating the same analysis after removing node 0 we can still reject the hypothesis that all nodes in each system had equal failure rates.

2.3.2.2 Failure characteristics of failure-prone nodes.

We are interested to find out whether the increased number of failures in some nodes is due to an increased number of failures of a particular type or due to generally increased failure rates. To answer this question we compare in Figure 2.9 the relative breakdown of the different failure types for failure prone nodes against the remainder of the system, and
Figure 2.10: The probability of different failure types in failure prone nodes compared to the rest of the nodes in a system.

We compare in Figure 2.10 for each failure type the probabilities of a node failure of this type in failure prone nodes vs the rest of the nodes in the systems. In Figure 2.10 each plot contains three pairs of bars for each of the three systems, where each pair corresponds to a timespan: day, week or month. The numbers on top of the bars indicate the factor increase in failure probability in a failure prone node compared to an average node.

The first observation we make based on Figure 2.10 is that node 0 exhibits increased failure probabilities for all types of failures, so the higher failure rate in those nodes cannot be attributed to a particular type of failure. However, we observe that the increase in failure probabilities is particularly high for environmental and network failures, with factors of increase in the 2000x and 500x-1000x range, respectively. Software failure rates are also significantly higher in node 0 than the remainder of the system (factors of 36X up to 118X). The increase in the probability of hardware failures is modest in comparison, but still significant with factors in the 5–10X range. To formalize our results we repeat the chi-square test for differences between proportions separately for each failure type. The only failure type where we fail to reject the null hypothesis that nodes fail with equal rates is for failures due to human errors; for all other failure types the test rejects the null hypothesis with 99% confidence.

Turning to Figure 2.9, which shows the relative breakdown of failures by root cause for the failure prone nodes compared to the whole system, we observe a higher percentage of software, environment and network failures in the failure prone nodes. This observation is in agreement with our findings in Figure 2.10, which indicate that those three failure
types have a higher factor increase in the failure prone nodes than other failure types. It is interesting to note that in the failure prone nodes the dominant failure mode shifts from hardware failures to software failures.

Discussion: Why do some nodes fail more frequently than others? One might wonder what the reason for the high variability in failure rates between nodes in the same system is, in particular since all nodes within a system typically use the same type of hardware. One possible explanation are statistical effects due to the strong correlations between failures in the same node (recall Section 2.3.1.2). Once a node is “unlucky” and starts to develop failures, a large number of correlated follow-up failures might bring the total failure rate of a node way above the average.

Another hypothesis we investigated is the effect of a node’s position in the machine room. For a few systems where we had information on the layout of nodes in the machine room we checked whether the location in the machine room or the location of a node within a rack played any role, but we could not find any clear patterns that certain areas in the machine room were more likely to be correlated with higher error rates.

Another hypothesis we tested is whether usage has an effect on the failure rate of a node and whether the failure prone nodes were used differently from other nodes. We discuss our analysis of usage in the next section.

2.4 Impact of Usage and Utilization

The effect of system workload on system reliability was studied in a series of papers by Iyer et al. [53, 54] and Castillo et al. [25]. However, these papers date back to the early 1980’s and don’t necessarily translate to modern HPC systems. To better understand the impact of usage on modern HPC systems we use LANL’s field data to study the effect of node utilization on its long term reliability, and also to investigate whether some HPC users more prone to node failures than others. We address both objectives in the following two subsections.

2.4.1 Effect of Usage on Node Reliability

To study whether the way a node is used affects its failure rate, we use job logs that are made publicly available by LANL for two of their HPC systems: system 8 (where we have a total of 763,293 job records), and system 20 (with a total of 477,206 job records). These two systems are representative of two larger groups of LANL systems, where all systems within the same group shared a similar hardware architecture and ran very comparable workloads.

We consider the effect of two simple usage metrics, one is the average node utilization (where we define a node as being utilized if at least one job is currently assigned to it,
and idle otherwise) and the other one is the number of jobs that were scheduled on a node throughout its lifetime. We begin by plotting the number of failures a node experiences against the node’s average utilization (see Figure 2.11-(a)) and against the number of jobs served by the node (see Figure 2.11-(b)). We have marked nodes with particularly high failure rates with special markers. This includes node 0, which we discussed in the previous section.

We observe that in both systems where we have usage information available the failure prone node 0 tends to be among the nodes with the highest utilization and the largest number of jobs assigned to it. We formalized our observation by looking at the Pearson correlation coefficient between the number of jobs assigned to a node and the number of failures experienced by the node. For both systems we observe clearly positive correlation coefficients of 0.465 and 0.12, respectively. However, repeating our analysis after removing node 0 reduces the correlation to insignificant levels, which lets us conclude that the strong linear correlation between usage and failures is mostly due to node 0. In discussions with operators at LANL we have been told that node 0 in most systems has a special role where it is used as a login node for users and/or is used to schedule and launch jobs.

Figure 2.11: The impact of usage on node reliability.
2.4.2 User Proneness to Failures

As a follow-up question on the relationship between usage and failures we used the job logs to test whether certain users are more likely to experience job failures than others. We only include job failures that are caused by failures in one of the underlying nodes, rather than a failure of a user’s application software. The two systems that have job logs available (systems 8 and 20) both have more than 400 different users. For each system, we focus on the 50 heaviest users in terms of the number of processor-days that they used on those systems.

The two graphs in Figure 2.12 show for each of the 50 heaviest users the average number of failures this user experienced per processor-day that this user utilized the system. Visual inspection shows a large discrepancy between the failure rates experienced by different users. We also formally verified that the difference in failure rates between users is statistically significant by using Poisson regression to fit a full (saturated) model (with users’ actual failure counts and usage periods), and a common failure rate model (where all users have the same failure rate). We then applied Analysis of Variance (ANOVA) test and found with 99% confidence level that the saturated model is significantly better than the common rate model, in both LANL systems.

In conclusion, we find that the way a node is exercised affects its failure rates. This might for example be because some users run applications that are more likely to exercise a buggy code path in some system software or because their application is more likely to exercise a hardware component in an access pattern that makes intermittent or hard errors more likely to manifest themselves.
2.5 Impact of Environmental Factors

Our goal in this section is to investigate how different environmental issues affect node reliability in HPC systems using field data analysis. While some anecdotal evidence from HPC operators points to the effect of factors such as room temperature, humidity and power quality on HPC reliability, there is a strong lack of large-scale empirical studies that quantify the magnitude and strength of these factors in the field.

In the remaining parts of this section, we take a closer look into these environmental issues, while focusing our analysis on three specific factors that are commonly reported by HPC practitioners to have an impact on system reliability: (1) power quality, (2) temperature, and (3) cosmic radiation.

2.5.1 Power quality and Reliability

\textit{Motivation.} In our analysis of LANL’s failure logs in Section 2.3.1, we observed that node failures categorized as ‘environmental’ caused a steep increase in the probability of follow-up failures. For example, a node with an environmental failure has a chance of 47.2% and 69.4% for group-1 and group-2 LANL systems, respectively, of experiencing another failure within a week. A closer look into what these environmental failures are shows that the majority of them are related to problems with power in the data center, in particular either power outages, power spikes or UPS failures:

![Breakdown of environmental failures in LANL systems](image)

In the remainder of this section we study how these power issues affect the two most common types of failures, hardware and software failures. In addition to power outages, spikes and UPS failures recorded as part of environmental failures, we also take into account the effect of problems with the power supply unit (PSU) of individual servers, which are recorded as hardware problems in LANL’s data.
2.5.1.1 How do power problems affect hardware failures?

Figure 2.14 (left) shows the probability that a node will experience a hardware failure within a day (left-most set of bars), a week (middle set of bars) and a month (right-most set of bars) after experiencing a power outage, a power spike, a power supply failure or a UPS failure, compared to the probability of a hardware failure in a random day, week, month (i.e. not necessarily preceded by a power issue).

We observe that generally after power issues the probability of seeing hardware failures in LANL nodes is significantly increased. Interestingly, while power outages and power supply failures caused a significant increase in hardware failures both in the short-term (within a day following the power problem) and in the long-term (within a month following the power problem), the effect of power spikes is more apparent at longer timespans. In the long-term, all four types of power issues lead to an increase in the hardware failure probability by factors of 5-10X.

What types of hardware failures are most affected by power problems? Figure 2.14 (right) shows the probabilities for different types of hardware failures to occur within a month of a power outage, power spike, power supply failure or a UPS failure, compared to the probabilities of those failures in a random month (not preceded by power issues).

We observe that a large range of hardware components, including memory DIMMs, node boards, and power supplies, show markedly increased failure rates following power problems. The only component that showed no clear signs of increased failure rates after any of the power problems are CPUs.

For the other components the degree at which failure rates increase depends on the type of power problem that preceded. After power outages the node board and power supply show the biggest increase in their failure rates (factors of 16-20X). These components also show similar failure rates following power spikes. Memory DIMMs show a higher failure rate...
following power spikes, compared to power outages, with an increase of 13.7X compared to 5X. For all components the increase in failure rates is strongest following a power supply failure, and ranges from more than 40X for fans and power supplies, to 14X and 28X for memory DIMMs and node boards. Two components show high failure rates following UPS failures: node boards (27.3X increase) and memory DIMMs (8.9 increase).

Do power problems cause issues in addition to node failures? When analyzing the LANL data to investigate the consequences of power problems, we also made another interesting observation. In addition to the clearly increased number of node outages due to failures following a power problem, we observe a large increase in the number of non-scheduled maintenance events related to hardware problems. Within a month after a power outage or power spike, around 25% of affected nodes need to undergo unscheduled downtime due maintenance. This is an increase of nearly 90X in the frequency of unscheduled maintenance compared to a random month in a node’s lifetime. In the month after a power supply failure maintenance activity is also markedly increased: a node has an 8% chance of requiring hardware-related maintenance work within a month after a power supply failure, which is lower than after a power outage or spike, but still nearly 30X higher than in a random month.

Failures in the UPS system had the strongest effect, increasing a node’s chance of undergoing unscheduled maintenance by a factor of 100X (28% chance). These results indicate that problems with power not only lead to hardware problems that cause a node to fail, but also a significant amount of downtime due to unscheduled maintenance.

2.5.1.2 How do power problems affect software failures?

We now turn our attention to study the impact of power issues on software failures in HPC systems. Figure 2.15 (left) shows the probability that a node will experience a software failure within a day (left-most set of bars), a week (middle set of bars) and a month (right-most set of bars) after experiencing a power outage, a power spike, a power supply failure or a UPS failure, compared to the probability of a software failure in a random day, week, month (i.e. not necessarily preceded by a power issue).

As was the case for hardware failures, we observe that after power issues the probability of seeing software failures in LANL nodes is significantly increased. We observe the strongest effect for power outages and UPS failures, which increase the probability of a software failure within a week by factors of 45X and 29X, respectively. Power spikes and power supply failures had a somewhat weaker effect, with factors of 10-20X, but still very strong. All four types of power problems show longer-term effects, as evidenced when looking at the software failure rates following the month of power problem, although the effects are weaker than the weekly ones (except for UPS failures).
What types of software failures are most affected by power problems? Figure 2.15 (right) shows a breakdown of software failures into their more detailed underlying root causes and for each of these underlying root causes the associated probability within a month after a power outage, power spike, power supply failure and UPS failure. We observe that the majority of the software-related outages following power issues are related to the system’s distributed storage system (DST). Some additional issues are related to Parallel File System (PFS) and the Cluster File System (CFS).

In summary, we observe that a large fraction of software issues created by power problems are related to data storage (either the distributed storage system or the file system), rather than general operating system issues or other software issues. While the data does not provide details on the nature of those storage and file system failures, the loss of power likely led to some inconsistency in the storage or file system state. All file and storage systems for HPC installations provide mechanisms to protect against loss of consistency or persistence in the case of crashes or power outages, so it is interesting to observe that despite those efforts power problems still remain a high risk factor for those systems.

How are power problems laid out in time and space?

To better understand the nature of power problems in HPC systems, we take a closer look into how these power failures occurred over time and across nodes in a system.

Figure 2.16 illustrates how the four different types of power problems (outages, spikes, UPS and power supply failures) are laid out in time and space using the data for all System 2 nodes. We present the results here for System 2 as it provides the largest data set on power issues. More precisely, the nodes in this system were responsible for approximately 50% of all power related failures reported in LANL’s dataset. We observe that the different power problems vary in how they are correlated in time and space. While power outages and UPS failures show clear correlations between nodes and also over time within the same node, power spikes tend to happen in more random unpredictable ways. Power supply failures are
the most common type of power-related failure and show only correlations within the same node. It is worth noting that when repeating this analysis for the rest of LANL systems in our data that reported power failures, we observe similar patterns.

### 2.5.2 Temperature and Reliability

**Motivation.** The second environmental factor we study is temperature. Developing a good understanding of how temperature affects system reliability is crucial for running large data centers efficiently. The world’s data centers are estimated to consume power equivalent to about seventeen 1,000 MW power plants, equaling more than 1% of total world electricity consumption, and to emit as much carbon dioxide as all of Argentina [59]. More than a third, sometimes up to one half of a data center’s electricity bill is made up by electricity for cooling [17,61]. For instance, for a data center consisting of 30,000 square feet and consuming 10MW, the yearly cost of running the cooling infrastructure can reach up to $4-8 million [76].

Not surprisingly, a large body of research has been devoted to reducing cooling cost. Approaches that have been investigated include, for example, methods to minimize air flow inefficiencies [76, 98], or incorporating temperature awareness into workload placement in data centers [20, 45, 79, 81, 95].

Interestingly, one key aspect in the thermal management of a data center is still not very well understood: controlling the setpoint temperature at which to run a data center’s cooling system. Data centers typically operate in a temperature range between 20C and 22C, some are as cold as 13C degrees [21,85]. Due to lack of scientific data, these values are often chosen based on equipment manufacturers’ (conservative) suggestions. Some estimate that increasing the setpoint temperature by just one degree can reduce energy consumption by 2 to 5 percent [21, 22].

While increasing temperatures might seem like an easy way to save energy, it comes with some concerns, the most obvious being its impact on system reliability. Unfortunately, the details of how increased data center temperatures will affect hardware reliability are
not well understood and existing evidence is seemingly contradicting. A recent study [98] indicated that in order to avoid thermal redlining, a typical server needs to have the air temperature at its front inlets be in the range of 20°C – 30°C. Every 10°C increase over 21°C decreases the long-term reliability of electronics by 50% [77]. Other studies show that a 15°C rise increases hard disk drive failure rates by a factor of two [13, 26]. On the other hand, a recent Google study [80] suggests that lower temperatures are actually more detrimental to disk reliability than higher temperatures. Theoretically, the most commonly cited model that relates hardware component aging and temperature is based on the Arrhenius equation [55] which predicts an exponential increase in failures as a function of temperature.

The goal of this section is to study the effect of temperature on various aspects of hardware reliability by analyzing a diverse set of field data collected at three different organizations: Google, Los Alamos National Labs (LANL), and SciNet (Canada’s largest supercomputing consortium). The data spans several dozen data centers and covers a diverse set of common reliability issues, including hard disk failures, latent sector errors in hard disks, uncorrectable errors in DRAM, DRAM replacements, and general node outages.

We first focus on two specific hardware components, hard disks (Section 2.5.2.1) and DRAM (Section 2.5.2.2), since these are among the most frequently replaced components in modern data centers [90,93]. Then, in Subsection 2.5.2.3 we use data on node outages in data centers to study the effect of temperature on overall server reliability.

2.5.2.1 Temperature and Hard-Disk Reliability

We focus our study on two common failure modes of hard disks: latent sector errors and complete disk failures.

a) Temperature and Latent Sector Errors

Background and Data. Latent sector errors (LSEs) are a common failure mode, where individual sectors on a disk become inaccessible, and the data stored on them is lost (unless the system can use redundancy mechanisms to recover it). LSEs happen at a significant rate in the field [15,80], with 3-4% of all drives experiencing them at some point in their life, and are expected to grow more common as disk capacities increase. While recent work [15] has studied the prevalence and some statistical properties of LSEs, there is no prior work on how temperature affects this important error condition.

To study the effect of temperature on the prevalence of LSEs, we obtained data collected from January 2007 to May 2009 at 7 different data centers (DCs) at Google covering three different disk models. For each of the disks, we have monthly reports of the average (internal) disk temperature and temperature variance in that month, the count of latent sector errors, the number of read and write operations during that month, and the age of the disk. All data were collected by polling the disks’ internal self-monitoring facility (SMART). It is
worth noting that the SMART tool reports the number of LSEs detected in a disk-month by keeping track of the I/O operations that produced an error due to a damaged sector. More detailed information about the measurement infrastructure and methodology Google uses to collect such data are described in Pinheiro et al. [80].

The table below summarizes our data:

<table>
<thead>
<tr>
<th>Model ID</th>
<th>#Data Centers</th>
<th>#Disks</th>
<th>#Disk Months</th>
<th>Avg. monthly LSE probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>18,692</td>
<td>300,000</td>
<td>0.0063</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>17,515</td>
<td>300,000</td>
<td>0.0177</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>36,671</td>
<td>300,000</td>
<td>0.0067</td>
</tr>
</tbody>
</table>

Table 2.2: Overview of Google LSE dataset.

**Analysis.** Figure 2.17-(a) shows for each of the three models the monthly probability of a disk experiencing an LSE as a function of the average temperature. The error bars in this figure (and in all other figures in this work) are computed using a 95% confidence level; larger bars for higher temperatures are due to lack of data. Since there are many data
center-specific factors beyond temperature that might affect reliability (workload, humidity, power spikes, handling procedures, etc.), we also break down the results for each model by data center. The three graphs in Figure 2.17-(b) show the monthly LSE probabilities for the three models, where each line corresponds to a different data center.

As one might expect, we observe a trend of increasing LSE rates as temperature rises. However, the magnitude of increase is much smaller than expected based on common models and estimates, in particular when isolating the instances of LSEs per model per data center. Models for the effect of temperature on hardware components usually assume an exponential increase in failures as a function of temperature (based on the Arrhenius equation [55]), and predict roughly doubling failure rates for every 10-15°C increase in temperature [13, 26, 98]. Visual inspection of our graphs shows for only 5 out of the 10 model/data center combinations a clear increase in errors with temperature: model 3, data center 2; model 4, data centers 8 and 9; model 6, data centers 3 and 5. We also observe that the increase in error rates tends to be linear, rather than exponential, except for very high temperatures (above 50°C).

To formalize our observation above, we fitted two different models to the data. The first is a simple linear model, i.e. we try to model the error rate \( y \) as a function of temperature \( t \) as \( y = a_1 + a_2 \cdot t \). Since one of the most common models for effects of temperature on hardware reliability, the Arrhenius model, is an exponential one, we also fit an exponential model to our data, i.e. we model the failure rate \( y \) as a function of temperature \( t \) as follows: \( y = a_1 \cdot e^{-a_2/t} \). The detailed results (including values for the parameters \( a_1, a_2, b_1, b_2 \), and the corresponding sum of squared errors (SSE)) are presented in Table 2.3. We find that in all cases the linear model provides a fit of comparable or even better accuracy, as measured by the SSE. The only exception is model 3, data center 2, where the exponential model provides a better fit. We attribute this to the sudden increase in LSEs for temperatures above 50°C. When repeating our analysis for only data points below 50°C, also for model 3, data center 2, the linear model provides a better fit.

**Observation:** For the temperature range that our data covers with statistical significance (< 50°C), the prevalence of latent sector errors increases much more slowly with temperature, than reliability models suggest. Half of our model/data center pairs show no evidence of an increase, while for the others the increase is linear rather than exponential.

In addition to comparing the quality of the linear versus the exponential fit, it is interesting to look at the slope of the linear increase in errors (parameter \( a_2 \)), i.e. the rate at which errors increase. One interpretation of \( a_2 \) is that it gives the additional fraction of drives that will develop LSEs for each 1 degree increase in temperature, e.g. \( a_2 = 0.01 \) means that for a 1 degree increase in temperature an additional 1% of the drive population in a data center would develop LSEs in a given month (that would not have had LSEs otherwise). We find that for 4 of the 10 model/data center combinations \( a_2 \) actually has a small negative value,
indicating a small decrease in error rates with temperature. For the remaining positive values, it is important to put the value of $a_3$ in relation to the average probability of a drive developing an LSE (provided in the third column in Table 2.3). Studying the values of $a_2$ for those cases where it is positive, we see that $a_2$ is always at least an order of magnitude smaller than the average LSE probability for that model/data center combination. That means the fraction of drives in the population that will develop LSEs due to a one degree increase in temperature, will be an order of magnitude smaller than the average observed in the dataset. However, an increase in the range of ten degrees or more in data center temperature would probably warrant some extra measures to protect against data loss due to LSEs.

In addition to the average temperature that a drive is exposed to, another important factor is the variability in temperature, since large variations in temperature can negatively affect IT equipment. To study the impact of temperature variability on LSEs we plot the monthly LSE probabilities as a function of the coefficient of variation (CoV) ² (see Figure 2.18). We chose the CoV, rather than variance or standard deviation, since it is normalized by the mean. A positive correlation between LSEs and temperature variance could just be due to the positive correlation between LSEs and mean temperature. Figure 2.18 shows a clear increase in LSE probabilities with increasing CoV for all models. We verify those visual trends by fitting a linear model to capture the relationship between LSEs and the CoV, and find a positive slope ($a_2$) for all model/data center pairs.

Observation: The variability in temperature tends to have a more pronounced and consistent effect on LSE rates than mere average temperature.

Our analysis so far has exclusively focused on the probability of a drive developing LSEs. Another interesting question is whether higher temperature leads to a higher number of LSEs, once a drive starts developing LSEs. To answer this question Figure 2.19 plots for

<table>
<thead>
<tr>
<th>Model</th>
<th>DC</th>
<th>Monthly Probability</th>
<th>Linear fit</th>
<th>Exponential fit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$a_1$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$7.99 \times 10^{-3}$</td>
<td>$-2.726 \times 10^{-2}$</td>
<td>$7.664 \times 10^{-4}$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$2.93 \times 10^{-3}$</td>
<td>$7.519 \times 10^{-2}$</td>
<td>$-1.055 \times 10^{-4}$</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>$2.51 \times 10^{-3}$</td>
<td>$7.322 \times 10^{-3}$</td>
<td>$-1.111 \times 10^{-4}$</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>$2.08 \times 10^{-2}$</td>
<td>$-5.614 \times 10^{-2}$</td>
<td>$1.755 \times 10^{-3}$</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>$1.73 \times 10^{-2}$</td>
<td>$-2.346 \times 10^{-2}$</td>
<td>$9.482 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

Table 2.3: Parameters from fitting linear and exponential models to monthly LSE probabilities as a function of avg. temperature.

²Recall that the coefficient of variation is defined as the standard deviation divided by the mean.
Figure 2.18: The monthly probability of LSEs as a function of variability in temperature, measured by the coefficient of variation.

those disk months that have errors the 25th and 75th percentile, and the mean. (We only include results for model 6, all others have comparable trends). We focus on the quartiles, rather than the mean, since we find the mean number of LSEs to be highly variable and hence easily biased by outliers. We observe that the line for all quartiles is flat, indicating that hotter drives with errors do not experience a higher frequency of errors than colder drives with errors.

Observation: Higher temperatures do not increase the expected number of LSEs once a drive develops LSEs, possibly indicating that the mechanisms that cause LSEs are the same under high or low temperatures.

Figure 2.17 provides another interesting observation: The rate of LSEs for the same model can vary greatly across data centers. For example, model 3’s error rate is significantly higher (more than 2x difference) for data center 2 than for the other data centers, and model 6’s error rates are significantly higher for data center 0 than for other data centers (again, more than 2x difference). This brings up the question whether factors, such as environmental
conditions or the age or usage of a drive affect how it reacts to temperature. While we have no data on environmental factors, such as humidity or the quality of the power, we have information on the age of each drive and its utilization and study the effect of those factors in Figures 2.20 and 2.21.

Our study of age and temperature in Figure 2.20 focuses on model 6, since the disks for this model span the widest range in age. We divide the drives into two groups, those that are less than 18 months old and those that are 18-36 months old, and plot LSE probabilities as a function of temperature separately for each group. We find that both lines show similar trends with no evidence that older drives are more sensitive to higher temperatures.

Observation: Within a range of 0-36 months, older drives are not more likely to develop LSEs under temperature than younger drives.
Figure 2.21 studies the effect of workload intensity. Figure 2.21 (left) divides disks into two groups, one with high read utilization and one with low read utilization, and plots the LSE probabilities separately for the two groups. We measure read utilization by the number of read operations per month and assign a disk to the low read utilization group if the number of read operations is below the median for the dataset, and to the high read utilization group otherwise. Figure 2.21 (right) performs the corresponding analysis for write utilization. Results are shown for model 6 only, but trends were similar for other models as well.

We find that drives with higher utilization are not more sensitive to higher temperatures. That is an interesting result beyond the study of temperature effects, as it has been an open question as to how workload intensity affects LSEs. Methods that are intended to protect against data loss due to LSEs, such as running a periodic “scrubber” that reads the entire disk to proactively detect LSEs, place additional load on a system, and a concern is that this added load might increase the rate of LSEs. Our results indicate that such worries are, likely, unfounded – particularly for conventional mechanical hard drives.

**Observation:** High utilization does not increase LSE rates under temperatures.

To add statistical rigour to our observations above on disk age and utilization effects, we performed an ANOVA test. The results indicate no correlation between LSEs and write utilization. There is evidence for a correlation with read utilization and age, however this is due to drives with lower read utilization and lower age experiencing slightly increased rates of LSEs.

### a) Temperature and Whole Disk Failures.

**Background and Data.** Hard disk failures include any kind of disk problems that are considered serious enough to replace the disk in question. Hard disk failures are a serious condition since they create the potential for data loss and happen at a significant rate: typically 1-5% of drives in a data center need to be replaced in a given year [80,92]. The only existing work that includes trends for the effect of temperature on hard disk failures based on field data is the work by Pinheiro et al. [80]. Surprisingly, this work found a strong drop in disk failure rates with increasing temperature, except for very high temperatures (above 45C). This is in contrast with common reliability models, which estimate disk failure rates to increase exponentially with temperature.

The goal of this section is to revisit the question of how temperature affects disk failure rates. In addition to obtaining a more conclusive answer to this question, we also look at the question from a broader angle, studying the effect of utilization, differences between models and data centers, and the age of a disk.

For our study, we have obtained data on disk replacements collected from January 2007
Figure 2.22: The monthly probability of a disk failure as a function of temperature separated by disk model.

to May 2009 at 19 different data centers (DCs) at Google covering 5 different disk models. For each disk we know the age of the disk, the average temperature and average utilization over the observation period as reported by the drive’s SMART system, and whether the disk was replaced during the observation period. While the time period is different from the study in [80] (there is actually no overlap in time), the measurement methodology and infrastructure used to collect the data is the same as the one Google used in their study.

The following table provides some summary information:

<table>
<thead>
<tr>
<th>Model</th>
<th>#DCs</th>
<th>#Disks</th>
<th>#Disk Months</th>
<th>Monthly disk fail prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>7972</td>
<td>173,945</td>
<td>0.0028</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5906</td>
<td>143,456</td>
<td>0.0023</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>93498</td>
<td>752,579</td>
<td>0.0004</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>69421</td>
<td>829,859</td>
<td>0.0011</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>95226</td>
<td>2,953,123</td>
<td>0.0012</td>
</tr>
</tbody>
</table>

Table 2.4: Overview of Google Disk Failure Logs.

**Analysis.** Figure 2.22 plots the monthly failure rate for each of the five models averaged across all data centers. Except for one model (model 3) we observe increasing failure rates with rising temperature. However, we observe that the increase in failures with temperature tends to be linear rather than exponential, except for very high temperatures (above 50°C). We validate this observation by fitting a linear and an exponential model to the data, following the same methodology as described in Section 2.5.2.1. Results are shown in Table 2.5. Since the slope of the curves tends to change for very high temperatures, we also repeated the analysis by including only data points below 50°C (see right half of Table 2.5). We find that in all cases the linear model provides a significantly better fit than the exponential model.
As explained in Section 2.5.2.1, when studying the rate at which failures increase with temperature (as given by the $a_2$ parameter) it is important to put the amount of increase in failures, in relation to the average failure rate in a system. When looking at the values for $a_2$ when fitting the linear model to data points below 50C (see Table 2.5), we notice that for all model/data center combinations $a_2$ is by two orders of magnitude smaller than the average failure rate (with the exception of one data point, model 4, data center 15). While average monthly failure rates are typically on the order of 0.1-0.2%, the additional fraction of drives one would expect to fail for each degree increase in temperature is on the order of one thousandth of a percent.

**Observation:** For temperatures below 50C, disk failure rates grow more slowly with temperature than common models predict. The increase tends to be linear rather than exponential, and the expected increase in failure rates for each degree increase in temperature is small compared to the magnitude of existing failure rates.

We also note that, unlike the Google study [80], we do not see a general trend for higher failure rates at lower temperatures. For example, the Google study reports more than a 50% drop in failure rate when moving from 25 to 35C. We believe that the reason is the aggregation of data for different models and data centers in the same curve in [80]. Since different drive models run at different temperatures (due to differences in their design) and different drive models can also vary greatly in their failure rate, it is possible that the data points at the lower end of the temperature spectrum contain more drives of a model that happened to run colder and have higher failure rates, hence biasing the results.

As was the case for LSEs, we find that for the same model, the monthly failure probabilities can vary greatly across data centers, even for the same temperature. This points to other factors, beyond temperature, that have an equally strong or stronger effect on disk lifetimes and motivates us to study two possible factors that we have data on: age and utilization. We followed the same methodology as for LSEs, and divided the drives for each model into those with high and low read utilization, high and low write utilization, and

<table>
<thead>
<tr>
<th>Model</th>
<th>DC</th>
<th>Monthly Prob.</th>
<th>Linear fit</th>
<th>Exponential fit</th>
<th>Monthly Prob.</th>
<th>Linear fit</th>
<th>Exponential fit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$a_1$ ($10^{-3}$)</td>
<td>$a_2$ ($10^{-5}$)</td>
<td>SSE ($10^{-2}$)</td>
<td>$b_1$</td>
<td>$a_2$ ($10^{-5}$)</td>
<td>SSE ($10^{-2}$)</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>2.79</td>
<td>-6.38</td>
<td>1.958</td>
<td>6.67</td>
<td>3.222</td>
<td>-116.4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3.32</td>
<td>-6.692</td>
<td>2.157</td>
<td>12.30</td>
<td>4.605</td>
<td>-123.3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1.60</td>
<td>-4.042</td>
<td>1.481</td>
<td>5.488</td>
<td>4.093</td>
<td>-128.8</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0.625</td>
<td>0.5464</td>
<td>0.025</td>
<td>0.3250</td>
<td>0.076</td>
<td>-7.242</td>
</tr>
<tr>
<td>1</td>
<td>0.869</td>
<td>0.9480</td>
<td>-0.0183</td>
<td>0.9065</td>
<td>0.06928</td>
<td>7.947</td>
<td>0.9194</td>
</tr>
<tr>
<td>2</td>
<td>0.919</td>
<td>2.555</td>
<td>-0.455</td>
<td>0.7095</td>
<td>0.0179</td>
<td>54.33</td>
<td>0.8768</td>
</tr>
<tr>
<td>3</td>
<td>1.45</td>
<td>-1.172</td>
<td>0.5886</td>
<td>6.440</td>
<td>0.3750</td>
<td>-45.18</td>
<td>7.123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.117</td>
<td>-2.123</td>
<td>0.9326</td>
</tr>
</tbody>
</table>

Table 2.5: Parameters from fitting a linear and an exponential model to monthly disk failures as a function of avg. temperature.
based on age. We found that the behaviour of a drive under temperature did not change depending on either utilization or age (with statistically significant data only up to 36 months).

**Observation:** Neither utilization nor the age of a drive significantly affect drive failure rates as a function of temperature.

### 2.5.2.2 Temperature and DRAM Reliability

**Background and Data.** In this section, we study how temperature affects the reliability of DRAM, which is one of the most commonly replaced hardware components in data centers and the most common hardware related cause of node outages [90, 92]. DRAM has two different error modes: correctable errors (CEs), where a bit on a DRAM chip is flipped, but can be corrected with internal error correcting codes (ECC); and uncorrectable errors (UEs), where multiple bits are flipped, and the number of erroneous bits is too large for the ECC to correct, causing a machine crash or shutdown. CEs can be caused by external disturbances, such as cosmic rays, or by hardware defects, such as a stuck bit. UEs usually involve underlying hardware defects, since it is highly unlikely that cosmic rays would simultaneously flip a large enough number of bits to cause an uncorrectable error. Therefore in many data centers it is a common policy to immediately replace a DRAM DIMM after the first occurrence of a UE.

Work in [94] looked at correctable errors in DRAM and showed that their frequency goes up with temperature, but found that this correlation disappears once one controls for utilization. In this section, we ask how temperature affects the long-term reliability of DRAM, rather than the likelihood of transient problems, i.e., do higher temperatures increase the rate at which DRAM wears out and needs to be replaced. We study the long-term reliability of DRAM by analyzing data on DIMM replacements, data on node outages that were attributed to DRAM, and data on uncorrectable errors (since the latter two tend to be indicative of hardware problems and typically lead to replacement of a DIMM). We have collected data from three different sources:

**Google:** Google routinely collects data on the occurrence of correctable and uncorrectable errors in all of their data centers, as well as periodic temperature measurements based on sensors on the motherboard. An overview of Google’s measurement infrastructure is provided in [94]. For our study we have obtained data for a sample set of Google’s systems, comprising a dozen different data centers. The data centers are based on five different hardware platforms, where a hardware platform is defined by the motherboard and memory generation. Details on the hardware platforms are considered confidential and we hence just refer to them as Platforms A, C, D, E, F.

**Los Alamos National Lab (LANL):** We use the same failure dataset described in our analysis of failure correlations in Section 2.3.1. As previously mentioned, the data made
available by LANL covers more than 20 of their HPC computing clusters and includes information on the root cause of a node outage and the duration of the outage. The data can be downloaded from LANL’s web page [6] and a more detailed description of the data and systems can be found in [90]. Uncorrectable DRAM errors are one of the most common root causes for node outages, and in this section we use only the subset of the data that consists of node outages due to DRAM.

For one of LANL’s clusters periodic temperature measurements from a motherboard sensor are available, allowing us to directly study the relationship between temperature and outages. We refer to this system as LANL-system-20, since the ID for this system on LANL’s web page is 20. For another 12 clusters information on the data center layout is available, including each node’s position in a rack. We use rack position as a proxy for temperature, since due to the cooling system design in those clusters the top of the rack tends to be hotter than the bottom. We have verified that this is the case by analyzing the data for LANL-system-20, where actual temperature measurements are available, and found a difference of 3.4°C between the top and bottom of the rack (see Figure 2.23 for actual temperature values). The 12 clusters are based on two different hardware platforms, which we refer to as LANL-Type-1 and LANL-Type-2.

LANL-Type-1 comprises seven clusters at LANL totaling 2720 nodes and 10880 processors. The nodes in the system are SMPs with 4 processors per node and are all based on the same hardware platform. The data for these systems spans the years 2002-2008 and corresponds to systems with IDs 3,4,5,6,18,19, and 20 on the LANL web page.

LANL-Type-2 comprises six clusters at LANL totaling 1664 nodes and 3328 processors. The nodes are SMPs with 2 processors per node and the data for these systems spans the years 2003-2008. The data is also available at LANL’s web page and corresponds to the systems with IDs 9,10,11,12,13, and 14 on the web page.

SciNet-GPC: The SciNet High Performance Computing Consortium provides computing facilities to researchers in Canada. Their General Purpose Cluster (GPC) is currently the largest supercomputer in the country [7]. We obtained parts replacement data from this
system which is manually entered by an administrator when broken hardware is replaced. The replacement log we obtained spans 19 months. The GPC consists of 3870 IBM iDataPlex nodes grouped into 45 racks. Each node contains 2 Intel Xeon E5540 CPUs totaling 8 cores and 16GB of ECC memory.

**Analysis.** Figures 2.24 show the monthly probability for node outages at LANL that are attributed to memory as a function of the node’s average temperature. In Figure 2.24 (left) we use the data for LANL-system-20, which has actual temperature measurements, and for Figure 2.24 (middle, right) we use the server’s position in a rack as a proxy for temperature for LANL-Type-1 and LANL-Type-2 systems. We find that none of the graphs shows clear evidence for increasing rate of node outages with increasing temperatures.

Results are similar for hardware replacement rates at SciNet. Figure 2.25 shows a node’s monthly probability of requiring a DIMM replacement as a function of its position in the rack. Again, we see no evidence of higher failure rates for higher (and hence hotter) rack positions.
Unfortunately, due to the size of the datasets the error bars in those graphs are relatively high. We therefore turn to the Google data on uncorrectable errors, which is a larger data set. Figure 2.26 (left) shows the monthly probability of an uncorrectable DRAM error for the five different hardware platforms at Google. We observe that for two of the models, model C and model F, error rates remain stable throughout the available range of temperature data (which is quite large ranging from 15°C to 60°C). Maybe surprisingly, model D and model A show contradicting trends, with the former exhibiting decreasing rates as temperature increases and the latter showing increasing rates as temperature rises.

To investigate the possible cause we break down the data by data center. Figure 2.26 (right) shows the resulting breakdown by data center for model D. We find that the error rates for individual data centers are mostly flat with temperature, with the exception of one data center (data center-2). It is the aggregation of data from different data centers that creates those apparently contradicting trends. Similarly, we observe for model A that higher temperature points are biased by one data center that is running at a higher temperature and tends to have generally higher error rates (even for low temperatures).

*Observation:* We do not observe evidence for increasing rates of uncorrectable DRAM errors, DRAM DIMM replacements or node outages caused by DRAM problems as a function of temperature (within the range of temperature our data comprises).

### 2.5.2.3 Temperature and node outages

**Background and Data.** Rather than focusing on a particular hardware component, this section looks at overall system reliability and availability as a function of temperature. For our study we use data from two different sources. The first source comprises the LANL datasets LANL-Type-1 and LANL-Type-2. Rather than focusing on records of node outages due to DRAM, we now include in our analysis any node outage that was attributed to a hardware problem. The second dataset is the SciNet-GPC replacement data, but rather than focusing on DRAM replacements we consider replacements of any hardware components.

**Analysis.** Figure 2.27 shows the effect of temperature on the rate of node outages at LANL. Figure 2.27 (left) shows the monthly probability of a node outages as a function of the node’s average temperature for system 20 in the LANL data set, as for this system temperature measurements are available. Figure 2.27 (middle, right) show the monthly probability of a node outages as a function of a node’s position in the rack (bottom to top position, i.e. colder to hotter) for LANL-Type-1 and LANL-Type-2 systems. We observe within the temperature range that our data spans no indication that hotter nodes have a higher probability of failing than colder nodes.

Results are similar for hardware replacements observed at SciNet (Figure 2.28): no
indication that nodes at the top of the rack experience more hardware replacements than those at the bottom of the rack.

For the LANL data, we also have information on the length of a node outage, i.e. how long did it take to bring the node back up. Figure 2.29 shows box plots \(^3\) for the total amount of downtime experienced by a node per month for system 20. We find that the downtime experienced by hot nodes does not differ significantly from the downtime experienced by cold nodes, as both medians and lower and upper quartiles of downtime tend to be similar.

**Observation:** We observe no evidence that hotter nodes have a higher rate of node outages, node downtime or hardware replacements than colder nodes.

One might ask whether node outages might be more strongly affected by variability in temperature, rather than average temperature. The only dataset that allows us to study

\(^3\)Recall that in a box plot the bottom and top of the box are always the 25th and 75th percentile, respectively, and the band near the middle of the box is always the 50th percentile (the median).
this question is the LANL data for system 20. Figure 2.30 (right) shows the monthly probability of a node outage for LANL-system-20 as a function of the coefficient of variation in temperature. The figure compares the node outage probability for the top 50% of nodes with the highest CoV and the bottom 50% of nodes with lowest CoV. We observe that nodes with a higher CoV in temperature have significantly increased rates of node outages. For comparison, we also plotted the probability of node outages as a function of average temperature in the same way (Figure 2.30 (left)) and observe no difference between hot and cold nodes.

**Observation:** We find that high variability in temperature seems to have a stronger effect on node reliability than average temperature.

### 2.5.2.4 How do temperature excursions affect failures?

Our analysis so far has focused on studying the effect of average temperature and the variability in temperature over time. We are now interested in investigating the impact of temporary excursions of very high temperatures. Rather than looking at average temperature readings, we study the effects of brief periods of high temperature by looking at the impact that a fan failure or a failure in the chiller system has on HPC nodes. Fan and chiller failures will lead to temporarily increased temperatures at a node, and depending on whether it’s a complete or partial failure can lead to extreme temperatures inside a node, making a node shutdown necessary.

We turn to LANL’s failure dataset where records of server fan failures and chiller failures are available. Figure 2.31 (left) shows the impact of fan failures and chiller failures on hardware failures. The graph shows the probability that a node will experience a hardware failure within a day, week and month following a fan or a chiller failure, compared to the probability of a hardware failure in an average day, week and month. We observe clearly increased hardware failure rates following fan and chiller failures for all timespans. Fan failures have a stronger effect for all timespans, with a factor of 40X increase in hardware failure rates on the day following a fan failure (compared to a random day). Chiller failures
had a weaker effect across the different timespans, with factors of 6-9X increase in hardware failure rates.

We also ask what type of hardware failures are likely to follow fan and chiller failures. Figure 2.31 (right) shows for each of the hardware components with corresponding records in the data the probability of failure within a month after a fan or a chiller failure, compared to a failure of that component in a random month.

We find that all hardware components, except for CPUs, show an increase in the failure rate following a fan failure. We find that for memory, node boards, and power supplies the order of magnitude of the increase is similar to the one observed after power problems, with factors of 10-20X. In addition, we observe significant increases in failure rates for two types of boards, MSC boards and midplanes, which we did not observe in the case of power problems. One of the largest increases in failure rates, a factor of 120X, occurs for fans, which is maybe not surprising given that we have observed previously that most failure types have the strongest correlation with events of the same type.

Chillers failures seem to only affect two components: memory DIMMs and node boards, with 5.3X and 10.8X increases in their probabilities, respectively.

Observation: Our analysis shows that hardware components are well able to tolerate higher average temperatures within the ranges that are typically observed in a data center. The harmful effects of temperature mostly stem from short periods of extremely high temperatures, for example due to the failure of a fan in the system.

### 2.5.3 External Factors: Cosmic Radiation

**Background and Data.** It is known that high rates of cosmic radiation can lead to soft errors due to bit flips in DRAM or on system buses. If the built-in error correcting codes (ECC) are not strong enough to correct the corrupted bits, those errors will lead to a machine crash or shutdown. Cosmic rays and their effect on system reliability are a major
concern, and, for example in the case of DRAM errors, most of the existing work on DRAM reliability focuses on the effects of cosmic radiation.

We use the failure data made available by Los Alamos National Lab, which we describe in more detail in Section 2.3.1, to study the impact of cosmic radiation levels on hardware failures in HPC nodes. Since LANL’s failure data spans a very long time period (nearly a decade), it covers almost an entire solar cycle (typically 11 years long), including several solar flares.

Records of high-energy neutron counts that are produced by cosmic rays in the atmosphere are collected at many neutron monitor (NM) stations around the world. We use data of 1-minute resolution neutron counts collected at a NM station in Climax, Colorado (geographically close to Los Alamos National Lab) [71]. We use this data to analyze whether periods of increased cosmic rays are correlated with a higher rate of hardware errors, in particular failures related to DRAM and the CPU.

**Analysis.** We begin by studying whether the likelihood of a node outage due to DRAM failure changes with neutron flux levels. Figure 2.32 (left) shows the monthly probability of a DRAM failure as a function of the monthly average neutron counts-per-minute, for LANL Systems 2, 18, 19 and 20. We focus our analysis on the LANL systems that span the longest lifetimes, or consist of the largest numbers of nodes, across all systems. We find that months with higher neutron rates are not associated with higher rates of DRAM failures.

These results might be unexpected, since cosmic rays are known to increase soft error rates in DRAM. One possible explanation is that while increased rates of cosmic rays might lead to a higher number of corrupted bits, the types of corruption caused by those events might usually be correctable with the built-in ECC. This explanation agrees with recent findings in [49], which provide evidence that most node outages that are due to errors in DRAM are likely caused by hard errors, i.e. problems with the underlying hardware, rather than random events, such as cosmic rays.
Cosmic ray-induced neutrons can also cause CPU faults, possibly leading to a machine crash or shutdown. We repeat our correlation analysis using data on node outages that were attributed to CPU failures, rather than outages due to DRAM problems (see Figure 2.32 (right)). We observe that in three systems (2, 18 and 19), CPU failures were slightly more likely to occur in months with relatively higher neutron rates.

Finally, we repeated this analysis with software failures to investigate if higher neutron rates were correlated with failures that were diagnosed as being software related (e.g. OS failures). We found no clear correlation with software failures, in general. When repeating this analysis for ‘OS’ failures, in particular, we find that the probability of an OS failure increases with neutron levels for one LANL system (system 20), but due to the size of the dataset we cannot verify the statistical significance of the results.

2.6 Combining Multiple Factors: Regression Analysis

Rather than studying the individual effect of different factors separately, we now ask the question of what the collective effect of multiple factors, combined, is on long-term HPC node reliability. More particularly, we are interested in studying and comparing the effect of factors related to temperature, usage, and physical layout, on the reliability of HPC machines.

The only dataset that allows us to explore this question is LANL’s failure data [6] where we have logs available on node outages, node usage, physical layout and ambient temperature for one of LANL’s HPC systems (system with ID 20).

We use regression analysis to model occurrences of node outages in system 20 as a function of node usage, physical location and temperature. More precisely, we use the total number of outages a node experiences during the data collection period (due to any type of failure) as our response variable, which we try to express by the set of predictor (explanatory) variables summarized in Table 2.6. We use two commonly used regression models, Poisson regression and negative binomial regression.

Both Poisson and negative binomial regression models belong to the family of generalized linear models, and are available as part of the computational toolbox for modeling count data in the software R for statistical computing [51].

The results of applying Poisson regression and negative binomial regression are shown in Tables 2.7 and 2.8, respectively. The two right most columns show the test statistic and the p-value, respectively, that the null hypothesis that each predictor’s coefficient is zero given that the rest of the predictors are in the model. The two right most columns show the test statistic and the p-value, respectively, which test if the null hypothesis is zero. The null hypothesis here is the hypothesis that each predictor’s coefficient is zero, given that the rest of the predictors are in the model.
Chapter 2. Failure Analysis in HPC Systems

<table>
<thead>
<tr>
<th>Variable</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fails_count</strong> (response variable)</td>
<td>Failures</td>
<td>This is the response variable; the total occurrences of node outages in a node’s lifetime.</td>
</tr>
</tbody>
</table>

### Input Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>avg_temp</strong></td>
<td>Temperature</td>
<td>The average ambient temperature of a node</td>
</tr>
<tr>
<td><strong>max_temp</strong></td>
<td>Temperature</td>
<td>The maximum temperature reported by a node</td>
</tr>
<tr>
<td><strong>temp_var</strong></td>
<td>Temperature</td>
<td>The variance of all temperatures reported by a node</td>
</tr>
<tr>
<td><strong>num_hightemp</strong></td>
<td>Temperature</td>
<td>The number of severe temperature warning messages reported by a node (i.e. when its ambient temperature exceeds 40C)</td>
</tr>
<tr>
<td><strong>num_jobs</strong></td>
<td>Usage</td>
<td>The number of jobs that were assigned to the node in the observation period</td>
</tr>
<tr>
<td><strong>util</strong></td>
<td>Usage</td>
<td>The utilization of a node during the observation period</td>
</tr>
<tr>
<td><strong>PIR</strong></td>
<td>Layout</td>
<td>Position in Rack: the position of a node inside the physical rack (1=most bottom, 5=top)</td>
</tr>
</tbody>
</table>

|                | Estimate | Std. Error | z value | Pr(>|z|) |
|----------------|----------|------------|---------|---------|
| (Intercept)    | 2.0232   | 0.8288     | 2.44    | 0.0146  |
| **avg_temp**   | 0.0546   | 0.0337     | 1.62    | 0.1046  |
| **max_temp**   | -0.0705  | 0.0339     | -2.08   | 0.0373  |
| **temp_var**   | 0.0253   | 0.0333     | 0.76    | 0.4479  |
| **num_hightemp** | 0.0210 | 0.0698     | 0.30    | 0.7639  |
| **num_jobs**   | 0.0004   | 0.0001     | 7.17    | 0.0000  |
| **util**       | -0.0268  | 0.0050     | -5.34   | 0.0000  |
| **PIR**        | -0.0262  | 0.0358     | -0.73   | 0.4654  |

Table 2.6: Summary of Regression variables

Interestingly, we observe similar results for both models: The predictors **num_jobs** (i.e. the number of jobs assigned to a node during the observation period) and **util** (i.e. the node’s average utilization) are each statistically significant in both models; with 99% confidence, we can reject our null hypothesis and conclude that each one of them is statistically different from zero given that the rest of the coefficients are in the model.

Since we know from Section 2.4.1 that node 0 in this system exhibited a strong linear correlation between usage variables and number of failures, we reran our regression models after removing node 0 from the data and found that utilization remains significant to the model, although the significance level drops slightly.

In addition to usage variables, we observe that for the Poisson model **max_temp** is statistically significant to the frequency of node outages. However, when rerunning the model with only the significant predictors, the significance level of **max_temp** in the Poisson model drops.

We find these results to be a strong indicator that a node’s usage and utilization levels have a stronger impact on a node’s failure rates than other factors, such as its ambient temperature or physical location inside a rack.
Chapter 2. Failure Analysis in HPC Systems

2.7 Large-Scale Job Failure Analysis

Motivation. As large-scale platforms continue to grow in scale and complexity, the applications running on them are increasingly being prone to errors and failures. A recent study by Snir et al. [96] predicts that large parallel applications running on an exascale machine may abort as frequently as once every 30 minutes. Minimizing the rate at which large-scale jobs fail is especially critical for long-running applications that can take up to weeks or months to finish executing successfully [92]. Emerging challenges, such as the increasing complexity of parallel application workflows [24] or the rising silent data corruption (SDC) rates in large-scale platforms [40], require substantial improvements in both system-level and application-level resilience.

So far in this chapter we have focused on analyzing system failures in large-scale clusters by studying the impact of different factors on node reliability. In this section, we turn to the application level; we are interested in answering the question of what makes jobs fail in large, parallel clusters. Towards this end, we use workload traces collected at real world systems to study the effect of different factors on job reliability and to identify which ones are predictive of job failures in parallel applications. We investigate factors such as job configuration parameters (e.g. the degree of parallelism), job resource utilization (e.g. memory, CPU, and I/O usage), and job fault-tolerance mechanisms.

We begin by describing the workload traces included in our study in Section 2.7.1. In Section 2.7.2, we use the traces to characterize the behaviour of unsuccessful large-scale jobs to understand how they fail in the wild. We then take a closer look into the possible root-causes and factors that are likely to result in a job’s unsuccessful termination, to better understand why these jobs fail, in Section 2.7.3. Finally, Section 2.7.4 focuses on the prediction of job and task failures using machine learning methods.

2.7.1 Description of workload traces

We rely on workload traces made available by three different sources: Google, Carnegie Mellon University (CMU), and Los Alamos National Labs (LANL).

|            | Estimate | Std. Error | z value | Pr(>|z|) |
|------------|----------|------------|---------|----------|
| (Intercept)| 1.5478   | 1.1930     | 1.30    | 0.1945   |
| avg_temp   | 0.0499   | 0.0462     | 1.08    | 0.2802   |
| max_temp   | -0.0510  | 0.0475     | -1.07   | 0.2828   |
| temp_var   | 0.0252   | 0.0449     | 0.56    | 0.5744   |
| num_high_temp| 0.0021   | 0.0937     | 0.02    | 0.9820   |
| num_jobs   | 0.0004   | 0.0001     | 3.86    | 0.0001   |
| util       | -0.0248  | 0.0073     | -3.42   | 0.0006   |
| PIR        | -0.0345  | 0.0488     | -0.71   | 0.4794   |

Table 2.8: NB Regression Coefficients
(a) Google: For one of Google’s large, multi-purpose compute clusters consisting of 12,000 machines, traces of all jobs submitted to the cluster during 29 days in May, 2011 are made publicly available [102]. Each job is comprised of one or more tasks, and each task is associated with a set of resource requirements. A task is assigned to run on a single machine, and data on task resource consumption is made available, such as CPU, memory, and disk usage (aggregated over 5-minute intervals). All values of usage data are normalized by the maximum value reported in each category, and fields on users and application names are obfuscated. The traces include data on when each job or task was submitted, scheduled, and finished (or was aborted). The exit status of each job (and of each task attempt within a job) is included in the traces and can be one of the following: failed, killed, evicted, or finished.

Jobs were submitted by hundreds of users, where a user can be a Google engineer or a service [82], and their tasks were assigned priorities from 0 (lowest) to 11 (highest). According to a paper that accompanied the release of the trace [82], priorities 0 and 1 are labeled as gratis (free); priorities 2–8 are dominated by MapReduce [29] jobs and are labeled as ‘batch’; priorities 9–11 are for ‘production’ jobs. Our discussions with Google scientists revealed that ‘production’ jobs in this cluster represent services that are designed to run continuously and eventually get aborted (killed). Therefore, in our work we focus on studying the reliability of ‘batch’ priority jobs, which are mostly dominated by MapReduce jobs submitted by Google engineers [82].

(b) CMU OpenCloud Traces: OpenCloud [5] is a 64-node research cluster at Carnegie Mellon University used by CMU researchers and managed by the Parallel Data Lab [3]. The cluster runs Apache’s Hadoop [1] distributed file system, and had different types of applications running on it during the trace collection period from areas such as computational biology, natural language processing, image processing, and machine learning [5].

The traces were collected over a period of 30 months and include two types of files: job configuration files, and job history logs. The configuration files contain information on the parameters set by the users for each submitted job, such as (but not limited to): the size of virtual memory for map and reduce tasks; the maximum number of attempts allowed for a task before the framework gives up on it; and boolean flags for whether or not to enable speculative execution. The history files contain data on job submission time, launch time, and finish time; the exit status of a job (succeeded, failed, or got killed); the number of map/reduce tasks in the job; the number of tasks in the job that completed successfully; and the number of non-successful task attempts.

(c) LANL Job Traces: LANL made publicly available three years worth of job traces for one of their HPC systems (system with ID 20 on their webpage [6]), which comprises of 256 nodes. The HPC cluster was used by scientists, typically to run CPU-intensive scientific
Chapter 2. Failure Analysis in HPC Systems

simulations. The workload traces contain data on job submission time, launch time, and exit time; the number of processors requested for this job; and the final status of the job. The table below summarizes the possible exit statuses for a job submitted to the LANL cluster:

<table>
<thead>
<tr>
<th>LANL Job Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>finished</td>
<td>All processes are scheduled</td>
</tr>
<tr>
<td>failed</td>
<td>One or more of the nodes running this job crashed</td>
</tr>
<tr>
<td>aborted</td>
<td>User aborted job (Ctrl/C)</td>
</tr>
<tr>
<td>killed</td>
<td>An application process was killed by a signal</td>
</tr>
<tr>
<td>syskill</td>
<td>Job was killed by an administrative user (either due to maintenance or due to a problem).</td>
</tr>
</tbody>
</table>

Table 2.9: Description of exit status meaning in the LANL job traces.

We now look at the basic statistics and overall success rates for the jobs submitted to the three clusters in our dataset. Table 2.10 below provides an overview of the clusters and their job traces, along with the breakdown of the job exit status.

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Data Timespan</th>
<th>Number of Nodes</th>
<th>Number of Users</th>
<th>#Scheduled Jobs</th>
<th>%Success Jobs</th>
<th>%Failed Jobs</th>
<th>%Killed or Aborted Jobs</th>
<th>%Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Multipurpose</td>
<td>29 days</td>
<td>12K</td>
<td>227</td>
<td>349K</td>
<td>63%</td>
<td>1.3%</td>
<td>35.5%</td>
<td>0.003%</td>
</tr>
<tr>
<td>Cluster (Batch Jobs)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMU OpenCloud Cluster</td>
<td>883 days</td>
<td>64</td>
<td>73</td>
<td>78K</td>
<td>88.8%</td>
<td>8.11%</td>
<td>3%</td>
<td>0.94%</td>
</tr>
<tr>
<td>LANL HPC System#20</td>
<td>1,022 days</td>
<td>256</td>
<td>446</td>
<td>290K</td>
<td>67.5%</td>
<td>0.62%</td>
<td>30.7%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 2.10: Overview of the workload traces and basic job statistics for the clusters in our dataset.

We find that job success rates vary across the different clusters in our data and fall within the 60–90% range. Jobs that ‘failed’ were in the range of 1–8% of total jobs, while jobs that were aborted or killed were as high as 30–35% in two of the clusters in our data. These high observed rates of jobs not completing successfully in large-scale clusters across different organizations motivated us to take a closer look into the workload traces to better understand how and why these jobs failed.

2.7.2 What characterizes unsuccessful jobs in large clusters?

Our goal in this subsection is to use real world traces to learn the characteristics of jobs that fail or get killed. We use different job attributes, job configuration parameters, and job runtime statistics to study whether and how ‘bad’ jobs (i.e. jobs that fail or are killed) exhibit execution patterns and/or configuration settings that are different than completed (successful) jobs. More precisely, we examine the following attributes: job duration, degree of parallelism, scheduling constraints such as job priority, requested resources, and resource utilization.


2.7.2.1 Job Duration

The duration of a job is the amount of task-minutes spent by the job in the cluster before the job exits. To compare the duration of successful and unsuccessful jobs, we study the distribution of task-minutes per job for jobs that complete successfully, jobs that fail, and jobs that get killed. Figure 2.33 studies the empirical cumulative distribution function (CDF) of job durations for each cluster in our traces. Each line in the graphs plots the CDF for jobs that share an exit status.

We find that the durations of successful jobs are consistently the shortest across all jobs submitted to a cluster. Jobs that get killed spend the longest times running before they are terminated (in the LANL HPC cluster the durations of both killed and failed jobs report somewhat comparable distributions). This insight is intuitive since hanging jobs are typically killed by users at some point during their execution when the user notices that the job is taking longer time than expected to finish. However, our results show that jobs which fail also spend longer times in the cluster than successful jobs. Note that jobs categorized as ‘failed’ are typically jobs that experienced a software crash [82]. We next explore if the number of tasks in a job contributes to this observed correlation.

2.7.2.2 Degree of Parallelism

We now study the relationship between the degree of parallelism of a job and its final status. We first consider the jobs submitted to the Google and the CMU Hadoop clusters, since jobs in these systems share a similar structure and a programming paradigm, where loosely-coupled jobs span one or more tasks.

In Google’s cluster, a job consists of one or more tasks that typically execute the
same binary with the same resource requirements and scheduling constraints (e.g. priority, scheduling-class, etc.). Applications that need to run different types of tasks, e.g. tasks that have different requirements, will usually execute them as separate jobs [102]. For example, MapReduce applications would execute masters and workers as separate jobs. Multi-task jobs are meant to have their tasks run simultaneously, where a single task can be running on a single machine at any point in time.

We find that 65% of Google’s batch jobs that we consider in this study consist of a single task. The remaining 35% of jobs that are multi-task, however, consume more than 98% of total cluster time during the trace collection period.

In the CMU OpenCloud Hadoop cluster, jobs consist of tasks that are categorized explicitly as either ‘mappers’ or ‘reducers’. In Hadoop MapReduce jobs, the number of map tasks are typically driven by the size and the number of files in the input data. The ideal number of reduce tasks as recommended in Hadoop’s documentation is a function of the buffer size of the output files in a job. Figure 2.34 shows the distribution of the number of map tasks and reduce tasks per job in CMU’s traces. We find that more than 80% of the OpenCloud jobs had at least two mapper tasks, and nearly 60% had one or zero reduce tasks. Note that the default number of reduce tasks in a job mentioned in Hadoop’s official documentation is one [1].

We now ask the question of whether the chances of completing successfully are different for single-task and multi-task jobs in these clusters. We begin by looking at job success rates separately for single-task and multi-task jobs, in both clusters. We find that in Google’s jobs, more than 90% of single-task jobs completed successfully, while more than 90% of multi-task jobs were killed; the portion of failed jobs was around 1.5% for both single- and multi-task jobs. In the CMU Hadoop cluster, however, we find that the breakdown of job status is almost identical for single- and multi-task jobs: 89-90% completed, 7-8% failed, and 2-3% were killed.

One thing to note here is that since multi-task jobs spend more time (i.e. task-weeks) in the cluster, they are naturally more likely to experience issues that could cause them to fail.
or hang. To account for this effect, we normalize the number of job interruptions observed by the total amount of task-weeks consumed by the jobs in each category. Figure 2.35 shows our results for both Google and CMU jobs. We find that, when normalized by system time, parallel jobs exhibit lower rates of job failures and job kills per unit time than do single-task jobs.

We now examine the jobs submitted to LANL’s HPC cluster. The LANL jobs ran on one or more nodes in the cluster (note that they do not span ‘tasks’); the number of nodes that each job was assigned to run on was determined by the number of processors that a job requested. Each node in LANL’s cluster had four processors and jobs typically requested processors in multiples of four. We therefore use the number of requested processors field in the traces as an indicator of the level of parallelism in a LANL job.

We find that 79% of the LANL jobs ran on a single node; the remaining 21% of jobs ran on varying numbers of nodes, ranging from 2 nodes and up to the entire cluster (i.e. 255 nodes). Figure 2.36-(left) compares the breakdown of the final status between jobs that ran on a single node and jobs that ran on multiple nodes in LANL’s cluster.

We observe 70% and 50% success rates in single-node and multi-node jobs, respectively. We also find that job kills, i.e. jobs that are either syskilled or killed, are almost doubled
in parallel jobs compared to single jobs.

Figure 2.36-(right) shows the rates of different types of job interruptions normalized by system time in each category. We find that with the exception of killed jobs, the interruption rates for LANL jobs that ran on single nodes were higher than in parallel jobs, for all categories.

Observation: For all the clusters in our dataset, we find that the percentage of parallel jobs that complete successfully is lower than the percentage of single-node jobs that complete. However, when normalizing by the system time spent on each group, we find that the rate of job interruptions per unit time is lower in parallel jobs, for the majority of the cases we examined.

2.7.2.3 Scheduling Constraints

We next look into the effect of different job scheduling constraints on a job’s final status. In particular, we consider the job’s priority, scheduling-class, and the amount of requested resources.

Job Priority: To study the effect of a job’s priority on its terminal status, we turn to the Google traces, which are the only traces in our dataset that include information on job priorities. In the Google cluster, when assigning tasks to machines, tasks with higher priorities are favored in resources over tasks with lower priorities. Additionally, the scheduler is designed such that over-committing resources on a machine is permissible. Therefore, in the case when there are not enough resources to satisfy the requests of all tasks running on a machine, tasks with lower priorities may be ‘evicted’ or even ‘killed’ [102].

Google’s batch jobs, which are the focus of our study, span a wide range of priorities: each job is submitted with a number ranging from 2 (lowest priority) to 8 (highest priority). Figure 2.37-(a) plots the breakdown of the jobs’ final status as a function of job priority level 4. We find that a job’s priority in Google’s cluster is positively correlated with the job’s chance of completing successfully.

Job Scheduling Class: In Google’s cluster, jobs have a scheduling class that determines how latency-sensitive a job is, with values ranging from 0 (least sensitive), to 3 (most sensitive). While a task priority determines if it is scheduled on a machine or not, the scheduling class is used locally by the machine to implement machine-local policies for accessing its resources. We find that in Google’s batch jobs, users rarely configured the jobs to have the most sensitive class of ‘3’ (only 0.4% of batch jobs).

Figure 2.37-(b) shows the breakdown of job status as a function of scheduling class. We find that the higher the scheduling class, the higher the chances of a batch job completing successfully. To investigate whether this is the result of higher priority jobs being assigned

---

4 We omit priorities 3, 5 and 7 from our analysis as the total number of jobs scheduled in these groups combined does not exceed 0.05% of all Google jobs.
higher scheduling classes, we took a closer look into the relationship between these two scheduling attributes, and found a positive correlation between the priority of a job and its scheduling class (Pearson Correlation Coefficient $R=0.53$).

**Observation:** In the Google cluster in our dataset, we observe a strong correlation between the priority level assigned to a job and the final status of the job. Jobs with higher priorities and more sensitive machine-local latency constraints had more chances of completing successfully.

**Requested Resources:** We now study the relationship between the amount of resources requested by a job and the final status of the job. In the LANL cluster, we only have data on the number of requested processors per job. In the Google cluster and the OpenCloud cluster, more detailed information is available. In Google, tasks are submitted with values for requested CPU, memory, and disk space, where these values represent the maximum amount of resources a task is allowed to consume on a machine. However, tasks are sometimes permitted to use more than what they requested if resources are available; e.g. tasks may use free CPU cycles on a machine [102].

In the OpenCloud cluster, job configuration parameters include the following fields for requested resources (more detailed definitions are found in Hadoop’s documentation [1]):

- `mapred.job.map.memory.mb`: The size of virtual memory for a single map task.
- `mapred.job.reduce.memory.mb`: The size of virtual memory for a single reduce task.

Figure 2.38 plots the distribution of the average requested resources by tasks in a job, for jobs that finish, fail or get killed, separately. We use boxplots $^5$ to examine and compare the distribution of requested resources.

$^5$Recall that in a box plot the bottom and top of the box are always the 25th and 75th percentile, respectively, and the band near the middle of the box is always the 50th percentile (the median).
Chapter 2. Failure Analysis in HPC Systems

Figure 2.38: Requested resources by tasks in a job versus the exit status of a job in the Google cluster and CMU’s OpenCloud cluster.

The first thing we observe from the graphs is that resource requests varied between successful and unsuccessful jobs in all the clusters in our data. In the Google cluster, we observe significantly higher distributions of requested memory and requested disk space in batch jobs that failed, compared to successful jobs (see Figure 2.38-(a)).

In the OpenCloud cluster, we find that jobs that were eventually killed report significantly higher requests for virtual memory than successful jobs or even jobs that failed, for both mapper and reducer tasks (see Figure 2.38-(b)). Note that the value of requested VM per task for more than 99% of jobs that completed or failed was left at the default value, which is 1024 MB according to Hadoop documentation [1]. In the LANL cluster, we observe the highest distributions of requested processors per job for jobs that get killed, followed by jobs that fail, as shown in Figure 2.38-(c).

Observation: For all the clusters in our dataset, we find that jobs with unsuccessful terminations requested more cluster resources than successful jobs. In cases where we have knowledge on the default values for some requested resources, we find that deviating significantly from the default configuration was correlated with job abortion. These observations suggest that knowledge of how much CPU, memory, and/or disk space a job requests can potentially be used to predict the job’s final status. (We study job failure prediction in Section 2.7.4.)
2.7.2.4 Job Resource Usage

After examining how a job’s configuration parameters correlate with the job’s exit status, we now turn our attention to the actual resource consumption of a job. Our goal is to study how the way a job utilizes resources in a large-scale cluster affects its final status. Data on resource utilization is available in the traces from Google and the CMU Hadoop cluster only. We discuss our analysis of each trace separately below then summarize our observations from both data sets at the end of this subsection.

a) Resource Usage in Google: In the Google cluster, there is usage data on the CPU, memory and disk utilization for each task running on a machine in the cluster, aggregated over 5-minute intervals. All values of usage data are normalized by the maximum value reported in each category.

Figure 2.39-(a) shows the distribution of the average CPU, memory, and disk I/O consumption for Google’s batch jobs, plotted separately for jobs that complete, fail, or get killed. We find that killed jobs exhibit higher CPU and memory utilization distributions, while failed jobs report higher I/O consumption. (It is worth noting that I/O consumption here refers to node-local disk utilization; Google’s traces do not contain any information on the GFS (Google File System) usage by jobs.)

In addition to studying the average resource consumption of jobs, we also ask the question of how the variability in usage between tasks that belong to the same job compare across successful and unsuccessful jobs. Figure 2.39-(b) plots the distribution of the coefficient of variation (CoV) \(^6\) in resource usage between tasks in a job. Note that we only include multi-task jobs in this analysis.

Interestingly, we find that failed jobs which had the highest average in I/O usage, report the lowest variability in I/O usage between tasks in the same job. On the other hand, tasks in a failing job had the highest variability in memory usage across them.

\(^6\)Recall that the coefficient of variation is defined as the standard deviation divided by the mean.
b) Resource Usage in the CMU Hadoop Cluster: We now turn to the CMU Hadoop cluster to study and compare the resource usage of completed and interrupted jobs. The CMU traces contain data on I/O task usage, both for Hadoop Distributed File System (HDFS) I/O counters and node-local filesystem counters. Below is a summary of the counters logged in the traces:

- **HDFS_BYTES_READ**: The number of bytes read from the HDFS by the mapper tasks, when a job starts.
- **HDFS_BYTES_WRITTEN**: The number of bytes written to the HDFS by the reducer tasks, when the final output of the job is produced.
- **FILE_BYTES_READ**: This is the number of bytes read from the local filesystem in a worker node. This typically corresponds to the amount of data read by reducer tasks from local disks on reduce nodes.
- **FILE_BYTES_WRITTEN**: This counter logs the sum of two quantities. First, the amount of bytes written by mapper tasks to local disks when the map output is produced. Second, during what is known as the ‘shuffle’ phase [1, 29], the reducer tasks will merge and spill the data to local disks on the reduce nodes, before the final output is produced. The bytes written locally by the reducers is therefore also included in this counter.

![Figure 2.40: Comparison of I/O usage in CMU Hadoop jobs that complete successfully, fail, or get killed.](image-url)
Using CMU’s traces, we have calculated for each Hadoop job the amount of bytes processed per task-minute, for each of these I/O counters separately.

The boxplots in Figure 2.40 show the distribution of I/O activity per job, broken down by job exit status. The first row shows the results for the HDFS counters, while the graphs in the second row correspond to the node-local filesystem counters.

We observe from the graphs that unsuccessful jobs used the I/O resources in the cluster differently than completed jobs. In particular, we find that failed jobs reported higher I/O writes to local disks in worker nodes, than did the rest of the jobs. This agrees with our observation in the Google cluster, where failed jobs had also reported higher local disk usage than jobs in other categories. (Note that in the Google cluster we had no information on the distributed Google Filesystem (GFS) activity to compare.)

We now examine the variability in I/O usage between tasks working on the same job in CMU’s Hadoop cluster. Figure 2.41 plots the coefficient of variation (CoV) in I/O activity between tasks in a job, for the four I/O counters available in our traces.

Figure 2.41: Comparison of I/O usage in CMU Hadoop jobs that complete successfully, fail, or get killed.
Interestingly, similar to our observation in the Google cluster, we find that failed jobs which had the highest average values for local disk writes, reported the lowest variability in local writes between tasks that belong to the same job. In other words, tasks in jobs that failed wrote more aggressively to local disks, on average, than tasks in completed jobs, with significantly lower variance in I/O activity between them. Writes to the HDFS, however, show a different behaviour: tasks in failed jobs have the highest variability between them in HDFS writes, compared to successful or killed jobs.

Observation: Our analysis of job resource usage shows that the ways in which jobs utilize resources in large clusters varies between jobs that complete successfully, jobs that fail, and jobs that get killed. For both the Google and CMU clusters in our dataset, we observe higher distributions of I/O activity in failed jobs with lower variability in disk usage between tasks in a job, compared to jobs that complete successfully. This observation can help improve the quality of job failure predictors in large-scale clusters, by exploiting resource usage monitoring in running jobs.

Summary of observed job failure characteristics: Our analysis so far of job traces from multiple organizations uncovered different properties of jobs that do not complete successfully in large-scale clusters (i.e. jobs that fail or get killed). We find that unsuccessful jobs typically run for longer durations, request more cluster resources, report higher average I/O activity, and exhibit lower I/O usage variability between tasks in the same job, compared to jobs that complete successfully in the same cluster.

After describing ‘how’ interrupted jobs behave differently than jobs which complete successfully, we next investigate different hypotheses on ‘why’ these jobs failed.

2.7.3 What are the root-causes behind job failures?

In this section, we take a closer look into the possible root-causes behind job failures in large-scale clusters, using the workload traces in our dataset and building on our observations of job failure properties from the previous section. We consider factors such as job misconfiguration, machine failures, application bugs, resource exhaustion, and user proneness to submitting bad jobs.

2.7.3.1 Job Misconfiguration: Resource Estimation

The configuration of large-scale jobs typically involves setting values for numerous parameters, many of which can interact with each other in ways that the user did not anticipate, once the job starts executing. In 2011, a manager in Cloudera [2], the leading provider and supporter of Apache Hadoop for the enterprise, reported that more than 30% of tickets handled by their support team were a result of some type of misconfiguration [100].
Job misconfiguration includes the mis-estimation of the amount of resources needed by the job’s tasks in order to complete their computation successfully. For example, the underestimation of required resources can get a job terminated if its tasks start to consume more resources than their allowed limits [83].

In Section 2.7.2, we found that unsuccessful jobs consumed and requested more cluster resources on average, than jobs which complete successfully. We now take a closer look into the accuracy of the estimated resources by these jobs, by computing the ratio between used resources and requested resources. The only dataset that allows us to compare information on requested and consumed resources is Google’s data on CPU and memory usage.

The graphs in Figure 2.42-(a) show the distribution of the ratio between the average CPU and memory consumption of a task divided by the requested amount of CPU and memory by the same task, respectively, in all of Google’s batch tasks. Note that we conduct this analysis on the task level since usage data in the Google cluster were logged on a per-task basis, so this allows for fine-grained examining of task-level resource consumption. The data points above the horizontal dashed red line in the graphs (which refers to ratio=1) correspond to tasks that underestimated the requested resource in their average consumption, while points below the line correspond to tasks that overestimated the same resource.

In addition to calculating the average usage of a task in relation to its requested resources, we are also interested in knowing if a task ever exceeded the requested amount of CPU or memory during its lifetime (not necessarily in the average case). This is important since the amount of resources requested by a task in the Google cluster practically translates to the maximum amount the task is permitted to use. According to the Google paper that accompanied these traces, tasks that use more memory than their limit can be terminated by the resource manager. Tasks that use more CPU time than their limit, on the other hand, may be just throttled. In fact, the resource manager may permit a task to use more than its CPU request sometimes if there is free CPU capacity on the machine. As a result, tasks with brief CPU bursts sometimes run with requested CPU of 0 [83]. We study the ratio between the maximum resource usage logged by a task at any point in its lifetime and its requested amount, in Figure 2.42-(b).

The first observation we make from Figure 2.42 is that almost all of the tasks in Google’s batch jobs made conservative estimates of their requested memory, when considering their average behaviour. In terms of average memory usage, tasks rarely exceeded their memory limits, regardless of the task terminal status. When looking at the maximum memory usage reported over a task’s lifetime, however, we find that 25% of tasks which fail do exceed their memory limit at some point (See right graph in Figure 2.42-(b)). Exceeding memory limits can be the result of the task initially underestimating the amount of memory needed, or due to something going wrong during execution, e.g. due to a bug in the program the task is executing.
(a) Ratio of average usage to requested resource.

(b) Ratio of maximum usage to requested resource.

Figure 2.42: Accuracy of CPU and memory estimation in Google’s Batch tasks.

On the other hand, we find that tasks were less conservative in their CPU estimates, which is not surprising given that the runtime environment permits consuming more CPU than initially requested. In fact, we find that completed tasks report the highest ratios of used CPU to requested CPU.

**Summary:** For the Google cluster in our dataset, we find that exceeding the allowed memory limits by tasks could be the root-cause behind the termination of up to 25% of the failed tasks. We find that even though failed tasks initially request more memory than other tasks (see subsection 2.7.2.4), they still exceed their memory limits more than do completed tasks or even killed tasks.
2.7.3.2 Fault-Tolerance Configuration

Another aspect of job configuration that we investigate for the purpose of identifying possible root-causes behind job failures, is job fault-tolerance configuration. We ask the question of whether the way a job is configured to deal with problems at runtime, such as slow or failing tasks, affects its chances of completing successfully.

In the Google and CMU Hadoop clusters in our data, reliability and fault-tolerance are provided by the underlying framework in the cluster through data replication across multiple nodes, and task re-execution (i.e. task retries upon failures). In the LANL HPC cluster, jobs represent tightly-coupled HPC applications, where fault-tolerance is achieved mainly through application level checkpointing; i.e. writing periodic checkpoints of application state to stable storage and restoring the application from that checkpoint upon failures. While we do know from the LANL webpage [6] that ‘checkpoint/restart’ was the used technique in these jobs, the LANL traces do not provide any data on the checkpointing configurations for these jobs, or any other parameters related to application fault-tolerance. The Google traces do not contain information on job fault-tolerance configuration either, but the effect of actual task re-execution on job reliability can be analyzed by considering the data available on task executions. (We examine the effect of task re-executions closely in the next subsection.)

The CMU Hadoop traces contain data on job configuration parameters that are collected from Hadoop’s default job_xml XML files. More precisely, we consider the following fault-tolerance and job reliability knobs that are available in the CMU traces, to study the effect of tweaking their values on a job’s final status:

- **map.tasks.speculative.execution, reduce.tasks.speculative.execution:** If true, speculative execution is enabled for mapper tasks and reducer tasks, respectively. Speculative execution in Hadoop is a way of dealing with cases where a task working on a parallel job ends up on a slow node, therefore slowing down the entire job. The Hadoop framework will then schedule redundant copies of that remaining slow task on multiple nodes, and collects the output from the first copy that finishes execution.

- **mapred.skip.attempts.to.start.skipping:** This is the number of task attempts after which a special mode will be enabled, called ‘skip’ mode. In skip mode, a task must report to the framework which data records it will process next. This helps the framework identify bad records if this task failed again, in order to inform the user and/or skip processing these records in future attempts.

- **mapred.map.max.attempts:** This is the maximum number of attempts that can be made by a mapper task; i.e. the Hadoop framework will try to execute a map task this number of times before giving up on it.

- **mapred.reduce.max.attempts:** Similar to the map max attempts, this is the maximum number of attempts that can be made by a reducer task.
Figure 2.43: Breakdown of the parameter values for the maximum number of task retries configured in the CMU Hadoop jobs.

Results for Speculative Execution: We begin by looking at the two speculative execution parameters. We find that overall, more than 99% of the jobs in the traces were left at the default values for these parameters, which is TRUE. When breaking down the jobs into different groups by their exit status, we find that the default enabling of speculative execution for mappers and reducers was present in 99.8%, 99.5% and 98% of completed jobs, killed jobs and failed jobs, respectively. The 2% of failed jobs that disabled speculative execution were divided almost equally between jobs that disabled it for both mappers and reducers, and jobs that disabled it only for mappers.

Results for Skip Mode Configuration: We find that 96% of the jobs in the CMU traces were left at the default value for the number of task attempts after which skip mode is enabled, which is 2 attempts. The rest of the values for this parameter in the traces were either 4, 15 or 16. We find that 96.5%, 97.6% and 95.5% of completed jobs, killed jobs and failed jobs had the default configuration, respectively, with the second most popular value being 16.

Configured Limit on Task Retries: We now consider the parameters on the maximum number of task attempts in CMU’s jobs, where the default value is 4 for both mappers and reducers. We find that overall 4% of the jobs changed the default value for the limit on map retries, while only 1% of the jobs changed the default limit on reduce retries.

Figure 2.43 shows the detailed breakdown of the maximum number of task retries that were found in the CMU job configuration traces, while separating jobs that completed from those that failed or got killed. For all the jobs in the trace, we find that the values set for task retry limits went as high as 100 for mappers and 130 for reducers. The bar plots show that killed jobs had the largest portion of jobs with limits on task retries that exceeded the default setting, followed by failed jobs, for both mappers and reducers:
Chapter 2. Failure Analysis in HPC Systems

Summary: Our results indicate that of all the fault-tolerance configuration parameters that we have data on in the CMU trace, the number of maximum task retries seems to have the strongest correlation with the unsuccessful termination of a job in the cluster (followed by the disabling of speculative execution). This observation motivated us to examine what the actual number of task retries looked like for these jobs when they were running in the cluster, and how that might have affected a job’s final status.

2.7.3.3 Recovery Mechanism: Task Retries

Our analysis of job configuration files from CMU showed that unsuccessful jobs were more likely to be configured with large limits on task retries, than did successful jobs. We now examine the effect of actual task retries on job reliability, using the traces available on task executions from CMU and Google.

To better understand the effectiveness of the task re-execution mechanism, we first ask the question of what the likelihood of a task attempt completing successfully looks like, after the task makes one or more previous failed attempts.

Figure 2.44 plots the probability of a task succeeding in the next attempt as a function of the number of past failed attempts.
of the number of past failed attempts the task had made, for both Google and CMU tasks. The left-most plot in each row shows the results when taking all jobs into consideration; the middle plots are for multi-task jobs only; and the right-most plots are for single-task jobs. This separation helps in studying the effectiveness of task retries when tasks are working alone versus when tasks are part of a parallel job.

It is worth noting that the probability of a task succeeding in the first attempt without past failures (not shown in the graphs) was found to be 98% and 97% in Google and CMU, respectively. We observe from the left-most graphs in Figure 2.44 that this probability drops dramatically to 42% and 28% in Google and CMU, respectively, in the first task retry (i.e. after making one failed attempt). In the second retry, a task’s success chance drops further to only 2% in Google and 11% in CMU. The probability of succeeding after more than two or three attempts becomes negligible in both clusters.

These results were similar when only considering multi-task jobs for both clusters in our dataset (middle graphs). Interestingly, when considering single-task jobs only (right-most graphs), we find that the probability of succeeding in the first task retry is much lower than in parallel jobs: 13% and 15% in Google and CMU, respectively. Note that the probability of a single-task succeeding without failures (not shown in the graphs), is 97% in Google and 90% in CMU, so these lower retry chances are not due to lower initial success chances. One possible explanation behind this observed behaviour in single-task jobs is the lack of race conditions that may affect the completion of parallel jobs, therefore making single-task jobs fail deterministically. Another possible reason why parallel jobs are more likely to complete after task retries is that parallel jobs can be configured by users to be marked as ‘completed’ upon the completion of a certain percentage of the job’s tasks, not necessarily all of the tasks. Unfortunately, information on this configuration parameter is not available in any of the datasets in our study.

Overall, these results show that retrying tasks more than once or twice can be a waste of resources, since the chances of succeeding drop significantly. To better understand the frequency of task retries in tasks that eventually fail, i.e. retries that end up wasting cluster resources, we study the distribution of the number of task retries in failed tasks in Figure 2.45, for both Google and CMU.

Note that we are not plotting task configuration parameters here, rather, this is the actual number of times a task was re-executed by the framework, before terminally exiting the cluster with a ‘fail’ status. We find that 70-90% of failed tasks had retried executing more than once, and 15-30% of failed tasks had retried executing more than twice. Some tasks tried re-executing up to 70–100 time before their terminal failure.

Effect of task failure on job reliability: So far we have analyzed the impact of task retries on task reliability. To better understand how this affects the reliability of the entire job
Chapter 2. Failure Analysis in HPC Systems

Figure 2.45: The distribution of the number of actual task retries in tasks that failed.

Figure 2.46: The relationship between task exit status and job exit status.

which a task belongs to, we now study the relationship between the status of tasks in a job and the job’s final status.

In both Google and CMU clusters, jobs spanned one or more tasks. In the CMU cluster, tasks either completed, failed, or got killed. In the Google traces, tasks either completed, failed, were killed, or got evicted.

Figure 2.46-(a) shows the fraction of jobs in Google’s cluster that experienced at least one of the possible task events, plotted separately for jobs that end up killed, jobs that fail, and jobs that complete. We repeat this analysis for the CMU Hadoop jobs in Figure 2.46-(b), with separate plots for mappers and reducers, since information on the type of the task is also made available in the CMU traces.

Based on Figure 2.46, we make the following observations about the relationship between task reliability and job reliability:

- In the Google cluster, it is rare for jobs that complete successfully to experience any unsuccessful task attempts: only 0.25% of completed jobs experienced at least one task...
failure. Things are somewhat different in the CMU cluster, where more jobs complete despite of unsuccessful task attempts: around 20-25% of completed jobs had at least one mapper task fail or get killed.

• We observe a strong correlation between having at least one task failure and the entire job failing. In Google, more than 98% of failed jobs had at least one task that terminally failed. In the CMU cluster, more than 95% of failed jobs had at least one mapper fail. Similarly, a single task kill is associated with the entire job getting killed.

Summary: For the clusters in our dataset where task re-execution is the common recovery mechanism, we find that attempting to re-execute a task more than once or twice is rather futile, as the chances of completing successfully become negligible. In both Google and CMU clusters, we find that the majority of failed tasks had tried to re-execute at least once, and that a single task failure is highly associated with the entire job failing.

2.7.3.4 Machine failures

Another possible root-cause behind job failures is machine failures. In the Google cluster, data on all machine removals from the cluster that happened during the data collection period are made available in the trace. The ‘removal’ of a machine could either be due to planned maintenance events or unplanned machine failures. According to the documentation that accompanied the Google traces [102], whenever a machine is removed, any task running on it will have a task ‘evict’ record logged in the traces, and the framework will re-submit the task for the scheduler to find another available machine for it.

We use the traces to correlate the time when a machine was removed from the cluster with the status of the tasks that were running on the machine. Our goal is to understand the effect of machine failures on job reliability by carefully examining the portion of tasks whose execution was interrupted due to machine removals. We identify these tasks by considering any task that was running on a machine when the machine logged a ‘removal’ event in the traces.

Table 2.11 shows the breakdown of such task events. For each event type, the left column shows what percentage of all machine related task events this type represents, and the right column reports the percentage of all events of this particular type that appear as a result of machine removals.

We find that 91% of task events correlated with machine removals are indeed logged as task ‘evicts’, and that these machine related evicts represent 16% of all task evicts in the traces. The rest of the evicts could be a result of higher priority tasks scheduled on the same machine. Note that task evicts generally represent a very small portion of task events in total for Google’s batch tasks: only 0.6% of all task events logged during the trace collection period are labeled as evicts.

Besides task evicts, we find that more than 8% of the events correlated with machine
removals were logged as task ‘kills’. One hypothesis is that these could be related to severe machine failures where no proper termination signal was sent to the tasks before the machine shut down. These machine related kills represent a small portion of task kills (0.04%).

Additionally, we find that only 0.3% of machine removal events are logged as task failures, and that these failures represent only 0.004% of all task failures in Google’s batch jobs. Finally, our analysis sheds light on the root-cause behind 1% of the tasks logged as ‘lost’ in Google’s traces, which is machine removals. Lost tasks represent only 0.02% of all task events in the Google cluster.

**Summary:** For the Google jobs in our dataset, we find that machine removals explain 16% of task evicts in the cluster, but can be identified as the root-cause behind task failure events very rarely, in particular only 0.004% of the time.

### 2.7.3.5 Cluster Resource Exhaustion

Our analysis of resource usage patterns in failed jobs in Section 2.7.2 showed that across multiple clusters operating in different organizations, jobs that failed consistently consumed more cluster resources than jobs which completed successfully. We also found that these jobs were configured to request more resources to begin with, for all the clusters in our dataset. In other words, these are jobs that were inherently designed to use the resources in the cluster more aggressively than the average submitted job. We found that failed jobs requested and consumed more I/O resources on average compared to other jobs, and exceeded their memory limits more frequently.

These observations suggest that resource exhaustion in large-scale clusters is a highly probable root-cause behind job failures. The aggressive utilization of disk drives and memory components in the cluster can result in different types of issues, including: running out of disk space or memory capacity on the physical machine, violating resource constraints and limits, or maybe even unmasking certain issues with the hardware components such as latent sector errors in hard drives [89] or correctable/uncorrectable errors in DRAM [49].

To further improve our understanding of the resource consumption patterns associated with failed tasks and jobs in large-scale clusters, we turn again to the Google cluster’s us-

<table>
<thead>
<tr>
<th>Google Task Event Type-X</th>
<th>Percent% of Machine Removal-Related Events</th>
<th>Percent% of Type-X Task Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVICT</td>
<td>91%</td>
<td>16%</td>
</tr>
<tr>
<td>KILL</td>
<td>8.4%</td>
<td>0.04%</td>
</tr>
<tr>
<td>FAIL</td>
<td>0.3%</td>
<td>0.004%</td>
</tr>
<tr>
<td>LOST</td>
<td>0.3%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 2.11: Breakdown of Google’s batch task events that were correlated with machine removals.
Figure 2.47: Distribution of the variability in resource consumption over a task’s lifetime in Google’s batch tasks.

How does resource consumption vary over time in a task’s lifetime?: So far we observed that failed tasks in the Google cluster consume more I/O on average and exceed memory limits more frequently than other tasks. We now ask the question of how this excessive usage of resources is manifested over the lifetime of an individual task, by studying the variability in resource utilization over time for each task.

In the Google traces, data on the average task usage of CPU, memory and I/O is logged over 5 minute timeslots. For many of the tasks that run for shorter periods than 5 minutes, records of the average resource consumption are also logged, often over the total lifetime of the task. In our analysis here, we consider tasks that reported more than one usage record in the traces.

Figure 2.47 plots the distribution of the calculated coefficient of variation (CoV) in task usage over time for Google’s tasks, broken down by task status. We find that the most notable difference in task usage variability is in I/O consumption (right most plot): failed tasks show lower variability in their I/O usage over time, compared to completed and killed tasks. This observation, coupled with our earlier analysis of average resource consumption, implies that failed tasks utilize I/O resources more aggressively than other tasks, with more steady rates over time. Failed tasks also report a higher median of CoV in memory usage than other tasks (middle plot), which agrees with our previous observation that failed tasks deviate from average memory consumption and exceed their maximum limits more frequently than others.
Chapter 2. Failure Analysis in HPC Systems

How does resource consumption vary between tasks that belong to the same job? Our analysis in subsection 2.7.2.4 on resource usage characterization showed that tasks working on a failing job report the lowest variability in I/O usage between them, when compared to other jobs.

One more question we are interested in investigating is how resource usage varied between tasks working on the same job, if some of these tasks completed successfully while others failed. This will help us further understand how failing tasks use cluster resources differently, by comparing their behaviour to successful co-tasks that were running the exact same binary.

More precisely, for each Google job in our traces with both failed and completed tasks, we calculate the ratio between the average resource consumption of the failed tasks divided by the average consumption of completed tasks in the job. Figure 2.48-(a) shows the distribution of this ratio for Google’s batch jobs. We find that in the majority of the jobs that we consider, failing tasks consume more I/O than completed tasks in the same job, with ratios of increase mostly in the 2–5X range. We compare the execution times between failed and completed tasks in the same job in Figure 2.48-(b), and observe less pronounced differences between them.

Summary: Our results show that cluster resource exhaustion is likely to be the root-cause behind the failure of a large portion of jobs in large-scale clusters. We find that failed jobs requested and consumed more memory and I/O on average than completed jobs, with more steady I/O usage rates over time (and across tasks in the same job). We also find that within a job, failing tasks consume more I/O than tasks that manage to complete successfully, if any.

2.7.3.6 User proneness to experiencing failures

So far we have identified several factors that affect job failures in the field, including job configuration parameters and resource usage patterns. We now turn our attention to the users who configured and submitted these jobs, to study if certain users were more likely to
experience more job failure rates and/or to utilize cluster resources differently. We focus on
the traces from Google and CMU where data on users and resource consumption is available.

The top two rows of graphs in Figure 2.49 plot for the Google and CMU users the num-
ber of job failures and job kills per task-week, for each user. Note that the data was sorted
by job failure rates, such that user at point x=0 has the highest failing rate. The graphs
below them plot for the same user on the X axis the average resources consumed by
the user’s jobs. In Google we have data on CPU, memory and I/O consumption; in CMU we
have data on HDFS and node local disk reads (Rd) and writes (Wr).

The first observation we make from the graphs is that job failure rates varied significantly
across users of a cluster, and that users with high failure rates are different than the users
with high killing rates, in both clusters.

In Google, we find that users with the highest failure rates are the same ones that report
the highest I/O utilization across all the users in the cluster, with factors of increase in
the 5-15X range, over the average I/O by any user at Google. We found that Pearson’s
Correlation Coefficient between user I/O consumptions and job failure rates is $R=+0.1$ (p-
value=0.1). On the other hand, the average memory and CPU usage by these failure-prone
users was lower than average, as shown in the figure. In the CMU Hadoop cluster, we

Figure 2.49: User job interruption rates and cluster resource usage in Google (left) and
CMU (right) clusters.
Chapter 2. Failure Analysis in HPC Systems

(a) Comparison of resource usage.
(b) Execution time.
(c) Comparison of resource usage (Apps with distinct users only).
(d) Execution time (Apps with distinct users only).

Figure 2.50: Comparison of resource usage and execution time between failed jobs and completed jobs that belong to the same application in the Google cluster.

find that the two amounts that exhibit a correlation are job killing rates by a user and the local filesystem writes per minute by the same user, with Pearson’s coefficient $R=+0.1$ (p-value=0.4).

Summary: Our results show that the way cluster resources are utilized varies across users and is correlated with the chances of their jobs completing successfully. For both Google and CMU clusters, users with the highest job interruption rates had utilized I/O resources in the clusters more heavily than the average user.

2.7.3.7 Application Issues

Another factor we consider in our analysis of possible root-causes behind job failures is the proneness of certain applications to failures, for example due to the presence of application bugs or unhandled exceptions in the application code. The only trace in our dataset that allows us to study this question is the Google data, where each job submission contained a field showing the ID of the application it belongs to. (Note that unfortunately there is no detailed information available on application exit codes, for example, which would be very helpful in exploring this question.)
The Google traces contain records of more than 4000 failed batch jobs representing 830 distinct applications. Of these applications, there are 608 ones that have records of other job executions completing successfully during the trace collection period.

We take a closer look into these Google applications that experienced both successful and failed job runs. For each application, we compute the average resource usage of its failed runs and divide that by the average usage of its completed runs. We also repeat this computation for execution time.

Figure 2.50 shows the distribution of these ratios, both for (a) resource consumption and (b) execution time. Furthermore, we repeat this analysis but when only considering applications that were executed by distinct users. This turns out to be a small subset (70 applications); the majority of applications were executed by a single user. The results for applications executed by distinct users are shown in Figure 2.50-(c,d).

**Summary:** More than 90% of failed applications in Google were invocations by single users (i.e. all jobs that belong to each application were submitted by one user). For the remaining applications whose jobs were submitted by distinct users, we find that the failed runs ran for shorter times, reported more I/O and CPU utilization, and consumed less memory than did the successful runs of the same application. These observations suggest that the way different users configured their jobs, even when running the same application, affects job behaviour significantly, and that most of the failing applications correspond to individual user submissions, which points again to user proneness to failures.

### 2.7.4 Job and task failure prediction

So far in this section we have studied the characteristics of failing jobs in large-scale clusters and investigated several possible root-causes behind job failures using real world traces. Our goal in this section is to try to predict these job failure events. Designing effective job failure predictors is necessary for minimizing resource waste in large-scale clusters, especially having observed that failing jobs tend to be long-running jobs that consume significant cluster resources.

We use the workload traces made available by Google (described in Section 2.7.1), to design and evaluate our job status predictor, since these traces contain detailed information on job and task executions and resource consumption. To classify and predict the final status of jobs and tasks in Google’s traces we utilize the machine learning analysis technique of *Classification and Regression Trees* (CART). Unless stated otherwise, we split our data randomly such that 70% of the rows are used for training the CART model, and 30% are used for testing.
2.7.4.1 Job level predictions

We begin by studying failure predictions on the job level. Our outcome (response) variable that we are interested in predicting is a job’s final status, and the explanatory (predictor) input variables which we feed our CART model are summarized in Table 2.12.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_status</td>
<td>Response</td>
<td>The outcome we are interested in predicting. A job’s status can either be COMPLETED, FAILED, or KILLED.</td>
</tr>
<tr>
<td>userID</td>
<td>Job Config</td>
<td>The ID of the user that submitted this job (either a Google engineer or a service).</td>
</tr>
<tr>
<td>logical_job_name</td>
<td>Job Config</td>
<td>The name of the program (application) that this job is running.</td>
</tr>
<tr>
<td>scheduling_class</td>
<td>Job Config</td>
<td>Each job has a scheduling class that drives machine local policies for allocating resources.</td>
</tr>
<tr>
<td>priority</td>
<td>Job Config</td>
<td>Job and task priorities determine if they are given preferences for resources.</td>
</tr>
<tr>
<td>num_tasks</td>
<td>Job Config</td>
<td>Each job spans one or more tasks. This determines the degree of parallelization of a job.</td>
</tr>
<tr>
<td>different_machines</td>
<td>Job Config</td>
<td>This flag, when enabled, means that all tasks in a job must be scheduled on different physical machines.</td>
</tr>
<tr>
<td>requested_cpu</td>
<td>Job Config</td>
<td>The amount of requested CPU capacity by a task in a job.</td>
</tr>
<tr>
<td>requested_memory</td>
<td>Job Config</td>
<td>The amount of memory requested by a task in a job.</td>
</tr>
<tr>
<td>requested_disk</td>
<td>Job Config</td>
<td>The amount of disk space requested by a task in a job.</td>
</tr>
<tr>
<td>task_failed_flag</td>
<td>Job Counters</td>
<td>Flag that is set if at least one task attempt failed in the job.</td>
</tr>
<tr>
<td>task_kill_flag</td>
<td>Job Counters</td>
<td>Flag that is set if at least one task was killed in the job.</td>
</tr>
<tr>
<td>task_evict_flag</td>
<td>Job Counters</td>
<td>Flag that is set if at least one task was evicted in the job.</td>
</tr>
<tr>
<td>avg_cpu_usage</td>
<td>Job Usage</td>
<td>The average amount of CPU used by a job’s tasks (5min window).</td>
</tr>
<tr>
<td>avg_memory_usage</td>
<td>Job Usage</td>
<td>The average amount of memory used by a job’s tasks (5min window).</td>
</tr>
<tr>
<td>avg_IO_usage</td>
<td>Job Usage</td>
<td>The average amount of I/O used by a job’s tasks (5min window).</td>
</tr>
<tr>
<td>CoV_cpu_usage</td>
<td>Job Usage</td>
<td>The variability in CPU usage among tasks in a job (5min window).</td>
</tr>
<tr>
<td>CoV_memory_usage</td>
<td>Job Usage</td>
<td>The variability in memory usage among tasks in a job (5min window).</td>
</tr>
<tr>
<td>CoV_IO_usage</td>
<td>Job Usage</td>
<td>The variability in I/O usage among tasks in a job (5min window).</td>
</tr>
</tbody>
</table>

Table 2.12: Summary of job status classification and prediction variables for the Google cluster.

In the rest of this section, we use standard metrics to evaluate our predictions:

- **Precision**: the percentage of predicted job failure events that are true.
- **Recall**: the percentage of actual job failure events that were successfully predicted.
- **Specificity**: the percentage of negative job failures (i.e. job successes or kills) that were correctly labeled.
- **Accuracy**: the percentage of predictions that are correct.

**Predicting job failures using configuration parameters only (pre-run).** We begin by asking the question: can we predict a job’s final status in the Google cluster using input data available before the job starts running? To answer this question, we consider job attributes under the category ‘Job Config’ in Table 2.12: the user and application IDs, the job’s priority and scheduling class, the number of tasks in the job, the amount of requested resources by a job, and whether or not there is an anti-affinity constraint on the job’s tasks.
Figure 2.51: Evaluation of job interruption predictions in the Google cluster using different input data as predictors.

Figure 2.51-(a) shows the prediction results when using pre-run job attributes. We find that the precision of our predictor is almost 100%, but the recall rate is low: only 20% of all failed jobs are correctly predicted by our CART classifier. Note that in the graph title we point out which variables were identified by CART as statistically significant to the classifier; in this case, the classifier used the amount of requested disk space by a job to predict if it fails or not.

It is worth noting that repeating this method for predicting job kills in the same cluster produces higher quality predictions with both precision and recall exceeding 90%, as shown in Figure 2.51-(b). To predict a job kill, CART utilized knowledge of the degree of parallelism in a job and the amount of CPU, memory, and disk requested.

So predicting a job kill is much easier using configuration data only. We turn again to job failures and explore how we can improve the recall rate, by exploiting some of the properties we observed in failing jobs earlier in Section 2.7.2.

**Predicting job failures using a 5 minute usage window.** Our earlier observation
on the high resource consumption of failing jobs motivated us to study if we can predict a job failure by monitoring its resource consumption over a certain time window after it starts running. We therefore feed our CART predictor, in addition to job configuration data, data on the average and standard deviation in CPU, memory and I/O consumption of tasks in a job over the first 5 minutes of job runtime, which we compute using the task usage traces.

Figure 2.51-(c) shows the results. We find that we can now correctly predict 10% more failed jobs than in the base case, with the same high precision.

**Predicting job failures using a task attempt failure flag.** Our analysis of failed jobs showed that a single task failure was strongly correlated with the entire job failing. Consequently, we explore if we can predict a job’s final status accurately as soon as one of its tasks makes a failed attempt. To study this question, we include in our input data to CART a single flag that is set to true if at least one task attempt failed in the job (didTaskFail).

We plot the results of our predictions when using this task failure flag in addition to job config attributes in Figure 2.51-(d). We find that including the didTaskFail flag improved our job failure prediction results significantly: we are able to correctly predict more than 80% of job failures with 85% precision.

This result shows that a single task attempt failure is a strong predictor of a job’s final status and can be used in conjunction with job configuration parameters to predict job failures accurately. However, to understand how effective such a predictor would be in practice, we examine when this first task failure happens during a job’s lifetime. For Google’s single-task jobs, we find that 70% of these jobs exit immediately after the job’s only task experiences a failure. Figure 2.52-(left), studies the distribution of the remaining time to job exit after the first task failure event when excluding single-task jobs from the data, and considering jobs that experienced at least one failed task attempt.
We find that 50% of failed jobs terminated within one hour of the first task failure. To interpret this result in terms of the remaining portions of job execution, we plotted the remaining percentage of job time after the first task failure in Figure 2.52-(right). We find that the median is within 60%; i.e., half of the parallel failed jobs in the cluster had run for 40% or less of their total execution time before the first task failure.

**Summary:** Our results on job level predictions show that a single task attempt failure, combined with job configuration data, can be used to accurately predict job failures in the Google cluster. We find that this method is worthwhile in parallel jobs in particular, where the average failing job had spent 60% of its execution time in the cluster after seeing the first task failure. This result motivates the question of whether we can predict the failure of a task in a job, which we explore next.

### 2.7.4.2 Task level predictions

On the task level, when repeating the same analysis discussed above for jobs, we found that using task configuration parameters results in zero recall and precision. Considering the first 5 minute usage data did not help, and when adding a task attempt failure flag, precision jumps to 88% and recall to 10%. To see if we can improve the quality of task level predictions, we experimented with two techniques: a) data oversampling, and b) sliding-windows.

**a) Data oversampling:** Oversampling is a way of adjusting the ratio between the positive and negative classes of data points in a training set, to correct for a bias in the original data. In our task level training dataset, the portion of failed tasks (i.e. positive labels) of all tasks in the trace is less than 2%.

We oversample failed tasks in our training set by experimenting with a ratio of non-failed tasks to failed tasks from 1 to 200, and we evaluate our predictor on an untouched testing set. Figure 2.53 shows the resulting precision and recall under the different ratios we experimented with.

We find that when using only task configuration parameters, precision barely improves (notice the x-axis limits). Recall goes as high as 80% under a training set with an equal number of failed to non-failed tasks. We make a similar observation for the 5 minute usage window scenario. When considering a task failure flag, there is an evident tradeoff between precision and recall over the range of ratios in our experiments (middle plot).

**Precision/Recall Tradeoff Discussion:** Learning the precision/recall tradeoffs is very helpful for the design and usage of failure prediction and mitigation strategies. For example, a conservative approach may favor a predictor with high precision even if it means compromising the portion of failures detected. Another, more relaxed approach may be willing to tolerate a higher rate of false positives (i.e. lower precision), in favor of detecting
more failed tasks. This decision partly depends on the desired action by the user (or the administrator), when failures are predicted. One possible action is terminating the job altogether, to save cluster resources. In such cases, the cost of false positives is expensive and could result in the undesired termination of healthy jobs and the (increasing) distress of cluster users.

**b) Sliding-window predictions:** The second approach we explore to improve task failure predictions utilizes time-series based techniques. Instead of only using the first 5-minutes of task usage data, we implement a sliding window method that moves an observation window over task usage data and feeds the CART predictor information on task utilization of CPU, memory, and I/O resources, dynamically; the predictor then updates the task failure prediction probability using the new usage data available, and so on. Additionally, each task’s configuration parameters are also fed to the CART model with every window prediction attempt, along with the updated usage data.

We experimented with windows of different sizes, from 5 minutes and up to 30 minutes. For each window size \( w \), we apply the prediction method to Google tasks that are at least \( w \) minutes long.

To evaluate the quality of our sliding window predictions on a per-task basis, we ask the following question: for each task in our training set, how many sliding windows made a correct prediction of the task’s final status? The left graph in Figure 2.54 shows the distribution of the number of windows that correctly predicted a failing task (i.e. the true positives); the right graph shows the distribution of the number of windows that falsely predicted the failure of a non-failing task (i.e. the false positives). Each line in the graphs corresponds to a different window size.

We find that 70–80% of failing tasks (left CDF) had at least one window with a correct failure prediction, and that more than 99% of non-failing tasks (right CDF) had zero windows with false predictions. (Note that the y-axis in the right CDF starts at 98%.)

We were also curious to examine *when* during a failing task’s lifetime does the first true
Chapter 2. Failure Analysis in HPC Systems

Figure 2.54: Per-task distributions of true positive and false positive sliding window predictions in Google’s cluster.

failure prediction take place. By taking a closer look at the logs we found that on average, the first window with a correct task failure prediction occurs half-way through a task’s lifetime.

Summary: For the Google cluster in our dataset, the close monitoring of task resource consumption using a combination of sliding windows and CART can improve the quality of task level predictions significantly. In 70–80% of the failing tasks in our data, the sliding window technique results in one or more windows with correct failure predictions (the range reflects different window sizes), with the first prediction happening on average half-way through a task’s lifetime. In 99% of non-failing tasks, the sliding windows produce no false failure predictions at all.

2.8 Summary and Implications

So far in this chapter our analysis of failure logs in different large-scale systems has shed light on the impact of different factors on system reliability and job reliability. We now summarize the key findings from our work in this chapter while deriving learned lessons and practical implications for the design and operation of large-scale clusters. At the end of this section, we provide a summary of the traces we utilized to study different aspects of large-scale system reliability.

2.8.1 Learned lessons and practical implications

- Failure Correlation: In agreement with prior work, we observe strong correlations between failures in HPC systems. In LANL’s clusters, during the day following a failure a node was 5–20X more likely to experience an additional failure, when compared to a random
day. Similar, albeit weaker trends exist across the nodes in the same rack: a node’s failure probability is increased by a factor of 3X during the day following another node failure in the same rack. Accounting for such spatial and temporal correlations can improve failure mitigation and tolerance techniques in HPC platforms significantly.

- **Failure-Type Correlations:** Interestingly, we observe that some types of failures increase the likelihood of follow-up failures more than others. In particular, power failures (such as power outages or voltage spikes) and network failures have a very strong effect on subsequent failures: 30–50% of nodes in LANL experience at least one failure in the week following a network or environmental failure, compared to only 2% in an average week. These observations are critical for creating effective failure prediction models, as they imply that such models should not only account for correlations between failures in time and space, but also consider the types of failures.

- **Power-Quality and Hardware Reliability:** We studied the impact of power outages, power spikes, UPS failures, and failures of a node’s power supply units on node reliability and found that they all lead to significantly increased hardware failure rates. Our observations on increased failure rates in memory DIMMs and node boards following power spikes, UPS failures and power supply problems suggest that after such events one might want to thoroughly inspect these hardware components for problems. Suspected fans should also be properly inspected in the case of a power supply failure since they were 40X more likely to fail in the following month, than in an average month. We find that bad or failing power supply can generally lead to many auto-correlated node outages and therefore should be quickly fixed or replaced.

- **Power-Quality and Software Issues:** Power outages have another interesting effect: significantly increased rates of software issues. A large fraction of the software failures following within a month of a power outage were either related to the distributed storage system or the file system. This observation might hold evidence that stronger mechanisms are required to protect storage and file system consistency in the face of power outages.

- **Temperature and Reliability:** The large cost of datacenter cooling motivated us to study the effect of temperature on node reliability. Based on our study of data spanning more than a dozen data centers at three different organizations, we find that the effect of high data center temperatures on system reliability are smaller than often assumed. For some of the reliability issues we study, namely DRAM failures and node outages, we do not find any evidence for a correlation with higher temperatures (within the range of temperatures in our datasets). For those error conditions that show a correlation (latent sector errors in disks and disk failures), the correlation is much weaker than expected. For (device internal) temperatures below 50C, errors tend to grow linearly with temperature, rather than exponentially, as existing models suggest. It is important to note that this does not
mean that high temperatures have no effect on hardware reliability or that the Arrhenius model is flawed. But it might mean that the effects of other factors dominate failure rates (e.g.: node usage, power issues, etc).

- **Temperature Variability and Excursions:** Rather than average temperature, we find that the variability in temperature might be the more important factor. Even failure conditions, such as node outages, that did not show a correlation with temperature, did show a clear correlation with the variability in temperature. Additionally, when studying the effect of node outages due to fan or chiller failures, which likely cause a brief period of very high temperatures inside a node, we do observe a strong increase in the subsequent rate of failures. Hardware components most strongly affected were MSC boards and midplanes but memory DIMMs, power supplies and node boards also experienced increased failure probabilities (>10X increase). This shows that hardware components are well able to tolerate higher average temperatures within the ranges that are typically observed in a datacenter. The harmful effects of temperature mostly stem either from high variability in temperature or from short periods of extremely high temperatures, for example due to a fan failure. Efforts in controlling such factors in HPC facilities are therefore critical in keeping hardware failure rates low.

- **Cosmic Radiation and Hardware Reliability:** Another environmental factor we studied is cosmic ray-induced neutron flux, which can lead to increased soft error rates. Interestingly, we observe no effect on failures due to DRAM errors, which might indicate that built-in error correcting codes are generally sufficient to mask bit flips in DRAM due to soft errors (and that those DRAM errors that lead to node outages are more likely due to hard errors). On the other hand, CPU failure rates, which did not show a strong correlation with other types of failures or environmental factors, such as power or temperature, are positively correlated with cosmic rays-induced neutron flux. This result points to the importance of employing mechanisms in CPUs that detect when cosmic-ray induced particles hit these devices, and fix any resulting miscalculations.

- **Failure-Prone Nodes:** We observe that some nodes fail significantly more frequently than others, even in systems where all nodes are identical in terms of their hardware. When we looked more closely at the most failure prone nodes in LANL’s systems, we found that they encountered higher-than-average rates of all types of failures (software, hardware, environment, network), but the increase was strongest for software, network and environment failures. One of the possible reasons that we investigated is the location of a node within the machine room, but we find no indication that certain areas in the machine room are more failure prone than others. Instead, we find that the failure prone nodes were typically used differently from the rest of nodes.
**Job Failure Characterization in Large Clusters:** Our trace-based analysis of job executions in large-scale clusters characterizes the behavior of failing jobs and identifies factors that show a strong correlation with a job's chances of completing. We find that the amount of resources requested and consumed by failing jobs were consistently larger than successful jobs across different clusters, particularly I/O consumption. Our analysis of job failures over users further shows that unsuccessful jobs and high I/O utilization levels are both correlated with a small subset of users who stress the I/O resources in the cluster more aggressively than the average user, and suffer from higher chances of job failures. Additionally, users who change the default values of job fault-tolerance and recovery parameters, such as task retry limits and speculative execution flags, also increase their chances of seeing job failures. Furthermore, when exploring the effectiveness of task retries in large clusters, we found that a lot of cluster resources are wasted on retrying failing tasks, and that the chances of succeeding after a second retry become negligible. An important practical takeaway from these observations is that more carefulness is required by users of large clusters in the way they configure and design their jobs. For example, the degree of parallelism in I/O intensive jobs can be chosen such that the amount of I/O processed per task is within the average I/O rates in the cluster.

**Job and Task Failure Predictions:** Our analysis of job failure predictions, based on traces from Google and using the practical machine learning method CART (Classification and Regression Trees), shows that a single task failure attempt is a strong predictor of the entire job failing. On the task level, we show that a combination of sliding-windows and CART can be used to predict task failures effectively. These results indicate that knowledge of a task's configuration parameters coupled with the monitoring of its resource usage behavior can be utilized to save significant cluster resources in large-scale installations.

### 2.8.2 Summary of traces and observations

Tables 2.13 and 2.14 in the next page provide an overview of all the traces we used in our study of failures in large-scale systems and a summary of the observations made based on these traces, respectively.
### Table 2.13: Overview of the different datasets used in Chapter 2.

<table>
<thead>
<tr>
<th>Data Source</th>
<th>Data Category</th>
<th>Data Source(s)</th>
<th>Data Collection</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANL</td>
<td>Failure Logs</td>
<td>LANL Failure Logs</td>
<td>9 years 2,918 node</td>
<td>Failures are correlated temporally and spatially in HPC systems.</td>
</tr>
<tr>
<td></td>
<td>Machine Layout</td>
<td>LANL Failure Logs</td>
<td>9 years 2,848 node</td>
<td>Specific types of failures are more predictive of follow-up failures.</td>
</tr>
<tr>
<td></td>
<td>Temperature</td>
<td>LANL Failure Logs</td>
<td>3.5 years 256 node</td>
<td>Power problems in large data centers are correlated with increased rates of hardware failures.</td>
</tr>
<tr>
<td></td>
<td>Job Logs</td>
<td>LANL System#20</td>
<td>3.5 years 256 node</td>
<td>High temperatures in data centers are correlated with increased rates of latent sector errors; however, the increase is smaller than what the theoretical reliability models (which are based on the Arrhenius equation) predict.</td>
</tr>
<tr>
<td>Google</td>
<td>Hard-Disk Errors</td>
<td>Google DRAM/HDD Data</td>
<td>2 years 72K HDD</td>
<td>High temperatures show no correlation with DRAM errors, HDD replacements, or server outages.</td>
</tr>
<tr>
<td></td>
<td>Hard-Disk Temperature</td>
<td>Google LSE/HDD Data, LANL System#20</td>
<td>2 years 270K HDD</td>
<td>Temperature fluctuations are more detrimental to hardware failures than increased temperature averages.</td>
</tr>
<tr>
<td></td>
<td>Hard-Disk Replacements</td>
<td>LANL Failure Logs, SciNet HW Logs</td>
<td>2 years 270K HDD</td>
<td>Cosmic-ray induced neutrons show no correlation with DRAM failures that lead to node outages.</td>
</tr>
<tr>
<td></td>
<td>DRAM Errors</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>2 years 200K DRAM</td>
<td>Few nodes in large-scale clusters tend to fail more frequently than others, and are typically used differently.</td>
</tr>
<tr>
<td></td>
<td>DRAM Temperature</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>2 years 200K DRAM</td>
<td>Job failures in parallel clusters are correlated with high resource consumption levels, particularly I/O.</td>
</tr>
<tr>
<td>Canada’s</td>
<td>Hardware Replacements</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>19 months 3,870 DRAM, HDD</td>
<td>A small subset of users who utilize a parallel cluster experience higher rates of job failures than the rest.</td>
</tr>
<tr>
<td>SciNet</td>
<td>Machine Layout</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>19 months 3,870 node</td>
<td>User proneness to submitting mis-configured jobs is an important root cause of unsuccessful job runs.</td>
</tr>
<tr>
<td>CMU</td>
<td>Job Logs</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>2.5 years 64 N/A</td>
<td>Job and task failures in parallel clusters can be predicted accurately by monitoring job resource consumption patterns and utilizing machine learning techniques.</td>
</tr>
<tr>
<td>National Oceanic &amp; Atmospheric Admin</td>
<td>Neutron Counts</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>10 years N/A N/A</td>
<td>Records of cosmic-ray induced neutrons in the atmosphere.</td>
</tr>
</tbody>
</table>

### Table 2.14: Summary of key observations from our analysis of failures in large-scale systems.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Data Source(s)</th>
<th>Section Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failures are correlated temporally and spatially in HPC systems.</td>
<td>LANL Failure Logs</td>
<td>Section 2.3.1</td>
</tr>
<tr>
<td>Specific types of failures are more predictive of follow-up failures.</td>
<td>LANL Failure Logs</td>
<td>Section 2.3.1</td>
</tr>
<tr>
<td>Power problems in large data centers are correlated with increased rates of hardware failures.</td>
<td>LANL Failure Logs</td>
<td>Section 2.5.1</td>
</tr>
<tr>
<td>High temperatures in data centers are correlated with increased rates of latent sector errors; however, the increase is smaller than what the theoretical reliability models (which are based on the Arrhenius equation) predict.</td>
<td>Google LSE Data</td>
<td>Section 2.5.2</td>
</tr>
<tr>
<td>High temperatures show no correlation with DRAM errors, HDD replacements, or server outages.</td>
<td>Google DRAM/HDD Data, LANL System#20</td>
<td>Section 2.5.2</td>
</tr>
<tr>
<td>Temperature fluctuations are more detrimental to hardware failures than increased temperature averages.</td>
<td>Google LSE/HDD Data, LANL System#20</td>
<td>Section 2.5.2</td>
</tr>
<tr>
<td>Cosmic-ray induced neutrons show no correlation with DRAM failures that lead to node outages.</td>
<td>LANL Failure Logs, NOAA</td>
<td>Section 2.5.3</td>
</tr>
<tr>
<td>Few nodes in large-scale clusters tend to fail more frequently than others, and are typically used differently.</td>
<td>LANL Job &amp; Failure Logs</td>
<td>Section 2.3.2</td>
</tr>
<tr>
<td>Job failures in parallel clusters are correlated with high resource consumption levels, particularly I/O.</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>Section 2.7</td>
</tr>
<tr>
<td>A small subset of users who utilize a parallel cluster experience higher rates of job failures than the rest.</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>Section 2.7</td>
</tr>
<tr>
<td>User proneness to submitting mis-configured jobs is an important root cause of unsuccessful job runs.</td>
<td>Google Job Logs, CMU Job Logs</td>
<td>Section 2.7</td>
</tr>
<tr>
<td>Job and task failures in parallel clusters can be predicted accurately by monitoring job resource consumption patterns and utilizing machine learning techniques.</td>
<td>Google Job Logs</td>
<td>Section 2.7</td>
</tr>
</tbody>
</table>
Chapter 3

Exploiting Failure Analysis in HPC Checkpoint-Scheduling

3.1 Introduction

3.1.1 Motivation

As the scale of High-Performance Computing (HPC) clusters continues to grow, their increasing failure rates and energy consumption levels are emerging as two serious design concerns that are expected to become more challenging in future Exascale systems. Efficiently running systems at such large scales critically relies on deploying effective, practical methods for fault tolerance while having a good understanding of their respective performance and energy overheads.

The most commonly used fault tolerance method is ‘checkpoint/restart’, where an application writes periodic checkpoints of its state to stable storage that it can restart from in the case of a failure. Under coordinated checkpointing, all compute nodes involved in a parallel application stop simultaneously to write their individual checkpoints, and that a failure of any one node involved in a parallel application requires all nodes to restart from their most recent checkpoints. While frequent checkpointing reduces the amount of lost computation in the case of a failure, it leads to a large amount of time spent checkpointing rather than performing computation. Conversely, the fewer checkpoints a system schedules, the higher the recovery overhead when failures happen.

Due to the importance of the problem, many checkpoint-scheduling policies have been proposed in the literature that focused on optimizing application runtime by minimizing the amount of time wasted in a system, either writing checkpoints or recovering lost computation.
after failures [28, 57, 64, 65, 70, 75, 101, 103, 104].

However, this prior work in the area of checkpoint scheduling suffered two main shortcomings. a) First, the high complexity of many of the proposed solutions in the literature made them difficult to implement in practice. Crude and ad-hoc rules of thumb are widely used in HPC organizations to decide on the frequency of checkpoints, such as “checkpoint once every hour”. Our discussions with HPC practitioners identified as a reason the high complexity of existing solutions. They often assume detailed knowledge about the underlying failure process, which is not readily available and which can furthermore change over the lifetime of a system. Moreover, existing solutions are perceived as too complex to implement, as they often do not provide straightforward closed-form solutions.

b) The second problem with existing work on HPC checkpoint-scheduling is that it primarily focused on optimizing application performance only—mainly by attempting to minimize the amount of time wasted either writing checkpoints or recovering lost work. Understanding the energy profile of the proposed solutions, however, or how to optimize them for other goals rather than performance, such as energy consumption or I/O time, remain open questions in the HPC community. Additionally, understanding the different energy/performance tradeoffs that result in an HPC machine when optimizing for one goal or the other, is also not yet well understood in practice.

Our goal in this chapter is to provide a comprehensive analysis of the performance, energy and I/O costs associated with a wide array of checkpoint scheduling policies, including policies that we propose as well as existing ones in the literature. Our proposed methods utilize failure log analysis in HPC systems to exploit different characteristics of real world failures in HPC checkpoint-scheduling. We account for various practical considerations in our design and analysis of the different methods and evaluate them using real world traces collected at production HPC installations. We also show that back-of-the-envelope formulas can be used to accurately estimate wasted work in checkpoint/restart systems.

The rest of this section will provide some background on checkpoint/restart definitions. We discuss related work in Section 3.2: subsection 3.2.1 provides an overview of related work on performance optimizations of checkpointing policies, while subsection 3.2.2 summarizes state of the art in energy optimized checkpointing in large-scale systems. We then present our own work on checkpoint-scheduling optimization and evaluation in HPC systems, first from a performance perspective (Section 3.3) then from an energy perspective (Section 3.4). Finally, we take a closer look into the implications that runtime-optimized and energy-optimized checkpointing methods have on the I/O subsystem in Section 3.5.
3.1.2 Checkpoint/Restart: Background

Our work in this chapter is focused on proposing and evaluating various checkpoint-scheduling policies in tightly-coupled HPC applications, which typically write coordinated checkpoints. We define these two concepts in more detail below as well as other checkpoint/restart terminology that we will be using extensively in this chapter.

- **Tightly-Coupled Computing**: In distributed systems design, a tightly-coupled system is one where all nodes or processes executing the workload are linked together and are dependent upon each other. A tightly-coupled HPC application therefore stops running when a single node fails or needs to be replaced. This is in contrast to loosely-coupled systems where processes work more independently of each other.

- **Checkpointing**: In computer systems, checkpointing is a fault-tolerance mechanism where an application writes a periodic snapshot of its state to stable storage, so that it can restart from that state in case of failure. Checkpointing is particularly critical for long running programs such as HPC applications to protect them against restarting from the beginning after failure events.

- **Coordinated Checkpointing**: In distributed systems, checkpoints can be coordinated or uncoordinated. Coordinated checkpointing is the most widely used fault tolerance protocol in tightly-coupled HPC systems, where all processes stop computing and synchronize to write a consistent state of the application to stable storage. If any one node fails, all nodes need to roll back and restart from the most recent checkpoint. In uncoordinated checkpointing, processes write their local checkpoints independently.

![Figure 3.1: Example of a compute cycle in a checkpoint/restart system.](image)

- **Checkpoint Interval** ($\Delta$): The elapsed time between two consecutive checkpoints in a system. (See Figure 3.1 for an example of a checkpoint/restart cycle.)

- **Checkpoint Cost** ($C$): The time needed to write one checkpoint of the application to stable storage. We use $T_{\text{ckpt}}$ to denote the total amount of checkpointing time in an application’s lifetime; i.e. the number of checkpoints it writes multiplied by $C$.

- **Restart Time** ($R$): This is the time required before an application can recompute lost work after a failure event, which includes application initialization code and any overhead associated with restarting the system after a failure.
• **Recovery Time** \((T_{\text{recomp}})\): The time needed for the application to redo any lost computation that has been done since the most recent checkpoint. We use \(T_{\text{recomp}}\) to denote the total amount of recompute time in a system; i.e. the number of failure events multiplied by the lost work units after each failure.

• **Total Wasted Time** \((T_{\text{wasted}})\): The total amount of time wasted in a checkpoint/restart system, either writing checkpoints or recovering lost computation in the case of failures.

### 3.2 Related Work

#### 3.2.1 Performance Optimization of HPC Checkpointing

A large body of work over the years has focused on studying the minimization of the completion time of HPC applications (or the minimization of the total wasted work), by finding the optimal interval between periodic checkpoints [28, 57, 64, 65, 75, 103, 104].

In the 1970s, Young [104] presented the first optimal checkpoint model that minimizes the total waste time in a system using a periodic, constant checkpoint interval. Young’s formula was derived under the assumption that system failures arrived according to a Poisson process with a fixed failure rate. In other words, Young assumed that the times between failures followed an exponential, memoryless distribution. Based on Young’s work, Daly [28] proposed a higher order approximation of the checkpoint interval that accounts for failures during recovery (i.e. does not assume fault-free checkpointing process, which Young [104] did), while still assuming that failure inter-arrivals follow an exponential distribution.

In a subsequent study [57], the authors explored the impact of sub-optimal checkpoint intervals on application performance. They conducted a simulation-based study to answer the question of whether and how underestimating or overestimating the checkpoint interval impacts application efficiency, and compared the results to analytical models. Their results suggest that underestimating a checkpoint interval, i.e. choosing a shorter interval that leads to checkpointing more frequently than the optimal case, can be more severe than overestimating the checkpoint interval [57].

A more general model, parametrized by the system’s process failure distribution, was proposed by Bouguerra et al. in [19]. Bouguerra derived a probabilistic model that minimizes the average completion time of a HPC application, independent from the failure inter-arrival process type. Through simulations, they evaluated the effectiveness of this model for two common failure distribution models: Poisson model and Weibull model. The parameters of the Weibull model were based on an empirical analysis of failure logs collected at LLNL’s ASC White supercomputer. The results of their work suggested that the optimal checkpoint interval can be periodic when the checkpoint cost (overhead) is constant, and when the distribution of failure inter-arrivals follows either an exponential or a Weibull distribution. On the other hand, work done by Ling et al. [64] concluded that theoretically
a constant checkpoint interval is actually optimal if and only if the system failure follows a Poisson process.

More research followed that moved away from studying periodic (constant) checkpointing to aperiodic (non-constant) checkpoint placement techniques [18, 65, 75]. Ozaki et al. [75] developed approximation methods to finding checkpoint sequences based on ‘variational calculus’ concepts and ‘min-max’ numerical methods. Lui et al. [65] proposed a checkpoint frequency function that generates a sequence of varying checkpoint intervals derived from a system’s actual reliability model. The optimal sequence of checkpoint intervals is generated using the theory of a stochastic renewal reward process, with focus on Weibull distributions of failure inter-arrivals.

More recently, a study by Bougeret et al. [18] developed a dynamic programming algorithm to minimize the expected execution time of a system using varying checkpoint intervals, while taking into account the degree of parallelism of the running job. For parallel jobs, they used different models to reflect different scenarios of parallelism: embarrassingly parallel jobs, generic parallel jobs, and numerical kernels. They evaluated their technique by running simulations that assume failure inter-arrivals follow an exponential model and a Weibull model for a wide range of system sizes (i.e. number of processors). They compared their approach with other checkpointing methods proposed in the literature, highlighting scenarios where their approach outperformed the other ones, such as in the case of Weibull failures in very large platforms [18].

The goal of our work in this chapter is not simply to propose further refinements over these existing approaches. Instead, our focus is to first carefully evaluate the performance of different checkpointing solutions using real world traces, while taking practical considerations into account. The methods we evaluate include previously proposed ones in the literature as well as new ones that we propose which exploit different characteristics of failures in real world systems (see Section 3.3). The second focus of our work in this chapter is the evaluation and optimization of various checkpointing methods from an energy point of view. We next discuss related work in the area of energy optimization of HPC checkpointing.

### 3.2.2 Energy Optimization of HPC Checkpointing

Traditionally, work on checkpoint policies has been focused on optimizing the application runtime by minimizing the associated overheads, namely the time that is spent writing periodic checkpoints and the time that is spent to recover the lost work in case of a failure.

However, checkpointing policies optimized for application runtime are not necessarily optimal from an energy perspective. For example, while time spent writing a checkpoint and time spent redoing work lost after a failure count equally towards the completion time of the application, they do not affect the power budget in the same way, since writing a checkpoint consumes less energy than doing computation [30, 32, 47, 67].
Understanding the energy overheads associated with different methods for scheduling checkpoints, as well as how they can be optimized for energy consumption, in combination with their effects on application runtime, are critical, yet not very well understood questions in the HPC community.

While there has been work on reducing the power consumed during an individual checkpoint, e.g. by using DVFS (Dynamic Voltage and Frequency Scaling) during a checkpoint [68], possibly combined with the use of energy-efficient NAND flash memory [88], less attention has been paid to the question of how to schedule checkpoints for energy efficiency.

To this date, we are aware of only two papers related to checkpoint scheduling in the context of energy consumption [14, 67]. Both papers derive a formula for determining a constant (static) checkpoint interval that optimizes for energy savings in the system (rather than runtime savings). Meneses et al. [67] propose an energy-optimized formula in order to compare the energy efficiency of checkpoint/restart to other fault tolerance protocols: message logging and parallel recovery. They show through analytical modeling that parallel recovery is expected to outperform both (static) checkpointing and message logging at extreme scales, in terms of performance and energy.

Aupy et al. [14] also propose an analytical model for a constant checkpoint interval, for the primary purpose of deriving projections on the energy efficiency of future HPC platforms. Neither papers evaluate their formulas using real world traces or investigate other adaptive (non-constant) checkpoint-scheduling techniques in their energy analysis.

In Section 3.4 in this chapter we use real world traces to conduct a detailed evaluation study of the energy/performance tradeoffs introduced in practice by a wide array of checkpointing methods, when optimized for energy or for runtime savings. We include in our study both static and adaptive checkpointing solutions, and analyze the impact that energy optimizations have on the I/O subsystem in HPC machines.

3.3 Performance Evaluation of Checkpoint Scheduling Policies

Performance overhead due to faults and fault tolerance in systems using checkpointing comes from two different sources: the time that is spent writing periodic checkpoints and the time that is spent to recover in the case of a failure (i.e. the time to revert back to the state of the most recent checkpoint and the time to redo all the lost work that has been done since the most recent checkpoint). Hence the amount of time that is wasted, i.e. any time that is not spent on doing actual computation, depends on the system’s failure rate, the amount of time it takes to write a checkpoint, and the frequency of checkpoints.

Optimizing the checkpoint scheduling process is therefore a crucial problem in HPC sys-
tems: while frequent checkpointing reduces the amount of lost computation in the case of a failure, it leads to a large amount of time spent checkpointing rather than performing computation. Conversely, the fewer checkpoints a system schedules, the higher the recovery overhead when failures happen.

In this section, we provide a detailed evaluation of the performance overhead associated with different checkpoint scheduling solutions that vary in their degree of complexity. We base our evaluation on real-world failure traces and discuss different practical considerations for each checkpointing method. We begin by looking at simple, static (constant) checkpointing intervals in Section 3.3.1 then study more adaptive (non-constant) techniques in Section 3.3.2.

3.3.1 Static Checkpointing

The first and simplest approach for computing the checkpoint interval is the closed-form solution proposed by Young [104] in 1974. Young’s formula determines the checkpoint interval $\Delta_{\text{Young}}$ based on only two quantities, the system’s mean time to failure (MTTF) and the checkpoint cost $C$:

$$\Delta_{\text{Young}} = \sqrt{2 \cdot C \cdot MTTF}$$

(3.1)

Young’s formula relies on the following set of unrealistic assumptions, which make its practical feasibility questionable and has spurred a sizable body of follow-up work that strives to resolve these assumptions:

- It is assumed that failures follow a Poisson process with a rate that is stable throughout the system’s lifetime. Prior work, including our own [33, 48] reports dependencies between failures and non-exponential inter-arrival time distributions. A number of papers [19, 66] extend the analysis to Weibull distributions, known to be a better model of empirical distributions.

- The possibility that failures happen during checkpoints (in which case the lost work would be equal to $\Delta$) is neglected. In practice, there are applications with a checkpoint cost that is high enough that the probability of experiencing a failure during a checkpoint is significant. Several recent papers [19, 28, 56, 66] take into account the probability of failure during checkpoints.

- It is assumed that failures happen on average half-way between two checkpoints. Work by Liu [65] removes this assumption by approximating the excess lifetime distribution of the time between failures.

In the remainder of this section our goal is to evaluate the performance of Young’s method, while assessing the impact of its unrealistic assumptions on real-world systems.
3.3.1.1 Evaluation of Young's accuracy for real traces

In this subsection we use trace-driven simulations to evaluate the performance of the solution provided by Young. The traces we use are available online [6] and cover nearly a decade worth of failure logs from 10 different HPC systems at Los Alamos National Lab (LANL). The data contains records of all node outages that occurred during the measurement period (a total of 18,316 node failures).

We selected these 10 LANL systems (out of a larger dataset) by studying the best Weibull fit to the failure data in each LANL system whose failure logs are made available, and choosing representative systems of varying Weibull distribution shapes (therefore representative of different failure inter-arrival distributions). Table 3.1 below describes the 10 clusters in our study in more detail, including the value for the Weibull shape parameter, which ranges from 0.545 to 0.889:

<table>
<thead>
<tr>
<th>LANL System ID</th>
<th>#CPU Months</th>
<th>MTTF (days)</th>
<th>#Failures</th>
<th>Weibull shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>636,928</td>
<td>0.58</td>
<td>7104</td>
<td>0.739</td>
</tr>
<tr>
<td>18</td>
<td>164,350</td>
<td>0.31</td>
<td>3997</td>
<td>0.817</td>
</tr>
<tr>
<td>19</td>
<td>122,196</td>
<td>0.33</td>
<td>3284</td>
<td>0.889</td>
</tr>
<tr>
<td>20</td>
<td>91,438</td>
<td>0.57</td>
<td>2478</td>
<td>0.646</td>
</tr>
<tr>
<td>12</td>
<td>22,787</td>
<td>2.66</td>
<td>259</td>
<td>0.615</td>
</tr>
<tr>
<td>3</td>
<td>12,179</td>
<td>2.46</td>
<td>299</td>
<td>0.823</td>
</tr>
<tr>
<td>9</td>
<td>5,710</td>
<td>2.43</td>
<td>280</td>
<td>0.546</td>
</tr>
<tr>
<td>11</td>
<td>5,622</td>
<td>2.51</td>
<td>268</td>
<td>0.564</td>
</tr>
<tr>
<td>10</td>
<td>5,608</td>
<td>2.85</td>
<td>237</td>
<td>0.545</td>
</tr>
<tr>
<td>21</td>
<td>1,763</td>
<td>1.00</td>
<td>110</td>
<td>0.685</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of the LANL clusters in our dataset.

Our simulations assume a “hero run” of an application that uses all available nodes. For each system, we use the entire log and simulate periodic checkpoints at fixed intervals that we compute using Young’s formula (recall Equation (3.1)) and record the fraction of time that is lost due to writing checkpoints or redoing lost computation. (Note that we do not include the time needed to restart the application to the state of the most recent checkpoint, as this time does not depend on the checkpointing interval and hence will be the same for any checkpointing policy).

We run simulations varying the checkpointing cost $C$ from as low as 20 seconds to as high as 60 minutes, and we obtain the MTTF for a system from the corresponding trace. For a measure of how good the provided solution is we compare it to the wasted work the system would have experienced under the optimal fixed checkpointing interval, i.e. $\Delta_{\text{Opt}}$ that leads for a given trace to the smallest fraction of wasted work. We obtain $\Delta_{\text{Opt}}$ by searching through the entire range of $\Delta$ values and using our simulator to identify the one that performs best for each LANL system in our data.

Figure 3.2 compares the fraction of time that is wasted under $\Delta_{\text{Young}}$ to that under $\Delta_{\text{Opt}}$, for all LANL systems, and under different $C$ costs. We observe that the fraction of wasted time under Young’s formula is very close to the optimal, in most cases within 2%.

Interestingly, even for input scenarios that deviate significantly from Young’s assump-
Chapter 3. Exploiting Failure Analysis in HPC Checkpoint-Scheduling

Figure 3.2: Wasted time under Young compared to wasted time under the optimal fixed checkpoint interval for all LANL systems in our dataset, and under different checkpoint cost assumptions.

...checkpointing using $\Delta_{Young}$ is near optimal. For example, systems whose shape parameter of the best Weibull fit to the data deviates the most from an exponential distribution (recall that an exponential would have a shape parameter of 1) do not exhibit worse performance than others. Also, we find that a higher checkpointing cost (which will increase the chance of failures during checkpoints) does not reduce performance of $\Delta_{Young}$ compared to $\Delta_{Opt}$.

It is worth noting that we repeated this evaluation for another static method commonly referenced in the literature as an analytical improvement over Young’s formula, which is the higher order approximation of the optimum checkpoint interval proposed by Daly [28]. We find that checkpointing using Daly’s solution produces very comparable results to Young’s interval, for all the LANL systems in our data. Table 3.2 at the end of this section reports the detailed results for $\Delta_{Young}$, $\Delta_{Daly}$ and $\Delta_{Opt}$, under different $C$ cost assumptions (see columns under “Static” methods).

Summary: Despite a number of unrealistic assumptions that the derivation of Young’s formula relies on, it achieves performance nearly identical to that under the optimal checkpoint interval (identified offline through exhaustive search). This is the case even for input scenarios that significantly deviate from Young’s unrealistic assumptions.

3.3.1.2 Sensitivity to accuracy in parameter estimation

The high quality performance of Young’s solution motivated us to take a closer look into this method. Young’s formula requires two types of information: the the cost $C$ of a checkpoint and the system’s MTTF. In this subsection we study how sensitive the performance of Young is to errors in the estimation of those parameters.
We apply the same trace-based simulation we relied on in Section 3.3.1.1, but rather than determining the checkpoint interval based on the actual values for $C$ and MTTF we assume that they were estimated with varying degrees of error. We range the degree of error between $(1/5)X$ (i.e. the $C$ or MTTF was underestimated by a factor of 5) to a degree of error of $5X$, i.e. the $C$ or MTTF was overestimated by a factor of 5.

Figure 3.3 shows the results when the MTTF is estimated with varying degrees of error while $C$ is estimated accurately in two representative LANL systems (systems 2 and 20). The X-axis shows the degree of error and the Y-axis shows the resulting wasted time normalized by the wasted time that would have resulted under error-free parameter estimation.

We observe that there is a large range of MTTF values that achieve almost identical, near optimal performance. For example, to stay within a range of 5-10% of the wasted work achieved under accurate parameter estimates one can tolerate errors of a factor of 2 (either over- or underestimation) in estimating the MTTF.

Performing the same sensitivity analysis for checkpointing cost $C$ (while assuming correct estimation of the MTTF) leads to the same results. The reason is clear when looking at how checkpoint intervals are calculated using Young’s formula (recall Equation (3.1)). Since $C$ and MTTF are multiplied, an error of a factor $X$ in the MTTF will lead to the same checkpoint interval as an error of a factor $X$ in the estimate of $C$.

**Summary:** We conclude that checkpointing based on Young’s formula is quite robust against reasonable errors in the estimation of $C$ or MTTF. This result motivates us to explore ways to implement Young in practice through adaptive checkpointing techniques, which we discuss in the next section.
3.3.2 Adaptive Checkpointing Policies

Section 3.3.1 demonstrates the good performance results obtained when using simple static methods with a fixed checkpoint interval. These results motivate us to investigate more advanced, non-constant checkpointing methods, which adapt the checkpointing interval dynamically.

We consider three classes of methods that inherently rely on the same principle: they exploit different characteristics of real world HPC failure processes to dynamically obtain (and continuously update) an accurate MTTF estimate and then plug this estimate into Young’s formula (recall Equation (3.1)) to determine the length of the next checkpoint interval. More precisely, we investigate methods that exploit the following failure properties: variability in system MTTF, decreasing hazard rates and failure autocorrelation.

3.3.2.1 Variability in the system MTTF

The results obtained under the static checkpointing methods in Section 3.3.1 made two assumptions about the system MTTF: it is known beforehand, and it is stable over the lifetime of a system. However, it is not feasible in practice to assume a-priori knowledge of a system’s real MTTF. Furthermore, empirical studies of failures in HPC systems report that a system’s MTTF varies over its lifetime [90]. We therefore propose that a checkpointing policy dynamically maintains an estimate of the system’s MTTF, based on a system’s recent failure history, using one of three implementations of Moving Averages:

- **SMA**: This approach uses a Simple Moving Average, i.e. it simply calculates the MTTF as the average value of the failure inter-arrival times within an observation window consisting of the last \( w \) days and uses that average as an estimate of the expected time to the next failure. \( w \) is a parameter that needs to be determined.

- **WMA**: This method works like SMA, but uses a Weighted Moving Average, i.e. it assigns a weight for each value in the observation window, with more recent observations having greater weights. WMA, therefore, considers recent failure inter-arrival times more predictive of the time to next failure, than older ones.

- **EMA**: EMA uses an Exponential Moving Average to estimate the MTTF, i.e. the weights of older observations decrease exponentially, giving past values a diminishing contribution to the calculated average. Unlike SMA and WMA which only consider values within the observation window, EMA is a cumulative calculation that includes all the historical observations (the entire history of failures).

Note that the only requirement of the above algorithms is that the system maintains a log of recent failure inter-arrival times (or a log of all failure times in the case of EMA) in
order to compute the MTTF estimate dynamically, making them straightforward to implement in practice.

We use trace driven simulations based on the LANL data to evaluate the performance of SMA, WMA, and EMA. Figure 3.4 shows the results for four LANL systems, systems 2, 18, 19, 20 (the four systems that the largest amount of data is available). Table 3.2 at the end of this section shows the complete results for all LANL systems.

Each graph plots for one of the systems the wasted time for each of the three algorithms as a function of the window size $w$ that was used. In the case of the EMA method, which does not rely on a specific time window, the parameter $w$ on the X-axis is used to vary the smoothing coefficient $\alpha$. In particular, $\alpha$ is computed as $\alpha = 2/(w + 1)$. 

Figure 3.4: Checkpointing using different techniques that exploit system MTTF variability over time, in the four largest LANL systems in our dataset.
To evaluate the quality of the produced solutions each graph shows for comparison the wasted time that would have resulted under Young’s static checkpoint interval, \textit{Young(stat)}; i.e. the method that calculates a fixed checkpoint interval to be used over the entire trace based on the system’s MTTF. We also compare our results to the case of checkpointing every one hour, a scenario commonly applied in practice. We observe the following:

• The performance of all moving averages methods and their ability to estimate the MTTF is nearly the same.

• For all systems one can achieve performance comparable to that of \textit{Young(stat)}, for a large range of \( w \) values.

• For some LANL systems (system 2 and system 20) running Young on the MTTF estimates actually performs slightly better than \textit{Young(stat)}. The reason is that for these two systems failure rates are more variable over the course of their life, as illustrated in Figure 3.5. Using a dynamic estimate of the MTTF, rather than the average across the entire trace, can actually slightly improve performance.

• The performance is not overly sensitive to the choice of \( w \) making tuning easy. Values for \( w \) of 30 days or more performed well for all systems. Except for system 2, only a few days worth of data provide nearly optimal performance, which is good news when introducing any of the proposed methods on a system with no prior recording of failures.

\textbf{Summary:} Using a combination of Young’s formula and simple moving averages of past failure one can achieve performance comparable to the (hypothetical) case where the optimum checkpoint interval is known a-priori.
### 3.3.2.2 Decreasing Hazard Rates

A number of previous studies [18, 19, 65] suggest placing checkpoints dynamically based on the statistical distribution of the time between failures. The motivation is that, unlike the exponential distribution, empirical distributions often exhibit decreasing hazard rates (as indicated by a shape parameter less than 1 in a Weibull distribution; see column “Weibull shape” in Table 3.1).

A decreasing hazard rate function predicts that if a long time has elapsed since the last failure then the expected remaining time until the next failure is long. The intuition is that in a system with decreasing hazard rates one can reduce the checkpoint frequency if a long time has passed without seeing any failures (as a long time without failures implies a longer expected time until the next failure).

We are mainly interested in exploring the general potential of hazard-rate based methods, without having to worry about issues due to the potential loss of accuracy when fitting a theoretical distribution to the empirical data, as required by previous approaches. We therefore rely directly on the failure trace data to determine for each system how exactly the expected time to the next failure depends on the elapsed time since the last failure.

Figure 3.13 plots the expected remaining time to the next failure as a function of the time since the last failure for systems 2, 18, 19, and 20; i.e. datapoint $(x, y)$ means that after running without failures for $x$ minutes the expected remaining time until the next failure (on top of the $x$ minutes already completed) is $y$ minutes. Not surprisingly, given the parameters of the fitted distribution in Table 3.1, we observe an increasing trend in all curves.

We use the curves in Figure 3.13 to implement an adaptive checkpointing method, called Hazard. Any time a new checkpoint interval calculation is made (i.e. after a failure or a checkpoint), the curves in Figure 3.13 are used to determine the expected time until the next failure as a function of how much time has elapsed since the last failure. This estimate is then plugged into Young’s formula to determine the length of the next checkpoint interval.

The column labeled “%wasted Hazard” in Table 3.2 at the end of this section shows the results obtained from running Hazard on the LANL traces, compared to our old Young(stat) from Section 3.3.1.

Overall, we observe that improvements are marginal. We identify as the main reason that while failure rates do change as a function of the elapsed time since the last failure, this change happens too slowly to have a large impact. For example, in systems 2 and 20 it takes 1000 minutes of failure free execution before the expected time until the next failure doubles from initially around 800 minutes to 1600 minutes (leading to an increase of 1.4X in the checkpoint interval). However, only 23% of all failure intervals are longer than 1000 minutes, so the number of opportunities where this knowledge can be brought to bear is limited.
Figure 3.6: Expected remaining time to fail given time since last failure.

We experimented with synthetically generated failure traces using smaller Weibull shape parameters than the shape parameters observed for the LANL data. We do find that improvements increase for smaller shape parameters. For example, for a shape parameter of 0.3 we observe an average improvement of 10% for Hazard over Young.

**Summary:** For the LANL systems in our data, improvements from placing checkpoints dynamically based on the hazard rate function are modest. We observe higher improvements in synthetic experiments with shape parameters much smaller than those observed in practice.

### 3.3.2.3 Failure Autocorrelation

In this section, we propose to take information about the burstiness of the failure process into account when making checkpointing decisions. To quantify the burstiness and degree of correlation between failures in LANL’s systems we plot in Figure 3.7 both the autocorrelation and partial auto-correlation functions of failure inter-arrivals in one of LANL’s systems, system 20. We observe strong positive auto-correlations between failure inter-arrivals. When repeating this analysis for the rest of LANL’s systems we found similar trends.

These observations motivate us to use autoregression (AR) to model the time between failures. We fit an AR model to the observed sequence of failure inter-arrivals for each LANL system and then use the fitted model to predict the time to next failure each time
a checkpoint scheduling decision is to be made; i.e., after a system failure occurs. The new checkpointing interval is determined by plugging the estimate from the AR model into Young’s formula. Results are shown in the column labeled “%wasted AR” in Table 3.2 at the end of this section.

We observe that in most cases AR provides the lowest level of wasted work among all policies. (For better readability, we have marked in each row in Table 3.2 the lowest observed level of wasted work in bold font). The improvements are largest for systems 11 and 21, with levels of wasted work that are 10-15% lower under AR than under Opt (the optimal interval identified through exhaustive search). In most cases we observe an average improvement of AR over Young(stat) of 4% and over Opt of 6%.

**Summary:** Among all methods, using autoregression to estimate the expected time until failure performs best. The improvements fall in the 3–6% range in most cases, but are more significant (in the 10–15% range) in some systems.

### 3.3.3 Performance Estimation Using Equations

So far in this chapter we have relied on simulations based on actual failure logs to determine the fraction of wasted time in a system when using different checkpoint scheduling policies. In practice, it is often useful to have a simple back-of-the envelope estimate of wasted work available without having to run simulations. Examples include situations where no failure logs are available for a system, or when one wants to experiment with parameters (e.g. the MTTF) that differ from the real system. This allows answering questions such as “How much does the fraction of wasted time drop if I could reduce the checkpoint overhead by a
<table>
<thead>
<tr>
<th>System ID</th>
<th>MTTF (min)</th>
<th>Weibull C (Shape)</th>
<th>Static</th>
<th>Dynamic (Moving Averages)</th>
<th>Advanced Methods</th>
</tr>
</thead>
</table>
|           |            |                   | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wasted | % wastewater overedd in terms of wasted work is marked in **bold** font.)
factor of two” or “How many processors can I run on while still keeping the wasted time below some threshold”.

In this section, we explore several approaches to estimate the fraction of wasted time in a system. We first discuss the mathematical derivation of different equations that we propose for approximating wasted time, then we evaluate the accuracy of the estimates they produce for real world systems using LANL’s failure traces.

3.3.3.1 Derivation of formulas for wasted work

Our goal is to estimate the fraction of time wasted in an HPC system performing periodic checkpoints with an interval $\Delta$. We begin with a basic formula that makes a number of simplifying assumptions about the sources of wasted time in a system, then derive more complex formulas that attempt to enhance these assumptions.

The first component of wasted work is due to the fact that once every $\Delta$ time units a checkpoint needs to be written, which takes time $C$. Hence, the system spends $C/\Delta$ fraction of its time checkpointing. Secondly, every time a failure happens (i.e. on average once every MTTF time units), some work is lost that needs to be recomputed. The amount of lost work is equal to the time since the last checkpoint. If failures are equally likely to happen anywhere in a checkpointing interval, the expected amount of lost work for each failure would be roughly $\Delta/2$. That means on average the fraction of time spent redoing lost work is $(\Delta/2)/MTTF$.

Combining these two sources of wasted work, we end up with the following simple formula for wasted work $W$:

$$W(\Delta) = \frac{C}{\Delta} + \frac{\Delta}{2 \cdot MTTF}$$  \hspace{1cm} (3.2)

In reality, however, checkpoints happen every $C+\Delta$ time units. We fix this as follows:

$$W(\Delta) = \frac{C}{C+\Delta} + \frac{\Delta}{2 \cdot MTTF}$$  \hspace{1cm} (3.3)

We further refine this equation by taking into account that checkpoints take place only during the fraction of failure intervals that are larger than $\Delta$, which we estimate by assuming an exponential distribution (an approximation we make for simplicity, as in theory a Weibull distribution is a better fit). This modification leads to the following formula for wasted work $W$:

$$W(\Delta) = e^{-\frac{\Delta}{MTTF}} \cdot \frac{C}{C+\Delta} + \frac{\Delta}{2 \cdot MTTF}$$  \hspace{1cm} (3.4)

We now turn to examine if we can refine our estimation of the second component of wasted work: the amount of computation that needs to be redone in the case of a failure. So far we have made the assumption that failures happen on average half-way between...
checkpoints; i.e. during regular computation. A more realistic assumption takes into account that failures can occur during checkpoints as well, especially when the checkpointing cost is high.

In the case that a failure happens during a checkpoint, the amount of lost work is equal to all the computation done by the application since the last checkpoint; i.e. $\Delta$ time units. This is different from the estimated amount of lost work when a failure happens between two checkpoints, which we previously approximated to be on average $(\Delta/2)$ time units. To account for both scenarios when estimating wasted time, we first derive the expected value of recompute time considering both the probability of failure during a checkpoint ($Pr_{\text{checkpt}}$), and the probability of failure during regular computation ($Pr_{\text{comp}}$).

We use $X_{\text{checkpt}}$ to denote the random variable that represents the amount of lost work when a failure happens during a checkpoint, and $X_{\text{comp}}$ for the amount of lost work when a failure happens during regular computation.

The expected value of recompute time $E(X)$ can therefore be expressed as:

$$E(X) = Pr_{\text{checkpt}} \cdot E(X_{\text{checkpt}}) + Pr_{\text{comp}} \cdot E(X_{\text{comp}})$$

We incorporate this new estimation of recompute time due to failures into our formula for wasted work $W$ as follows:

$$W(\Delta) = e^{-\frac{\Delta}{\text{MTTF}}} \cdot \frac{C}{C + \Delta} + \frac{1}{\text{MTTF}} \cdot \left( \frac{C \cdot \Delta}{C + \Delta} + \frac{\Delta^2}{2 \cdot (C + \Delta)} \right)$$  \hspace{1cm} (3.5)

Note that Equation 3.5 accounts for the probability of a failure happening during a checkpoint by considering the amount of lost computation that needs to be redone in such cases; however, it does not account for the checkpointing time spent by the system when writing partial checkpoints that get interrupted by failure events.

We account for the fraction of time spent writing partial checkpoints (due to failures) into the first component of wasted time, in the following equation:

$$W(\Delta) = e^{-\frac{\Delta}{\text{MTTF}}} \cdot \frac{\text{MTTF} + \Delta}{\text{MTTF}} \cdot \frac{C}{C + \Delta} + \frac{\Delta}{2 \cdot \text{MTTF}}$$  \hspace{1cm} (3.6)

Finally, Equation (3.7) combines all the refinements discussed above:

$$W(\Delta) = e^{-\frac{\Delta}{\text{MTTF}}} \cdot \frac{\text{MTTF} + \Delta}{\text{MTTF}} \cdot \frac{C}{C + \Delta} + \frac{1}{\text{MTTF}} \cdot \left( \frac{C \cdot \Delta}{C + \Delta} + \frac{\Delta^2}{2 \cdot (C + \Delta)} \right)$$  \hspace{1cm} (3.7)
### 3.3.3.2 Trace-driven evaluation and analysis of formulas

In this section, our goal is to evaluate the accuracy of the wasted work estimates produced by Equations (3.2–3.7) for real world production systems. More precisely, we use LANL’s failure traces to compare the fraction of wasted time that results from trace-driven simulations to the fraction of wasted time estimated by each equation.

Figure 3.8 studies the accuracy of the estimations from the equations for all LANL systems in our data under different checkpointing cost $C$ assumptions. The box-plots show the percentage of error in the estimation of wasted work by the equations, with respect to the wasted work that results under Young’s interval when using trace-based simulations.

![Figure 3.8: Errors in estimation of wasted work using equations with respect to the wasted work under Young when using trace-based simulations, for different checkpointing cost assumptions. (Note that a positive sign indicates overestimation while a negative sign indicates underestimation.)](image)

We make several observations based on the graphs in Figure 3.8. First, we find that
Figure 3.9: Errors in estimation of time spent writing checkpoints using equations plotted against each LANL system’s best Weibull shape fit

the estimates produced by the different equations are both comparable and quite accurate for smaller checkpoint costs. For example, the average magnitude of error stays within 3% for all equations under \( C = 20 \) seconds. As the checkpoint cost increases, the error in the estimates overall increases, as well as the variation between the results from different equations. For example, for a large checkpoint cost of 60 minutes, Equations (3.2, 3.7) deviate on average from the simulation results by as far as 15-17%, while Equations (3.4, 3.5) maintain an average error magnitude of only 4-5%.

In fact, we find that Equations (3.4, 3.5) consistently produce the lowest magnitude of error for all LANL systems and under different checkpointing cost \( C \) assumptions (note that Equation (3.4) tends to underestimate wasted time, hence the negative error sign). On the other hand, the highest errors are reported when using the basic equation we started with, Equation (3.2), followed by the last equation we derived which incorporates all of our discussed refinements, Equation (3.7).

These observations motivated us to take a closer look into the way these equations estimate the two components of wasted time, i.e. checkpointing time and recompute time due to failures, separately. This will help us better identify the sources of error in each equation and also examine why some equations tend to underestimate wasted time while others overestimate it. Furthermore, to study how the nature of the underlying failure process in a system affects the quality of the estimates produced by the different equations, we take into account in our analysis the value of the Weibull shape parameter for the best fit of failure data in each LANL system.

Figures 3.9 and 3.10 plot the percentage of error in estimating checkpointing time and the percentage of error in estimating recompute time, respectively, as a function of the Weibull shape value in LANL’s systems and assuming a checkpointing cost \( C \) of 10 minutes. Note that multiple equations can share the same method to estimate either component of wasted time (but each equation has a unique combination of methods to estimate both
Equations (2,3,4,6)

Equations (5,7)

Figure 3.10: Errors in estimation of time spent recovering lost work using equations plotted against each LANL system’s best Weibull shape fit.

components). Each graph in the figures corresponds to one estimation method, identified by the numbers of the equations that use this approach.

**Results.** The first observation we make based on the graphs in Figure 3.9 is that the significant overestimation of checkpointing time is the main source of Equation (3.2)’s relatively high inaccuracy. Further refinements to checkpoint time estimation considered in subsequent methods reduced the average magnitude of error by factors of 2-2.5X.

We find that the methods used to estimate checkpoint time in Equation (3.3), Equations (3.4,3.5) and Equations (3.6,3.7) produce comparable magnitudes of error, with average values of 6.1%, 6% and 5.2%, respectively. The method in Equations (3.4,3.5), however, consistently underestimates checkpointing time for all the systems in our data. It is helpful to note these differences in error signs for practical considerations; for example, if one needs to set an upper bound on I/O time in a HPC machine, a conservative overestimation of checkpointing time could be more useful to utilize than an underestimation.

For estimating recompute time due to failures, we observe from Figure 3.10 that the refinements used in Equations (3.5,3.7) do not introduce improvements over the basic approach used in Equations (3.2,3.3,3.4,3.6). When examining the reasons behind this result, we find that the method in Equations (3.5,3.7) tends to overestimate the probability of a failure happening during a checkpoint for the LANL systems in our data. This observation also explains why Equation (3.7) produces relatively high errors, since both methods it combines tend to overestimate the two quantities of wasted time, raising the total amount of error significantly. (Note that other equations that produce comparable error magnitudes to Equation (3.7) but with opposite signs for the two components of wasted time will yield smaller total error.)

Finally, we observe from the graphs in Figures 3.9 and 3.10 that under all equations, the accuracy in estimating both quantities of wasted time is correlated with the Weibull shape fit in a system. We observe positive correlations between the shape value and the magnitude...
of error in estimating checkpointing time; for example, we find that Pearson’s correlation coefficient $R$ between these two variables is 0.76 ($p$-value=0.01) and 0.65 ($p$-value=0.04) when using Equation (3.3) and Equations (3.4,3.5), respectively.

On the other hand, we observe strong negative correlations between the shape value and the magnitude of error in estimating recompute time: $R=-0.75$ ($p$-value=0.011) for Equations (3.2,3.3,3.4,3.6), and $R=-0.66$ ($p$-value=0.03) for Equations (3.5,3.7). The reason behind this is that a smaller shape parameter implies a heavier tail in the distribution function of failure inter-arrivals in a system, while the methods to estimate recompute time assume that, during regular computation, failures arrive on average half-way in a checkpoint interval, leading to higher estimation errors.

**Summary:** The fraction of wasted time in an HPC system, whether due to writing check-points or redoing lost work, can be approximated with simple back-of-the envelope formulas based only on the MTTF and checkpointing cost $C$. Additional knowledge of the properties of failure inter-arrivals in a system can be helpful in understanding the expected accuracy of different estimation techniques for wasted time.

### 3.4 Energy Awareness in Checkpoint Scheduling Policies

Our work so far has focused on evaluating (or estimating) the performance of checkpointing policies in practice. We now study the problem of HPC checkpoint scheduling from another angle: energy efficiency. Traditionally, the goal behind work on checkpoint scheduling has been to minimize an application’s runtime or completion time. However, runtime-optimized checkpointing policies are not necessarily optimal for energy savings. In this section, we revisit all the checkpointing policies discussed so far, both static intervals and adaptive techniques, from an energy perspective. We study how these policies can be optimized for energy savings and analyze the different energy/performance tradeoffs that result when they are deployed in practice.

#### 3.4.1 Static Checkpointing

We begin by studying how simple static checkpoint intervals can be optimized for energy savings, starting with Young’s traditional formula.

##### 3.4.1.1 Optimizing Young’s formula for energy

We show in Section 3.3.1 that Young’s traditional runtime-optimized formula (Equation (3.1)) provides near-optimal results in terms of completion time, by attempting to balance the two types of wasted work associated with checkpointing: the amount of time the application...
spends writing checkpoints ($T_{\text{checkpt}}$), and the amount of time that is lost after a failure and needs to be recomputed ($T_{\text{recomp}}$).

However, minimizing wasted time (and hence completion time) does not necessarily minimize the consumed energy. To see why, note that the power $P_{\text{checkpt}}$ consumed during checkpointing is lower than the power $P_{\text{comp}}$ consumed during computation, since the storage system of an HPC installation consumes significantly less power than computation [27].

More precisely, measurements in modern HPC installations suggest that the power consumption during a checkpoint operation compared to idle power ($P_{\text{idle}}$) is typically in the $1.05 - 1.15 \times P_{\text{idle}}$ range [30, 32, 67]. The power consumption during HPC computation, on the other hand, was found to be in the $2 - 4 \times P_{\text{idle}}$ range [47, 67], depending on the underlying hardware architecture. The most recent results reported from the SPECPower benchmark [8] show that the ratio between server power at average and high load levels and server idle power is in the 4 - 6X range. Furthermore, the difference between power consumed during computation versus idle time is expected to grow in future architectures.

Formally, our goal in this section is to minimize wasted energy $E_{\text{waste}}$, i.e. energy that is spent either on writing checkpoints ($E_{\text{checkpt}}$) or redoing work that was lost after a failure ($E_{\text{recomp}}$), and hence did not directly contribute to an application’s forward progress, for an application running on $N$ number of nodes:

$$E_{\text{waste}} = E_{\text{checkpt}} + E_{\text{recomp}} = (N \cdot P_{\text{checkpt}} \cdot T_{\text{checkpt}}) + (N \cdot P_{\text{comp}} \cdot T_{\text{recomp}})$$  \hfill (3.8)

Intuitively, due to the difference between $P_{\text{comp}}$ and $P_{\text{checkpt}}$, the checkpoint interval that minimizes energy consumption is shorter than Young’s, since checkpointing uses less energy than computation. We next derive a formula for an energy-optimized checkpoint interval for a given ($P_{\text{comp}}/P_{\text{checkpt}}$) ratio:

- Assuming an application spends $C$ time units writing a checkpoint roughly every $\Delta$ time units, we get $T_{\text{checkpt}} = C/\Delta$.
- Assuming failures are equally likely to happen anywhere in a checkpoint interval, the amount of work lost every time a failure happens (i.e. on average once every MTTF time units) is $\Delta/2$; then $T_{\text{recomp}} = \frac{\Delta}{2 \cdot \text{MTTF}}$.
- Accordingly, Equation (3.8) for wasted energy becomes:

$$E_{\text{waste}(\Delta)} = (N \cdot P_{\text{checkpt}} \cdot C/\Delta) + (N \cdot P_{\text{comp}} \cdot \Delta/(2 \cdot \text{MTTF}))$$

And the first order derivative:

$$E'_{\text{waste}(\Delta)} = (-N \cdot P_{\text{checkpt}} \cdot C/\Delta^2) + (N \cdot P_{\text{comp}}/(2 \cdot \text{MTTF}))$$

- Solving $E'_{\text{waste}(\Delta)} = 0$ to find the value of $\Delta$ that minimizes $E_{\text{waste}}$, we get the following
approximation for an energy-optimized checkpoint interval, $\Delta_{\text{Energy}}$:

$$
\Delta_{\text{Energy}} = \sqrt{2 \cdot C \cdot MTTF \cdot \left( \frac{P_{\text{checkpt}}}{P_{\text{comp}}} \right)}
$$

(3.9)

The remainder of this section will provide, among other things, a trace-based evaluation of the energy/performance tradeoff provided by Young’s formula and Equation (3.9), including the accuracy of Equation (3.9) in estimating the energy-optimized checkpoint interval.

### 3.4.1.2 Trace-driven study of energy/performance tradeoffs

In this section, we use trace-driven simulations based on the LANL failure data to answer the following three questions:

1. How do the energy/runtime tradeoffs vary as a function of the checkpoint interval, over a wide range of checkpoint intervals?

2. How does the runtime-optimized checkpoint interval fare in terms of energy consumption?

3. How close does our proposed solution in Equation (3.9) come to minimizing energy consumption?

We run trace-driven simulations for each LANL system to estimate the wasted energy for a wide range of checkpoint intervals, including Young’s traditional runtime-optimized interval (we refer to Young’s interval as $\Delta_{\text{Runtime}}$ in the remainder of this Section), and $\Delta_{\text{Energy}}$. We vary the ratio between compute power and checkpoint power ($P_{\text{comp}}/P_{\text{checkpt}}$) from 2X to 8X to represent a wide range of system configurations (the 8X ratio is representative of future hardware architectures), and we experiment with values for the checkpointing cost $C$ from as low as 20 seconds to as high as 60 minutes. (Note that we do not make assumptions about the absolute values for $P_{\text{comp}}$ and $P_{\text{checkpt}}$ as Equation (3.9) only needs the ratio between them).

Figure 3.11 shows the results of the trace-driven simulation for one representative LANL system (system with ID 20). (Results for other systems are similar and can be found in the tech-report [37]). Each graph corresponds to a different checkpoint cost $C$; the solid blue line in each graph corresponds to the wasted time (shown on the left Y-axis) and each of the other (green) lines corresponds to the wasted energy (shown on the right Y-axis) for a different $P_{\text{comp}}/P_{\text{checkpt}}$ ratio. The values for the wasted time (left y-axis) and wasted energy (right y-axis) are each normalized by the wasted time and wasted energy that result when using Young’s traditional runtime-optimized interval $\Delta_{\text{Runtime}}$, respectively. The values for $\Delta_{\text{Runtime}}$ and $\Delta_{\text{Energy}}$ are marked with a star and a circle, respectively, on the X-axis.
The first observation we make from Figure 3.11 is that there is a relatively wide range of checkpoint intervals that leads to near-optimal amounts of wasted time, while the amount of energy consumed for checkpoint intervals in this range can vary significantly. For example, when assuming a checkpoint cost \( C \) of 10 minutes, we find that the fraction of wasted time under any checkpoint interval in the range [90, 190] minutes is within 5% only of the optimal wasted time. Energy consumption, however, for the same range of intervals varies from a 5–16% decrease in energy waste, to a 16–25% increase in energy waste, compared to the energy wasted under \( \Delta_{\text{Runtime}} \) (the ranges reflect different \( P_{\text{comp}}/P_{\text{checkpt}} \) ratios).

The second observation is that while Young’s interval (\( \Delta_{\text{Runtime}} \)), marked with a star on the X-axis, does produce near optimal results from a performance perspective, the same interval can be quite far from optimal in terms of energy consumptions, in particular for larger \( P_{\text{comp}}/P_{\text{checkpt}} \) ratios. For example, for a checkpoint cost \( C \) of 10 minutes, \( \Delta_{\text{Runtime}} \) wastes 15% and 25% more energy than the optimal energy waste achieved when assuming \( P_{\text{comp}}/P_{\text{checkpt}} \) is 4X and 8X, respectively.

The third observation is that \( \Delta_{\text{Energy}} \) does introduce energy savings (i.e. the decrease in wasted energy in the system w.r.t. the wasted energy under \( \Delta_{\text{Runtime}} \)), in the following ranges:

- 5–7% for \( P_{\text{comp}}/P_{\text{checkpt}}=2X \) (under different \( C \) values);
- 10–12% for \( P_{\text{comp}}/P_{\text{checkpt}}=3X \);
- 15–19% for \( P_{\text{comp}}/P_{\text{checkpt}}=4X \);
- 33–34% for the extreme case of \( P_{\text{comp}}/P_{\text{checkpt}}=8X \).

While the energy savings that \( \Delta_{\text{Energy}} \) provides do not come for free in terms of runtime overheads, the price one pays is on-par with the gains one reaps. We observe runtime overheads (i.e. the increase in wasted time in the system w.r.t. the wasted time under \( \Delta_{\text{Runtime}} \)), in the following ranges:

- 6–8% for \( P_{\text{comp}}/P_{\text{checkpt}}=2X \) (under different \( C \) values);
- 10–15% for \( P_{\text{comp}}/P_{\text{checkpt}}=3X \);
- 25–30% for \( P_{\text{comp}}/P_{\text{checkpt}}=4X \).

The figure illustrates the trade-offs between runtime and energy consumption for different checkpoint intervals and power-ratios.
• 16–24\% for $P_{\text{comp}}/P_{\text{checkpt}}=4X$;
• 32–56\% for $P_{\text{comp}}/P_{\text{checkpt}}=8X$.

It is important to recall that the above are increases in wasted time, and not total completion time. E.g. if for a given system and application a total of 10\% of the time is wasted under $\Delta_{\text{Runtime}}$, then even in the extreme case of a $P_{\text{comp}}/P_{\text{checkpt}} = 8X$ not more than 13.2 – 15.6\% of the time is wasted under $\Delta_{\text{Energy}}$. (Note that we use the terms energy savings and runtime overheads in the rest of this work to describe the decrease in wasted energy and the increase in wasted time, w.r.t. the wasted energy/time under Young’s $\Delta_{\text{Runtime}}$, respectively).

**Summary:** Methods that schedule checkpoints to optimize application runtime are not optimal for energy purposes, and optimizing the checkpoint interval for energy introduces a tradeoff between energy savings and runtime overheads (due to the added checkpointing time). We find that the percentage of increase in wasted time is either comparable or modestly higher than the percentage of decrease in wasted energy, for the majority of the $(C, P_{\text{comp}}/P_{\text{checkpt}})$ configurations that we experimented with.

**3.4.1.3 Optimizing for energy with a bound on runtime**

After observing the performance cost of optimizing the checkpoint interval for energy savings, one might ask whether we can optimize checkpoint scheduling for energy purposes by minimizing the checkpoint interval, while enforcing a bound on the maximum allowed performance degradation (i.e. runtime overhead).

More precisely, for a given threshold $t$ on the increase in the expected fraction of wasted time $W$ in a system, compared to the wasted time that would have resulted under Young’s $\Delta_{\text{Runtime}}$, we want to find the smallest checkpoint interval $\Delta_{\text{bound}}$ that satisfies the following inequality:
\[
W(\Delta_{\text{bound}}) < t \cdot W(\Delta_{\text{Runtime}})
\]  
(3.10)

Solving this problem requires an estimate of the wasted time as a function of the size of the checkpoint interval. We therefore turn to the equations that we derived in Section 3.3.3 for estimating wasted time in a system, and use Equation (3.2), which produces a conservative estimate of wasted time. Since we assume that $C$ and MTTF are known for a given system, and $\Delta_{\text{Runtime}}$ can be computed using Equation (3.1), it is straightforward to determine $\Delta_{\text{bound}}$ using Inequality (3.10) and Equation (3.2).

To evaluate the quality of this approach, we determine the value for $\Delta_{\text{bound}}$ for several $t$ values and then use trace-driven simulations, based on the same data and testbed used in previous sections, to obtain the amount of wasted time and wasted energy resulting from $\Delta_{\text{bound}}$.

Table 3.3 presents the results for runtime threshold values of 3\%, 5\% and 10\%, under a checkpoint cost $C$ of 10 minutes and a $P_{\text{comp}}/P_{\text{checkpt}}$ ratio of 3X. The two right most
columns show the runtime overhead and energy savings for $\Delta_{bound}$ (w.r.t. the wasted time and wasted energy from $\Delta_{runtime}$), respectively.

The first thing we observe from Table 3.3 is that enforcing a bound on runtime did not significantly affect the quality of the energy savings achieved. In fact, under this configuration, a threshold on the runtime overhead as small as 3% was sufficient to introduce energy gains in the 7–11% range. As the threshold goes up to 10%, the corresponding energy savings either exhibit a modest increase (in the 0.3–4% range), or do not exhibit an increase at all (see systems 9, 10, 12). This is explained by the trends in the energy lines in Figure 3.11: as the runtime threshold increases, the checkpoint interval decreases until the added overhead due to checkpointing time raises the overall energy consumption of the system significantly.

We also observe from Table 3.3 that using Equation 3.2 to put a bound on runtime worked quite accurately in 23 scenarios out of the 30 scenarios we experimented with. For the 7 cases where the actual runtime overhead exceeded the estimated overhead, we observe that 5 of these cases belong to two LANL systems (systems 9 and 10), which warranted a closer look into their failure behaviour. Going back to Table 3.1, we find that systems 9 and 10 have the lowest values for the Weibull shape parameter among all systems. As noted in
Section 3.3.3, a smaller shape parameter is associated with a higher estimation error, as the equation assumes that failure inter-arrivals are exponentially distributed.

**Summary:** Optimizing the checkpoint interval for energy with a bound on runtime can be done using simple formulas that estimate wasted time in a system. We find that relatively small margins of increase in runtime (due to increased checkpointing frequency) can lead to high energy savings.

### 3.4.2 Adaptive Checkpointing

#### 3.4.2.1 Adaptation for energy

Section 3.4.1 demonstrates the potential for energy savings by carefully choosing the checkpoint interval. The results motivate us to investigate more advanced methods that adapt the checkpoint interval dynamically. More precisely, we consider the adaptive methods that we propose in Section 3.3.2: the moving averages (SMA, WMA, EMA), decreasing hazard rates (Hazard), and autocorrelation (AR). While our focus in Section 3.3.2 is to optimize and evaluate these methods for application runtime savings, we now study their effect on energy consumption and how they can be adapted for minimizing energy waste in HPC systems.

The three classes of methods that we consider all rely on the same principle: they exploit different characteristics of real world HPC failure processes to dynamically obtain (and continuously update) an accurate MTTF estimate and then plug this estimate into Young’s formula (recall Equation (3.1)) to determine the length of the next checkpoint interval. As such, each of these methods can easily be adapted for optimizing for energy usage, rather than application runtime, by simply applying the MTTF estimates they provide to Equation (3.9), rather than Equation (3.1).

#### 3.4.2.2 Trace-driven study of energy/performance tradeoffs

In this section, we use trace-driven simulations to examine the energy/performance tradeoffs of the adaptive checkpointing methods summarized above, both when optimized for runtime or for energy consumption.

Table 3.4 shows the results achieved under the runtime-optimized and energy-optimized versions of the adaptive methods when running trace-based simulations for the 10 LANL systems in our data, assuming $P_{comp}/P_{checkpt}$ is 3X and the checkpoint cost $C$ is 5 minutes. Note that for the sake of completeness we include in our comparative analysis Young’s static interval as well as the higher order approximation proposed by Daly [28].

1) **Runtime-optimized methods.** The three columns under ‘Runtime-optimized methods’ in Table 3.4 describe the runtime overhead, the energy overhead, and the fraction of time spent writing checkpoints when the different methods are optimized for runtime. We observe the following:
### Table 3.4: Comparison of the performance/energy tradeoffs introduced by different checkpointing methods under $(P_{\text{comp}}/P_{\text{checkpt}})=3X$, $C=5$ minutes. *(Note: a negative sign in the time/energy overhead columns indicates savings)*

<table>
<thead>
<tr>
<th>LAND System ID</th>
<th>Method</th>
<th>Runtime-optimized Methods</th>
<th>Energy-optimized Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>runtime overhead%</td>
<td>energy overhead%</td>
<td>I/O fraction%</td>
</tr>
<tr>
<td></td>
<td>(w.r.t. Young-runtime)</td>
<td>(w.r.t. Young-runtime)</td>
<td>(of total time)</td>
</tr>
<tr>
<td>3</td>
<td>-14.075</td>
<td>-17.211</td>
<td>4.981</td>
</tr>
<tr>
<td>9</td>
<td>-17.968</td>
<td>-10.983</td>
<td>2.434</td>
</tr>
<tr>
<td>12</td>
<td>-10.750</td>
<td>-10.671</td>
<td>8.480</td>
</tr>
</tbody>
</table>

*Note: negative sign indicates savings.*
• Despite being optimized for runtime, all the adaptive methods, with the exception of Hazard, introduce both runtime and energy savings (i.e. decrease in wasted time and wasted energy) for the 10 LANL systems in our data, compared to the static methods.

• The average savings in wasted time and wasted energy when using the moving averages (SMA, WMA, EMA) over Young’s interval are 5% and 7%, respectively. The best performing MA method overall is EMA, which results in runtime savings up to 8–13% and energy savings up to 10–18% (see systems 9, 10, 11, 12).

• We find that AR method produces average improvements of 6.5% and 7.6% in runtime and energy wastes, respectively. In some systems, exploiting failure autocorrelation resulted in significantly high runtime savings (9–17%) and energy savings (10–17%); see systems 3, 10, 11, 21.

• For the Hazard method, we find that the runtime improvements are on average quite marginal (1%), and no energy savings are introduced by the runtime-optimized version of this method for any LANL system in our data (energy overhead is 10.8% on average).

2) Energy-optimized methods. We repeat our analysis after optimizing all the methods for energy savings, as described in Section 3.4.2, and observe the following (see the columns under ‘Energy-optimized methods’ in Table 3.4):

• The energy-optimized methods do produce higher energy savings than the runtime-optimized ones with varying degrees of runtime overheads. Again, the adaptive methods outperform the static ones.

• The moving windows (SMA, WMA, EMA) now result in an average of 15.4% energy savings, but with an average runtime overhead of 11%. We find that EMA offers the best energy/runtime tradeoff among all MA methods.

• The average energy/runtime tradeoff for AR is comparable to the tradeoff for EMA: we find that AR results in an average of 17% energy savings, 7.5% runtime overhead. In two LANL systems, however, AR performs significantly better where it results in 23% and 26% energy savings while also introducing 1.75% and 4.7% improvements in runtime waste over Young (see systems 11 and 21).

• For Hazard, we observe energy savings of 12% on average across all systems, which is lower than MAs and AR, but the performance results are better than for other methods (on average 1% savings in wasted time).
3) Results for different power ratios. The results in Table 3.4 were obtained under the assumption that \( P_{\text{comp}} \) is 3X higher than \( P_{\text{checkpt}} \). We now repeat this analysis for other power ratios. Figure 3.12 plots for each checkpointing method the wasted runtime (Y-axis) versus the wasted energy (X-axis) that result under this method when it is optimized for runtime (hollow markers), and when it is optimized for energy (filled markers), in one of the LANL systems in our data, system 20. Each graph corresponds to a different \( P_{\text{comp}}/P_{\text{checkpt}} \) ratio, from 2X (far left) to 8X (far right). The dashed black lines in the graphs show the energy/runtime results for a range of static checkpoint intervals. (Results for the other LANL systems can be found in the tech-report [37]).

The graphs in Figure 3.12 show that the energy-optimized adaptive methods consistently result in higher quality energy/runtime tradeoffs than static checkpoints, under different power ratios (i.e. higher energy savings with lower runtime costs). We also observe that the MA methods and AR exhibit similar trends for different power ratios, while Hazard consistently reports modestly lower energy savings (than MA and AR), but with a much smaller runtime overhead.

Summary: Using adaptive techniques to schedule checkpoints dynamically introduces runtime and energy improvements over the static methods, both when optimized for runtime or for energy savings. The highest energy savings are introduced by the energy-optimized MA and AR methods, while energy-optimized Hazard saves less energy but with a significantly lower runtime cost, if any.
3.4.2.3 Practical Considerations

The high quality energy/performance results obtained under the adaptive checkpointing techniques motivated us to explore how easily they can be adopted in practice. We next examine different implementation issues for all the checkpoint scheduling policies included in our study so far.

1) ‘Hazard’ in practice: Our analysis shows that the energy-optimized Hazard results in a good balance between energy savings and runtime overhead (if any). But how feasible is implementing Hazard in practice? Recall that Hazard assumes a-priori knowledge of the system’s hazard rate function (i.e. knowledge of how the expected time to fail, $ETTF$, changes as a function of the time since the last failure, $TSLF$). We explore two ways to implement Hazard in practice, whenever accurate knowledge of the system’s hazard function is not available:

1.a) Calculating the hazard function dynamically: We propose calculating the values for the $(ETTF|TSLF)$ table dynamically using past failure observations. Practically, this method requires keeping a log of the system’s failure history over time and updating the $(ETTF|TSLF)$ table after a failure event. The updated table would then be used to estimate the ETTF every time a checkpoint interval is computed (i.e. after a failure or a checkpoint). We refer to this approach as Hazard(dynamic).

1.b) Approximating the hazard function using the Weibull shape parameter: This approach estimates the hazard function using the shape parameter for the Weibull fit of failures in a system. We assume knowledge of the shape parameter and use it to compute the hazard function (i.e. the $(ETTF|TSLF)$ table) prior to the application run, instead of relying on actual failure traces. We call this approach Hazard(shape).

Figure 3.13 evaluates the quality of these two approaches using trace-driven simulations for the four largest LANL systems in our data in terms of CPU-days (systems 2, 18, 19, 20), by comparing the energy/runtime results of Hazard(dynamic) and Hazard(shape) to the results obtained under the original Hazard method. The bars for Hazard(shape) are the mean value of 20 runs; the error bars represent 95% confidence levels.

We find that the two approximation methods result in generally comparable energy savings to Hazard, with the exception of system 18 where Hazard(dynamic) introduces significantly lower energy savings. The runtime overhead for Hazard(dynamic) was close to the original Hazard in 3 of the 4 systems, while the runtime overhead for Hazard(shape) was consistently higher.

2) MAs and AR in practice: The only requirement of the MA methods is to maintain a log of the most recent times of failures (for SMA, WMA), or a complete record of the times of failures (for EMA), which is easy to implement in practice. Implementing AR is more involved
Figure 3.13: Evaluation of different approaches to estimating energy-optimized $Hazard$ in practice for four LANL systems, under $(P_{comp}/P_{ckpt})=3X$.

as it requires constructing an accurate autoregression model of the failure inter-arrivals in the system.

3) Young and Daly in practice: The simplicity of the static methods Young and Daly make them appear attractive for implementation purposes. However, these formulas rely on the a-priori knowledge of the system’s real MTTF (unlike the MA methods which compute the MTTF online), which makes them more challenging to apply in practice.

Summary: From a practical point of view, the MA methods are the easiest to implement of all the methods we consider in our study. This observation, coupled with the high quality energy/runtime tradeoffs introduced by the MA methods, particularly EMA, makes them strong practical candidates for energy-optimized checkpointing. To achieve lower runtime overheads, however, we propose different techniques to implement the method $Hazard$ in practice.

3.5 Implications for the I/O Subsystem

Energy-optimized checkpoint scheduling increases the load on the I/O subsystem, since the system writes more frequent checkpoints to avoid the higher power-cost of redoing lost work (in case of a failure). In this section, we study the impact of energy-optimized checkpointing on the I/O subsystem, and investigate how to minimize this impact.

3.5.1 Static Checkpointing

We begin with a comparison of the I/O cost under static checkpointing when using the energy-optimized interval ($\Delta_{Energy}$) versus the runtime-optimized interval ($\Delta_{Runtime}$). The column “I/O fraction%” in Table 3.4 shows the fraction of time the system spends writing checkpoints, when assuming a checkpoint cost of $C=5$ minutes and $P_{comp}/P_{ckpt}=3X$. We observe that, not surprisingly, the fraction of time spent on I/O under $\Delta_{Energy}$ is higher
Figure 3.14: Breakdown of wasted work across a wide range of static checkpoint intervals for two LANL systems (Y-axis is logscale).

than under $\Delta_{\text{Runtime}}$, however it does not significantly exceed 10% in any of the cases. 10% is a typical target load that HPC storage systems are designed for [68].

For a more general understanding of how I/O time contributes to wasted time as a function of the checkpoint interval, Figure 3.14 plots the breakdown of wasted time into I/O time and lost computation across a wide range of checkpoint intervals. Results are shown for LANL systems 2 and 9, representatives of systems with higher and lower Weibull shape parameter, respectively. Each graph contains vertical lines marking the values for $\Delta_{\text{Energy}}$, when assuming different power ratios (green dashed lines), and under $\Delta_{\text{Runtime}}$ (thick red line).

We make two observations. First, even under the aggressive assumption of $P_{\text{comp}}/P_{\text{ckpt}} = 8X$ the fraction of time spent on I/O remains within 7% and 13% in systems 9 and 2, respectively. (Note the log scale of the Y-axis).

Second, the graphs in Figure 3.14 show significant potential for using the checkpoint interval as a tuning knob to control the I/O load of a system: We observe that I/O time can be reduced significantly with only moderate runtime overhead, just by increasing the checkpoint interval. For example, within a 10% increase in the overall wasted time (compared to the wasted time under Young’s $\Delta_{\text{Runtime}}$) the fraction of time spent on I/O can be reduced by nearly 50% in systems 2 and 9. Naturally, these savings come at the expense of increased energy cost: a 31% and 28% increase in energy waste in systems 2 and 9, respectively (under $P_{\text{comp}}/P_{\text{ckpt}}=3X$). However, in systems that are I/O bottlenecked this might be a cost worth paying.
3.5.2 Adaptive Methods

In this section, we study the I/O cost for the various adaptive checkpointing methods that we discussed in Section 3.4.2. Looking back at the column labeled “I/O fraction%” in Table 3.4, where we reported the fraction of time spent writing checkpoints under each method, we make several observations.

We find that the adaptive method Hazard consistently introduces the lowest fraction of I/O time across all methods, both when optimized for runtime or for energy. The reason is that this method reduces the checkpointing frequency whenever the expected time to fail is high. In fact, we find that the fraction of time spent on I/O under the energy-optimized Hazard remains below 10% in each system in our data.

The MA methods on the other hand, when optimized for energy, consume I/O time that is comparable to the I/O time under the energy-optimized static methods (MAs however introduce higher energy savings than the static methods or Hazard). The method AR mostly consumes less I/O time than the MA methods but more than Hazard. From an I/O perspective, the method Hazard is therefore the best candidate for HPC systems that are I/O bottlenecked.

3.5.3 Optimizing for energy with a bound on I/O

This section explores checkpoint placement for energy savings that places a bound on I/O time. We approximate the fraction of time a system spends checkpointing as \( C/(\Delta + C) \), since it spends on average roughly every \( \Delta + C \) time units, \( C \) time units to checkpoint. Hence to stay within a certain bound \( b_{IO} \) on the fraction of time spent checkpointing, we need to ensure that \( C/(\Delta + C) < b_{IO} \). We therefore choose the checkpoint interval \( \Delta_{EnergyIO} \) that optimizes energy within a bound on the I/O cost as follows:

\[
\Delta_{EnergyIO} = \max\{\Delta_{Energy}, C/b_{IO} - C\}
\]  
(3.11)

Table 3.5 shows the simulation results from using Equation (3.11) for all the LANL systems in our data. We chose I/O thresholds that are moderately lower than the fraction of time spent on I/O under \( \Delta_{Energy} \), which translated to a threshold of 10% in systems 2, 18, 19, 20, and 5% in systems 3, 9, 10, 11, 12, 21. The table summarizes the results for runtime/energy/I/O under \( \Delta_{Energy} \) and under the interval that optimizes energy with a bound on the estimated I/O time (\( \Delta_{EnergyIO} \)).

We find that the I/O times under the new interval \( \Delta_{EnergyIO} \) are within the desired I/O bounds. We also observe that the overall runtime overhead for \( \Delta_{EnergyIO} \) is significantly lower than the runtime overhead under \( \Delta_{Energy} \) (due to the reduced checkpointing time), but interestingly, the energy savings under \( \Delta_{EnergyIO} \) drop only modestly (with the exception of systems 18 and 19). This agrees with our findings in Section 3.4.1.3 where we
### Chapter 3. Exploiting Failure Analysis in HPC Checkpoint-Scheduling

#### Table 3.5: Optimizing the checkpoint interval for energy with a bound on I/O for all LANL systems (simulation results under $C=10$ minutes, $P_{\text{comp}}/P_{\text{ckpt}}=3X$).

<table>
<thead>
<tr>
<th>ID</th>
<th>I/O System</th>
<th>I/O %</th>
<th>Length (min)</th>
<th>$\Delta_{\text{Energy}}$ (w.r.t. Young)</th>
<th>$%$ time overhead (w.r.t. Young)</th>
<th>Energy saving (of total time)</th>
<th>I/O time % (w.r.t. Young)</th>
<th>$\Delta_{\text{Energy IO}}$ (w.r.t. Young)</th>
<th>$%$ time overhead (w.r.t. Young)</th>
<th>Energy saving (of total time)</th>
<th>I/O time % (w.r.t. Young)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>10</td>
<td>54.71</td>
<td>10.601</td>
<td>-12.343</td>
<td>14.533</td>
<td>90</td>
<td>0.113</td>
<td>-1.867</td>
<td>9.051</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>10</td>
<td>55.997</td>
<td>9.571</td>
<td>-13.735</td>
<td>14.276</td>
<td>90</td>
<td>0.745</td>
<td>-1.981</td>
<td>9.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>77.278</td>
<td>12.397</td>
<td>-11.813</td>
<td>11.433</td>
<td>90</td>
<td>5.188</td>
<td>-9.915</td>
<td>9.498</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>159.682</td>
<td>16.703</td>
<td>-8.34</td>
<td>5.777</td>
<td>190</td>
<td>8.263</td>
<td>-8.507</td>
<td>4.884</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>155.324</td>
<td>13.675</td>
<td>-12.862</td>
<td>5.936</td>
<td>190</td>
<td>8.41</td>
<td>-6.421</td>
<td>4.872</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>97.897</td>
<td>8.594</td>
<td>-15.73</td>
<td>8.939</td>
<td>190</td>
<td>-3.663</td>
<td>-1.015</td>
<td>4.675</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed how moderate runtime overheads (w.r.t. Young’s runtime interval) can lead to high energy savings.

**Summary:** The I/O pressure in checkpointing systems can be effectively reduced either by using adaptive methods, such as *Hazard*, which we show to have inherently lower I/O requirements, or by using a simple method we propose to determine a (static) checkpoint interval that optimizes energy savings, while enforcing a bound on I/O time.

### 3.6 Conclusion and Future Directions

Checkpoint scheduling policies in HPC platforms have been traditionally designed to optimize application runtime. However, with energy efficiency becoming a key driver in the design of future Exascale architectures, optimizing the checkpointing process needs to be revisited and analyzed from an energy point of view. Additionally, different practical considerations need to be taken into account when proposing HPC checkpointing solutions in the literature, whether they are designed to optimize system performance or energy consumption.

Our goal in this part of our work is to provide a comprehensive evaluation of the tradeoffs introduced by a wide array of checkpointing policies in HPC platforms, from multiple angles: performance, energy consumption and practicality. We provide insights into the energy overhead, as well as the performance impact, associated with different checkpointing policies using traces from 10 real world HPC clusters. We study policies that vary in their design and complexity, from basic, static checkpoint intervals to more advanced techniques that exploit HPC failure properties to adapt the checkpoint interval dynamically.

We show that optimizing checkpoints for energy savings, with or without a bound on application runtime, can be done through simple formulas that we propose and evaluate using real world traces. Interestingly, our analysis shows that relatively small margins of
increase in runtime (due to increased checkpointing operations) can produce high energy savings. In our comparative analysis of the different checkpointing policies, we find that the energy-optimized adaptive policies result in higher quality energy/runtime tradeoffs than the static (constant) policies. We explore different practical considerations for these adaptive techniques and identify candidate methods that are easy to implement in practice (while producing high energy savings and relatively low runtime overheads), such as exponential moving averages.

We also assess the impact of the energy optimized checkpointing on the I/O subsystem in HPC machines and identify opportunities for I/O bottlenecked systems to achieve high I/O savings, either through adaptive methods that have low I/O requirements (e.g. hazard-rate based methods), or by using a simple, effective formula that we propose to optimize static checkpoints for energy with a bound on the I/O load.

**Checkpoint/Restart for future HPC platforms:** Coordinated checkpointing to stable storage is the most prevalent fault-tolerance technique used in modern HPC systems. Traditionally, coordinated checkpointing is designed such that all processes in a distributed application are synchronized to write a global checkpoint of the application’s critical data to remote storage, typically a parallel file system (PFS). However, as HPC systems grow in size, the performance and energy overheads associated with checkpointing to a PFS can become prohibitively expensive, even when using adaptive policies that have relatively lower energy/performance overheads compared to constant checkpoints.

It is reported that the fastest supercomputers from the Top500 machines list [9] take an average of 20 minutes to write a single checkpoint [23,39], and that cost is not expected to reduce significantly in future platforms due to the stability of the I/O subsystem organization over time [39]. In fact, some recent studies predict that an exascale machine will be spending approximately 50% of its time either writing checkpoints or recovering from failures [12,72]. Therefore, for checkpoint/restart to remain viable in future HPC generations, drastic improvements need to take place in the checkpointing process. Our results in this work show that adaptive checkpoint scheduling techniques that exploit HPC failure characteristics online can produce high improvements in the performance and energy overheads for current HPC systems. Our results also indicate that further optimizations to the checkpoint scheduling problem are not likely to introduce additional significant gains for future HPC generations performing coordinated checkpoints.

One factor that can make up for the increasing component count in future large-scale systems is the checkpoint cost, i.e. the time needed to perform a checkpoint. Investing in lowering the cost of writing checkpoints will therefore be increasingly critical for the efficient use of future HPC generations. Various techniques have been proposed for reducing the
checkpoint size such as checkpoint compression [50], data aggregation [52], or incremental checkpointing [70, 101]. Other studies proposed hybrid, multi-level checkpointing schemes where several technologies are utilized at different layers of the system architecture. Checkpoint files are stored at multiple ‘levels’ that offer different reliability/performance tradeoffs: the first level is typically a node-local fast storage medium (e.g. local flash drives), whereas the last level is usually the remote parallel file system. Intermediate layers can include storing checkpoints in neighboring nodes. Examples of such multi-level checkpointing libraries are SCR [69] and FTI [16]. The popularity of these libraries has increased a lot recently and proposals to standardize their interface have been discussed in the HPC community [24]. Additionally, some researchers are exploring how future nonvolatile memory technologies such as PCM (Phase Change Memory) can be exploited in multi-level checkpointing [31, 58].
Chapter 4

Conclusion

Our work in this thesis demonstrates how the analysis of field data collected from real world large-scale systems can be immensely helpful for understanding different aspects of their behaviour in the wild. Our focus in this thesis is on two growing concerns associated with the design and operation of high-performance computing (HPC) systems in particular, and large-scale data centers, in general: reliability and energy efficiency. As HPC architectures grow in size and complexity, capturing the effect of these factors and the interplay between them in practice is becoming increasingly challenging, and will be more challenging for future Exascale platforms that are expected to operate within tight power constraints and to experience failures in unprecedented rates.

In the first part of this thesis, we utilize various system logs and failure datasets collected at multiple organizations to develop an in-depth understanding of how and why failures happen in large-scale systems. We focus on understanding correlations between different types of failures, and provide insights into how different factors impact hardware and software reliability in the field. We consider both node reliability and application reliability, in order to develop a comprehensive understanding of the resilience of different layers within large computing platforms.

Our results on environmental factors in large data centers, such as temperature and power quality, provide critical insights into which system components are more likely to be affected by these factors, and what this effect looks like under real world scenarios. We find that the impact of temperature is smaller than what the theoretical models assume, which shows significant potential for achieving high energy savings, as a large fraction of a data center’s energy bill goes into cooling. Furthermore, our analysis of power failures identifies various correlations between different types of power related issues and failures of data center components. We quantify the strength and magnitude of these correlations, and discuss practical implications and learned lessons based on our observations.
In addition to studying node and system reliability, we also investigate job reliability in large-scale clusters. Our analysis of job failures using workload traces from multiple organizations identifies factors that correlate significantly with the completion of massive scale jobs, such as resource consumption patterns and job configuration parameters. We further show that certain parameters can be highly predictive of a job's final status in the cluster, such as the amount of requested resources and the reliability of a job's tasks.

In the second part of this thesis, we use the knowledge of system failure properties obtained from our failure analyses to optimize HPC fault-tolerance and recovery mechanisms. We focus on 'checkpoint/restart', the most widely used fault-tolerance technique in HPC systems. We design checkpoint-scheduling policies that exploit various characteristics of real world systems, with the goal of optimizing the checkpointing process either for performance or for energy savings.

We use trace-driven simulations to evaluate the quality of a wide array of checkpointing policies, including ones that we propose as well as existing policies in the literature, while investigating different practical implementation issues. We provide a comprehensive study of the different tradeoffs introduced by checkpoint-scheduling policies when they are optimized for energy or for performance goals. We show that adaptive techniques that exploit system failure properties dynamically provide higher quality tradeoffs between energy savings and runtime overheads, than do static checkpoint intervals.

The findings reported in this thesis indicate that using real world traces to conduct large-scale analysis studies is crucial for the efficient design and operation of large computing systems. This is especially critical due to the difficulty of replicating large-scale system problems inside a lab environment. Our results show that the effect of different factors on system reliability and energy consumption is better understood when looking at the actual data collected from different subsystems in these platforms, and that the behaviour of these systems in the wild can deviate from what the theoretical models predict. We also show that using real world logs can help us identify practical opportunities for performance improvements and energy savings, while understanding the corresponding tradeoffs that result when optimizing for these different factors in practice. We hope that these results will motivate the HPC community to collect and share more field data with researchers and/or to make them publicly available, to further pave the way towards reliable, energy-efficient Exascale computing.
Bibliography


[63] Yinglung Liang, Yanyong Zhang An, Sivasubramaniam Ramendra, K. Sahoo, Jose Moreira, and Manish Gupta. Filtering failure logs for a Blue Gene/L prototype. In


