A SYSTEMATIC AND AUTOMATED METHODOLOGY FOR CALIBRATING SIMULATORS OF PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

MAY 2026

By

Jesse McDonald

Dissertation Committee:

Henri Casanova, Chairperson
Edoardo Biagioni
Jason Leigh
Anthony Peruma
Yuriy Mileyko

Keywords: High Performance Computing, Simulation, Calibration, Workflow Scheduling

ACKNOWLEDGMENTS

Many people contributed to the existence of this dissertation both directly and indirectly. Without any one of these, I likely would never have made it this far.

First, I would like to thank my Chair, Advisor, and Mentor, Henri Casanova; who introduced me to the world of High Performance Computing, then took me on for his research; who insisted on always aiming high and targeting the most prestigious publication venues; and who took every chance to find extra opportunities for me.

I would also like to thank my committee members for their willingness to help me through this process.

Next I would like to thank my Family. My wonderful mother Lina; who never stopped supporting me or believing in me (even when I did), and helped foster my love of learning. My amazing father Rob who was along side her the entire time. And my older brother Bobby who was always there learning all the things I missed.

I would also like to thank all my undergrad professors. Specifically Samuel Seth Long who provided an opportunity for me to fund my bachelors, who pushed us along by never hand waving advanced topics as "too advanced for this class", and who showed me how amazing research is. Also Kacey Diemert who was the first professor to tell me I was good at math, and Heather Moon who insisted I should get a degree in math.

Lastly, I would like to thank all those who worked on this research with me. Maximilian Horzela who provided a wonderful opportunity to test our research in the field, Jeff Wong who stressed Simcal to its limits, and Frédéric Suter, Loic Pottier, and Rafael Ferreira Da Silva who were behind the scenes the whole time supporting the research in subtle, but important ways.

ABSTRACT

Simulation plays a central role in Parallel and Distributed Computing (PDC) research and development. Conducting experiments in simulation offers many potential advantages over real-world experiments, such as, increased reproducibility, flexibility, and efficiency. However, the validity of simulation-based studies hinges on the accuracy of the underlying simulator. Considerable work has focused on designing accurate and scalable simulation models. But the critical task of calibrating an entire simulator, so that simulated behaviors match real-world behaviors as much as possible, has not received as much attention. Current practices for simulator calibration are typically ad hoc, manual, and rarely documented, resulting in widespread use of simulators whose results may be unreliable. This dissertation proposes a general methodology and framework for automating the calibration of simulators of PDC systems. The key results demonstrate that automated calibration is feasible, substantially improves simulation accuracy compared to current practice, and enables new uses of simulation in the field. The proposed methodology is validated via case studies across multiple PDC domains. Beyond improving simulator accuracy, this work also leverages automated calibration to explore the questions of determining the appropriate level of sophistication for a simulator and of assessing the potential benefit of simulation-driven scheduling at runtime.

TABLE OF CONTENTS

Acknowledgments . . . iii
Abstract . . . iv
1 Introduction . . . **1**
1.1 Context and Motivation . . . 1
1.2 Objective and Contributions . . . 2
1.3 Implications of this Work . . . 3
1.4 Organization of this Dissertation . . . 3

I Simulation Calibration **5**
2 Simulation . . . **6**
2.1 Simulator Definition . . . 6
2.1.1 Gravity Simulator Running Example . . . 6
2.1.2 Interest Metric . . . 8
2.1.3 Simulator Sophistication . . . 8
2.2 Simulation of PDC Systems . . . 9
2.2.1 Motivation for Using Simulation . . . 9
2.2.2 State of the Art . . . 10
2.3 PDC Simulation in this Work . . . 12
2.3.1 PDC Simulation Technology . . . 13
2.3.2 Target PDC Simulation Use Cases . . . 14
2.4 Conclusion . . . 15
3 Simulation Calibration . . . **16**
3.1 The Simulation Calibration Problem . . . 16
3.1.1 Simulators that Only Appear Correctly Calibrated . . . 17
3.2 Ground-truth Data Selection . . . 18
3.3 Loss Function Selection . . . 20
3.4 Parameter Selection . . . 24
3.5 Calibration of Simulators of PDC Systems . . . 26
3.5.1 PDC-Specific Considerations . . . 26
3.5.2 Need for Calibration . . . 28
3.5.3 State of the Art . . . 30
3.6 Conclusion . . . 32
4 Simulation Calibration Methodology . . . **33**
4.1 Methodology . . . 33
4.1.1 Ground-truth Data . . . 33
4.1.2 Loss Function and Optimization Algorithm Selection . . . 35
4.1.3 Parameter Selection and Ranges . . . 37
4.2 Simcal . . . 39

4.2.1 Simulator Wrapper . . . 40
4.2.2 Calibration Wrapper . . . 40
4.2.3 Parallelism Wrapper . . . 45
4.2.4 Calibration Evaluation . . . 45

II Evaluation of Calibration Methodology 47
5 Experimental Case Study: Scientific Workflows . . . 48
5.1 Ground-truth Data . . . 48
5.2 Simulator Description and Parameters . . . 49
5.3 Instantiating the Automated Calibration . . . 51
5.3.1 Parameter Selection . . . 51
5.3.2 Parameter Ranges . . . 52
5.3.3 Loss Functions and Algorithms . . . 52
5.3.4 Calibration Time Budget . . . 53
5.4 Use of Ground-truth Data . . . 53
5.5 Conclusion . . . 57
6 Experimental Case Study: MPI Applications . . . 59
6.1 Ground-truth Data . . . 59
6.2 Simulator Description and Parameters . . . 60
6.3 Instantiating the Automated Calibration . . . 61
6.3.1 Parameter Selection . . . 62
6.3.2 Parameter Ranges . . . 62
6.3.3 Loss Functions and Algorithms . . . 63
6.3.4 Calibration Time Budget . . . 64
6.4 Use of Ground-truth Data . . . 64
6.5 Conclusion . . . 65
7 Experimental Case Study: High Energy Physics Application . . . 67
7.1 Ground-truth Data . . . 68
7.2 Simulator Description and Parameters . . . 69
7.3 Instantiating the Automated Calibration . . . 70
7.3.1 Parameter Selection . . . 70
7.3.2 Parameter Ranges . . . 70
7.3.3 Loss Functions and Algorithms . . . 71
7.3.4 Calibration Time Budget . . . 75
7.4 Use of Ground-truth Data . . . 76
7.5 Conclusion . . . 77

III Applications of Calibrated Simulators 78
8 Simulator Sophistication . . . 79
8.1 Case Study #1: Scientific Workflows . . . 79
8.2 Case Study #2: MPI Applications . . . 82

8.3 Case Study #3: High Energy Physics Application . . . 85
8.4 Conclusion . . . 88
9 Using Simulation for Scheduling at Runtime . . . 89
9.1 Introduction . . . 89
9.2 Related Work . . . 91
9.2.1 Online Performance Monitoring . . . 91
9.2.2 Offline Simulations . . . 92
9.2.3 Online Simulations . . . 93
9.3 Proposed Approach: SDPS . . . 96
9.3.1 Implementation Considerations . . . 96
9.3.2 When to apply SDPS . . . 98
9.3.3 Objective . . . 99
9.4 Experimental Methodology and Scenarios . . . 100
9.4.1 Experimental Methodology . . . 100
9.4.2 Platform Configurations . . . 102
9.4.3 Workflow Instances . . . 104
9.4.4 Algorithms . . . 105
9.5 Results . . . 107
9.5.1 Diversity of the Algorithms in the Portfolio . . . 107
9.5.2 Impact of Simulation Error . . . 109
9.5.3 Impact of Simulator Sophistication . . . 115
9.5.4 Adapting to Dynamic Platform Changes . . . 118
9.5.5 Simulation Overhead . . . 119
9.6 Conclusion . . . 120

IV Conclusion 122
10 . . . 123
10.1 Summary of Contributions . . . 123
10.2 Future Work . . . 124
10.2.1 Simulation Calibration Improvements . . . 125
10.2.2 Applications of automated simulation calibration . . . 126
11 Summary of Published Work . . . 128

Appendices 129
A Gravity Example Ground-Truth Data . . . 130
B Simulation-Driven Scheduling Applied to Dynamic Platforms . . . 131

Bibliography 135

LIST OF TABLES

5.1 Workflow specifications used for ground-truth executions logs. 49
5.2 Simulation models implemented in WorkflowSimulator. 50
5.3 Calibration error vs. algorithm and loss function. 53

6.1 Simulation models implemented in MPISimulator. 61
6.2 Calibration parameter ranges 62
6.3 Calibration error vs. algorithm and loss function. 63

7.1 Hardware Platform Configuration Specifications. 69
7.2 Calibration parameter ranges 71
7.3 Calibration error vs. algorithm and loss function. 75

8.1 Level-of-detail options for the simulators used in Case Study #1. 80
8.2 Level-of-detail options for the simulators used in Case Study #2. 83
8.3 Level-of-detail options for the simulators in case study #3. 86
8.4 DCSim Sophistication Error 87

9.1 Cluster configurations used for SDPS case study 102
9.2 Workflow instances used for SDPS case study 103
9.3 SDPS comparison across levels of sophistication 117
9.4 Simulated makespans vs simulation time 119

LIST OF FIGURES

2.1 Gravity example simulator 7
2.2 Simulation usage by domain 11

3.1 Projectile motion calibrations using insufficiently diverse ground-truth data . 19
3.2 Projectile motion curves using a too simple loss function 22
3.3 Projectile motion curve using a too complicated loss function 23
3.4 Low loss acceleration coefficients given fixed velocity and position coefficients 25
3.5 Examination calibration practices in research publications 30

4.1 Example architecture of Simcal 43

5.1 Loss value vs. time for all ground-truth data for the Epigenomics workflow. . 53
5.2 Calibration loss for different training datasets for the Seismology workflow . 55
5.3 Training dataset cost vs. loss 56

6.1 Loss value vs. time when using all ground-truth data for 128 compute nodes. 64

7.1 Execution platform 68
7.2 Loss value vs. time when using all training data 75

8.1 WorkflowSimulator Sophistication Error 81
8.2 MPISimulator Sophistication Error 84

9.1 Diversity of the scheduling portfolio used for the SDPS case study 107
9.2 *dfb* vs. experimental scenarios, for $e = 1.0$. 110
9.3 *dfb* vs. experimental scenarios, for $e = 0.3$. 111
9.4 *dfb* vs. simulation error (e) 111
9.5 Summary of SDPS performance with added noise 112
9.6 SDPS's *dfb* vs. experimental scenarios, for $e = 1.0$ and $e' = 0.3$. 114
9.7 SDPS's average *dfb* for all values of (e) and (e') 114
9.8 SDPS's average *dfb* for all values of (e) and (e') for workflow instance W_8 . 115
9.9 Comparison of experimental scenarios using SDPS with error mitigation . . 116

B.1 Dynamic platform makespan results for the W_1 workflow. 132
B.2 Dynamic platform makespan results for the W_4 workflow. 133

CHAPTER 1
INTRODUCTION

In this chapter, we first provide context and give the motivation for this work. We then state our main objectives, highlight our key research contributions, and discuss the implications of this work. Lastly, we outline the organization of this dissertation and summarize the focus of all subsequent chapters.

1.1 Context and Motivation

Parallel and Distributed Computing (PDC) research and development activities are often empirical in nature, and thus typically involve executing application workloads on hardware platforms. Simulating these executions, as an alternative to conducting real-world experiments, offers many potential advantages: it makes it possible to explore hypothetical scenarios, can yield 100% reproducible and observable results, and can require less time, labor, carbon footprint, and/or funding. As such, simulation is a popular approach for research and development across many PDC domains [20]. However, in order to be useful, a simulator should accurately model the system it is simulating. The challenge of designing simulation models that can be sufficiently accurate and scalable is well-recognized in the field of PDC, and many simulation approaches and simulation framework have been proposed that offer different solutions for tackling this challenge [25, 16, 12, 100, 18, 11, 13, 124, 61, 85, 71, 109]. Another challenge, which is often overlooked, is that for a potentially accurate simulator to be accurate in practice it needs to be *calibrated* [106, 92]. This challenge stems from the fact that simulator calibration is a difficult optimization problem.

Our examination of the state of the art in the field of PDC reveals that simulator calibration is frequently not performed or, if performed, not documented. In the rare cases in which authors document their calibration procedure, they typically describe a tedious, at least partially manual, process. Using poorly calibrated simulators can lead to low accuracy, thus rendering conclusions drawn from simulation results questionable at best. Consequently, the overall usefulness of simulation in the field is impaired, and the validity of some of the simulation results published in the PDC literature is unclear. This situation can be remedied if a sound approach for automatically calibrating simulators of PDC systems is developed. The main motivation for the research in this work is to develop such an approach.

1.2 Objective and Contributions

Our objective is to design a generic automated simulator calibration methodology, and a framework for using this methodology in practice, for improving the accuracy of simulators of PDC systems. The main hypothesis of this dissertation is:

> *Automated calibration of PDC simulators is feasible, can increases simulation accuracy significantly when compared to current practice, and can unlock new simulation-driven use cases in the field.*

To verify this hypothesis, we focus on four following broad research questions (RQs):

> **RQ1:** Can an automatic simulation calibration methodology be developed and used to increase the accuracy of PDC system simulators?
>
> **RQ2:** How much ground-truth data is required to perform sufficiently precise calibrations of PDC system simulators?
>
> **RQ3:** Does the ability to calibrate PDC system simulators make it possible to quantify the impact of the implemented level of simulation sophistication on simulator accuracy?
>
> **RQ4:** Can high-accuracy (likely due to being well-calibrated) PDC system simulators be used to inform application scheduling decisions made by runtime systems used to run scientific workflow applications.

Our answers to the above research questions form the contributions of this dissertation. Specifically:

- To answer **RQ1** we formally define the simulation calibration problem and propose a general methodology for solving it, tackling several conceptual and practical challenges in the process. We then apply this calibration methodology to three relevant case studies for PDC system simulators in three different domains: scientific workflows, MPI benchmarking, and high-energy physics platform provisioning, each with their own real-world ground-truth data. We demonstrate that our methodology vastly improves simulator accuracy when compared to calibration performed by a human expert.
- To answer **RQ2** we apply our automated calibration methodology to the same three case studies, but using various ground-truth datasets of various sizes. This allows us to quantify the impact of using reduced ground-truth data for calibration.

- To answer **RQ3** we apply our automated calibration methodology to the same three case studies, but using simulators implemented at various levels of sophistication. We demonstrate that our methodology makes it possible to draw conclusions regarding which level of sophistication is sufficient for different simulated components.

- To answer **RQ4** we investigate the use of simulation-driven scheduling in workflow management systems, by which such a system can simulate executions of itself at runtime to pick the most promising algorithm from a portfolio of candidate scheduling algorithms. We demonstrate that this scheduling approach is feasible in practice and that it can provide significant performance improvements when compared to the state-of-the-art scheduling approach in workflow management systems.

1.3 Implications of this Work

If the hypothesis in the previous section is confirmed, this work could have direct impact on the field as follows. PDC researchers frequently base research conclusions on results obtained through simulation; any improvement to the accuracy of the PDC simulators being used would increase the quality of these conclusions. Likewise, PDC practitioners base decisions on PDC simulation, for instance when making platform provisioning decisions; higher-accuracy simulation would lead to a better foundation for these decisions, potentially saving both large amounts of money and/or natural resources otherwise wasted by poor hardware provisioning choices. The above improvements in both PDC research and PDC practice will naturally translate to improved runtime systems and platform architectures. These improvements will in turn directly benefit computational applications across many fields, e.g., virtually all scientific fields.

1.4 Organization of this Dissertation

The chapters of this dissertation are organized in four parts (relevant previous and related work is cited and discussed in each chapter, as needed):

Part I defines the simulation calibration problem and details our methodology for solving it. Specifically:

- Chapter 2 provides a definition of a simulator, gives the motivation for using a simulator for PDC research, and outlines the decisions that need to be made to do so effectively.

- Chapter 3 defines the simulation calibration problem, identifies and explains the key challenges involved when calibrating a simulator, and reviews the state of the art of simulator calibration in the field of PDC.
- Chapter 4 details our proposed methodology for solving the simulation calibration problem, and gives useful guidelines for addressing the involved challenges.

Part II evaluates and validates our methodology via three case studies. Specifically:

- Chapter 5 presents a case study in calibrating a simulator of scientific workflow executions on a cloud platform.
- Chapter 6 presents a case study in calibrating a simulator of MPI (Message Passing Interface) applications on a leadership-class supercomputer.
- Chapter 7 presents a case study in calibrating a simulator of High-Energy Physics workloads executions on a commodity platform.

Part III explores two applications of (accurate) PDC simulators. Specifically:

- Chapter 8 investigates the question of determining the appropriate level of simulation sophistication that should be implemented in a simulator.
- Chapter 9 assesses the potential benefit of online simulation-driven portfolio scheduling in the context of scientific workflow applications.

Finally, **Part IV** consists of Chapter 10, which summarizes our contributions and discusses future work directions.

Part I

Simulation Calibration

CHAPTER 2
SIMULATION

Simulation is a vital tool in many domains. It allows for experimentation in areas or at scales where it would be impractical or even impossible to perform real-world experiments. Although simulation approaches and technologies have domain-specific features, fundamental aspects are shared across domains. In this chapter, we first define what a simulator is in general, distinguishing between its inputs and its parameters. We then discuss the need for and the state of the art of simulation in the context of PDC research and development. Lastly, we discuss the specific simulators and simulation technologies we use in this work to both drive and evaluate our research contributions.

2.1 Simulator Definition

Simulators span a wide range of designs and applications. In general a simulator is a software artifact that is configured by a set of parameters, takes a set of inputs, and produces output with the intent of replicating the behavior of some target system. The simulator can be conceptualized as a family of functions that map a set of inputs to a set of outputs for some given set of parameters.

2.1.1 Gravity Simulator Running Example

While discussing simulators, it will be useful, for illustration purposes, to consider a simple example. Suppose there is some system we know (or at least think) can be modeled reasonably well using a quadratic polynomial with three inputs. The simulator code would then define the family of functions:

$$F(t, v, h) = At^2 + Bvt + Ch \ , \tag{2.1}$$

where t, v, and h are the inputs, and A, B, and C are the parameters. The parameters are effectively constants that control the behavior of a simulator, whereas the input is the initial conditions of the simulation. Once values are chosen for all parameters, the simulator is said to be *instantiated* as it now defines a specific function. For example, to model the elevation

of a projectile launched vertically under Earth's gravity, the parameters could be assigned as ($A = -\frac{1}{2}9.8$, $B = 1$, $C = 1$), so that the instantiated simulator is the function:

$$F(t, v, h) = -\frac{1}{2}9.8t^2 + vt + h \ . \tag{2.2}$$

We use the function in Eq. 2.1 to define an example simulator, which we call GravitySimulator, that will be used for illustration purposes in this and future chapters. GravitySimulator takes as input the initial elevation of the projectile, $h \geq 0$, and the initial launch velocity, $v \geq 0$. It produces as output a time-stamped series of the projectile's elevation, until this elevation is 0 (i.e., the projectile has hit the ground). GravitySimulator samples the above function at discrete-time steps until the function's output is below 0. To this end, it takes in an additional parameter to define the time step duration, Δt. As a result, the input t is not an input to the simulator but is instead derived from Δt. In all that follows, we use $\Delta t = 0.01$ unless otherwise specified.

Figure 2.1 depicts the simulator, which is configured by the parameters A, B, C, and Δt, takes in the inputs v and h, and outputs the elevation of the projectile at each time step until the elevation is ≤ 0.

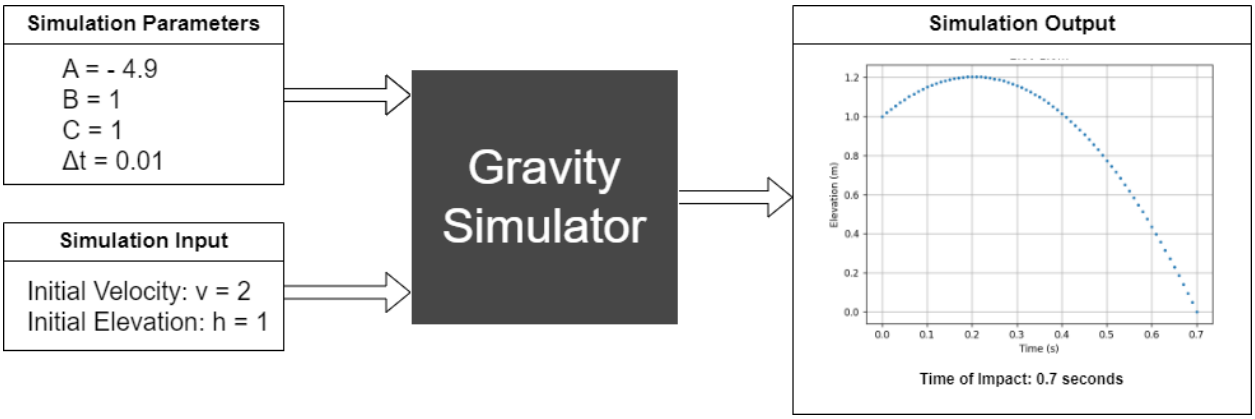


Figure 2.1: GravitySimulator used as a running example throughout this and subsequent chapters. Simulation parameters and input shown with sample values; simulation output shown as a elevation vs. time plot.

As Eq. 2.1 uses a well-known physics formula, we happen to know the correct parameter values ($A = -\frac{1}{2}9.8\text{m/s}^2$, $B = 1\text{m/s}$, $C = 1\text{m}$), leading to GravitySimulator being instantiated as in Eq. 2.2. But, for a general simulator, the correct parameter values are not known a priori. Consequently, when this simulator is discussed in future chapters, we assume it is necessary to discover the values of A, B, and C without any knowledge about the laws

of physics. Furthermore, in this simple example, we could easily algebraically manipulate the function to find exact solutions rather than discrete ones. Instead, and as in the general case, we treat the above function (and simulator) as a black box that can only be sampled.

2.1.2 Interest Metric

A simulator should produce output that is useful for the user. A simulator may produce output that is more detailed than necessary for a particular use case. More precisely, the user may be interested in a metric that is only a subset of or is derived from the output of the simulator. We refer to this metric as the user's *interest metric*. For example, suppose a user who is using GravitySimulator is interested in how long the projectile remains above a certain elevation. That duration is their particular interest metric. The simulator, however, outputs a time-stamped sequence of elevations. The interest metric is then computed as a single scalar based on the simulator's output. There could be many interest metrics that are relevant to different use cases, such as the maximum height reached by the projectile, or the full trajectory of the projectile (in which case the interest metric is exactly the simulator's output). More generally, an interest metric may also be computed based on the output from multiple runs of a simulator. For instance, if GravitySimulator took as input a single time and returned the projectile's position at that time, then multiple invocations of the simulator would be necessary to compute the above interest metrics.

2.1.3 Simulator Sophistication

To be useful, a simulator should mimic the behavior of the real-world system it purports to simulate as closely as possible, in which case we say the simulator has high *accuracy*. In some cases, a simulator can be easily expressed as a mathematical formula, which may have convenient properties such as being continuous or differentiable. But in many relevant use cases, to be accurate, a simulator should be more complicated than can be expressed with a simple formula, and thus may lack such properties.

At simulator implementation and instantiation time, there is frequently a choice between accuracy and computational complexity. For example, do you simulate a hurricane as a particle cloud, or do you simulate it as a single object with size, spin, and direction? Generally speaking a more complex simulation is potentially more accurate, but also slower to compute. The computational complexity of a simulator can also be a function of its parameters. Consider again GravitySimulator. Simulating with a $\Delta t = 1$s parameter value may be

less accurate than simulating with a $\Delta t = 0.01$s value, but will be 100 times faster.

Overall, based on its implementation and/or the value of some of its parameters, a simulator can simulate real-world behaviors at various levels of detail, which we refer to as the simulator's *sophistication*. A higher sophistication implies higher computational complexity, with the hope that it also has higher potential accuracy.

2.2 Simulation of PDC Systems

In this section, we highlight the commonplace motivations for using simulation in the context of PDC research and development, and review the current state of the art.

2.2.1 Motivation for Using Simulation

Simulation is used extensively for research and development in many fields, and the PDC field is no exception. The reasons for using simulation in this field, rather than conducting real-world experiments, include:

- **Increased Experimental Scope** – Simulation enables experiments to be performed on platforms that are not available for real-world experimentation. These include platforms that exist but are not available to all users for conducting experiments, e.g., experiments using the entirety of the El Capitan [64] cluster (currently number one on the Top500 list [104]). These also include platforms that do not or may never exist, but are still of interest for research purposes, e.g., hypothetical future cluster designs. In addition, simulation can allow for the exploration of unrealistically ideal or disruptive environments, e.g., platforms with zero-latency interconnects, or platforms where critical components "crash" at particular times or with particular frequencies.

- **Reduced Development Costs** – Simulation can decrease the time and effort required to develop an experimental system. For example, instead of having to physically build a new network topology, it can be simulated in software. Likewise applications with known performance characteristics can be simulated based on those characteristics without requiring full-fledged implementations of applications and/or runtime systems, or access to requisite datasets.

- **Reduced Experimentation Cost** – Simulation can decrease the time, energy, and cost of performing experiments. This is because, in many cases, simulation execution

time can be orders of magnitude shorter than simulated execution time, thereby consuming fewer compute resources. For example, Chapter 9 contains experiments that could have been done in the real-world instead of in simulation. Had we done so using the same hardware, it would have take roughly 170 years to gather the data. Instead, all data in that study was gathered in a month and a half using simulation.

- **Perfectly Repeatable** – Simulated executions are typically deterministic, making simulation experiments perfectly repeatable, which is not the case for real-world experiments. A simulator may implement randomized behavior, but these can always be made fully deterministic (e.g., via the use of seeded random number generators).
- **Perfectly Observable** – As a simulator is purely software, the state of any part of the simulation can be observed at any time without impacting the simulated execution. This makes it possible to obtain measurements from all simulated components, which is not possible in real-world experiments. This is especially useful when paired with repeatability: When new observations of a part of the system are needed, previous experiments can be perfectly repeated to produce the new observational data.

2.2.2 State of the Art

Simulation of PDC systems have been used for many research and development, and even educational, purposes, including:

- Informing platform provisioning decisions for particular application workloads;
- Comparing alternative hardware components and/or architectures for PDC platforms;
- Estimating environmental impacts of computing;
- Analyzing the performance of popular PDC programming models and runtime systems;
- Evaluating scheduling and resource management techniques;
- Providing experimental platforms for PDC education.

These purposes have been pursued for different classes of platforms, including traditional clusters, cloud platforms, distributed computing federations, as well as individual machines with multi-core processors. The above diversity in purposes and platforms is well-documented. For instance, the popular SimGrid [18] PDC simulation framework keeps track of objectives

and platforms targeted by research publications that use it. Figure 2.2 shows publication counts per domain/platform combination for the 2016-2022 time period by authors who use the SimGrid framework.

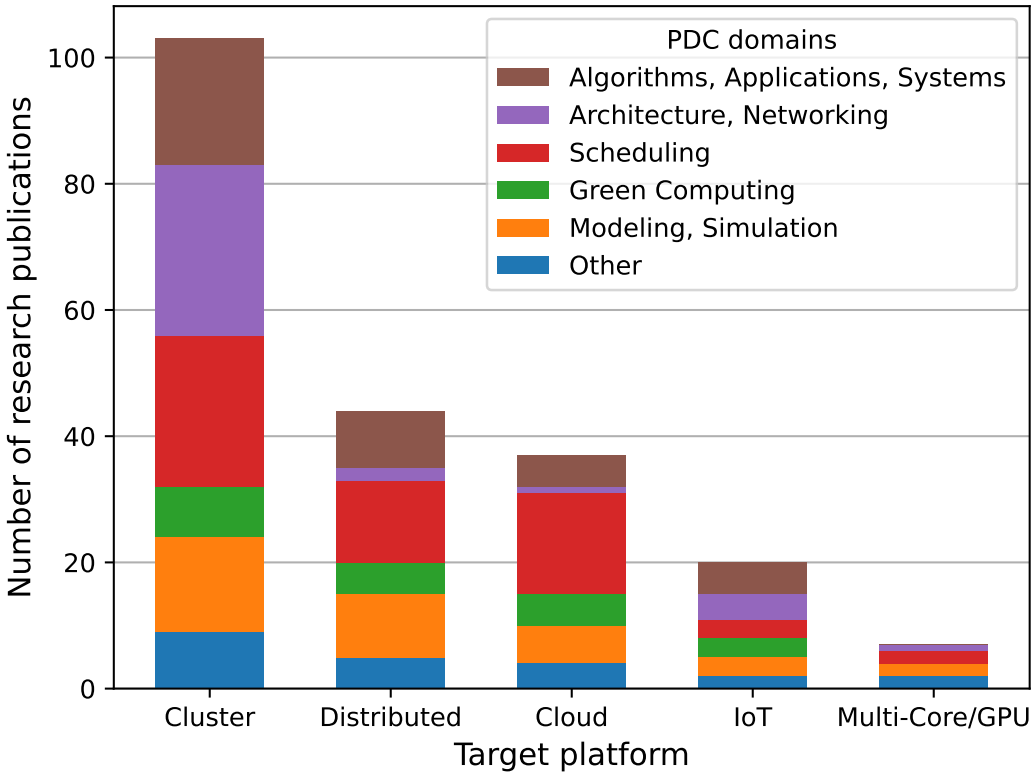


Figure 2.2: Counts of research publications that include simulation results obtained with the SimGrid simulation framework for different domain/platform combinations for the 2016-2022 time period. Figure reproduced from [20].

There is a large body of literature devoted to the simulation of PDC platforms and applications with many PDC simulation frameworks. Some of these frameworks have garnered sizable user communities and are still actively maintained at the time of writing. They can be placed into two broad categories based on the level of details of their underlying simulation models, with different implications on accuracy, scalability, and versatility.

Perhaps the most natural approach is to implement simulation models at high levels of detail, in an attempt to reproduce near-exact real-world behaviors so as to achieve high accuracy. Many simulation frameworks have followed this approach for simulating network resources (packet-level simulators [90, 77]), compute resources (cycle-accurate simulators [9, 15]), and I/O resources (block-accurate simulators [44]). These frameworks are not *versatile* from a PDC standpoint because they each focus on a particular class of hard-

ware components, and as such are typically used by researchers who specialize in studying these components. While it is conceivable to combine multiple such frameworks to create a versatile PDC simulation framework, doing so is impractical due to scalability limitations as simulation time can be orders of magnitude longer than simulated time. Some PDC simulation frameworks have been developed that simulate network communications at the packet level for high accuracy, with typically a severe loss in scalability, but simulate other components at lower levels of details [84, 62].

One solution to address the scalability issue posed by simulation models that implement a high level of detail is to employ Parallel Discrete Event Simulation (PDES) [40]. This approach has been used successfully by PDC simulation frameworks developed for simulating high-performance computing (HPC) applications and platforms [25, 16, 12, 55, 100]. The use of PDES allows these frameworks to scale up the platform on which the simulation is executed, possibly requiring a platform of the same scale as that of the platform being simulated. By contrast, many other PDC simulation frameworks, as discussed hereafter, aim to achieve simulation scalability even when executing the simulation on a single compute core.

An alternate approach for achieving scalable simulations, without scaling up compute resources, is to implement simulation models at a relatively low level of detail. That is, rather than simulating "microscopic" behaviors of the target system, these models instead rely on mathematical models that aim to capture "macroscopic" behaviors. For instance, rather than simulating the lifecycle of individual network packets, they compute instantaneous data transfer rates based on network path bandwidths and latencies and on current network usage. A key challenge is to develop such coarse-grain simulation models that are reasonably accurate in spite of their low level of detail [113]. Several popular PDC frameworks have been developed using this approach [18, 11, 13, 124, 61, 85, 71, 109].

2.3 PDC Simulation in this Work

In this work, we use several simulators of various target PDC domains for driving and evaluating research contributions via realistic and relevant use cases. These simulators are not implemented from scratch, but instead are built on top of popular PDC simulation frameworks. The contributions of this work are completely agnostic to the simulation framework used to implement a simulator and to specifics of a simulator's implementation. Nevertheless, for completeness we describe hereafter the specific simulation frameworks that are used for our target simulators, and the particular use cases we consider using these simulators.

2.3.1 PDC Simulation Technology

Many of the simulators used in this work are built using a simulation framework called WRENCH [20, 122]. WRENCH itself builds on the SimGrid simulation framework [18, 95]. SimGrid can also be used directly to implement simulators, as is the case for some of the simulators used in this work.

SimGrid

SimGrid [18, 95] is a discrete-event simulation framework dedicated to the simulation of PDC platforms and applications. The term *discrete-event* refers to a simulation in which time is advanced based on the determination of when the next relevant simulated event will occur. This determination comes from simulations models that provide estimates of the duration of simulated activities. For instance, say a simulation model for data transfer duration consists in dividing the data size by the bandwidth. In this case, if the only simulated activity is a 10GB data transfer over a 100MB/s network path, the simulation model estimates the duration of that transfer to be 100 seconds. The simulated time is then advanced by 100 seconds, i.e., jumping forward to the next event (in this case, the data transfer completion), which has $O(1)$ time complexity. This is by contrast with a *discrete-time* approach, in which a time-step duration is chosen and the simulated time is incremented by that time-step duration at each iteration, so that the same data transfer simulation would have $O(n)$ time complexity, where n is the number time-steps that must be simulated. For PDC simulation purposes, discrete-time simulation is typically prohibitively slow whereas discrete-event simulation can be fast, resulting in simulation times that are orders of magnitude shorter than simulated times.

SimGrid comes with implementations of simulation models for PDC hardware and software components. For instance, it makes it possible to simulate compute hosts with some CPU speed and core count, with assorted hardware like RAM and storage; or network topologies that comprise network links and routers. Each of these components' specification includes some "speed" parameter (e.g., GFlop/s for a core, GB/s for a network link). These parameters influence the behavior of the aforementioned discrete-event models. These models account for complex real-world behaviors, including resource sharing and contention behavior. They have been designed to achieve high scalability (i.e., low time an space complexity). They have also been thoroughly validated [112, 111, 41, 65, 31, 91, 51, 102, 101, 27, 17], meaning that they will lead to high simulation accuracy provided their configuration parameter values are chosen appropriately.

WRENCH

Many PDC researchers wish to simulate the execution of various application workloads on distributed platforms, and often end up re-implementing the same simulation abstractions and mechanisms. Furthermore, these target PDC systems can be complex, meaning that implementing simulators of these systems can be labor-intensive when using a low-level simulation framework like SimGrid. To alleviate this issue, WRENCH [17, 122] is a framework that builds on SimGrid and was designed to provide pre-built, high-level, reusable simulation abstractions. Using these abstractions, users can implement simulators of complex PDC systems quickly and easily. These abstractions come as highly configurable "services" with well-defined and simple APIs, which users can re-use as building blocks in their simulators.

2.3.2 Target PDC Simulation Use Cases

To drive and demonstrate the contributions of this work we develop and/or re-use simulators that target specific PDC domains, as described hereafter.

Workflow Scheduling

Scientific workflows and their executions on PDC platforms have supported some of the most significant discoveries of the past decades [6, 29]. In this context, many researchers have used simulation to explore relevant questions (often to investigate scheduling and resource management strategies), which has motivated the development of simulation framework specifically for this purpose [22, 2, 121]. WorkflowSimulator [119] is a WRENCH-based simulator designed to evaluate various workflow scheduling strategies. WorkflowSimulator is used as a case study for our calibration methodology in Chapter 5 and later to study the impact of simulator sophistication in Chapter 8.

MPI Implementation Performance

Many researchers have used simulation to investigate the performance of MPI (Message Passing Interface) applications, relying on one of the many simulation frameworks developed for this purpose [59, 34, 52, 25, 12, 31]. MPISimulator [81] is one such simulator that has been developed using SimGrid's SMPI module to investigate the performance of MPI applications on the Leadership-class Summit HPC cluster. MPISimulator is used as a case study for our calibration methodology in Chapter 6 and later to study the impact of simulator sophistication in Chapter 8.

High Energy Physics

PDC platforms are used to support the high compute and storage demands of many High Energy Physics (HEP) applications, for instance for processing data generated by the Large Hadron Collider (LHC) experiments and physics simulations. DCSim is a WRENCH-based simulator that estimates the execution time of workloads frequently used by the Compact Muon Solenoid (CMS) collaboration [108] to analyze this data. DCSim is intended to investigate potential improvements in the hardware and software configurations of the Worldwide LHC Computing Grid [117]. DCSim is used as a case study for our calibration methodology in Chapter 7 and later to study the impact of simulator sophistication in Chapter 8.

Scheduling Using Simulation

Countless researchers have studied scheduling problems and proposed scheduling heuristics for improving the performance of applications on PDC systems. These heuristics however, have no guaranteed performance and may even lead to poor performance for particular scenarios. An intriguing use case for simulation is to implement several heuristics and use simulation at runtime to pick which one should be used. To evaluate the merit of this approach, which we term Scheduling Using Simulation, we have implemented a WRENCH simulator similar to WorkflowSimulator that is capable of using a portfolio of scheduling algorithms and speculatively simulating itself running under each. Using this simulator we evaluate the merit of Scheduling Using Simulation, in the context of Scientific Workflow scheduling, in Chapter 9.

2.4 Conclusion

In this chapter, we have defined what a simulator is in the general case and in the specific PDC case. We have motivated the use of simulation for PDC research and development, reviewed the state of the art, and provided the specific PDC use cases that we target in this work. In the next chapter we motivate and define the simulation calibration problem, which must be solved for simulators to produce output that is representative of real-world systems.

CHAPTER 3
SIMULATION CALIBRATION

In order to ensure that simulations are accurate, they must first be calibrated. Broadly speaking calibration is the process by which one selects the parameters for a simulator so that the simulator's output matches the behavior of the real-world system being simulated. In this chapter, we first outline what is required to calibrate any simulator. We then describe the challenges inherent to individual requirements, including PDC-specific considerations and examples. Finally, we examine state-of-the-art approaches for addressing these challenges in the context of PDC simulations.

3.1 The Simulation Calibration Problem

Ideally, simulated behavior would match real-world behaviors as closely as possible. Real-world behaviors are described through observations of the real-world system of interest under various conditions. We term the set of available such observations the *ground-truth data*. Given simulated executions for the same conditions as those used to obtain the ground-truth data, the calibration problem is an optimization problem: how to pick simulator parameter values that minimize the discrepancy between simulated and real-world behaviors?

While many algorithms can be used to solve optimization problems, before applying them to calibrate a simulator the simulator calibration problem must be instantiated. Doing so entails answering the following three questions:

1. Ground-truth Data Selection: How much, and what type of ground-truth data is required for calibration?

2. Loss Function Selection: How should the discrepancy between simulated and real-world behaviors be quantified so that the calibrated simulator's output yields realistic interest metric values and behaviors?

3. Parameter Selection: Which simulator parameters should be calibrated, and what ranges of values should be considered?

In what follows, we use again GravitySimulator from Section 2.1.1. Recall that this simulator takes as input a projectile's initial velocity (v) and initial elevation (h) and produces as output a sequence of time-stamped elevations. It is configured by parameters A,

B, and C, as well as a Δt parameter, which controls the resolution of the simulation. Due to the laws of physics, we happen to know the correct parameter values for A, B, and C. But for this example we assume it is necessary to discover their values using GravitySimulator as a black box that can only be sampled. We use the projectile's impact time as the interest metric. The ground-truth data is sampled from a more sophisticated model of gravity that considers air resistance (see Appendix A).

3.1.1 Simulators that Only Appear Correctly Calibrated

In addition to solving the optimization problem above, there is another concern: Making sure that the calibrated simulator is actually useful. The typical use case of a simulator is as follows: (i) calibrate the simulator's parameters to fit the available ground-truth data as best as possible; (ii) determine if the simulator is sufficiently accurate; and (iii) if so, trust that the simulator can be used to explore hypothetical scenarios that go beyond the available ground truth. For instance, for GravitySimulator, the goal could be to use the simulator to explore scenarios where the gravity is different (say, the moon vs. the earth), by changing the value of the A parameter. Unfortunately, it is the case that a simulator can be calibrated as best as possible with respect to the ground-truth data, and thus produce seemingly accurate results, but is fundamentally flawed. That is, changing parameter values to simulate scenarios that are not represented in the ground-truth data can lead to aberrant simulation results.

While the above problem can arise in GravitySimulator, is more evident in the following trivial example, which we call the Transportation Example: Consider a moving company whose goal is to minimize moving time. They use a simulator whose behavior is defined by three parameters: the speed at which a truck can be loaded, the speed at which a truck can be unloaded, and the speed at which a truck can drive. The simulator takes in two inputs: the amount of cargo to transport, and the distance between two locations. Once calibrated, the company will use the simulator to observe the effect of changing the parameter values, the goal being to determine which of these three steps of the moving processes should be improved in the real world. As for GravitySimulator, we assume that the simulator is a blackbox that can only be sampled.

Under certain conditions (discussed in later sections in this chapter), it is possible to calibrate this simulator perfectly but draw invalid conclusions when using it in the above experimental process. For example, suppose the simulator's correct calibration is: Loading speed α, Driving speed β, Unloading speed α because, say, in the real world, loading and

unloading a cargo takes the same time. But a calibration has been computed as: Loading speed $\alpha/2$, Driving speed β, Unloading speed ∞. These two different calibrations yield calibrated simulators that simulate the same time to move a given cargo a given distance. Yet, one of these calibrated simulators is fundamentally flawed as it models that loading a cargo takes twice as long, but unloading a cargo is instant. Using the flawed simulator for experiments would show that improving the loading speed will significantly improve moving speed, but that there is no benefit to improving unloading speed, despite these being equally important in reality.

This problem is cross-cutting and partly related to each of the other problems we discuss in the upcoming sections. For this reason, each of these sections includes a sub-section entitled "Transportation Example" in which we address how to mitigate the above problem in various ways.

Although this Transportation Example may seem arbitrary and trivial, it is in fact highly relevant to PDC simulators. The Transportation simulator is nearly analogous to a PDC simulator where a file (the cargo) is read from storage at some host (loaded at the first location) then transmitted over a network to another host (driven some distance at some speed) and saved to storage at that host (unloaded at the second location). The only difference between these scenarios is that the driving time in the Transportation Example does not depend on cargo size whereas network transfer time does depend on file size. Nevertheless, many of the issues we discuss in the context of the Transportation Example in the upcoming sections are also relevant for the calibration of PDC simulators. Additionally, a common use case for such a PDC simulator is hardware provisioning (e.g., quantify the impact of hardware upgrades), which is analogous to the above use case for the Transportation simulator and thus can suffer from exactly the same issues.

3.2 Ground-truth Data Selection

To be able to compare simulated behavior to expected real-world behavior, one must first collect data that describes real-world behavior, i.e., ground-truth data. Generally, ground-truth data is obtained from observations of the real-world system being simulated. But it can also be obtained from simulated behavior generated by more sophisticated simulators (as we do for our gravity example). For calibration purposes, using more ground-truth data potentially enables a better fit of the simulator to the real-world system, rather than overfitting to just the ground-truth data. However, obtaining data from a real-world system

can be difficult and/or costly. As such, there is another optimization problem: Collecting the minimum amount of ground-truth data that will prevent overfitting.

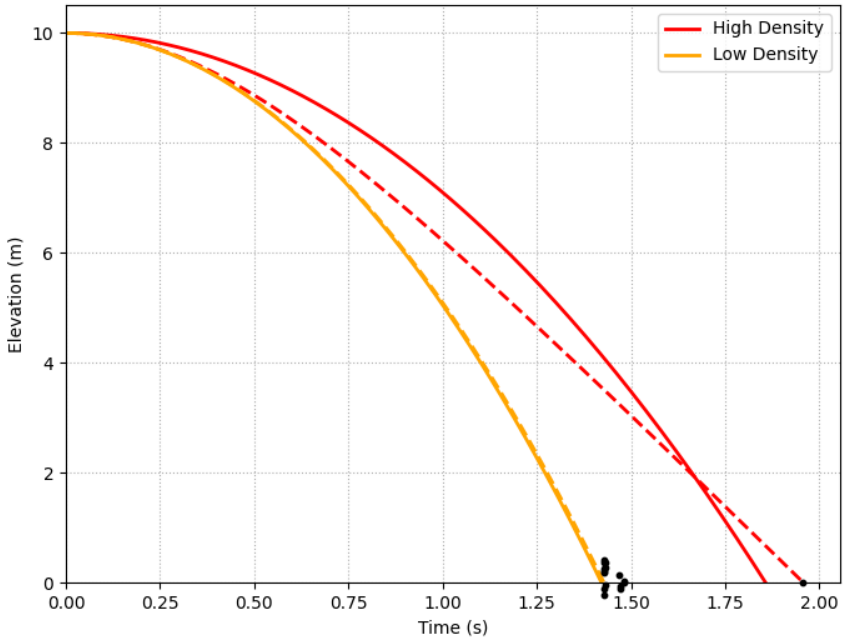


Figure 3.1: Projectile motion curves from several parameter sets that create similar loss values according to the impact time loss function with initial elevation of 100m and no initial velocity. Curves in red correspond to calibration using ground-truth data that contains only only projectiles with density $2kg/m^3$ and curves in orange correspond to calibration using ground-truth data that contains only projectiles with density $2,000kg/m^3$. The dashed line of each color indicates one of the ground-truth trajectories considered for that calibration. The ground impact times of the other ground-truth data have been plotted as black dots (with added random jitter for better visibility) on the time axis. These black dots include points that are not in either of the ground-truth datasets used for the above calibrations.

One reason why one may overfit a simulator to the ground-truth data is that the available ground-truth data is not sufficiently diverse, even though it may be plentiful. Considering our gravity example again, Figure 3.1 shows two calibrations of GravitySimulator where all factors are the same except one calibration is computed based on ground-truth data for objects with density $2kg/m^3$ and radius $5m$ and the other for objects with density $2,000kg/m^3$ and radius $5mm$. Since our simple simulator does not consider air resistance, gravity appears to be lower for the less dense objects. In the extreme, such lack of diversity in the ground-truth data can cause the impact of initial velocity and gravity to be greatly underestimated. In general, lack of diversity in any dimension of the ground-truth data

can lead to this problem, regardless of whether the simulation uses this data directly (e.g., air resistance is ignored by GravitySimulator but must be substantially diverse in the ground-truth data to yield good accuracy).

Regardless, even with plentiful and diverse ground-truth data, one should not expect to be able to calibrate a simulator perfectly. First, the underlying simulation model may not be able to perfectly reproduce real-world behavior due to insufficient sophistication. Consider GravitySimulator again. At its current level of sophistication, it does not consider air resistance. Consequently, no single set of values for its parameters will make the simulation accurate for both a large low-density object and a small high-density object. Second, real-world systems are non-deterministic, so the obtained ground-truth data is typically noisy. It is generally not possible to perfectly match all of the ground-truth data with a single calibrated simulator, even if calibrated as best as possible.

3.2.1 Transportation Example

For the Transportation Example, lack of diversity in the ground-truth data can lead to an additional problem. Consider the case where the combined time to load and unload the truck is the same as the driving time in the ground-truth data. In this case, a seemingly valid calibration could set the loading and unloading speed to infinity, and set the driving speed to half the actual driving speed. Similar to what was discussed in Section 3.1.1, using this simulator to explore, for instance, the effect of modifying the loading speed, will lead to incorrect observations and conclusions. This problem can be addressed by ensuring that the ground-truth data has sufficient diversity by including observations for multiple cargo/distance ratios.

3.3 Loss Function Selection

Given the output of any (useful) simulator, there should be a way to quantify how close that output is to the ground-truth data. We define an *experiment* as the simulator's input (including all simulator parameter values), the simulator's output, and the ground-truth data that corresponds to the conditions described by the simulator's input. Given an experiment, a *loss function* needs to be defined that quantifies, as a scalar, the discrepancy between the simulator's output and the ground-truth data. Given multiple experiments, these scalars then need to be *aggregated* into a single loss value. The goal of the calibration process is then to minimize this (aggregated) loss value.

A key question is: given a calibration problem, which loss function should be used? For most relevant problems there are many viable loss functions. Consider GravitySimulator again. Viable loss functions could be computed based on many different metrics, such as the time at which the projectile strikes the ground, the velocity at which the projectile strikes the ground, the maximum height the projectile has reached, the time at which the maximum height was reached, the position at any given time, the velocity at any given time, etc. These values can be compared to their ground-truth counterparts in various ways (absolute difference, relative difference, relative square difference, etc.), and then aggregated together in various ways as well (sum, weighted sum, maximum, minimum, etc.) to yield a single loss value for an experiment. Any of these many options could be a reasonable loss function to use for defining the calibration problem.

Recall that the user's objective is to minimize the discrepancy between the simulated and the ground-truth interest metric (see Section 2.1.2). It is therefore tempting to simply use the discrepancy between the simulated and ground-truth interest metric as the loss function. However, it is likely that a loss function that captures more behaviors of the target system will yield a better calibration. Therefore, it is also tempting to use a complex loss function that can accounts for many behaviors of the target system. But accounting for all these behaviors may, perhaps surprisingly, yield a calibration that makes the simulator less accurate for the interest metric that is relevant to the user. Let us illustrate these two extreme options using GravitySimulator.

Let us first consider the error on the interest metric (the projectile's impact time) as the loss function. We computed several calibrations based on all available ground-truth data. Figure 3.2 shows the simulator output for each of these calibrations, which shows that all their impact times are almost perfectly accurate, thus indicating low loss. Specifically, the brown, purple, blue, and red curves all have identical loss values. The parameter values in these calibrations are very different, and so is the resulting simulated behavior. As a result, although for the available ground-truth data these calibrated simulators yield "good" results, no more than one correctly models the real-world behavior and conclusions drawn from the others are likely incorrect.

Let us now consider the other extreme, where we use a complex loss function that accounts for many of the behaviors of the real-world system. Specifically, we use a loss function that equally weighs the relative difference between the position and velocity of the projectile at every time step, the time and position of the apex of the trajectory, and the time and velocity of the impact. Figure 3.3 shows results for a calibration that achieves low loss according

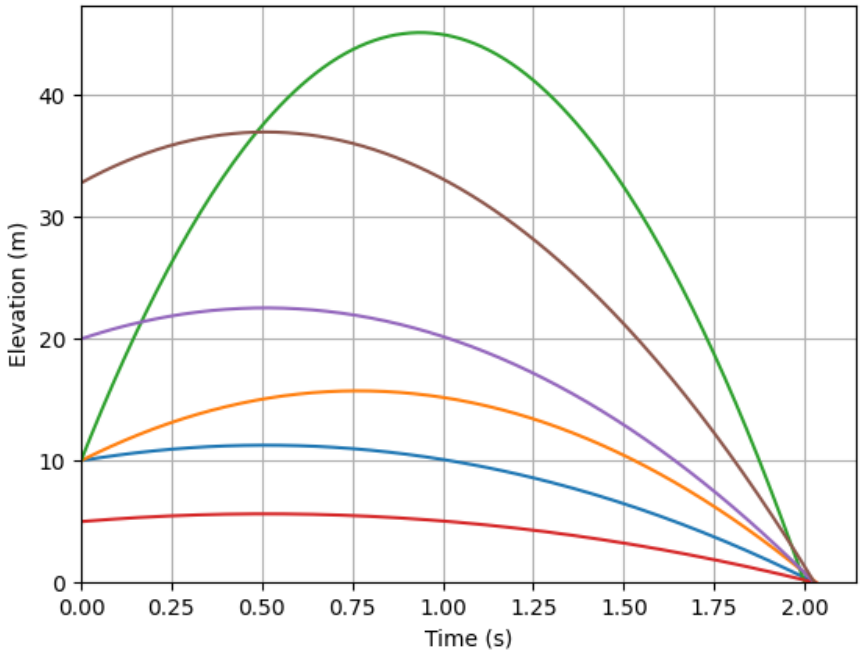


Figure 3.2: Projectile elevation vs. time with initial velocity 5m/s and initial elevation 10m, using the error on the interest metric (the projectile impact time) as the loss function. Curves are shown for different calibrations, all leading to similar loss values. The blue curve is the "real" calibration.

to this "advanced" loss function (the blue curve), and for a calibration computed using the simple loss function that is just the interest metric (the orange curve).

The calibration computed with the complex loss function achieves a loss of 9.8, while the "real" calibration yields a loss of over 10^9 according to that same complex loss function. Note that even the "good" loss values seem high compared to the previous loss function, but it is a consequence of the loss being a combination of many metrics. In terms of error on the interest metric, the calibration computed using the complex loss function yields a value of 0.18, which is worse than the 0.11 value for the calibration computed using the simple loss function. Despite this more complex loss function, this calibration does not perform as well, both in behavior and impact time. Overall, while a simplistic loss function may not be desirable, an overly complicated loss function may also not be desirable.

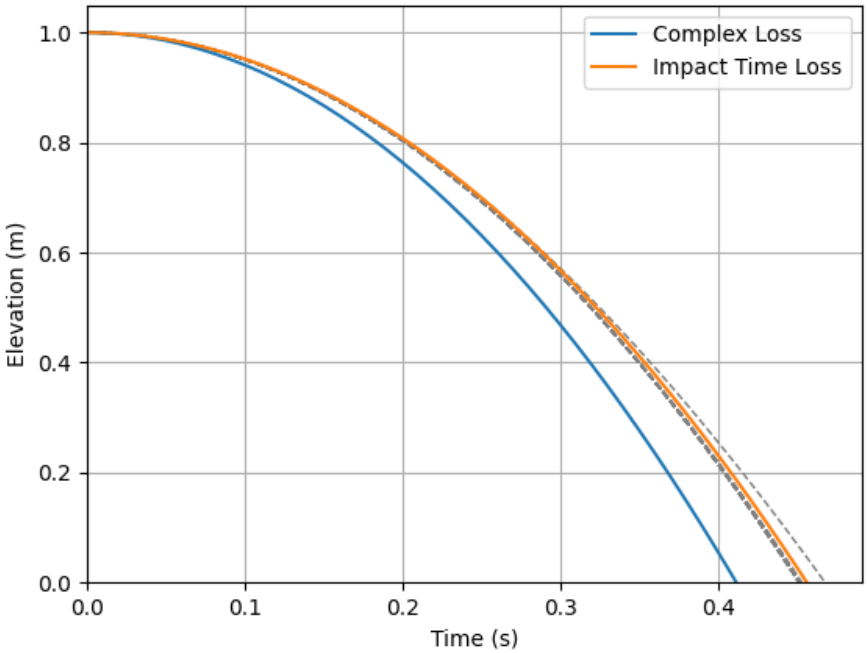


Figure 3.3: Projectile elevation vs. time with initial velocity 0m/s and initial elevation 10m. The blue curve corresponds to the calibration computed using an "advanced" loss function. The orange curve corresponds to the "real" calibration. Projectile trajectories from of the ground-truth are shown in gray.

3.3.1 Transportation Example

When discussing the Transportation Example in the context of ground-truth data selection (Section 3.2.1), the simulator was calibrated using the error on the interest metric (total time) as the loss function. A more advanced loss function that considers the times required to load, drive, and unload the truck individually could help prevent the problem highlighted in that section. Such a loss function would heavily penalize parameter sets that make any individual action significantly faster or slower than its real-world speed. This can mitigate some of the impacts of insufficiently diverse ground-truth data, as it does in this example. However it may not mitigate all of the impacts, as demonstrated for GravitySimulator in Section 3.2.

3.4 Parameter Selection

All simulators have parameters, some of which are to be calibrated and others are not. For instance, for our example gravity simulator, we calibrate A, B, and C, but do not calibrate the exponents of t (which are set to 2, 1, and 0). Virtually all simulators have similar parameters whose values are chosen when implementing the simulator, that impact simulation results, but are not intended to be calibrated as they are (believed to be) inherently known and correct. While it may seem desirable to calibrate all parameters so as to get the best accuracy possible, the computational complexity of the calibration problem is driven by the number of parameters to calibrate. A simulator with more parameters will thus take longer to calibrate, and typically exponentially so. Additionally, more parameters can add degrees of freedom to the simulator. This increases the risk of computing calibrations that fit the ground-truth data "for the wrong reasons" (as further discussed in Section 3.4.1).

Overall, the number of parameters to be calibrated should be as small as possible, but deciding which parameters should be calibrated and which ones should not can be a challenge. Figure 3.4 shows loss values (left side) and calibrated A parameter values (right side) for 16 different pairs of fixed (non-calibrated) values for parameters B and C. These results are obtained using the simple loss function in the previous section, that is, the error on the impact time interest metric. The main observation is that for any value of B and C we can find a value for A that minimizes the loss function. However, we observe that different calibrated A values yield the same loss (e.g., values shown on the anti-diagonal of the right-hand side plot yield the same losses as shown on the anti-diagonal of the left-hand side plot). By forgoing the calibration of parameters B and C, we obtain loss values that are likely much larger than they would have been had we calibrated these two parameters.

Note that some parameters are known to have an impact on both simulation accuracy and on simulation time. Consider GravitySimulator again. This simulator uses discrete time steps. We have arbitrarily set the step size to 0.01 second, which matches our ground-truth data. But increasing the step size would decrease simulation time, which can be desirable for some use cases of the simulator. However, an increase in step size can also decrease accuracy since the impact time interest metric can only be as precise as the chosen step size. Setting the step size to 1 second would reduce the simulation time by $\sim$100x, but would also reduce accuracy. For instance, when using the "real" parameters values (i.e., arguably the best possible calibration), the error on the interest metric when using 1 second time steps increases from 0.12% to 0.46%. It many thus be advisable to consider that different time step values define different levels of sophistication of the simulator (in essence, different

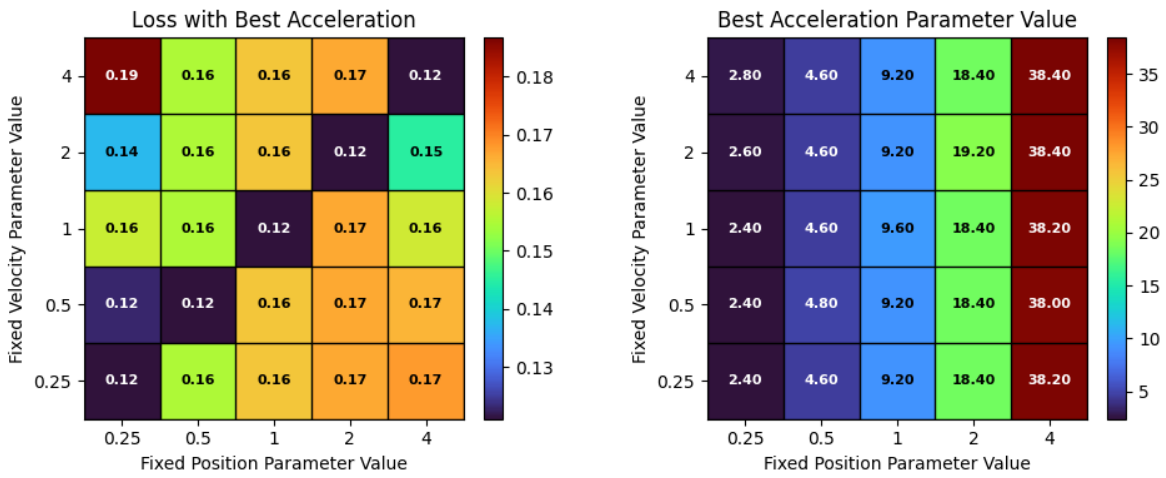


Figure 3.4: Left hand-side: loss when calibrating parameter A for fixed values of parameters B and C; Right hand-side: calibrated A value for fixed values of parameters B and C.

versions of the simulator), rather than considering the time step as a parameter to calibrate for which there is a single correct value.

Once the parameters to be calibrated have been selected, the range of possible values for each parameter must be determined. While not strictly necessary in theory, in practice, all calibration techniques rely on finite ranges of values to search a bounded parameter space. Smaller parameter ranges allow the calibration process to search the parameter space faster, but parameter ranges should be large enough so that the parameter space has a high chance of containing the optimal parameter values. Parameter ranges are typically chosen based on the user's available knowledge about the target system. As this knowledge is often imperfect and/or incomplete, in practice it is expected that the user will specify parameter ranges that are relatively large.

3.4.1 Transportation Example

In the simple gravity example discussion in the previous section, we highlighted that it is important to include relevant parameters in the calibration process. But in other cases, including too many parameters is harmful. For instance, consider our Transportation Example. The simulator uses three parameters, but two of these parameters are likely related to each other: the truck loading and unloading speeds are likely to be not be completely independent of each other but instead related by some constant multiplicative factor. As seen for GravitySimulator in the previous section, calibrating all three parameters can lead

to an infinite number of seemingly correct calibrations due to too many degrees of freedom. If the multiplicative factor is known, then the calibration should be performed only for two parameters.

3.5 Calibration of Simulators of PDC Systems

The research in this dissertation focuses on simulators of parallel and distributed computing (PDC) systems, i.e., computers distributed over a network working in concert to execute some application workload. In the following sections, we list PDC-specific considerations for the simulation calibration problem, we motivate the need for calibrating simulators of PDC system, and we review the current state of the art.

3.5.1 PDC-Specific Considerations

In the previous sections, we have used simple illustrative examples (gravity simulation and transportation simulation) to expose the reader to the simulation calibration problem in contexts that would be familiar and easily understood. Calibrating simulators of PDC systems is conceptually similar, but with several domain-specific considerations for the three questions listed in Section 3.1.

Ground-truth Data Selection

The ground-truth data necessary for calibrating simulators of PDC systems is based on execution logs collected from real-world systems running application workloads. Typically these logs include time stamps of relevant execution events such as beginning and end of computational tasks, I/O operations, and/or network transfers, and possibly the hardware resources used for these. At a minimum a log should include the start and end time of the whole execution, which makes it possible to compute the overall execution time, or *makespan*.

To collect this data, applications must be executed on real-world PDC systems. To minimize interference from other application executions, these executions should take place on dedicated hardware. Nevertheless, each execution should be repeated over several trials since real-world PDC systems always experience noise [26, 70]. Furthermore, executions should be conducted for several application configurations so as to ensure that the ground-truth data is sufficiently diverse, as discussed in Section 3.2. As a result, obtaining the ground-truth execution logs can be costly in terms of compute resources, energy, time, and/or

monetary cost. A key question is thus that of the minimal amount of ground-truth data that can be used to produce a good and generalizable calibration. We investigate this question in future chapters.

Loss Function Selection

PDC researchers and developers who use simulators for conducting their experiments typically use interest metrics that quantify some notion of overall application performance (the application makespan, the application throughput, the resource utilization, etc.). Unsurprisingly, there are many viable options for defining the loss function based on overall execution metrics (including the interest metrics) as well as individual metrics (individual task start and end times, task ordering, utilization of specific hardware resources, etc.). Which loss function is best for which use case is an open question, which we investigate in upcoming chapters.

Parameter Selection

When calibrating any simulator, and in particular PDC simulators, it is important to determine which parameters should be calibrated, and which should not. The goal is to define few parameters, so as to reduce parameter space dimensionality, but enough parameters that the calibrated simulator can be accurate. Parameter selection is typically based on available knowledge that the user may have about the target real-world PDC system. For instance, the user likely knows discrete properties of the real-world system, such as the number of compute nodes and the number of CPU cores. Similarly, the user may know that the compute nodes are homogeneous and that their compute speeds can be described with a single parameter, although this is likely only an approximation [70]. Other approximations can be done knowingly by the user. For instances, in some of the simulators used in this dissertation, there are hundreds if not thousands of possible parameters that define the size in byte of particular control messages. The user may know that the real-world system has a well-provisioned network with high bandwidth and low latency. As a result, because control messages have a small size, it is reasonable to assume they have the same small constant size, or even not simulate their transfer at all.

Note that even if parameter space dimensionality is not a problem, there can be drawbacks to including some parameters to calibrate. For instance, some applications may have execution-dependent behavior, observed in the ground-truth data but not simulated by the simulator. Consider a computational task that produces an output data file that is seen in

the ground-truth data, over multiple executions, that have sizes between 10MB and 20MB. One may decide to create a parameter that should be calibrated to define the size of that file. If one is also calibrating the I/O bandwidth of the hard drive to which that file is written, then there are too many degrees of freedom in the calibration problem, e.g., writing a 10MB file to a 100MBps disk will take approximately the same time as writing a 20MB file to a 200MBps disk. Consequently, there will be an infinite number of solutions to the calibration problem if both these parameters are included. A solution here, instead, would be to fix the file size parameter (e.g., to the average observed file size).

Finally, the user must determine ranges of values for each parameters. This determination is typically based on rough knowledge of hardware and software specifications. Although these specifications may be advertised and known to the user, they could be inaccurate or not paint a full picture of the real-world performance behavior. But they can still provide a basis for defining reasonable parameter ranges that can include the best calibrated value. The simulation calibration approach proposed in this dissertation is evaluated with broad parameter ranges so as to test "worst case" scenarios. These broad ranges can be based merely on knowledge of the current state of technology and be defined in terms of orders of magnitude. For example, given the current commodity cluster interconnect technology, the network in the University of Hawai'i's Koa commodity cluster is probably faster than 10Mbps but slower than 10Tbps.

3.5.2 Need for Calibration

PDC simulators have parameters that correspond to the hardware being simulated and to the software being simulated. Values for these parameters must be picked so that simulated behavior are as representative of real-world behaviors as possible. There are generally three approaches for selecting parameter values in practice:

1. Use advertised hardware specifications and software configurations of the real-world systems used to obtain ground-truth data;

2. Benchmark the hardware and software stacks used to obtain ground-truth data, so as to determine parameter values empirically; and

3. Perform calibration to compute parameter values that minimize loss compared to the obtained ground-truth data.

Among these three approaches, the first one is the most straightforward but it has many caveats. For instance, although hardware specifications are advertised, they typically represent peak performance under ideal conditions. In real-world executions of useful workloads, only a fraction of the peak performance is achievable and that fraction is use case-dependent. Another problem is that advertised specification and/or configurations do not necessarily include the metrics that are relevant to the simulation parameters. For example, software runtime systems typically do not advertise their various operational overheads, and yet these overheads are typically key parameters that must be provided to a simulator. Finally, the user may not know the exact specifications or configurations of the real-world system used to obtain the ground-truth data. For instance, often precise information on the network topology is not available.

The second approach of using benchmarks is more involved but should in principle address the caveats of the first approach. However, benchmarks implement specific, often simplified, conditions that are not necessarily representative of hardware/software performance for non-benchmark applications [72]. Furthermore, it can also be difficult to isolate the components being benchmarked. For instance, consider a network benchmark that is executed between two endpoints connected by some network path. The benchmark results provide information on the end-to-end latency and on the bandwidth of the bottleneck link on the network path. However, a simulator may take parameters that describe the network topology in more details (i.e., a list of links each defined by its own latency and bandwidth), and the values of these parameters cannot be determined from the benchmark results. Another difficulty arises because a simulator necessarily uses abstractions of the real-world hardware and software. For example, in real-world executions, execution performance on a CPU is partially dependent on data locality (i.e., whether data is found in some level of cache). But a simulator may have been implemented in a way that abstracts these details away and simply does not simulate CPU caches or data locality. As a result, the observed CPU speed achieved for a particular benchmark does not directly corresponds to the CPU speed parameter provided to the simulator.

The consideration of the level of abstraction of the simulation raises a more general difficulty with the first two approaches above: a parameter of the simulator may not map directly to a single, known characteristic of the real-world system. For instance, if a simulator abstracts a complicated network topology as a single network link with a latency and a bandwidth, it is difficult to come up with reasonable values for these two parameters based on knowledge (assuming this knowledge even exists) of the real-world network topology. In

situations where the simulator does not abstract components of the real-world system, one may surmise that it is possible and perhaps straightforward to pick appropriate values for these parameters based on known specification of the target system. But, even in these situations many authors have found that picking good parameter values is (surprisingly) challenging [4, 39, 42, 66, 57].

Thus, we are left with the third option, that is, simulation calibration. As noted in previous sections, simulation calibration comes with several challenges. This dissertation seeks to provide solutions to these challenges. In our experimental evaluation of these solutions in future chapters, for completeness, we include comparisons to the first and second approaches above (and, unsurprisingly, find them inadequate). In the next section we discuss the current state of the art for the calibration of simulators of PDC systems.

3.5.3 State of the Art

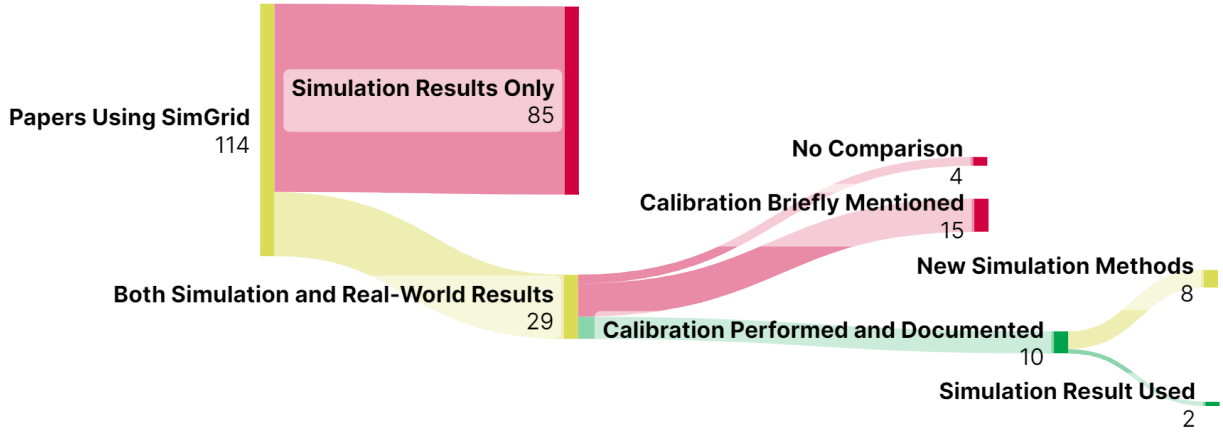


Figure 3.5: Examination of 114 research publications in the 2017-2022 time period that include results obtained with SimGrid.

Assessing the state of the art of calibration of PDC simulators is difficult. Fortunately, the SimGrid project maintains a list of research publications that include results obtained using its widely recognized and used PDC simulation framework [96]. At the time of investigation, this list included 610 publications. We selected the peer-reviewed journal or conference/workshop publications for the 2017-2022 6-year time period, which amounts to 114 publications. As an attempt to assess the state of the art, we have examined these 114 publications in detail to determine how they perform simulator calibration (many authors

refer to calibration as "parameter picking" or "parameter tuning"). Our results are summarized in Figure 3.5. Of the 114 publications, 85 include only simulation results, which likely indicates that calibration was not performed. In some of these works the goal is only to simulate a simplified model of an abstract system. In others, real-world data is used as input to the simulation, but no comparison with the real-world execution that has generated that data is done (or can be done). In yet other works, the goal is to simulate a system with hardware/software technology that does not (yet) exist. One reason why papers do not include real-world results is because simulation is often used precisely because such results cannot be obtained in practice. In many of these works, however, the simulation is intended to be representative of real-world systems. If calibration was not performed it is not clear what systems the simulation results represent.

29 of the 114 works we reviewed include both simulation and real-world results. We use a broad definition of the term "real-world", which includes not only results obtained on real-world hardware platforms, but also results obtained using emulation and results obtained using low-level simulation (e.g., results obtained using packet-level simulation of networks). 25 of these 29 works perform or allow comparison of simulation and real-world results. 15 of these 25 works either do not detail any calibration procedure or merely mention that picking better simulation parameters improves accuracy. Some of these works, however, present simulation results that exhibit high accuracy, which may indicate that calibration was performed even if not mentioned.

Overall, out of the 114 publications we reviewed, only 10 explicit perform some calibration and give details. Half of these describe manual painstaking procedures by which simulation parameter values are picked based on quantitative and qualitative comparisons of real-world and simulation execution logs and metrics, and sometimes on inspecting the source code of the target system's software stacks. The other half do perform similar procedures but also rely on simple statistical techniques (i.e., regressions). It is important to note that, for 8 of these 10 works, the main research contribution is a novel simulation model. Calibration is thus necessary to validate this simulation model. In the end, among the 106 publications that target a non-simulation-related research topic, we found only 2 that performs a solid and documented calibration procedure so as to ensure that simulation results are accurate.

The above indicates that simulator calibration is likely not performed routinely in the PDC field. Those works that perform simulation calibration typically employ labor-intensive, partially manual, procedures. When calibration is not performed, parameter values may be picked based on best guesses or simply by using the default values for models provided

by simulation frameworks. These defaults can come from calibration with respect to some real-world systems available to the developers of the simulation framework (as it is the case with SimGrid). Consequently, published simulation results obtained using default parameter values may be valid for some system configurations, but not necessarily for that of the particular systems of interest. Furthermore, not all simulation models are provided by the simulation framework, and custom models are also developed for each particular simulator. These custom models may not come with any (calibrated or even reasonable) default parameter values.

3.6 Conclusion

In this chapter, we have defined the general simulation calibration problem and identified the challenges that must be addressed for solving this problem, namely: ground-truth data selection, loss function selection, and parameter selection. We have then discussed the simulation calibration problem in the specific context of PDC research and development. An examination of the state of the art in that field reveals that most PDC simulators are either not calibrated, or calibrated using labor-intensive, at least partially manual, and often undocumented processes.

The current state of the art of calibration of PDC simulators provides the main motivation for the work in this dissertation. In the next chapter, we propose a methodology for addressing the challenges of simulation calibration in a view to produce automatically calibrated PDC simulators.

CHAPTER 4
SIMULATION CALIBRATION METHODOLOGY

In previous chapter, we defined the simulation calibration problem and identified the challenges it entails. In this chapter, we detail our methodology for addressing these challenges in view to solve this problem. We also present the framework we have developed, Simcal, for performing automated simulation calibration in practice.

4.1 Methodology

Chapter 3 identifies three questions that must be answered when calibrating a simulator:

1. Ground-truth Data Selection;
2. Loss Function Selection; and
3. Parameter Selection.

In this section, we outline our approach to answering these questions as well as a fourth question, to which we only previously alluded to:

4. Optimization Algorithm Selection: Which optimization algorithm works best for a given calibration problem?

Section 4.1.1 focuses on question 1, Section 4.1.2 focuses on questions 2 and 4 simultaneously, and Section 4.1.3 focuses on question 3. The discussion in these sections provides the foundation for our simulation calibration methodology, which we evaluate and validate in Chapters 5-7.

4.1.1 Ground-truth Data

The problem of ground-truth data selection is, unfortunately, heavily dependent on the particular system being simulated. Furthermore, the amount and type of ground-truth data required cannot be determined precisely prior to attempting the simulator calibration. We can offer general methods to determine if the ground-truth data is insufficient post calibration (see Section 6.5). But a general methodology for pre-determining what ground-truth data to collect and use seems out of reach. This said, hereafter we give several general guidelines,

which are common-sense and/or have emerged from our performing simulation calibration for the particular use cases described in future chapters.

In this section, we use the term *component* to refer to any element of the real-world system (e.g., hardware resource, software service) whose operations and behavior is simulated by the simulator, i.e., it has a corresponding simulated component. The behavior of the simulated component is configured via one or more parameters, whose values are to be calibrated (e.g., a network link with bandwidth and latency to be calibrated, a software service with overhead to be calibrated).

Our general guidelines are as follows:

• **Ensure that all components are stressed** – If a component exists in the simulator, controlled by parameters that must be calibrated, then the ground-truth data must include data from real-world executions that use the corresponding real-world component. Otherwise there is no way to quantify the impact of changing the values of the component's parameters on the accuracy of the simulation since nothing in the ground-truth data depends on that component. At best this results in a component with arbitrary parameter values.

• **Ensure components cannot compensate for each other** – If a group of components are used in conjunction (in real-world and thus also in simulated executions), then it may be possible for multiple (or even an infinite number of) calibrations to be equivalent in terms of the loss function. This creates a situation where the optimization algorithm must choose one of these. This choice is effectively arbitrary to the algorithm, and yet only one of these calibrations is actually correct. Selecting an incorrect calibration will cause the simulator to not generalize to situations where the components are not used in conjunction.

We have already identified this problem in Section 3.1.1, in the context of our Transportation example, which is analogous to situations that occur frequently in simulators of PDC systems. A solution to this problem is to ensure the ground-truth data is sufficiently diverse, as explained in Section 3.2.1. What diversity is required, however, depends on the specific platform and workload being simulated.

• **Ensure no component "masks" another** – In a similar situation to resources compensating for each other, it is possible for a component's correct configuration to be masked by another slower component. In this situation, there are again infinite possible calibrations so long as the slowest component happens to be the bottleneck of all components being masked.

There are several potential solutions to this, the simplest is to reduce the components to a single component that has the bottleneck speed. Considering the file server example,

suppose the user's use case in only concerned with how long a file takes to arrive from the file server, and not the inner workings of that server. In this case, the file server internal parameter values can be set arbitrarily high, and one need only calibrate the network bandwidth parameter. This is functionally the same as ignoring the bottleneck, but decreases the number of parameters to calibrate.

In the case where this solution is not viable, more ground-truth data is required to stress the individual parts of the bottleneck chain independently, such as adding network-only traffic to the file server. Another solution is to switch to a more abstract component (see further discussion in Section 4.1.3).

• **Account for noise within real-world platforms** – Real-world parallel and distributed computing platforms are all subject to "noise" [26, 70]. It is thus necessary for the ground-truth data to include multiple repeats of the same executions, to ensure that the ground-truth data is representative of the platform and not of a single noisy observation.

• **Account for required simulation execution time** – Most of our previous recommendations point to collecting and using more rather than less ground-truth data, which has two downsides. First, *collecting* more ground-truth data entails higher cost (in terms of labor, time, energy consumption, carbon footprint, hardware resources, and/or funds). Second, *using* more ground-truth data increases the number of simulator invocations when searching for a good calibration. If the time allotted to performing the automated calibration is limited, then the produced calibration may be (vastly) sub-optimal.

There is thus a tension between using too little and too much ground-truth data. This tension is necessarily use case-specific, and we explore it in future chapters.

4.1.2 Loss Function and Optimization Algorithm Selection

Recall from Section 3.3 that there are many options for defining a loss function to quantify simulation accuracy, and we expect that each user will construct a set of candidate loss functions for their use case. Selecting the most appropriate loss function is difficult because the values of different loss functions are not easily comparable. Even for the best possible calibration (i.e., the best set of simulation parameter values), different loss functions can have different (non-zero) values. Furthermore, when different loss functions lead to different calibrations one does not know which one is the closest to the (unknown) best calibration.

To resolve the above difficulty, we have developed a process we call *Synthetic Calibration*, for comparing the performance of multiple loss functions even though their output values

cannot be compared directly. The first step in our methodology is to generate *Synthetic Data*. This is a set of "fake" ground-truth data based solely on simulator outputs obtained for a particular set of parameter values. There are different possible approaches for picking these values. For instance, a domain expert could pick these values based on knowledge of the real-world system. Another approach is to compute a preliminary calibration and use the parameter values it produced as a basis for generating the synthetic data. Using this approach, however, some amount of noise should be added to these parameter values. This is to ensure that the optimization algorithm used to obtain the preliminary calibration does not give it preferential treatment by accident due to the specific way the optimization algorithm generates candidate parameter values to evaluate. We expect the method for adding noise to parameter values to be of little importance, and in our work we simply round all parameter values to 2 or 3 significant figures. For each scenario present in the real ground-truth data, the simulator is executed with the selected parameter values. Its output is post-processed to generate Synthetic Data in the same format as the real-world ground-truth data. In this manner, we obtain our synthetic ground-truth data.

The choice of loss function also can effect the performance of the optimization algorithms, thus the problem of which optimization algorithm to use must also be solved simultaneously. Therefore, our methodology considers all possible combinations of loss function and optimization algorithm: the calibration is performed for each such combination based on the generated synthetic data. Unlike with the real-world ground-truth data, the correct calibration for the synthetic data is known, and it is thus possible to quantify the goodness of the computed calibrations.

Specifically, using this known calibration T and a computed calibration C, we defined a distance metric D as:

$$D(C,T) = \sum_{i=1}^{\dim(\mathbf{T})} \left| \frac{C_i - T_i}{T_i} \right|$$

This metric allows us to compare each computed calibration to the correct calibration and select the combination of loss function and optimization algorithm that produces the calibration that is the closest to the true calibration. We use the above metric, instead of the L_1 or L_2 norm, because each dimension of the calibration search space has an unknown scale and importance. By using this metric, which is essentially a relative L_1 norm, the unknown scale is compensated for and each dimension is assumed to be equally important.

The above process allows us to find, experimentally, the best loss function and optimization algorithm pair to use for computing a calibration based on the real ground-truth data.

Note that the loss function in that pair may now be used to compare all optimization algorithms when calibrating based on the real ground-truth data to further confirm the choice of optimization algorithm if desired.

4.1.3 Parameter Selection and Ranges

Similar to the problem of selecting ground-truth data discussed in Section 4.1.1, the selection of parameters and their ranges is heavily use case-dependent. Nevertheless, we can offer a few helpful guidelines.

Parameter Selection

Recall from Section 3.4 that the goal of parameter selection is to determine a small set of parameters to calibrate. As discussed in Section 3.5.1, PDC simulators can have large numbers of potential parameters.

Some initial parameter selection is to be performed by the user, using expert knowledge about the simulator and the real-world system one which the ground-truth data was collected. For instance, although a system can comprise many compute hosts, the user may know that these hosts are homogeneous and thus described via the same parameter values. Similarly, the user may know that control message exchanged between processes are small enough that they should not have a significant impact on the performance of the execution, justifying forgoing their simulation or setting the value of the parameters that specify their size to a fixed (not to be calibrated) low number.

Parameters should be selected so that no parameter is fully redundant with another leading to a situation where one parameter only depends on the other. For example, let us consider a minor change to the PDC simulator discussed in Section 3.1.1, which is analogous to our Transportation example simulator. That is, let us add a second network link between the two hosts, as is the case in real networks in which end-to-end network paths comprise multiple physical links. Each of these two links has its own bandwidth. Any network transfer between these hosts will only depend on the bandwidth of the slower link, and the latency will be the sum of the latencies of both links. Assuming there is no ground-truth data that only uses one of these two links, there is an infinite number of possible calibrations that perform equally well so long as the minimum bandwidth of these two links is correct, and the sum of their latencies is correct. We have identified three ways to address redundant parameters:

- One can gather more ground-truth data so that there is data that exerts each component individually. This is often not viable, for instance, for network infrastructure.
- One can assume a fixed relation between these (such as homogeneity) thereby reducing the number of parameters to calibrate.
- The simulation component can be replaced with an approximation with fewer parameters (i.e., replacing the two network links with a single network link directly connecting the hosts).

Once a calibration has been obtained, the user may find out that the simulator has unacceptably low accuracy. In this case, the user may decide to include parameters that were initially excluded and repeat this process. Or the user may decide to increase the level of sophistication of the simulator by adding more components to the simulator, with their new parameters to potentially calibrate.

Finally, as discussed in Section 3.4, some selected parameters can have an impact on the execution time of the simulator. These parameters often have no equivalent in the real system but only control the resolution of the simulation (e.g., Δt for GravitySimulator as described in Section 2.1.1). These parameters may also represent a known property of the system being simulated where simulating it at the proper level drastically increases simulation time (e.g., the size of a buffer used to perform I/O operations). These parameters can change the runtime of the simulator by several orders of magnitude. In general, setting such a parameter to a value that causes the simulator to run slower also increases the potential accuracy of the simulator. This introduces a trade-off between runtime and accuracy. If calibration is performed using such a parameter, this parameter may be set to a value that increases runtime substantially. Recall that one benefit of simulation is the decreased cost in time and computation to run experiments, an increase in runtime will decrease (and can potentially eliminate) this advantage. Conversely, increasing the runtime of the simulator often also increases the time required to calibrate the simulator to a desired level of accuracy. This can result in a situation where actual accuracy is decreased despite the increase in potential accuracy.

A seemingly straightforward solution is to perform the calibration using fixed values for those parameters that allow for a faster simulator, and then change these parameter values to produce a slower (but hopefully more accurate) calibrated simulator. We investigate the effectiveness of this approach in Section 8.3. Another approach is to simply not calibrate these parameters. Instead, using different values for these parameters can be considered as

different versions of the simulator, each of which implements a different level of sophistication and is then calibrated separately.

Parameter Ranges

Even though narrower parameter ranges make solving the calibration problem easier due to a smaller parameter space to search, automated calibration should be effective even if the user specifies wide parameter ranges. This is because it is unrealistic to expect the user to have precise knowledge of the real-world system and precise insights on what good parameter values are. A possible and straightforward approach is to use the imprecise methods described in Section 3.5.2 for picking parameter values. Then choose ranges that include these values and are wide enough that if the parameter values that produce the lowest error fall outside these ranges then the simulator is likely incorrect. For instance, one could pick minimum values that are a few order of magnitude lower than, and maximum values that are a few order of magnitude larger than, the picked values. Consider, as an example, a parameter that defines the bandwidth of a 100GBps Ethernet network link in a HPC cluster. For this parameter one could use a $[1GBps, 1TBps]$ range. Once a preliminary calibration has been obtained with these broad ranges, if future rounds of calibration are needed, then the ranges could be adjusted based on the parameter values obtained in that preliminary calibration, e.g., to narrow ranges or to expand a range in one direction if a calibrated parameter value lies at one edge of the range. There are cases in which setting a parameter value below or above some threshold has no effect on the simulated executions as its effects are masked by another component (e.g., setting an I/O bandwidth parameter when simulating CPU-bound scenarios). If these thresholds can be determined, then parameter ranges can be adjusted so that their edges are set to these threshold values.

4.2 Simcal

To aid in automated simulation calibration, we have developed a Python framework called Simcal (a portmanteau of SIMulator CALibrator). Simcal is designed to be generic: it can invoke any simulator and calibrate it using a variety of calibration methods. The design goal is for Simcal to make it possible to apply our methodology to any use case with minimum amounts of effort. This is achieved via a set of utilities and wrappers, as described hereafter.

4.2.1 Simulator Wrapper

The implementation and input/output schemes of a simulator are completely use-case dependent and arbitrary, as there are no simulator standards in the field to date. A user's simulator could thus be an executable or a library, whose input is provided in any conceivable manner (command line arguments, piped arguments, files, parameters on the stack, etc.), and that produces output in any format. Consequently Simcal provides a *simulator wrapper*, i.e., a Python class that the user can derive. The user must override the wrapper's `run()` method with their own implementation to invoke their simulator. To make this process easier, the wrapper creates a temporary *environment* in which the simulator will be executed. This environment provides helper methods for managing temporary files and folders in such a way so as to not conflict with concurrent simulator invocations, for invoking bash scripts, for allowing the user to provide piped input to the simulator and retrieve piped output, etc.

The simulator wrapper is a Python object, and as such can store any use case-specific configuration variables, as well as loss function implementations. The `run()` method can invoke the simulator multiple times (e.g., for different ground-truth data items), and must return a loss value computed based on the simulator's output for these invocations. This method takes as input a map of parameter values, i.e., the candidate values for the parameters that are to be calibrated (see the next section for how the user can define calibration parameters). These parameter values must be transformed into the specific input data required by the simulator, which the user must implement as it is use case-specific. Similarly, the output from the simulator invocations must be processed, as implemented by the user, so as to compute the loss value achieved.

Throughout this section, unless otherwise specified, any reference to the simulator means calling the simulator wrapper's `run()` method with a particular set of parameter values.

4.2.2 Calibration Wrapper

The calibration problem is fundamentally a multidimensional optimization problem. As such, it should be possible to use any optimization algorithm to solve it. Many such algorithms are already implemented in various Python packages, but the corresponding functions expect input in different forms. For this reason, Simcal provides a *calibration wrapper*, which provides a common interface for implementing or integrating any optimization algorithm. This calibration wrapper uses the optimization algorithm selected by the user to invoke the simulator with candidate parameter values multiple times, constructs a list of the best

candidates over time (i.e., those that have improved the loss over the best loss previously achieved), and returns the set of parameter values that achieve the best loss overall.

The calibration wrapper controls the execution of the underlying optimization algorithm rather than completely handing over program flow control to it. As optimization algorithms are not guaranteed to converge to a solution within an acceptable amount of time, the calibration wrapper can enforce a time-based cutoff (where calibration runs for a set amount of time) or an invocation-based cutoff (where a set number of simulator invocations are performed). Due to the nature of our research, as explained in future chapters, we have only implemented time-based cutoffs in the calibration wrapper, but it is straightforward to extend the implementations to consider invocation-based cutoffs as well.

Implemented Optimization Functions

The current implementation of Simcal includes implemented versions of the following optimization algorithms:

- **Random (RAND):** Parameter values are sampled randomly, which can be parallelized trivially.
- **Grid (GRID):** Parameters values are sampled in a grid pattern, which can be parallelized trivially. To allow for a time limit rather than only an iteration limit, the grid starts at a coarse resolution and the resolution is doubled at each iteration.
- **Gradient Descent (GRAD):** At each iteration, a random set of parameter values are chosen as a starting point. From there the neighborhood around the current parameter values is sampled to approximate the gradient. The parameters are then moved in the direction of the gradient, iteratively. This implementation implements a classical backtracking line search optimization and upon convergence (or reaching a loss plateau) restarts at a new random set of parameter values. This algorithm can be parallelized by running multiple independent descents, which has higher parallel efficiency than attempting to parallelize an individual descent.
- **Genetic Algorithm (GA):** Initial sets of parameter values are sampled randomly and a number of candidates that perform the best are selected. These best performers are then "bred" by randomly exchanging parameter values and "mutated" by perturbing parameter values, to obtain a new set of parameter values representing a new "generation", iteratively. The implementation optionally allows the best few sets to

enter the next generation unmodified. It also optionally implements annealing to slow down variation over time. This algorithm can be parallelized trivially by evaluating individuals in parallel.

- `Skopt` **Bayesian Optimization (BO):** The `Skopt` library within the `Scikit` Python package provides a Bayesian Optimization function, Simcal implements a thin wrapper around this function. Bayesian Optimization uses an incrementally updated surrogate model for learning the relationship between the "loss function's" (simulator's) inputs and outputs. This model is used to prune the search space and identify promising regions, balancing exploration and exploitation. While the exploration samples input configurations that can potentially improve the accuracy of the surrogate model, the exploitation samples input configurations that are predicted by the model to be high-performing. The wrapper adds time-based cutoffs and utilizes the ask/tell interface provided by `Scikit` to allow for trivial parallelism. It allows for four different base regressor functions: Gaussian process regressor (BO-GP), gradient boosted quantile regressor trees (BO-GBRT), random forest regressor (BO-RF), and extra trees regressor (BO-ET) (all of which are implemented in `Skopt`). Custom regressors are also supported, which is made possible by the `Skopt` implementation.

All random sampling above can be seeded to enforce repeatable and deterministic outcomes.

The main components of Simcal, including a user-implemented simulator to calibrate and the use of a 3rd-party optimization algorithm, are depicted in Figure 4.1

Parameter List

The calibration wrapper must be instantiated for a particular set of parameters that are to be calibrated. Simcal provides a *parameter wrapper*, in the form of a *Parameter* class, which allows the user to specify value range bounds and/or different kinds of range definitions. Instances of this class can be treated seamlessly as a scalar value, or as a formatted string as defined by the user. This duality is needed because simulators have specific requirements and formats for their input (e.g., a bandwidth-defining parameter value may need to be passed to the simulator formatted with a particular unit string suffix such as *10Mbps*). This wrapper also transparently implements normalization of the parameter range, so that, internally, all parameter values are scaled between 0 and 1. Such normalization can be

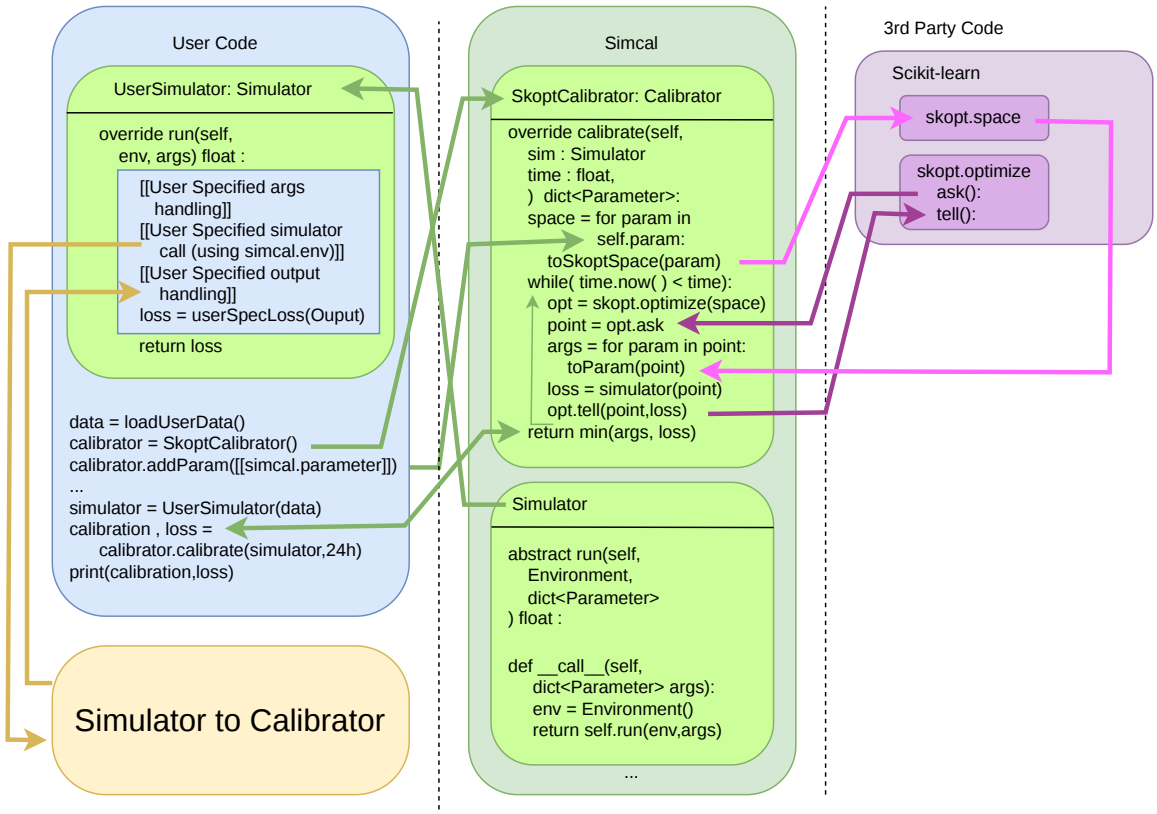


Figure 4.1: Example architecture for Simcal using Skopt. The user's simulator is connected to Simcal through the user's code, and the 3rd party Skopt framework is connected to the users code through Simcal.

beneficial depending on the optimization algorithm used for performing the calibration, but is optional. Specifically, Simcal makes it possible to define parameters of the following types:

- **Categorical:** Categorical parameters represent a collection of fixed input values that are passed to the simulator unmodified, such as a flag that specifies which version of a simulator feature to use. Values for a categorical parameter are treated as unordered and are expected to already be in the form the simulator needs. Categorical parameters cannot be normalized.

- **Ordered:** Ordered parameters are almost the same as categorical parameters, except that they are in a definitive order, e.g., a parameter that defines some discrete level of parallelism would have the three values "Low", "Medium", and "High". As such, normalization can be performed and maps equal-sized ranges to each option in order

(e.g., [0,0.$\overline{3}$) $\mapsto$ "Low",[0.$\overline{3}$,0.$\overline{6}$) $\mapsto$ "Medium", and [0.$\overline{6}$,1] $\mapsto$ "High").

- **Ordinal:** Ordinal parameters are a special case of Ordered parameters where the underlying list of values is of a numeric type. As such, string formatting, as defined by the user, is automatically applied prior to passing parameter values to the simulator as input.

- **Linear:** Linear parameters are a continuous range of values between a start and end point, specified by the user. Values in this range are distributed linearly with respect to the normalization. For instance, in a range 0 to 10 that is normalized to a 0 to 1 range, the value mapped to the normalized 0.5 value is 5. Linear parameters support user-defined string formatters as well as an option to force the parameter value to always be an integer.

- **Exponential:** Exponential parameters are a continuous range of values between a start and end exponent, specified by the user. Values in this range are distributed as for Linear parameters, but then have a base 2 exponential function applied. For instance, in a range 0 to 10 that is normalized to a 0 to 1 range, the value mapped to the normalized 0.5 value is $2^5 = 32$. Exponential parameters are useful for exploring large parameter ranges that span multiple orders of magnitude without overly exploring some portions of the range. For instance, say that a parameter specifies some network bandwidth, and that the user has specified that the parameter value lies between 1Mbps and 10Gbps. In this case, while it may be useful to sample both the 1Mbps and 2Mbps values, it is likely less useful to sample both the 10Gbps and 10.001Gbps values, even though in both cases the two values differ by 1Mbps. Exponential parameters support user-defined string formatters as an option to force the output to be an integer.

Regardless of the parameter type above, user-specified metadata information can be attached to a parameter, which can be useful for invoking the simulator correctly. This metadata is bundled with the formatted parameter values, unmodified, and can be used in any use case-specific manner.

The calibration wrapper takes as input a set of parameter definitions, samples parameter values, and passes them as a dictionary of key-value pairs to the simulator as input. Note that the calibration wrapper can transform the parameters in different ways for using different optimization algorithms. For instance, the `Skopt` Bayesian Optimization framework uses its own parameter list data structure, which the Simcal wrapper builds based on the parameter

set provided by the user. Similarly, `Skopt` produces vector-based output that is transformed back into a dictionary of formatted parameter values prior to being passed to the simulator as input.

4.2.3 Parallelism Wrapper

The calibration process is often parallelizable. The optimization algorithms implemented in Simcal are actually embarrassingly parallel (Section 4.2.2 describes, for each algorithm, how it can be parallelized). This is because multiple parameter values can typically be sampled concurrently.

The parallelization scheme should be specific to the user's available computation platform. To this end, Simcal provides a *parallelism wrapper* to allow calibrators to easily parallelize the calibration process in a transparent manner. We refer to this wrapper as the *coordinator* since it coordinates where and how simulator invocations are executed.

The coordinator provides the expected await/async interface where functions can be dispatched asynchronously and the results can be either gathered when they become available or awaited. The current version of the simulation wrapper includes a trivial, single-threaded coordinator, and a coordinator that uses a fixed-sized thread pool as provided by the `multiprocessing` Python package. The simulation wrapper, however, has been designed with various multi-machine distributed computing platforms in mind, and should thus be easy to extend to these platforms. An important feature of the coordinator is that it puts a bound of the number of outstanding simulator invocations, so as to bound the memory footprint of the calibration process. That is, while the optimization algorithm can generate a possibly infinite stream of points to sample (e.g., the Grid algorithm, the Random algorithm), the coordinator does not simply submit the corresponding simulation invocation to the thread pool, as doing so can quickly exceed the available memory capacity. Instead, the main thread in the calibration wrapper blocks until a simulator invocation has completed. In essence, the wrapper implements its own producer-consumer with bounded buffer implementation as a front-end to the compute platform backend (e.g., a thread pool).

4.2.4 Calibration Evaluation

For our work with DCSim, our collaborator (i.e., the physicist who implemented and uses DCSim) wanted to be able to quantify how "stable" a calibration is based on some statistics computed on other nearby calibrations with similar loss values. To this end, we have added

an evaluation method to Simcal that we call the "Loss Cloud." The Loss Cloud takes a calibration and two loss values. It finds all calibrations that have a lower loss than the first of these two loss values within a hypercube around the calibration using grid search. The size of this hypercube is determined by binary searching each parameter "axis" until a point with approximately the second loss value is found on each axis. All calibrations found within the cloud are output for further analysis. It is up to the user to decide what analysis to perform on this data. For instance a user may wish to quantify the sensitivity of the computed calibration, that is, whether small changes in the parameter values can cause a large increases in loss, which would then denote a more sensitive, less "stable," calibration.

Part II

Evaluation of Calibration Methodology

CHAPTER 5
EXPERIMENTAL CASE STUDY: SCIENTIFIC WORKFLOWS

Scientific workflows and their executions on PDC platforms have supported some of the most significant discoveries of the past decades [6, 29]. In this context, many researchers have used simulation to explore relevant questions, as seen in the large number of works focused on evaluating and/or designing workflow scheduling algorithms [69, 114, 68, 82, 5, 97, 94, 48, 1]. This has motivated the development of simulation frameworks specifically for the purpose of simulating workflow executions on parallel and distributed computing platforms [22, 2, 121].

In this chapter, we apply our automated calibration methodology described in Chapter 4 to a simulator that was designed to simulate workflow executions in a view to compare different scheduling algorithms. Section 5.1 describes our ground-truth data that consists of real-world workflow execution logs. Section 5.2 describes our target simulator, which we call WorkflowSimulator. Section 5.3 explains how we instantiate our automated calibration methodology and describes the obtained results. Section 5.4 discusses the impact of the type and the quantity of the ground-truth data used to perform the calibration. Finally, Section 5.5 summarizes the results and puts them in perspective with a non-automated calibration performed by a user. Part of the content in this chapter has been published in [76].

5.1 Ground-truth Data

The ground-truth data, which is publicly available at [47], was produced using WfCommons tools [23] to generate and execute realistic workflow benchmarks that are based on the structures and the execution logs of real-world workflows. Benchmarks were generated for five different scientific applications, five different workflow sizes (i.e., numbers of tasks), five different amounts of per-task CPU work, and four different total data footprint sizes (i.e., sum of the sizes of all workflow data files). Additionally, benchmark were generated for two synthetic workflow patterns, a "chain" linear task graph and a "forkjoin" fan-out/fan-in task graph, each for three different workflow sizes, five different amounts of per-task CPU work, and three different total data footprint sizes.

These benchmarks were executed with the Pegasus [30] (v5.0.3) / HTCondor [56] (v24.0) Workflow Management System (WMS) on Chameleon Cloud [21]. Pegasus and the HT-

Table 5.1: Workflow specifications used for ground-truth executions logs.

Workflow Application	**Workflow Size (#tasks)**[1]	**Work / Task (sec.)**[2]	**Data Footprint (MB)**[3]
Epigenomics [116] (bioinformatics)	43, 64, 86, 129, 215	0.6, 1.15, 1.73, 7.22, 73.25	0, 150, 1500, 15000
1000Genome [116] (bioinformatics)	54, 81, 108, 162, 270	0.9, 1.47, 2.11, 8.02, 80.94	0, 150, 1500, 15000
SoyKB [116] (bioinformatics)	98, 147, 196, 294, 490	0.53, 1.06, 1.6, 6.55, 74.21	0, 150, 1500, 15000
Montage [116] (astronomy)	60, 90, 120, 180, 300	0.59, 1.12, 1.75, 7.07, 73.13	0, 150, 1500, 15000
Seismology [116] (seismology)	103, 154, 206, 309, 515	0.74, 1.28, 1.91, 8.34, 86.25	0, 150, 1500, 15000
Chain (synthetic)	10, 25, 50	0.83, 1.36, 1.85, 5.74, 48.94	0, 150, 1500
Forkjoin (synthetic)	10, 25, 50	0.84, 1.39, 2.05, 7.61, 70.76	0, 150, 1500

1. For each application the smallest workflow size is the smallest size that can be generated by WfCommons, and the other sizes are approximately 1.5x, 2x, 3x, and 5x larger (WfCommons enforces constraints on workflow size to ensure that the generated task-graphs are representative of the original application).
2. Based on an execution on a single core of a worker node.
3. Given as the sum of the sizes of all the data files used by the workflow, including intermediate files.

Condor Central Manager were installed on a "submit node," and n HTCondor workers were deployed on n "worker nodes" (48-core Intel Icelake processors running Ubuntu 22.04) on which tasks execute. Each benchmark was executed five times for a deployment with $n = 1, 2, 4, 6$ workers (except for the chain benchmark, which only uses $n = 1$ worker). All workflow input data was initially stored on disk at the submit node, and all workflow data was transferred (automatically by Pegasus) between the submit node and the worker nodes until all output data was eventually stored on the submit node. Some benchmark executions with high data footprint and small workflow sizes are not available due to limits imposed on individual file sizes by the runtime systems and/or cloud infrastructure. In total, execution logs were collected from 9,200 workflow executions (i.e., 1,840 different executions, each repeated five times). Table 5.1 gives relevant details on this ground-truth data.

5.2 Simulator Description and Parameters

WorkflowSimulator was developed in C++ using the WRENCH simulation framework [17]. The implementation of the simulator (and of the simulator calibrator used in the next section) is publicly available at [119]. WorkflowSimulator takes as input a workflow specification, as a WfCommons JSON file, and a number of workers. It was designed to simulate workflow executions conducted with Pegasus and HTCondor on Chameleon Cloud, as described in Section 5.1. To do so, it implements particular simulation models for the network, the storage, and the compute infrastructure:

- **Network Infrastructure** – There is no precise information regarding the physical network topology that interconnects the submit node and the workers on Chameleon Cloud besides the fact that all nodes are in the same data-center, and perhaps in the same rack. As a result, WorkflowSimulator implements a commonplace and relative versatile topology composed of a single shared network link out of the submit node and then a dedicated network link to each worker.

- **Storage Infrastructure** – In the real-world platform disks are attached to each node and can be used for storing application data. WorkflowSimulator simulates such storage capability at both the submit node and each worker node, along with software services for reading/writing workflow data on the disks.

- **Compute Infrastructure** – The compute infrastructure consists of compute services on workers, accessed by the WMS (i.e., Pegasus) to execute workflow tasks on workers using HTCondor. WorkflowSimulator simulates this infrastructure by reusing simulation abstractions and models provided by the WRENCH simulation framework specifically for simulating HTCondor deployments.

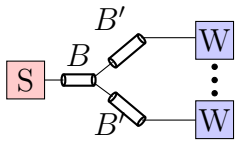
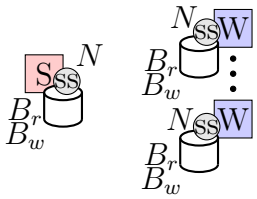
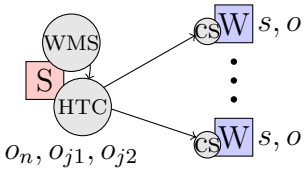
Simulated Network Topology		B: bandwidth B': bandwidth
Simulated Storage System		B_r: read bandwidth B_w: write bandwidth N: max # concurrent disk reads/writes
Simulated Compute System		s: core speed o: task startup overhead o_n: HTCondor negotiator overhead o_{j1}: HTCondor job pre-overhead o_{j2}: HTCondor job post-overhead

Table 5.2: Simulation models implemented in WorkflowSimulator.

Table 5.2 provides a visualization of the above simulation models. These models, as well as the foundational models provided by the WRENCH simulation framework, are configured by many parameters, each of which potentially requires calibration.

5.3 Instantiating the Automated Calibration

In this section, we apply the methodology in Section 4.1 to WorkflowSimulator, computing calibrations using 48 cores of a dedicated Intel Xeon Gold 2.8GHz CPU. Each loss function evaluation by the optimization algorithm entails invoking WorkflowSimulator for each ground-truth data point, which has non-zero computational cost. As optimization is not guaranteed to converge in an acceptable amount of time, the user may wish to allocate a fixed time or iteration budget to the calibration process. After this budget is exceed the best achieved calibration is simply returned. To enable fair comparison of different calibration options we allocate a fixed time budget (as further discussed in Section 5.3.4).

The following sections detail how we apply the guidelines and procedures described in section 4.1

5.3.1 Parameter Selection

While WorkflowSimulator and WRENCH come with hundreds of parameters, we must select which parameters should be calibrated. To do so, we use available knowledge of the system on which the ground-truth data is obtained to prune the parameter set, so as to consider only the most likely relevant parameters. For example, we have limited the number of parameters by assuming that the workers are homogeneous, as specified in the Chameleon Cloud allocations used for obtaining the ground-truth data. That is, the set of workers is described by a few parameters, independent of the number of workers. Another example is the size of all simulated control messages, each of which is described by an individual parameter. However, we know that in practice all these control messages are small and that our platform's network has low latency and high bandwidth, thus justifying using the default 0-byte value for all these control messages as they have negligible impact on the execution. In the end, following this approach, we are left with 10 parameters that are considered for calibration, as listed in Table 5.2.

5.3.2 Parameter Ranges

We intentionally pick broad parameter ranges with upper bounds much larger than, and lower bounds much smaller than the performance of current hardware. As discussed in Section 4.1.3, in practice, a user may have very little knowledge upon which to select parameter ranges, and we expect them to pick such broad ranges. Specifically, we use the following ranges: network and disk bandwidths are 2^x bits per second and core speeds are 2^x ops per second for $20 \leq x \leq 40$, network latencies are between 0ms and 10ms, overheads are between 0s and 20s, and the maximum number of concurrent I/O operations at a disk is between 1 and 100.

5.3.3 Loss Functions and Algorithms

We now need to define candidate loss functions. As explained in Section 3.3, there are typically many options. The ground-truth data contains the *makespan* (i.e., overall execution time) and the execution times of each individual task. For workflow i, let m_i denote the ground-truth makespan, and $\hat{m}_i$ denote the simulated makespan; and for task j in workflow i let $t_{i,j}$ denote the ground-truth execution time, and $\hat{t}_{i,j}$ denote the simulated execution time, of the task. For workflow i, let $e_i = |(m_i - \hat{m}_i)/m_i|$ denote the error on the makespan, and $e_{i,j} = |(t_{i,j} - \hat{t}_{i,j})/t_{i,j}|$ denote the error on a task's execution time. Among the many possible options for defining a loss function we consider:

- $\mathcal{L}_1$: $\text{avg}_i(e_i)$
- $\mathcal{L}_2$: $\max_i(e_i)$
- $\mathcal{L}_3$: $\text{avg}_i(e_i + \text{avg}_j(e_{i,j}))$
- $\mathcal{L}_4$: $\max_i(e_i + \text{avg}_j(e_{i,j}))$
- $\mathcal{L}_5$: $\text{avg}_i(e_i + \max_j(e_{i,j}))$
- $\mathcal{L}_6$: $\max_i(e_i + \max_j(e_{i,j}))$

These loss functions return the average or maximum error. $\mathcal{L}_1$ and $\mathcal{L}_2$ only consider the makespan error, hence do not account for the temporal structure of the execution. $\mathcal{L}_3$ to $\mathcal{L}_6$ combine the makespan error and the task execution time errors in various ways that a user may employ.

For optimization algorithm, we considered RAND, GRID, GRAD, and the 4 BO regressors available in Simcal (see Section 4.2.2). In preliminary experiments, GRID and GRAD performed exceptionally poorly and were excluded from further experiments. Additionally, all Bayesian Optimization regressors produced identical calibrations; as such, only results for BO-GP are reported.

As explained in Section 4.1.2, we use synthetic benchmarking to compare the calibration

Table 5.3: Calibration error vs. algorithm and loss function.

Alg. \ Loss	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$	$\mathcal{L}_6$
RAND	541.24	111.43	610.82	883.40	130.55	883.40
BO-GP	**30.96**	935.10	935.10	89.76	89.76	89.76

error of different loss functions. Table 5.3 shows calibration error for different algorithm / loss function combinations (lower values are better). The calibrations are computed using synthetic ground-truth data generated for all workflows. Overall, BO-GP outperforms RAND. Using BO-GP with the $\mathcal{L}_1$ loss function leads to the best calibration error (even though $\mathcal{L}_1$ is the simplest loss function). Consequently, our methodology leads us to select $\mathcal{L}_1$ as the loss function and BO-GP as the optimization algorithm. This combination of loss function and algorithm is used in all that follows.

5.3.4 Calibration Time Budget

As expected, the longer the time budget the better the calibration and/or the better the use of a larger training dataset. We picked a 24-hour time budget for our experiments, which is sufficient for the loss of the calibration to converge. For instance, Figure 5.1 shows loss vs. time for a particular workflow (similar behavior is seen for other workflows). The loss improves rapidly in the first two hours, and only improves marginally afterwards.

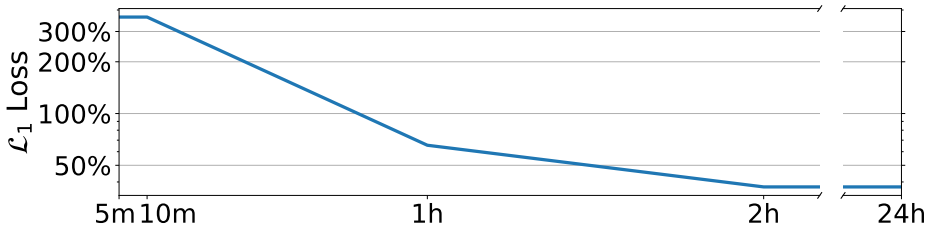


Figure 5.1: Loss value vs. time when using all ground-truth data for the Epigenomics workflow.

5.4 Use of Ground-truth Data

As discussed in Section 4.1.1, a simulator should be calibrated with respect to a sufficiently large and diverse set of ground-truth executions. Obtaining ground-truth data has a cost

as it requires labor, time, and hardware resources. Furthermore, using more ground-truth data increases the evaluation time of the loss function (due to a larger number of simulator invocations). There is thus an incentive to compute calibrations based on only a few and/or small-scale executions. The risk is to overfit to these executions, obtaining a calibration that is "correct for the wrong reasons" and thus non-generalizable to other executions (as discussed in Section 3.1.1).

In this section, we consider several options for selecting ground-truth data using two different schemes: (i) "single-sample" training based on executions for a single number of workers (n) and a single number of tasks (m); and (ii) "rectangular-sample" training based on executions for all numbers of workers $\leq n$ and for all numbers of tasks $\leq m$. Given our available ground-truth data, for a given workflow, this yields 27[1] different options for the training dataset. All options were evaluated against the same validation dataset defined as the "large" executions, i.e., all executions for the largest number of workers and/or the largest number of workflow tasks (excluding executions for the smallest number of workers and the smallest number of tasks). For instance, for the Seismology workflow, ground-truth executions are for 1, 2, 4, and 6 workers; and for 103, 154, 206, 309, and 515 tasks. Our testing dataset comprises all executions on 6 workers with 154 or more tasks, and all executions with 515 tasks on 2 or more workers. This dataset was selected because simulation is often used to explore larger scenarios that ground-truth data can be collected for. We have constructed "lossmaps" (i.e., loss value heat maps) for each workflow showing the evaluation loss for all 27 options for all workflows. Figure 5.2 shows the lossmap for the Seismology workflow as an example.

An interesting observation from these results is that using larger training datasets (in the number of data points, as in the rectangular-sample scheme, or in the scale of the executions for these data points) can be detrimental. This may seem counter-intuitive, but is explained as follows. A simulation for a single number of workers and single number of tasks does exert all components of the simulated systems with many simulated task executions, data transfers, and I/O operations. The executions for other numbers of workers and/or tasks are not necessarily qualitatively different, in which case there is little new behavior to learn. Including these executions leads to an unnecessarily large training dataset. This is detrimental because the loss function evaluation involves simulating the execution of all

[1]The training data options for each workflow consist of all data for 3 values of node count, and 4 values of task count, as well as 2 additional extreme points, for a total of $3 \times 4 + 2 = 14$. The 2 schemes for using this data brings the total to 28, however the smallest task count on 1 node option is identical under both schemes, and thus is only considered once.

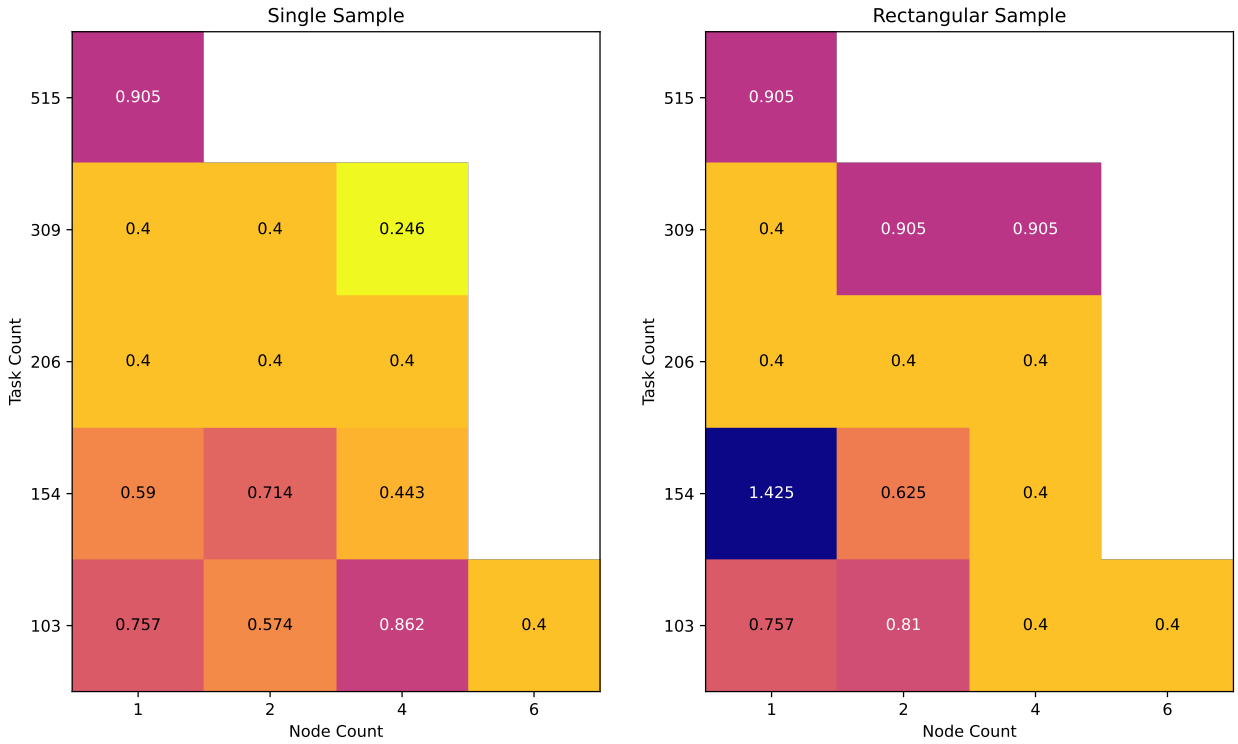


Figure 5.2: Training dataset vs. loss for the Seismology workflow. Left-hand side results correspond to the loss for calibrations computed with single-sample training datasets (the loss value indicated at point (n, m) is for training with the dataset that includes all executions with n compute nodes for m tasks). Right-hand side results correspond to calibrations computed with rectangular dataset (the loss value indicated at point (n, m) is for training with the dataset that includes all executions with $\leq n$ compute nodes for $\leq m$ tasks).

workflows in the training dataset. For a given time budget, the optimization algorithm will thus perform fewer iterations and may produce a worse calibration. Overall, the results in Figure 5.2 show that for the Seismology workflow, using rectangular datasets brings at most occasional benefits over using single-sample datasets, and often leads to higher loss. Similar observations holds for the other workflows. For Seismology in particular, the lowest loss is achieved when calibrating based on the single-sample dataset for $n = 4$ workers (the second largest n in our ground-truth data) and $m = 309$ tasks (the second largest m value in our ground-truth data). For other workflows, other single-sample datasets may achieve the lowest loss (typically for $n > 1$ workers).

Additionally, obtaining the ground-truth data for a training dataset has a resource cost, with higher cost for more workflow executions, more workers, and/or longer workflow executions. Figure 5.3 shows results as a scatter plot where the horizontal axis is the achieved

loss and the vertical axis is the cost of obtaining the training data, for all workflows. We measure the cost of obtaining the ground-truth data for a particular training dataset as the sum, over all included workflow executions, of the number of workers used multiplied by the makespan of the execution in seconds.

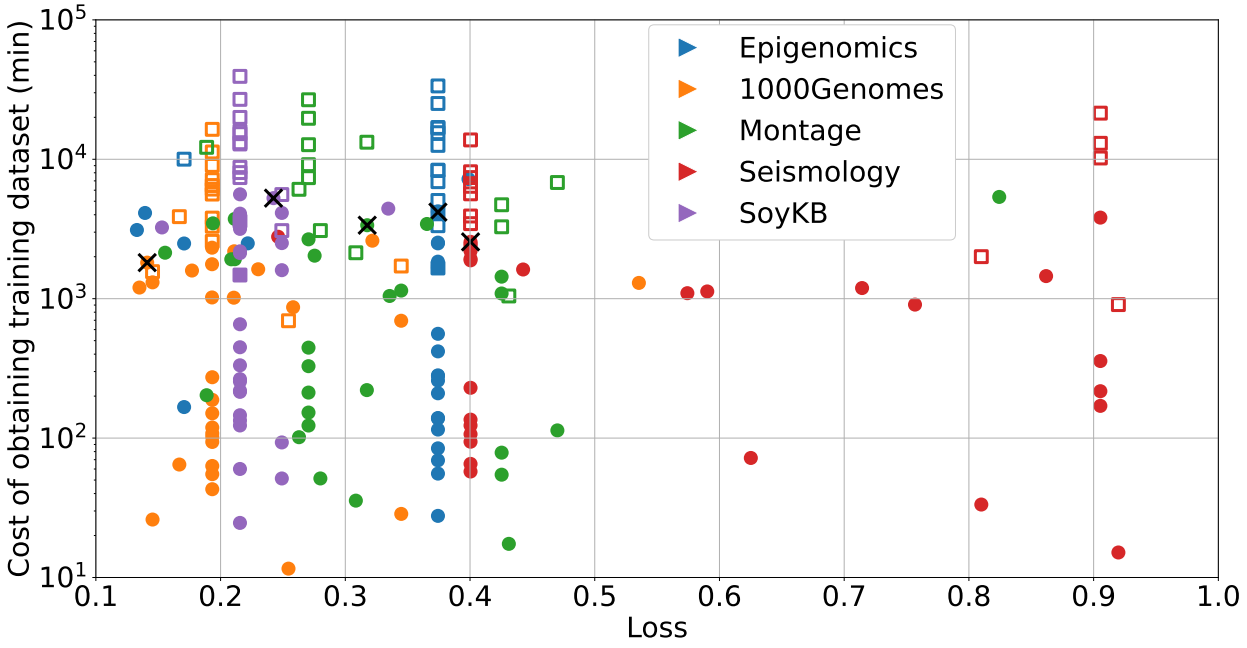


Figure 5.3: Training dataset cost vs. loss, for each workflow. For each workflow, data points are shown for the single-sample scheme as filled circles and for the rectangular-sample scheme as unfilled squares. Crosses indicate the training datasets used in the previous section.

From the results in Figure 5.3, we can see that using the cheapest (running the smallest workflow configuration on a single worker), and thus necessarily least diverse, training datasets is a poor choice. Save for the SoyKB workflow, the cheapest training dataset for each workflow in Figure 5.3 is among the ones with the highest loss, but there is a large number of low-cost datasets that all perform similarly.

Overall, we conclude that, for this case study, a relatively cheap training dataset that only includes executions for a single number of workers and for a single number of tasks is sufficient to obtain a good calibration.

A way to further reduce the training dataset cost is to, for each worker count and task count, execute fewer workflow configurations. As seen in Table 5.1, our ground-truth data spans a range of sequential work and data footprint values. We computed calibrations using only executions for one sequential work value and one data footprint value. In more than 98% of the cases the resulting loss on the validation dataset becomes significantly

larger, sometimes up to an order of magnitude. As expected, the worst results are when the training dataset only includes executions with zero sequential work and/or zero data footprint, meaning that some components of the system are never simulated. We conclude that the training dataset must include executions with diverse data to compute volume ratios. Otherwise, calibrations are overfit to a single such ratio and thus non-generalizable.

Yet another way to reduce the training dataset cost is to employ simple benchmarks, such as the chain or forkjoin workflows described in Section 5.1. We computed calibrations with a training dataset that contains only chain executions, forkjoin executions, or both. The ground-truth data for real-world workflows is used as the validation dataset. Using only chain executions increases the loss by more than one order of magnitude due to the training dataset not including any parallel task executions. Using only forkjoin executions leads to loss increases between 1.2x and 3.5x depending on the workflow. Computing calibrations based on both chain and forkjoin executions leads to worse results, due to the loss function evaluation being more costly. Overall, the use of simple benchmark ground-truth data for calibrating simulations of real-world applications is attractive but has a non-negligible negative impact on accuracy, at least within the scope of this case study.

5.5 Conclusion

This simulator achieves error around 23% and relatively low variance across workflows. One source of simulation inaccuracy is that the simulator does not reproduce the ground-truth task schedules exactly. To do so, the simulator would have to implement the exact same scheduling algorithms as that in Pegasus and HTCondor, and use the exact same data structures, which is not implemented in WorkflowSimulator.

Another source of simulation inaccuracy is that ground-truth data collected from real-world PDC systems is necessarily noisy [26, 70]. Specifically, in our ground-truth data we observe non-negligible variance among repeats of a single workflow execution (recall that our ground-truth data includes 5 repeats for each execution). Because WorkflowSimulator is deterministic by design, there is thus unavoidable simulation error, as already discussed in Section 3.2. Let us consider an ideal, hypothetical simulator that when simulating a particular workflow execution, always yields a makespan equal to the average of the makespans across all repeats of that same execution in the ground-truth data. Based on our available ground-truth data, this ideal simulator would achieve error around 14% across all workflows. We conclude that a 23% error, as achieved by WorkflowSimulator when calibrated

using our approach, is reasonably close to the ideal case.

To assess the value added by our automated calibration methodology, we consider an approach in which one would set all parameter values based on available hardware specifications and benchmarks (i.e., documented on the Chameleon Cloud website and/or obtained directly on the nodes). This is likely in line with what authors do when they do not mention calibration (e.g., as in more than 70% of the publications reviewed in Figure 3.5.3). Using this approach, the average percent relative error between simulated and ground-truth makespans ranges from 58% to 106% over the five workflow applications, and the average over all five is 73%. This error is significantly higher than that of our automatically calibrated simulator.

CHAPTER 6
EXPERIMENTAL CASE STUDY: MPI APPLICATIONS

A large class of scientific applications are implemented as distributed-memory, message-passing parallel programs that are to be executed on High Performance Computing (HPC) clusters. The Message Passing Interface (MPI) [80] standardizes the development of these applications. Many implementations of MPI are available, that use different approaches and algorithms [89]. A crucial question is: which approaches, algorithms, and ways to use them yield the best performance under which conditions? As a result, it is not surprising that many researchers have used simulation to investigate the performance of MPI applications. This has motivated the development of simulation frameworks specifically designed for simulating the execution of MPI applications [59, 34, 52, 25, 12, 31].

In this chapter, we apply our automated calibration methodology described in Chapter 4 to a simulator that was designed to simulate MPI application executions on a leadership-class supercomputer in a view to making platform provisioning decisions (for the interconnect). Section 6.1 describes our ground-truth data that consists of real-world MPI application execution logs. Section 6.2 describes our target simulator, which we call MPISimulator. Section 6.3 explains how we instantiate our automated calibration methodology and describes the obtained results. Section 6.4 discusses the impact of the type and the quantity of the ground-truth data used to perform the calibration. Finally, Section 6.5 summarizes the results and puts them in perspective with a non-automated calibration performed by a user. Part of the content in this chapter has been published in [76].

6.1 Ground-truth Data

The ground-truth data that was made available to us, and which is now publicly available at [47], is from runs of the Intel MPI Benchmarks (IMB) [58] executed on the pre-exascale Summit leadership-class supercomputer at the Oak Ridge National Laboratory. The IMB measures the performance of various MPI point-to-point communication functions and patterns for ranges of message sizes. Specifically, the collected ground-truth data consists of execution logs of the PingPing, PingPong, BiRandom, and Stencil IMB benchmarks with 2^x-byte messages, for $x \in \{10, 11, \ldots, 22\}$, on 128, 256, and 512 compute nodes.

Summit is an IBM system with ∼4,600 compute nodes, each equipped with two IBM

POWER9 processors, for a total of 42 CPU cores, and with six NVIDIA Tesla V100 GPUs. Summit's interconnect uses a non-blocking three-level Fat-Tree topology: Level one switches connect 18 nodes in each cabinet along with 18 director switches comprising 36 level two and 18 level three switches to connect cabinets together. The switches in this network have a limit to the total bandwidth they can provide to all hosts in addition to a per host bandwidth. On platforms like Summit, the bulk of the performance for current applications comes from the use of the multiple GPUs at each node. The usage of CPUs is often limited to the control of the execution flow of the application, to send and retrieve data to and from GPUs, and to manage inter-node data exchanges using MPI. On Summit, with six GPUs per node, often only six out the 42 available CPU cores are used. The ground-truth executions match this practice by running the IMB with six MPI ranks per node.

6.2 Simulator Description and Parameters

MPISimulator was developed in C++ using the SMPI [31] simulation framework, which makes it possible to compile and execute unmodified MPI programs on simulated "hardware". The implementation of the simulator and of the simulator calibrator is available at [81]. SMPI takes as input a specification of a simulated hardware platform and executes the MPI application code as is, with each MPI rank executing as a thread. Each MPI call is intercepted and its duration is simulated, while each block of code between two calls is timed to simulate computation delays. Using SMPI as a foundation, MPISimulator implements particular simulation models for the network infrastructure, the compute infrastructure, and MPI's adaptive protocol behavior:

- **Network Infrastructure** – Unlike in the previous chapter, the network topology of the platform used to obtain the ground-truth data is thoroughly documented. As such, MPISimulator uses the three-level Fat-Tree network topology described in Section 6.1.

- **Compute Infrastructure** – The compute infrastructure consists of several compute nodes. Each node has two CPUs that can communicate via an X-Bus SMP bus, and are each connected to a shared Network Interface Card (NIC) via a PCIe bus, which is similar to the real-world architecture of the Summit compute nodes.

- **MPI Adaptive Protocol** – MPI implementations use different communication protocols for different message sizes for performance reasons (e.g., switching from eager

to rendezvous mode), which can be modeled as a multiplicative factor applied to data transfer rates [31]. Results in [31] indicate two threshold message sizes, or change points, above which the multiplicative factor changes.

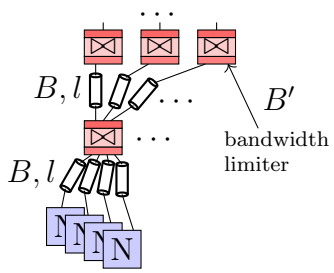
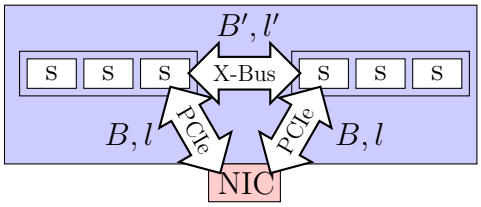
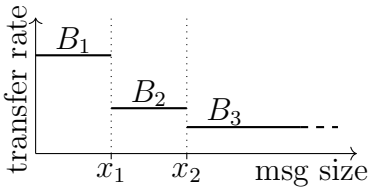
Simulated Network Topology		B: bandwidth l: latency B': bandwidth
Simulated Compute Node		s: core speed B: bandwidth l: latency B': bandwidth l': latency
Simulated Adaptive Protocol		B_1: bandwidth factor B_2: bandwidth factor B_3: bandwidth factor x_1: message size x_2: message size

Table 6.1: Simulation models implemented in MPISimulator.

Table 6.1 provides a visualization of the above simulation models. These models, as well as the foundational models provided by the SMPI simulation framework, are configured by many parameters, each of which potentially requires calibration.

6.3 Instantiating the Automated Calibration

In this section, we apply the methodology in Section 4.1 to the MPISimulator. Calibrations are computed using 48 cores of a dedicated Intel Xeon Gold 2.8GHz CPU. As for the case study in the previous chapter, to enable fair comparison of different calibration options, we allocate a fixed time budget to the calibration process.

6.3.1 Parameter Selection

While MPISimulator and SMPI come with hundreds of parameters, we must select which parameters should be calibrated. To do so, we use available knowledge of the system on which the ground-truth data is obtained to prune the parameter set, so as to consider only the most likely relevant parameters. For example, we have limited the number of parameters by assuming that the workers are homogeneous, as specified in the Summit allocations used for obtaining the ground-truth data. Additionally SMPI has many internal parameters. These parameters were independently calibrated by the developers of SMPI based on multiple commodity clusters [99]. We assume these default parameter values are accurate and do not include them as parameters to calibrate. In the end, following this approach, we are left we 13 parameters that are considered in the calibration, as listed in Table 6.1.

6.3.2 Parameter Ranges

We use broad parameter ranges: for all bandwidth, latencies, and compute speeds, we use min values that are at least one order of magnitude lower than, and max values that are at least one order of magnitude higher than the actual hardware specification of Summit, as listed in Table 6.2.

Table 6.2: Calibration parameter ranges

Parameter	Minimum	Maximum
Compute Node Parameters		
CPU speed (s)	20Gf	100Gf
PCIE Bandwidth (B)	1GBps	160GBps
pcie latency (l)	1ns	1ms
xbus bandwidth (B')	20Gf	100Gf
xbus latency (l')	20Gf	100Gf
Network Link Parameters		
Tree Link Bandwidth (B)	1Gbps	12800Gbps
Host Link Bandwidth (B')	1Gbps	100Gbps
Link Latency (l)	10ns	1ms
MPI Adaption Parameters		
Bandwidth Factor (B_1, B_2, B_3)	0.2x	1x
Protocol Change Points (x_1, x_2)	1kB	4MB

6.3.3 Loss Functions and Algorithms

We now need to define candidate loss functions. As explained in Section 3.3, and as in the previous chapter, there are often many options. For each benchmark and for a given message size, our ground-truth data consists of data transfer rate measurements, or samples, for multiple repeated executions. Due to platform noise, there is variance in these samples. Our simulator, instead, produces a single data transfer rate value since SMPI simulations are deterministic and repeatable by design. We need to quantify how representative this simulated value is of the set of measured samples. To do so we use the *explained variance*, which is defined as a/b, where a is the L1 distance between the samples and the model value, and b is the L1 distance between the measured samples and their mean. The lower (i.e., closer to 1) the explained variance, the more closely the simulated value matches the measured sample. Let $ev_{i,j}$ denote the explained variance between the ground-truth data transfer rates and the simulated data transfer rate for benchmark i executed with message size j. We consider four loss functions defined based on four different ways to aggregate the explained variance:

- $\mathcal{L}_1$: $\text{avg}_i(\text{avg}_j(ev_{i,j})$
- $\mathcal{L}_2$: $\text{avg}_i(\max_j(ev_{i,j})$
- $\mathcal{L}_3$: $\max_i(\text{avg}_j(ev_{i,j}))$
- $\mathcal{L}_4$: $\max_i(\max_j(ev_{i,j}))$

For optimization algorithm, we considered RAND, GRID, GRAD, and the 4 BO regressors available in Simcal (see Section 4.2.2). In preliminary experiments, GRID and GRAD performed exceptionally poorly and were excluded from further experiments. Additionally, all Bayesian Optimization regressors produced identical calibrations; as such, only results for BO-GP are reported.

Table 6.3: Calibration error vs. algorithm and loss function.

Alg. \ Loss	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$
RAND	60.96	58.17	43.80	70.74
BO-GP	**1.87**	126.65	31.58	47.11

As explained in Section 4.1.2, we use synthetic benchmarking to compare the calibration error of different loss functions. Table 6.3 shows calibration error for different algorithm/loss combinations (lower values are better). The calibrations are computed using synthetic ground-truth data generated for all benchmarks. Overall, the best combination, which we use in all that follows, is the BO-GP algorithm with the $\mathcal{L}_1$ loss function. This

is similar to our finding in the previous chapter (i.e., use Bayesian Optimization with a loss function that accounts for average error).

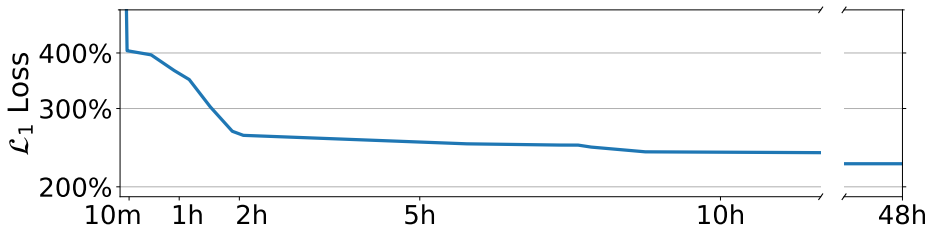


Figure 6.1: Loss value vs. time when using all ground-truth data for 128 compute nodes.

6.3.4 Calibration Time Budget

As expected, the longer the time budget the better the calibration and/or the better the use of a larger training dataset. We used a 48-hour time budget for our experiments, which is sufficient for the loss of the calibration to converge. For instance, Figure 6.1 shows loss vs. time for all benchmarks. The loss improves rapidly in the first two hours, and only improves marginally afterwards.

6.4 Use of Ground-truth Data

As discussed in Section 4.1.1, a simulator should be calibrated with respect to a sufficiently large and diverse set of ground-truth executions. As in the previous chapter, we explore the impact of using different training datasets for computing calibrations, to determine the extent to which a calibrated simulator produces results that generalizes beyond the ground-truth data. We consider generalization to (i) different benchmark types and (ii) different execution scales. First, we simulate the 128-node execution of the Stencil application benchmark using a calibration computed based on the 128-node executions of the BiRandom, PingPing, and PingPong benchmarks. We find that simulated execution achieves a relative error (averaged over all message sizes) of 58.8%. By contrast, using a calibration based on the Stencil ground-truth data, achieves an error of 28.6%. Second, we simulate the 256- and 512-node execution of each benchmark using a calibration computed based on their 128-node executions. We observe significant increases in simulation error for all benchmarks. For instance, for the BiRandom benchmark, while error averaged over all message sizes is 15.2% when simulating

128-node executions, when simulating 256- and 512-node executions the error is 30.8% and 59.4%, respectively.

These results show that the calibrated simulator does not generalize well beyond the ground-truth data. It may still be useful for some purposes (e.g., studying the impact of the network link bandwidth on particular benchmark executions at particular scales). But overall, this is a negative result for this simulator: it cannot fulfill its intended purpose of studying MPI performance scaling for the target HPC cluster, even for simple point-to-point communications. Perhaps MPISimulator does not implement the simulation at a sufficiently high level of detail, or perhaps more parameters should be considered for calibration. In this particular case, however, based on discussion with our collaborators at the Oak Ridge National Laboratory where the ground-truth data was collected, we suspect that information on how the ground-truth data was obtained is incomplete and/or inaccurate, leading to simulated executions that are qualitatively different from ground-truth executions. Regardless, this is a positive result for our methodology: automatically calibrating MPISimulator to the best of its ability with respect to the available ground-truth data makes it possible to systematically evaluate its intrinsic accuracy, and to reach the above (negative) result.

6.5 Conclusion

The results in the previous section show the use of our methodology to "narrow down" the sources of inaccuracy by removing parameter calibration uncertainty from the equation. That is, if simulation error remains high even after automated calibration of the parameters, then one can conclude that further tuning of the values of these parameters is unlikely to yield significant improvements. Instead, one should pursue different avenues, including investigating alternate loss functions, calibrating more parameters, obtaining different ground-truth data, or increase the level of detail of the simulation.

The calibration was successful in finding an accurate simulator for the available ground-truth data, even though this accuracy does not generalize beyond that ground-truth data. The most accurate simulators achieve errors below 15%, with relatively low variance across all benchmarks. As previously discussed in Section 3.2, an unavoidable source of simulation inaccuracy is that the ground-truth data is noisy while the simulator is deterministic. As in the previous chapter, we consider an ideal, hypothetical simulator that when simulating a particular workflow execution, always yields a data transfer rate equal to the average of the data transfer rates across all repeats of that same benchmark execution in the ground-

truth data. Based on our available ground-truth data, this ideal simulator would achieve error around 8.6% across all workflows. We conclude that a 16% error, as achieved by MPISimulator when calibrated using our approach, is reasonably close to the ideal case.

As in the previous chapter, we have performed a straightforward calibration that a user may do based on Summit's specifications. The average percent relative error between simulated and ground-truth data transfer rates ranges from 91% to 97% over the three benchmarks. This is a similar improvement magnitude as what was observed for the case study in the previous chapter.

CHAPTER 7
EXPERIMENTAL CASE STUDY: HIGH ENERGY PHYSICS APPLICATION

Distributed computing platforms are used to support the enormous compute and storage demands of many High Energy Physics (HEP) applications, such as for processing data generated by the Large Hadron Collider (LHC) experiments. These experiments generate large amounts of data. For example, the LHC experiments conducted for the Compact Muon Solenoid (CMS) collaboration [108], generate ∼30PB of data per year, which requires ∼270 million CPU-hours annually to process. The generated data, which describes particle collision events, can be split into chunks that can be processed and stored independently of each other. The processing of one event entails multiple data reduction/transformation steps until a final analysis step produces an output that can be stored on a single computer and analyzed to generate scientific results. This data processing workload is performed on a multi-site distributed computing platform, the Worldwide LHC Computing Grid (WLCG) [117], using various software infrastructures, such as HTCondor [56] for distributing computation and XRootD [123] for distributing storage. Researchers need to estimate the execution times of current HEP workloads of interest on (subsets of) WLCG to plan experiments, to explore various hardware resource provisioning options, and to ensure that future CMS workloads can achieve acceptable performance. Achieving this objective via real-world experiments would be prohibitively resource-consuming, especially since WLCG is used daily in production for running mission-critical workloads. Furthermore, real-world experiments cannot be used to explore hypothetical (future) scenarios. As a result, the objective can only be achieved by conducting simulation experiments.

In this chapter, we apply our automated calibration methodology described in Chapter 4 to DCSim [54], a simulator that was developed by HEP researchers at the Karlsruhe Institute of Technology to simulate the execution of HEP Workloads on WLCG. DCSim is intended to explore the effect of various caching policies and hardware configurations to inform future upgrades to WLCG sites. Section 7.1 describes our ground-truth data obtained from real-world execution logs. Section 7.2 describes DCSim. Section 7.3 explains how we instantiate our automated calibration methodology and presents the obtained results. Section 7.4 discusses the impact of the type and the quantity of the ground-truth data used to perform the calibration. Finally, Section 7.5 summarizes the results and puts them in perspective with a

non-automated calibration performed by a physicist (who is also the developer of DCSim). A preliminary version of the work in this chapter has been published in [74].

7.1 Ground-truth Data

The ground-truth data was obtained with a workload comprised of 48 jobs, where each job takes 20 files as input, each of size of ∼427MB, and produces only a few bytes of output. This workload was executed on WLCG using one compute site and a remote storage size, interconnected together via a wide-area network. The compute site hosts three compute nodes that are homogeneous with one exception: two of the compute nodes have 12 cores while the third one has 24 cores. All three nodes host a local HDD cache, and are connected together via a local network. This platform configuration is depicted in Figure 7.1. The workload was executed for ICD (Initially Cached Data) values ranging from 0 to 1 in 0.1 increments. The ICD denotes the fraction of input files that are initially stored in local caches. There are 3 datasets with different specifications: *copy* jobs, jobs that use 0% of the CPU and only perform file copies; *test* jobs, where the CPU time is moderate with respect to the I/O time; and *slow* jobs, where the CPU time is high with respect to the I/O time. In this case study, we reserve the *slow* jobs for validation and compute calibrations based on the *copy* and *test* jobs.

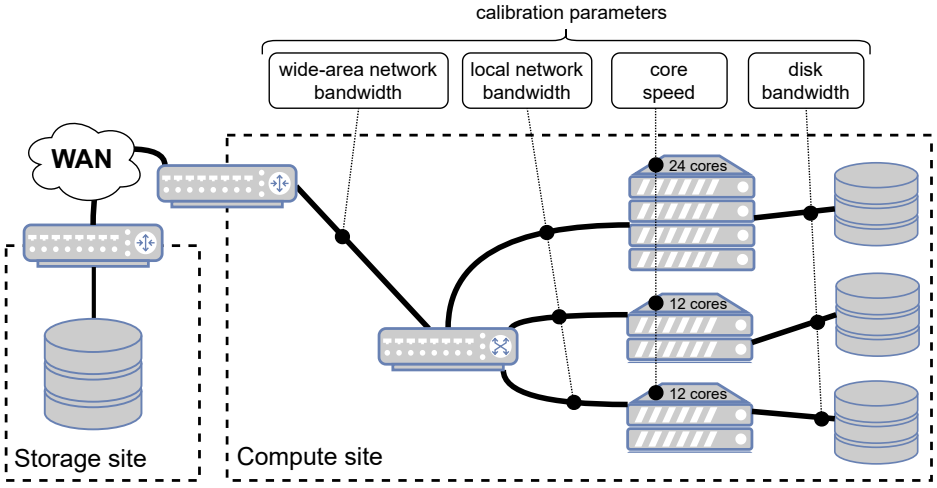


Figure 7.1: Execution platform

Each execution of the workload on the platform was conducted for each of four different configurations of the hardware platform. Specifically, two different network interfaces can be used for the compute site to connect to the remote storage site (1 Gbps or 10 Gbps), and at each compute node the HDD can be used as the cache, or a RAM disk can be used as

Table 7.1: Hardware Platform Configuration Specifications.

Platform	**Cache location**	**WAN interface**
SCFN	HDD	10 Gbps
FCFN	RAM	10 Gbps
SCSN	HDD	1 Gbps
FCSN	RAM	1 Gbps

the cache. These platform configurations are summarized in Table 7.1 (FC and SC stand for Fast Cache and Slow cache, respectively, and FN and SN stand for Fast Network and Slow Network, respectively).

The dataset, which has not been made publicly available but was provided to us by HEP researchers at the Karlsruhe Institute of Technology, reports the makespan of the whole execution, as well as the CPU time and runtime for each individual job. Unfortunately, this dataset is incomplete and contains a few aberrant data points that had to be filtered out. As a result, there is not necessarily sufficient usable data for each of the four combination shown in Table 7.1. We discuss the limitations of the dataset further throughout the rest of this chapter.

7.2 Simulator Description and Parameters

DCSim was developed in C++ using the WRENCH simulation framework [17] by HEP researchers at the Karlsruhe Institute of Technology. The implementation of the simulator is available at [28]. DCSim takes as input a description of a workload to execute and of the WLCG platform on which to execute it. A *workload* consists of a set of independent jobs, where each job consists in reading input files of given sizes, performing some volume of computation per byte of input, and writing an output file of a given size. The user can specify data and compute volumes either as constant values or as probability distributions from which values are sampled. A WLCG platform consists of multiple sites interconnected over a wide-area network. One or more of these sites host a storage service that stores all initial input data for all jobs. Each site comprises multi-core compute nodes, each of which can use its local disk to cache input data. Finally the simulator takes as input a number between 0 and 1, for the ICD. The simulator produces as output a sequence of time-stamped execution events. In particular, with this output one can compute individual job execution times and the overall makespan.

7.3 Instantiating the Automated Calibration

In this section, we apply the methodology described in Section 4.1 to the DCSim simulator. Calibrations are computed using 48 cores of a dedicated Intel Xeon Gold 2.8GHz CPU. As in the previous chapters, to enable fair comparison of different calibration options, we allocate a fixed time budget to the calibration process.

7.3.1 Parameter Selection

Like the simulators described in Chapters 5 and 6, DCSim has hundreds of parameters, some inherited from WRENCH, and we must select which parameters should be calibrated. To do so, we use available knowledge of the system on which the ground-truth data is obtained to prune the parameter set, so as to consider only the most likely relevant parameters. We make the same base assumptions as Chapter 5 such as homogeneity and negligible impact of control messages. But since one compute node is known to be different from the other two, it has its own CPU speed parameter. There are also different configurations for the platform, which in effect leads to four different platforms (see Table 7.1). These platforms share some common components and include some specific components. Because of these shared common components, we opted to calibrate all the platforms simultaneously using additional parameters for those components that are unique to a platform. That is, we define a parameter for each unique component in the union of the platforms, and calibrate all platforms together. Specifically, each execution scenario is simulated four times, once for each of the four platforms, simulating only the components that are relevant to each specific platform. As the only network activity is large file transfers on a well provisioned local-area network, the network latencies are assumed to be negligible and are not calibrated. Following this approach, we are left we 7 natural hardware parameters that must be considered for calibration. Examining the ground-truth data reveals that the XRootD storage system leads to a certain amount of compute overhead when accessing data. While this is not necessarily surprising, this overhead is different depending on whether the data is retrieved from a local cache or from a remote server. This leads to two additional parameters for these overheads, for a total of 9 parameters to calibrate.

7.3.2 Parameter Ranges

We initially picked broad parameter ranges with upper bounds being much larger than, and lower bounds being much smaller than the performance of current hardware. As discussed

in Section 4.1.3, in practice, a user may have very little knowledge upon which to select parameter ranges, and we expect them to pick broad ranges. However, in this case study, after multiple rounds of calibrations for different scenarios and based on a preliminary case study [74], we narrowed the parameter ranges significantly. Table 7.2 summarizes the parameters and ranges used. All parameters use exponential distributions (as described in Section 4.2.2).

Table 7.2: Calibration parameter ranges

Parameter	Minimum	Maximum	Platforms
Compute Node Parameters			
Homogeneous CPU core speed	500Mflop/sec	5000Mflop/sec	All
Other CPU core speed	500Mflop/sec	5000Mflop/sec	All
Network Link Parameters			
Local network bandwidth	1Gbps	100Gbps	All
Slow WAN network bandwidth	33Mbps	10Gbps	FCSN, SCSN
Fast WAN network bandwidth	250Mbps	10Gbps	FCFN, SCFN
Cache Parameters			
Disk cache bandwidth	8Mbps	67Mbps	SCFN, SCSN
RAM cache bandwidth	67Mbps	34Gbps	FCFN, FCSN
XRootD Overhead Parameters			
XRootD local overhead	1flop	1Gflop	All
XRootD remote overhead	1flop	1Gflop	All

7.3.3 Loss Functions and Algorithms

It may seem like the loss function for this case study should be similar to those used for the case study in Chapter 5, where the simulated makespan is compared to the real-world makespan, or the simulated runtime of each job is compared with the real-world runtime of that same job in the ground-truth data. Specifically in this case study, it is sensible to use both individual job runtimes and individual job CPU time, simulated and real-world, to define a loss function, as distinguishing these two times make it possible to capture the time spent reading input files and the time spent processing the input files. Unfortunately, a limitation of the ground-truth data is that there is some randomness in the runtime of the jobs. As such in both simulated jobs are only statistically similar to real-world jobs.

Thus we cannot form an association in a way that is consistent between the real-world jobs and the simulated jobs. Due to this limitation, the loss function selection problem is significantly more challenging than in the previous case studies. Hereafter we first provide an explanation and justification of each loss function option that we consider, and give more rigorous definitions afterwards.

Loss Functions and Rationales

$\mathcal{L}_1$ **Makespan** – The time from start to finish of the entire workload execution. This is a straightforward loss function to calculate but it does not consider any data about individual jobs.

$\mathcal{L}_2$ **Chamfer** – The Chamfer metric was originally created for the purpose of image comparison [8]. It takes a set of unordered points and computes the distance between nearest neighbors. Calculating this metric using the runtime and CPU time, for all jobs executed on a machine in a particular test case, should quantify how similar the simulated job executions are to the ground-truth job executions. As it does not rely on ordered points in either set, this loss function can be used without having to match a simulated job to its real-world counterpart.

$\mathcal{L}_3$ **ddKS** – KS tests are typically used to determine the probability that two samples were drawn from the same underlying probability distribution [60]. While KS tests only work on single dimensional data, the ddKS test extends KS tests to handle multi-dimensional data [49]. Calculating the probability that the ground-truth job executions and simulated job executions are from the same distribution, is potentially a good way to determine the discrepancy between simulated and ground-truth executions. We compute this loss function on the set of simulated and real-world jobs for the runtime and the CPU time (hence the need to handle multidimensional data).

$\mathcal{L}_4$ **DoubleSorted** – A simple way to compare two datasets is to sort them both. This allows low values to be compared to low values and high values to be compared to high values. However, two-dimensional data cannot be sorted easily. If one dimension is used, the other dimension will be un-ordered. We define the DoubleSorted loss function as follows. It sorts the job runtimes and job CPU times separately; it then simply calculates the Mean Relative Error (MRE) of both independently and combines the result. Naturally the weakness of this approach is that it is likely that the runtime

and CPU time of a single ground-truth job are being compared with the runtime and CPU time of two different simulated jobs.

$\mathcal{L}_5$ **Hausdorff** – The directed Hausdorff metric calculates the distance between two objects as the "largest smallest" distance between them [87]. For our data, this means computing the distance of the (runtime, CPU time) point in the ground-truth executions that is the farthest away from the (runtime, CPU time) point in the simulated executions. This loss function handles multidimentional data by design, since it is simply a euclidean distance in a vector space.

$\mathcal{L}_6$ **MREa** – This loss function is a "blunt instrument" similar to makespan. Since the real-world and simulated jobs cannot be matched to each other, this loss function simply considers all jobs on a machine and computes the average job run time. It then computes the MRE of those averages across all samples. This loss function ignores the CPU times.

$\mathcal{L}_7$ **SumMREa** – This loss function is similar to MREa, but it adds the average job runtime to the average job CPU time prior to calculating the MRE.

$\mathcal{L}_8$ **Sorted** – This loss function seeks to address the potential issue in the above Double-Sorted loss function, where one ground-truth job execution may be compared to the executions of two different simulated jobs. Rather than sorting each list independently, it sorts only on runtime, then calculates the MRE of both runtime and CPU time using this sort. This means that each ground-truth job is only being compared to one simulated job, with the caveat that the CPU times are no longer sorted.

Loss Function Definitions

Prior to giving formal definitions of our loss functions, we first establish some general definitions. Let d denote all ground-truth execution data, and $\hat{d}$ denote all simulated execution data. Both d and $\hat{d}$ have the same structure, therefore let $\mathbb{D} = \{d, \hat{d}\}$. For $D \in \mathbb{D}$, let us define $D_t \in D$ to be all data in D relating to a given test case t. Let $(D_t)^m$ be the makespan of D_t. For machine m in D_t, let $D_{t,m}$ denote all data relating to jobs that executed on m in test case t. Let $D_{t,m,i}$ denote job i within $D_{t,m}$. Let $(D_{t,m,i})^r$ refer to the runtime of $D_{t,m,i}$ and $(D_{t,m,i})^c$ denote the CPU time of $D_{t,m,i}$. For example, $(\hat{d}_{t,m,i})^r$ refers to the runtime of a job within the simulated execution data, and $(d_{t,m,i})^c$ refers to the CPU time of a job within the ground-truth execution data. Recall that jobs in the ground-truth and simulated

execution data cannot be easily matched to each other, thus for any given i, $(d_{t,m,i})^r$ and $(\hat{d}_{t,m,i})^r$ may not be directly comparable. We use $\text{sort}^x(D_{t,m}) \hookrightarrow S_{t,m}$ to denote a sequence $S_{t,m}$ where $\forall S_{t,m,i}, S_{t,m,j} \in S_{t,m}$, if $i > j$ then $(S_{t,m,i})^x \geq (S_{t,m,j})^x$ and $S_{t,m,i}, S_{t,m,j} \in D_{t,m}$ (where x is either c or r). Thus $\text{sort}^r(\hat{d}_{t,m})$ denotes sorting all jobs within a machine m and test case t in the simulated data by their runtimes.

Using these definitions and external functions for Chamfer distance, ddKS, and directed Hausdorff distance, we can now formally define the loss functions used in this case study.

$\mathcal{L}_1$ (Makespan):
$\text{avg}_t \left(|\frac{(d_t)^m - (\hat{d}_t)^m}{(d_t)^m}| \right)$

$\mathcal{L}_2$ (Chamfer):
$\text{avg}_{t,m} \left(\text{ChamferDistance}(d_{t,m}, \hat{d}_{t,m}) \right)$

$\mathcal{L}_3$ (ddKS):
$\text{avg}_{t,m} \left(\text{ddKS}(d_{t,m}, \hat{d}_{t,m}) \right)$

$\mathcal{L}_4$ (DoubleSorted):
$\text{avg}_{t,m} \left(\text{avg}_i(|(\text{sort}^r(d_{t,m})_i)^r - (\text{sort}^r(\hat{d}_{t,m})_i)^r|) + \text{avg}_i((|\text{sort}^c(d_{t,m})_i)^c - (\text{sort}^c(\hat{d}_{t,m})_i)^c|) \right)$

$\mathcal{L}_5$ (Hausdorff):
$\text{avg}_{t,m} \left(\text{DirectedHausdorffDistance}(d_{t,m}, \hat{d}_{t,m}) \right)$

$\mathcal{L}_6$ (MREa):
$\text{avg}_{t,m} \left(\left| \frac{\text{avg}_i((d_{t,m,i})^r) - \text{avg}_i((\hat{d}_{t,m,i})^r)}{\text{avg}_i((d_{t,m,i})^r)} \right| \right)$

$\mathcal{L}_7$ (SumMREa):
$\text{avg}_{t,m} \left(\left| \frac{\text{avg}_i((d_{t,m,i})^r) - \text{avg}_i((\hat{d}_{t,m,i})^r)}{\text{avg}_i((d_{t,m,i})^r)} \right| + \left| \frac{\text{avg}_i((d_{t,m,i})^c) - \text{avg}_i((\hat{d}_{t,m,i})^c)}{\text{avg}_i((d_{t,m,i})^c)} \right| \right)$

$\mathcal{L}_8$ (Sorted):
$\text{avg}_{t,m} \left(\text{avg}_i(|(\text{sort}^r(d_{t,m})_i)^r) - (\text{sort}^r(\hat{d}_{t,m})_i)^r|) + \text{avg}_i(|(\text{sort}^r(d_{t,m})_i)^c) - (\text{sort}^r(\hat{d}_{t,m})_i)^c|) \right)$

In terms of optimization algorithm, we consider RAND, GRID, GRAD, and the 4 BO regressors available in Simcal (see Section 4.2.2). In preliminary experiments, GRID performed extremely poorly and was excluded from further experiments. Additionally, BO-ET and BO-RF consistently exceed the memory bound (64GB) during calibration runs and are therefore excluded.

Table 7.3: Calibration error vs. algorithm and loss function.

Alg. \ Loss	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_3$	$\mathcal{L}_4$	$\mathcal{L}_5$	$\mathcal{L}_6$	$\mathcal{L}_7$	$\mathcal{L}_8$
RAND	10.20	3.81	35.18	14.05	12.02	2.58	90.51	14.26
GRAD	21.64	18.30	57.13	8.58	13.95	4.24	40.25	20.67
BO-GBRT	68.69	3.61	79.68	9.76	11.88	12.45	44.66	10.88
BO-GP	25.26	6.93	33.41	11.84	**2.30**	7.18	17.35	8.55

As explained in Section 4.1.2, we use synthetic benchmarking to compare the calibration error of different loss functions. Table 7.3 shows calibration error for different algorithm/loss combinations (lower values are better). The calibrations are computed using synthetic ground-truth data generated for all training data. While several achieve low calibration error, the best combination is the BO-GP algorithm with the $\mathcal{L}_5$ loss function. We use this combination in all that follows. This is similar to one of our findings in the previous chapters, i.e., use Bayesian Optimization.

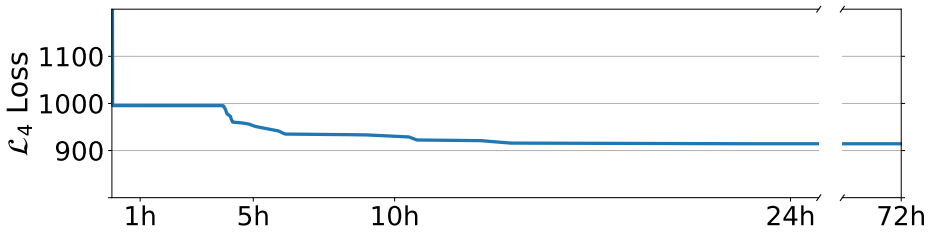


Figure 7.2: Loss value vs. time when using all training data

7.3.4 Calibration Time Budget

As expected, the longer the time budget the better the calibration, up to a point. We used a 24-hour time budget for our experiments, which is sufficient for the loss of the calibration to converge. Figure 7.2 shows loss vs. time when trained on all data up to 72 hours. The loss improves significantly several times over the first few hours, but only improves marginally after 12 hours.

7.4 Use of Ground-truth Data

As discussed in Section 4.1.1, a simulator should be calibrated with respect to a sufficiently large and diverse set of ground-truth executions for the computed calibration. As in the previous chapters, we explore the impact of using different training datasets for computing calibrations, to determine the potential effect of gathering less ground-truth data, which is always desirable from the user's perspective. For this case study, our ground-truth data is more limited when compared to the case studies in the previous two chapters, and there are few opportunities to reduce it further. One opportunity is afforded by the multiple ICD runs for each dataset. ICD = 0.0 corresponds to the situation where all application data is initially stored on the remote data server, which ends up maximizing the load on the external network. ICD = 1.0 corresponds to the situation where all application data is initially cached locally, which thus imposes no load on the external network. ICD values between these two extremes will impose loads on different system components to different degrees. Thus, if our conclusions from Chapter 5 are generalizable, we would expect that this simulator should only require a low number of diverse ICD values (perhaps 2 or 3) to provide enough diversity for calibration, and ICD = 1.0 should likely not be used alone. To test our expectation, we compute several calibrations based on reduced ground-truth data (i.e., for a subset of the ICD values), and evaluate how well they fare on our validation dataset, i.e., the full execution data for the *slow* jobs.

We consider ground-truth data for a single ICD value, for two ICD values (0.0 and 1.0), for three ICD values (0.0, 1.0, and one of the other ICD values), for four ICD values (0.0, 0.3, 0.5, 0.7, 1.0), six ICD values (0.0, 0.2, 0.4, 0.6, 0.8, 1.0), seven ICD values (0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0), and all eleven ICD values. All these combinations, except using data for only ICD = 0.0, performs relatively similarly, with evaluation loss values between 965 and 1,336, evaluated using $\mathcal{L}_5$. When evaluating using $\mathcal{L}_1$, which is a metric that many users would use to quantify simulation error, these combinations yield percent simulation between 21% and 28%. It is perhaps surprising that there is not a larger difference between all these options, especially given that some of them use a single ICD value. The exception is when using data for only ICD = 0.0, which performs poorly with an evaluation loss of 7,372 (192% according to $\mathcal{L}_1$). But recall that, in this case study, a difficulty is that the ground-truth data that was made available to us is incomplete. After examining the ground-truth data we have determined that this unexpected result is primarily caused by missing ground-truth data. Specifically, many data points for ICD = 0.0 are, unfortunately, not available across all platforms. So the fact that using ICD = 0.0 performs unexpectedly poorly is an artifact

of our ground-truth data. We conclude that, in this case study, it is possible to compute good calibrations from ground-truth datasets that are relatively small, i.e., even datasets that include execution data for a single ICD value. However, the incomplete evaluation set makes it difficult to quantify the true simulator accuracy.

7.5 Conclusion

Like in the previous chapters, to assess the value added by our automated calibration methodology, we compare its results to a manual calibration. The developer of DCSim, who is also its main user, has attempted to calibrate the simulation manually based on platform specifications, on their own knowledge of the system, and on ad-hoc "performance plots" from real-world executions. Due to the incomplete nature of the evaluation set, we will present both the evaluation loss, and the (potentially overfit) full dataset loss. The obtained calibration has an $\mathcal{L}_5$ loss of 3800 / 2660 (evaluation set loss / full dataset loss). For an easier interpretation of these numbers we calculated the $\mathcal{L}_1$ loss values achieved by the calibrations, as the $\mathcal{L}_1$ loss function is a metric that many users would use to quantify simulation error. According to this metric, the manual calibration leads to 98% / 85% simulation error. By contrast, the calibration automatically computed using our methodology, after selecting an appropriate loss function and using a minimal, but complete, set of ground-truth data yields an $\mathcal{L}_5$ loss of 1000 / 930, which equates to $\mathcal{L}_1$ loss of ∼22% / ∼35%[1]. We conclude that our automated calibration methodology, like for the case studies in the previous two chapters, affords significant improvement over manual calibration.

After completing this case study we examined the ground-truth data thoroughly. We discovered that it is only about 68% complete, with 25% of experiments being completely unrepresented for some platform configurations and only 58% fully represented. This is an expected challenge inherent to using real-world ground-truth data. We note that completing the ground-truth data is not always possible, due to systems having been changed and/or being no longer available, as is the case for the ground-truth data made available to us in this case study. Using what little data we have multiple samples for, we calculated the error of an ideal simulator at 16%. Thus we conclude that despite the quality of the ground truth data, our methodology managed to achieve a relatively low loss, and thus simulation error, in spite of incomplete ground-truth data.

[1]Perhaps surprisingly, $\mathcal{L}_1$-based accuracy is higher for just the evaluation set, this is likely related to why using $\mathcal{L}_5$ yields better calibrations than $\mathcal{L}_1$ when using our methodology

Part III

Applications of Calibrated Simulators

CHAPTER 8
SIMULATOR SOPHISTICATION

As discussed in Section 2.1, the accuracy of a simulator is partly dependent on the *sophistication* level at which the simulation is implemented. However, determining which level of sophistication is appropriate, or even comparing two simulators that use different levels of sophistication is a challenging proposition. This is because the level of sophistication of a simulator is not the only source of simulation inaccuracy. Another key source of inaccuracy is how well the simulator is calibrated. The automated calibration methodology proposed and evaluated in preceding chapters can be applied to address the above challenge because it greatly reduces the inaccuracy due to poor calibration. Consequently, it makes it possible to perform sound comparisons of simulators implemented at different levels of sophistication, and select which level of sophistication to use in practice in a rational manner.

In this chapter we re-examine our three case studies from Chapters 5-7, focusing on the level of detail at which the simulators are implemented. Specifically, in each case study we consider simulators implemented at several levels of sophistication, apply our automated calibration methodology to each of these simulators, compare the resulting calibrated simulators to each other, and draw conclusions regarding the levels of details that are appropriate in practice. Part of the content in Sections 8.1 and 8.2 has been published in [76], and a preliminary version of the work in Section 8.3 has been published in [74]

8.1 Case Study #1: Scientific Workflows

In this section we return to the WorkflowSimulator simulator from Chapter 5. Recall that our knowledge of the Chameleon Cloud platform is limited. As such, in Chapter 5 we had used a particular abstraction of the network and storage infrastructure, and had also assumed that HTCondor has a significant impact on executing time. In this section we revisit these assumptions, considering different options that implement simulated components at various levels of details. Specifically, for the network topology we consider three options: (i) a single shared network link; (ii) a star topology of dedicated network links between the submit node and each worker; and **(iii) a single shared network link out of the submit node and then a dedicated network link to each workers**. For the storage system, we consider two options: (i) only the submit node has storage capabilities; and **(ii) the**

submit node and the workers all have storage capabilities. For the compute system, we consider two options: (i) the WMS submits tasks directly to the workers; and **(ii) the WMS uses HTCondor to access the workers**. The options shown in **boldface** above correspond to the implementation of the simulator in the original case study in Chapter 5. In total, we consider 12 versions of the simulator. Table 8.1 depicts the above options, and for each option shows the parameters that must be calibrated.

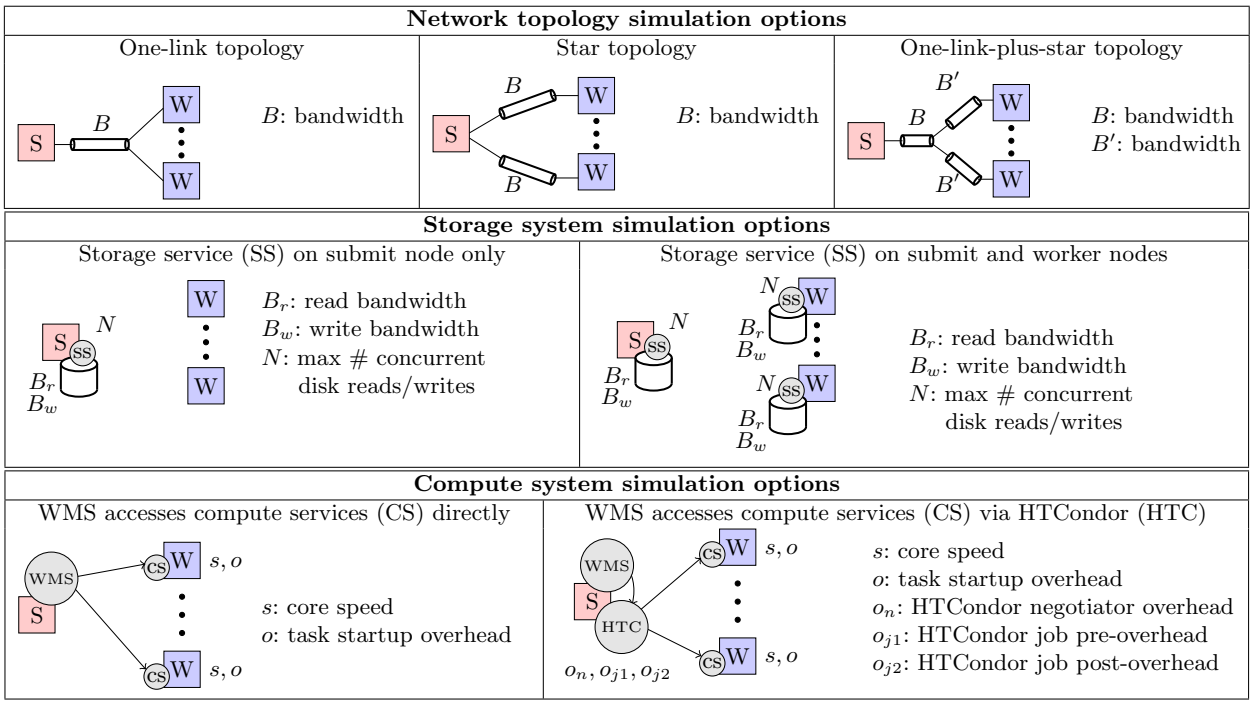


Table 8.1: Level-of-detail options for the simulators used in Case Study #1.

In practice a user would calibrate a simulator, decide whether a different level of detail is likely more desirable, implement a new simulator, and repeat. For the purpose of this case study we instead calibrate all 12 simulator versions. These calibrations should be computed based on a subset of the ground-truth data (the "training dataset") and their simulation accuracy should be evaluated on the remaining data (the "validation dataset"). As in Section 5.1 we use the "large" workflow executions as the validation dataset. For the training dataset we use the workflow executions for the second largest number of workers and the second largest number of tasks. As seen in the results in Chapter 5, this training set is a reasonable choice that would not require the extensive ground-truth data experiments to discover and performed well in those experiments. Figure 8.1 shows, for all 12 calibrated simulator versions, the relative percentage error between simulated and ground-truth makespans

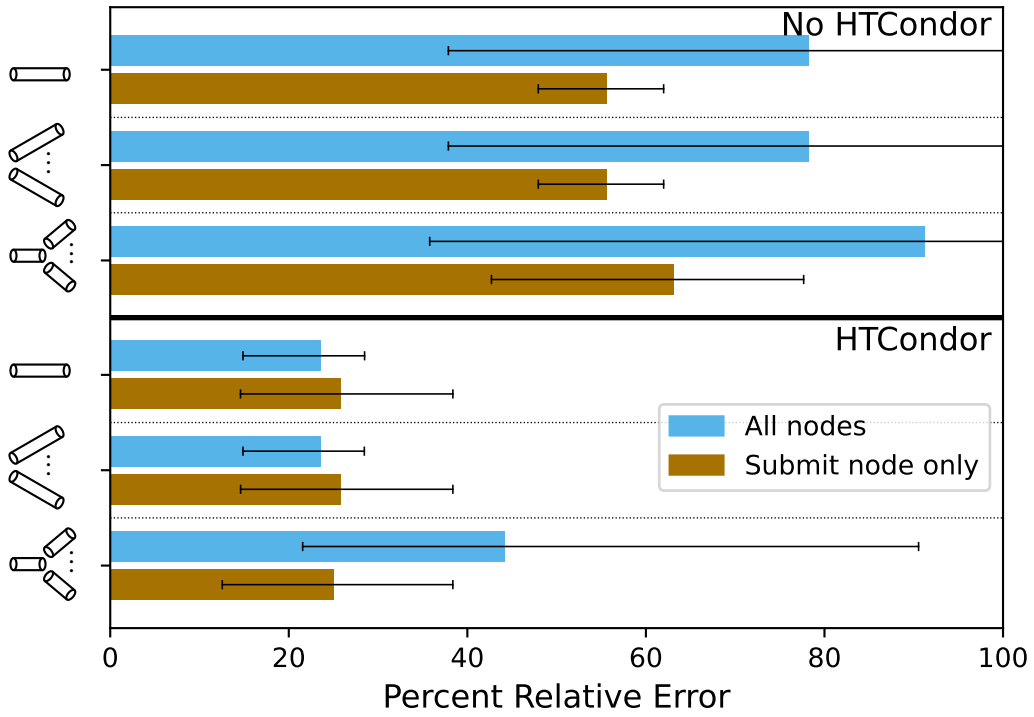


Figure 8.1: Percent relative error between simulated and ground-truth makespans (bars show averages values over all workflows; error bars show min and max values). The top half of the figure is not simulating HTCondor, and the bottom half of the figure is simulating HTCondor. Each pair of bars represents one of the three options for simulating the network (depicted as simplified network diagrams on the vertical axis). Each bar in a pair is for a different option for simulating the storage system, either on the submit node only (brown) or on all nodes (blue). Lower error is better.

($\mathcal{L}_1$). This is the metric that most users would likely care about (and is conveniently the loss value in percentage). The first observation is that simulating HTCondor is crucial. This is because the simulated HTCondor component simulates overheads that occur at different phases of a task's execution. Considering only the bottom half of the figure, we find that using a relatively low level of detail for the network topology is sufficient. This is because the real-world network is sufficiently provisioned to support all concurrent data transfers in the ground-truth workflow executions. As a result, the one-link and the star topology leads to equivalent results. Our more complex topology with a shared link and a dedicated link in series does worse. This may seem surprising because setting the bandwidth of the shared link of the dedicated links to a high bandwidth effectively yields a star topology and setting the bandwidth of the shared link to a low bandwidth yields a one-link topology. But this more complex topology increases the dimensionality, and thus the difficulty, of the simula-

tion calibration problem without bringing any accuracy benefit (at least given our available ground-truth data). Finally, we find that simulating a storage system on all nodes brings only marginal benefit over simulating a storage system on the submit node only. In the real-world deployment there is a storage system at all nodes. But simulating storage only on the submit node provides a reasonable approximation, at least in the scope of our available ground-truth data.

From these experiments, it is now possible to make several conclusions about how the system should be: HTCondor and the overheads it incurs should be simulated; Network topology should be abstracted as a simple network; and while it is better to simulate storage capabilities at all nodes, the gain in simulation accuracy is marginal. Using these conclusions, it is possible to calibrate the platform to achieve average error of $\sim 23\%$. This is a significant improvement over the worse case of $\sim 91\%$ average error, i.e., the case in which the least appropriate level-of-detail options are chosen.

8.2 Case Study #2: MPI Applications

In this section we return to the MPISimulator simulator from Chapter 6. Recall that for this simulator, the target platform (Summit) is thoroughly documented, and MPISimulator was thus implemented at a relatively high level of detail. However, as we found in the previous section, a high level of detail is not necessary best due to the increased difficulty of the calibration problem. In this section we consider other versions of the simulator by considering various abstractions of the components that implement lower levels of detail. Specifically, for the network topology, we consider four options: (i) a single shared backbone link; (ii) a single shared backbone link and dedicated links to each compute nodes;(iii) a 4-ary tree network; and **(iv) a fat tree topology with several layers of switches**. For simulating the compute nodes, we consider two options (i) a multi-core node with NIC, which abstracts away architectural details; and **(ii) a two-socket node where the sockets communicate via an X-Bus SMP bus, and are each connected to the NIC via a PCIe bus**. Lastly, for the MPI Adaptive Protocol change points, we consider two options: (i) protocol changes occur at two known change points set at 16kB and 128kB, leading to three bandwidth factors to calibrate; and **(ii) the same model but where the two change points are unknown and must thus be calibrated as well as the three bandwidth factors**. The options shown in **boldface** above correspond to the implementation of the simulator in the original case study in Chapter 6. In total, we consider 16

versions of the simulator. Table 8.2 depicts the above options, and for each option shows the parameters that must be calibrated.

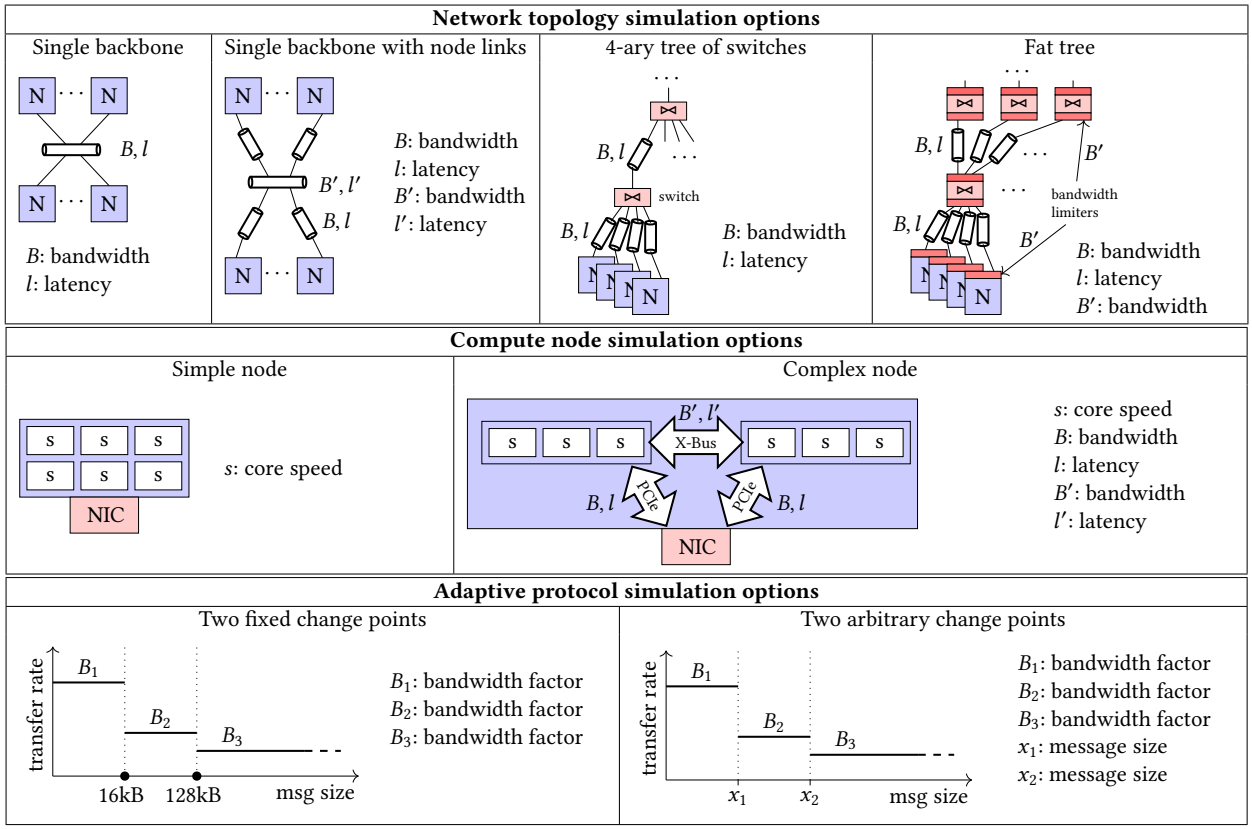


Table 8.2: Level-of-detail options for the simulators used in Case Study #2.

We compare the accuracy of all 16 simulators, each calibrated using our automated calibration methodology. As noted in the previous case study, in practice a user would instead incrementally implement, calibrate, and evaluate simulators. In Chapter 6.4 we found that MPISIMULATOR is unable to generalize beyond the ground-truth data it has been calibrated on. As such, we forgo using separate training and validation sets in this section. Instead we present "overfit" results where both sets consist of 128-node ground-truth executions for PingPing, PingPong, and BiRandom.

Figure 8.2 shows relative percentage error between simulated and ground-truth data transfer rates. This is an easier accuracy metric to interpret than the loss value (which is based on explained variances), and what most users would measure. The first observation is that all simulators exhibit relatively similar results, more so than in the previous case study, with average relative errors between 13% and 24%. Simulating complex, rather than simple, compute nodes is better in most cases. The one exception is when simulating simple

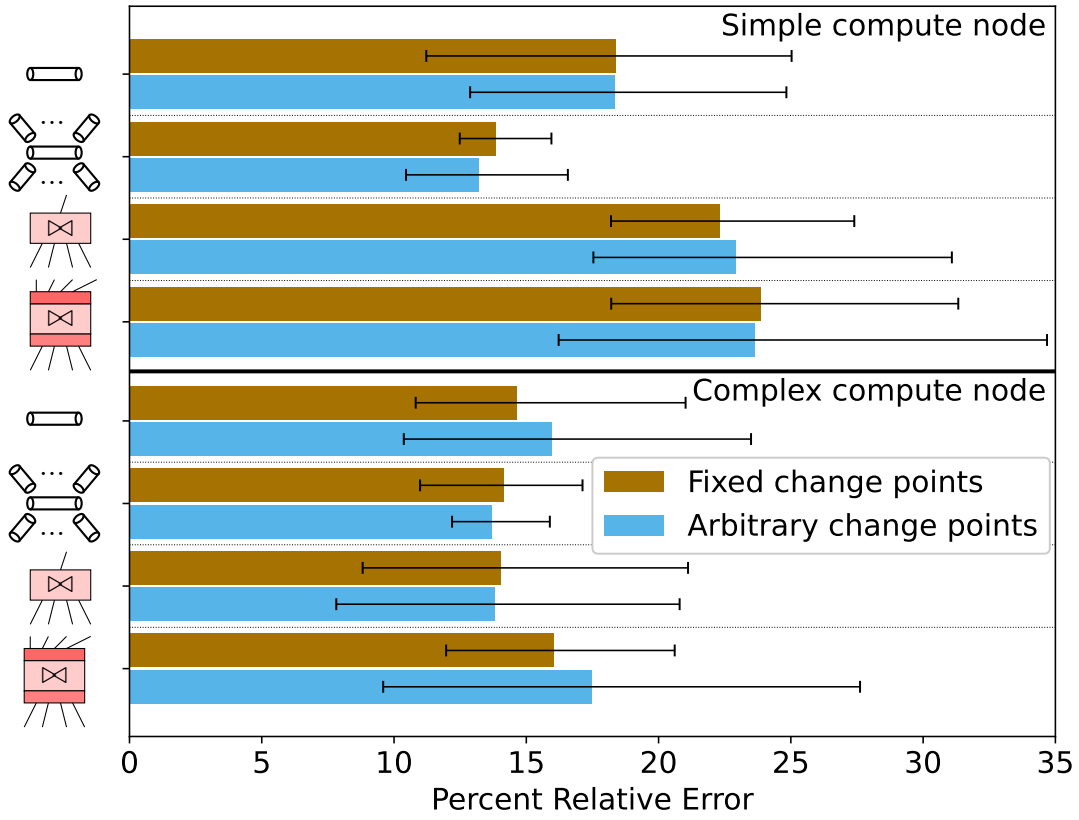


Figure 8.2: Percent relative error between simulated and ground-truth data transfer rates (bars show average values over all benchmarks; error bars show min and max values). The top half of figure is simulating a simple compute node, and bottom half of the figure is simulating a complex compute node. Each pair of bars is for one of the four options for simulating the network topology (depicted as simplified network diagrams on the vertical axis). Each bar in a pair is for a different option for simulating the adaptive MPI protocol, with either fixed or arbitrary change points.

compute nodes and a backbone topology with individual links (i.e., the second pair of bars in the top half of the figure). This particular case happens to lead to average error lower than the best simulator that simulates complex compute nodes (but a higher variance). In terms of MPI protocol simulation, the differences are small in terms of averages. But using fixed, rather than arbitrary, change points, leads to lower variance in most cases. The increase in the level of detail due to non-fixed change points is not worthwhile, at least given our calibration time budget. Finally, we see that the best results are achieved when simulating a backbone topology with individual links, which seems to strike a good compromise between calibration dimensionality and potential accuracy. Simulating the network at a higher level of detail, using a 4-ary tree or a fat-tree topology, leads to worse calibrations.

From these experiments, it is now possible to draw several conclusions about the platform: Complex compute nodes should be simulated; Network topology should be abstracted as a backbone network with individual links; and while it is better to used fixed MPI adaptation points, the gain in simulation accuracy is marginal. Using these conclusions, it is possible to calibrate the platform to achieve average error of $\sim 13\%$, a moderate improvement over the worst-case $\sim 24\%$ average error, i.e., the case in which the least appropriate level-of-detail options are chosen.

8.3 Case Study #3: High Energy Physics Application

In this section we return to the DCSim simulator from Chapter 7. The topology and structure of the subset of WLCG used as a platform for DCSim cannot be altered without affecting the target use case. But there are additional parameters for DCSim that were not calibrated in Chapter 7 and that modify the level of sophistication of the simulation. Specifically the XRootD implementation uses a block size parameter, B. Each file in XRootD, like in most storage systems, is partitioned into blocks. The jobs in the workload process input files block by block, so that reading and processing data is done in a pipelined fashion. Similarly there is a buffer size parameter, b, which specifies the internal buffer size used by a storage service, for the purpose of pipelining I/O and network operations, as done in production storage systems. The lower the B and b values, the higher the level of sophistication of the simulation. WRENCH supports a fluid model of network communication and I/O operation streaming, i.e., an approximation that does not consider buffer size at all. This approximation, which we will denote as $b = \text{None}$, allows for faster simulation.

The B and b parameters correspond to software configuration parameters in the real-world system. Picking realistic values for them could be done by inspecting the software configuration files. This was not done for the WLCG platform for this case study. But values are likely a few MBs or on the order of KBs (e.g., the default XRootD block size is $B = 2\text{MB}$). Yet, in Chapter 7, all simulator invocations used $B = 10^{10}$ bytes and $b = \text{None}$. It may seem surprising that we opted to use such a large B value, when in practice it is likely on the order of MBs. This choice was motivated by fact that the B and b parameters drive the number of discrete events that must be simulated. Given a job in the workload that needs to process s bytes of data, the number of simulated events for this job's execution is $O(s/B + s/b))$. If B and/or b are low relative to s, the simulation time can become prohibitively high.

Given the above, to goal is not to find B and b values that are as realistic as possible. Instead, the goal is to set their values so that the simulation time is below some user-defined threshold, and then calibrate all other parameters automatically. The question is whether this calibration can still lead to good simulation accuracy. In other words, can the automated calibration of the other parameters compensate for the potential loss of accuracy due to simulating the execution at a higher granularity (i.e., larger block and buffer sizes) than the real-world system?

To answer this question we consider different combinations of B and b values. The specific values used and the corresponding average simulation runtimes can be found in Table 8.3. Streaming (b = None) simulations are significantly faster than buffering simulations ($b \neq$ None), and as such we were able to run experiments with lower B values for streaming than for buffering.

Table 8.3: Level-of-detail options and observed simulation times for DCSim.

B	b	Avg. simulation time
10^{10}	10^8	∼3 sec
10^9	10^7	∼15 sec
10^8	10^6	∼2 min
10^7	10^5	∼15 min
10^{10}	None	∼2 sec
10^9	None	∼5 sec
10^8	None	∼5 sec
10^7	None	∼30 sec
10^6	None	∼5 min
10^5	None	∼15 min

We ran our automated calibration procedure for each of the considered B / b combinations including executions for all ICD values in the ground-truth data for computing calibrations. Table 8.4 shows $\mathcal{L}_1$ loss (i.e., percent error on the makespan) for all B / b combinations. In general we observe that loss increases as the simulator sophistication (and therefore simulation runtime) increases. There are a few exceptions, such as $B = 10^5, b =$ None, but these exceptions exhibit only minor improvements in the loss. The overall lowest loss calibration is achieved when using $B = 10^{10}$ and $b = 10^8$ (3-second simulator execution time), however only by a thin margin.

With larger B and b values the simulation has a much lower granularity than the real-world system, which would seem to imply worse accuracy. However, with these large values

Table 8.4: Percentage makespan error ($\mathcal{L}_1$) when evaluating the calibration for each sophistication level against the full dataset for DCSim.

B	b	$\mathcal{L}_1$ Loss
10^{10}	10^8	29.84%
10^9	10^7	29.95%
10^8	10^6	67.44%
10^7	10^5	76.67%
10^{10}	None	30.59%
10^9	None	33.59%
10^8	None	41.62%
10^7	None	76.67%
10^6	None	76.55%
10^5	None	67.93%

the simulation time is short, meaning that the calibration procedure can better explore the parameter space, allowing it to find parameter values that lead to better accuracy in spite of the higher granularity. A lower granularity in the real-world system means better utilization of the hardware resources due to finer-grain pipelining of I/O, network, and compute activities. Pipelining thus increases the effective speed of I/O, network, and compute resources for each job. In simulation this same increase can be achieved instead by using a higher granularity and, at the same time, increasing the speed of the corresponding simulated hardware resources, at least within some bounds. In this case study, doing so allows the user to obtain a calibrated simulator that is both fast and accurate. Because our calibration procedure is automated once defined, it would be straightforward for users of the simulator to explore the accuracy-speed design space to achieve whatever user-specific trade-off is the most desirable. Making this determination manually, without using our automated calibration approach, would be prohibitively labor-intensive.

An interesting additional question that is relevant for this case study is: Once a simulator has been calibrated, can its B and b values be changed to lower values to increase simulation accuracy? This would allow a simulator to be calibrated using a level of sophistication (low B and/or b) that allows for rapid calibration, and then switch to a more sophisticated (high B and/or b) but slower version for production use. To explore this possibility, we considered the best performing calibration when simulating buffering ($B = 10^{10}, b = 10^8$) and when simulating streaming ($B = 10^{10}, b =$ None). We then re-evaluated each with the highest level of sophistication in each case, i.e., buffering ($B = 10^7, b = 10^5$) and streaming ($B = 10^5, b =$

None), leaving other parameter values unchanged. The new loss becomes 50.15% for buffering and 64.47% for streaming. These loss values are better (i.e., lower) than the ones in Table 8.4 that were obtained when computing the calibration using the simulator at those higher levels of sophistication. But, unfortunately, these values are markedly higher than the original loss values. That is, taking a calibration computed using the simulator at a low level of sophistication (high B and b) and then using it for a simulator at higher level of sophistication (lower B and b), simulation accuracy is decreased overall. In other words, the answer to the question at the beginning of this paragraph is negative. The simulated component's differences caused by using a lower level of sophistication are compensated for by using different hardware parameters. Changing the sophistication parameters after calibration defeats this compensation. As a result, the simulator's overall accuracy is decreased.

From the above results we can draw several conclusions. The main conclusion is that it is possible for automated calibration to compensate for decreased level of simulation sophistication (in this case increased granularity of the simulation of I/O and network operations). We also discovered that the level of sophistication used for DCSim in Chapter 7 was sufficient, which was not a given. Lastly, we observed that attempting to increase accuracy by increasing sophistication post calibration does not work in this case study, and we surmise that it is unlikely to work in general.

8.4 Conclusion

Determining which level of sophistication is appropriate in a simulator, or even comparing two simulators that use different levels of sophistication is a challenging proposition. In this chapter, we have used our automated simulator calibration methodology to address this challenge. We have re-examined our three case studies from Chapters 5-7, but focusing on the level of detail at which the simulated components are implemented. For each case study, our approach makes it possible to perform a sound comparison of different levels of details, allowing a user to pick the level of detail at which each simulated component should be implemented in a quantitative and rational manner.

CHAPTER 9
USING SIMULATION FOR SCHEDULING AT RUNTIME

Now that we have a methodology that can (significantly) improve the accuracy of a simulator via automated calibration, in this chapter we examine a potential application of calibrated simulators: Using simulation at runtime to inform scheduling decisions made by Workflow Management Systems. Section 9.1 provides an introduction to the problem of workflow scheduling. Section 9.2 discusses related work. Section 9.3 describes our approach, which we evaluate via the experimental methods described in Section 9.4. Section 9.5 presents experimental results. Finally, Section 9.6 summarizes our contributions and discusses future work. Part of the content in this chapter has been published in [73].

9.1 Introduction

As already noted in Chapter 5, scientific workflow applications have been used by computational scientists to support some of the most significant discoveries of the past several decades [6, 29], and are executed daily to serve a wealth of scientific domains. Many workflows have high computational demands and are executed in production on platforms that range from single HPC clusters to federations of such clusters and clouds. Setting up, orchestrating, monitoring, and optimizing workflow executions on these platforms is managed by runtime systems often called Workflow Management Systems (WMSs) [38, 37]. These systems automate workflow executions including (i) selecting hardware resources; (ii) picking application configuration options (e.g., numbers of cores to be used by multi-threaded tasks); (iii) allocating resources to application activities; and (iv) monitoring application executions. This means that when workflow tasks are ready to be scheduled for execution, decision must be made in most of these categories. It is widely accepted that these scheduling decisions should be made by using appropriate *scheduling algorithms* so as to optimize performance, monetary cost, energy consumption, reliability, etc. The past decade has witnessed a proliferation of WMSs [120], but there is no consensus on which scheduling algorithms should be implemented in these systems.

Scheduling problems are generally NP-hard, and thus most proposed algorithms employ (typically non-guaranteed) heuristics. The design of scheduling algorithms for workflow ap-

plications has received an enormous amount of attention [68, 115, 69, 82, 5, 97, 94, 48, 1]. Most of the proposed algorithms reuse ideas and principles from the extensive DAG (Directed Acyclic Graph) scheduling literature [98]. Yet, when examining existing WMSs, there is a clear disconnect between research and practice. Given the complexity of hardware platforms on which workloads are to be executed and the complexity of workflow applications themselves, scheduling research results are typically obtained relying on simplifying assumptions so that scheduling problems are rendered more formalizable and tractable. Furthermore, published evaluation results cannot cover all relevant situations a WMS could encounter in practice. The literature is thus rife with scheduling algorithms that have been evaluated within the scope of their underlying assumptions, but whose potential effectiveness in practice for particular use cases remains unquantified. Given the above, there is little incentive for WMS developers to pay close attention to scheduling research. Our own observation of current and popular WMSs has shown that naive, and thus possibly vastly sub-optimal, scheduling algorithms are typically implemented.

A way to resolve the disconnect between scheduling research and scheduling practice is to obviate the challenge of picking a particular scheduling algorithm to implement in a WMS. Instead, one can implement a set of scheduling algorithms, estimate how each algorithm would fare for the particular use case at hand at runtime, and select the most desirable algorithm. This approach has been termed "portfolio scheduling" [32]. We consider WMSs that automate the execution of workflow applications that perform I/O, communication, and computation operations that use and contend for various hardware resources. In this context, determining how a scheduling algorithm would perform using an analytical model to compute application execution metrics (e.g., the makespan) is challenging, due to the need to account for complex phenomena such as disk bandwidth contention, network sharing, network protocol effects, overlap between computation, I/O, and network communication activities, etc. An alternative is to use discrete-event simulation, by which the behavior of a particular workflow application when executed on a particular hardware platform emerges from simulated discrete events. Application executions metrics can then be easily computed from the produced event trace. As seen in previous chapters, these metrics are likely not be perfectly accurate even if the simulator has been calibrated (and likely very inaccurate if the simulator has not been calibrated)

9.2 Related Work

Several authors have proposed to select a scheduling algorithm (or the parameter values that define a particular instantiation of a scheduling algorithm) at runtime. Approaches for performing this selection have been investigated in various parallel and distributed computing domains and are often termed "portfolio scheduling". The key question is that of the mechanism that should be employed to select the best algorithm among those included in the portfolio. Authors have proposed different answers to this question by using one of three approaches, which we review in the next sections: online performance monitoring, offline simulations, and online simulations.

9.2.1 Online Performance Monitoring

Previously proposed approaches perform algorithm selection via *online performance monitoring* of the algorithms in the portfolio. In [43] the authors use such an approach in the context of batch scheduling for HPC clusters to dynamically select one among 12 different batch queue reordering policies. Their approach keeps track of how well each policy has fared over previous time windows in terms of job wait times, and selects the policy for the next time window using reinforcement learning. Specifically, an ϵ-greedy strategy is employed to achieve both exploitation (select with high probability the policy that has worked best in the past) and exploration (select with low probability a policy at random). Reinforcement learning is also used in [10] for selecting one among 10 loop scheduling algorithms in the context of iterative distributed-memory parallel applications. The authors define a metric that captures the robustness of a scheduling algorithm to variations in the execution time of the loop iteration. Using this metric as a reward they then evaluate the effectiveness of several reinforcement learning techniques for selecting the loop scheduling algorithm. In [79], the authors propose an approach for selecting one among 12 scheduling algorithms in the context of OpenMP parallel loops. Based on online monitoring of the load imbalance after each iteration they propose several methods for selecting the algorithm to use for the next iteration, including a method based on fuzzy logic. These works are for different application contexts but they all rely on the same assumption: observing how algorithms have fared over time windows or application steps in the near past makes it possible to determine which algorithm is best for the near future. This is clearly a reasonable assumption for the iterative applications considered in [10, 79]. In [43] this assumption is more questionable since batch scheduling decisions can have long-term implications, which explains why the results therein

show that a simulation-based approach yields better results.

In this chapter we focus on workflow applications for which it is difficult to assess the effectiveness of a scheduling algorithm based on its past performance. The particular task graph structure of a workflow can cause early scheduling decisions to have significant long-term (positive or negative) impact on the overall execution time. As a result, and as observed in [19], although a particular scheduling algorithm may seem efficient (e.g., in terms of amount of computational work performed per time unit) during an early phase of the execution, other algorithms that are similarly or less efficient during that phase can lead to significantly shorter execution times overall. This provides a clear motivation to use simulation to assess future algorithm performance instead of online monitoring of previously observed performance.

9.2.2 Offline Simulations

Several authors have proposed using results from *offline simulations* to drive portfolio scheduling decisions at runtime. In [14], the authors focus on batch scheduling for HPC clusters and study the problem of picking the batch queue ordering policy at runtime. They generate large numbers of synthetic workload datasets and run offline simulations of their executions for permutations of the job scheduling order. The goal is to determine a score that, based on a job's characteristics, quantifies the impact of scheduling that job first. They then use a machine learning technique (nonlinear regression) to derive a model of that score. At runtime, the queue ordering policy is decided by predicting the score for each job in the queue and picking the job with the best predicted score. Also in the context of batch scheduling for HPC clusters, in [107] the authors conduct offline simulations of the execution of several real-world workload datasets using two different queue ordering schemes. The simulation results show that the average job parallelism can be used to decide on which scheme to select at runtime. The proposed approach consists in defining a time window length, computing the average job parallelism over that time window, and using the computed value to pick which of the two queue ordering schemes should be used for the next time window. In [105], the authors propose to use a supervised learning approach for selecting which dynamic loop scheduling algorithm to use at runtime. The training dataset consists of simulated executions for various problem sizes, hardware platform specifications, and workload and resource variability ranges. At runtime, the trained model can then be used to decide which algorithm in the portfolio (which in the presented results consists of eight algorithms) should be selected. The main challenge for these approaches is to deter-

mine what kind and what amount of training data should be used to obtain good results in practice. In some cases, the setting is simple enough that only a few parameters are to be considered (e.g., 4 parameters in [105]). It is then reasonable to expect that even a relatively small training set can produce good results. But this is not always the case. For instance, in the context of batch-scheduling, there are many kinds of possible workloads with different distributions of job arrival times, levels of parallelism, requested durations, actual durations, and correlations between all these features. It is thus likely that in [14, 107] better results could be achieved with larger and/or more diverse datasets.

This chapter focuses on portfolio scheduling to be implemented in Workflow Management Systems. These systems must support applications that have a large space of possible configurations in terms of task graph structures (number of tasks, number of levels, distribution of parallelism across these levels, density of dependency edges), task compute volumes, and data compute volumes; and execute these applications on platforms that also have a large space of possible configurations in terms of scale, heterogeneity, network topologies, and storage architectures. As a result, only a very large training dataset could potentially lead to good results. But obtaining a sufficiently large real-world dataset is prohibitively difficult in the general case. An alternate approach could be to generate a synthetic dataset from large numbers of high-accuracy offline simulations.

9.2.3 Online Simulations

Using online simulation for portfolio scheduling can be seen as a compromise between online performance monitoring and offline simulation approaches. Like the former it does not require any a-priori model of which algorithms in the portfolio will perform well in particular situations, and like the latter it uses simulation to capture long-term impact of scheduling decisions. In [78] the authors propose to simulate periodically at runtime the execution of iterative parallel applications when using one of 13 possible dynamic loop scheduling algorithms, and pick the best algorithm to use for the next time period. Their main objective is to adapt to changing platform conditions. The key difference between the work in this chapter and that in [78], besides the targeted application domain, is that the authors therein assume that simulation error is low (below 2%). This assumption makes sense in their context due to the focus on compute-intensive applications for which the influence of possibly complex memory hierarchies, network protocols and network topologies is small. Instead we consider the execution of distributed applications for which network and I/O usage can drive the overall performance. As a result, simulation error can be higher since simulating complex

network and I/O behaviors accurately is challenging. Furthermore, unlike in [78], and as explained at the end of Section 9.2.1, we must simulate the application execution until its completion. As a result, simulation errors are likely to accumulate and high simulation error (much higher than 2%) is likely. One of our contributions in this chapter is to show that simulation-driven portfolio scheduling remains viable even with high simulation error.

Most previous works that have employed online simulation for portfolio scheduling are in the context of batch scheduling for HPC clusters. The works in [43, 103, 33, 32, 107] all propose similar approaches by which a portfolio of standard batch scheduling algorithms is considered. At runtime, while scheduling is done using a particular scheduling algorithm during the current time period, all other algorithms are also used to compute the schedule during that time period. At the end of the time period, the schedules produced by all algorithms are compared using metrics such as average job wait time or bounded slowdown, and the best performing algorithm is selected for the next time period. The work in [36] proposes to use online simulation to tune the parameters of a single batch scheduling algorithm at runtime, using genetic algorithms to determine optimal parameter values. The fitness function evaluation entails running simulations of the recently executed workload. In all these works, because jobs are submitted to the HPC cluster by users, at the time the next algorithm must be chosen, the future workload is unknown. Therefore, online simulations are used to simulate the past workload, assuming that it is representative of what the future workload will be. In this chapter, instead, we use online simulation to simulate the future workload. This is because the future workload is known (as the workflow application is fully specified) and because, as explained at the end of Section 9.2.1, the performance of a scheduling algorithm in earlier phases of the execution may not be indicative of its performance overall. Another major difference between this work and that in [43, 103, 36, 33, 32, 107] is that in those works the "simulation" is not a discrete-event simulation but is instead merely the use of each algorithm to compute a schedule. That is, there is no actual simulation of the hardware platform, but merely a computation of a Gantt chart of the schedule based on job sizes (number of compute nodes) and durations (requested by the user). The only source of inaccuracy in this computation is the job durations requested by the users. This does not, however, correspond to the standard notion of simulation inaccuracy, which stems from the fact that the simulation is only an approximation of a real-world system. Instead, the inaccuracy in these works is in the input to the simulation, which is no different from the inaccuracy of the input to the real system. The one exception among the works cited above is [43], in which after the schedule has been computed so as to estimate the wait time of all

the jobs, then a random uniformly distributed noise (up to 20%) is added to each job wait time. We also inject random noise to experiment with various levels of simulation accuracy. In our context, simulation inaccuracies arise because the simulation cannot perfectly capture the behavior of a complex system in which an application workload performs communication, I/O, and computation activities that use and contend for distributed hardware resources. Furthermore, information on the current state of the execution, on the platform configuration, and on the application's behavior, which are all needed to instantiate a simulation, is not perfect. As a result, simulation models are inherently biased and simulation inaccuracy can be large. For this reason, and unlike all the aforementioned works, we also investigate the possibility of mitigating simulation error at runtime.

The authors in [83] use online simulation for the purpose of workflow scheduling. The authors propose to use simulation to improve the decisions made by a single scheduling algorithm. A workflow scheduling algorithm makes decisions regarding which task should be scheduled next based on task finish time estimates. These estimates are traditionally obtained based on analytical models of task compute, I/O, and/or communication times. Given the complexity of current platforms, developing accurate analytical models is a steep challenge, especially if needing to account for contention and network protocol effects. The authors in [83] propose to obtain network transfer time estimates at runtime using discrete-event simulation, thereby developing in essence a simulation-enhanced scheduling algorithm. A major difference with our work is that the purpose is not to perform portfolio scheduling. But both our work and that in [83] share a key motivation for using simulation for scheduling workflows on distributed computing platforms: the ability afforded by simulation to estimate network (and in our case also I/O) times more accurately than would be possible with only analytical models.

Finally, the idea of driving scheduling decisions using online simulations has been explored by authors outside the field of parallel and distributed computing, such as for manufacturing and logistics (e.g., to solve versions of the job shop scheduling problem for cyber-physical systems). A cursory review of that literature shows that simulation-driven scheduling has been proposed and used since at least the early 1990's. To take just one recent example, the work in [86] proposes simulation-driven scheduling in the context of wind farms operation.

9.3 Proposed Approach: SDPS

Consider a distributed computing platform with hardware resources (compute, storage, network) accessible via various software services for starting computations, storing data, and moving data. A Workflow Management System (WMS) is used to automate the execution of a workflow application on this platform. The WMS must make decisions regarding the allocation of application activities to the hardware resources in time and space, with the goal of optimizing some user-defined metrics of performance.

We propose simulation-driven portfolio scheduling (SDPS) by which the application execution is simulated at runtime for each scheduling algorithm in a portfolio. A description of the application and of the available hardware resources is constructed based on (likely imperfect) available information, so as to instantiate a simulator of the remaining application execution. Simulations are then executed for each algorithm in the portfolio, assuming in each case that the considered algorithm is used until application completion. These simulations are executed on the host on which the runtime system itself executes (typically some multi-core host that orchestrates the application execution on other "remote" resources). Each simulation produces a trace of time-stamped simulated discrete events, from which execution metrics can be computed. The scheduling algorithm that achieves the best such metrics (in simulation) is then selected and used for making scheduling decisions at runtime from then on.

Hereafter, we discuss implementation considerations, discuss when SDPS should be used at runtime, and state the specific research questions that we investigate in this chapter.

9.3.1 Implementation Considerations

For a WMS to implement SDPS it must: (i) make it possible to implement and activate different scheduling algorithms; (ii) have access to information regarding the platform, the application and the current state of the execution to instantiate a reasonable (but not necessarily perfectly accurate) simulation of the remainder of the execution; and (iii) include an implementation of a simulator of this execution instantiated based the available information. We discuss these three requirements hereafter.

Scheduling algorithms

All WMSs provide ways for users to specify workflow tasks and task control- and data-dependencies. Information provided to the WMS about each task can include resource re-

quirements (in terms of compute cores, memory, and disk space), amounts of CPU and I/O work, and/or traces from previous executions. To execute a workflow WMSs are configured with or can discover the resources provided by the platform on which the workflow is to be executed. At runtime, WMSs keep track of which tasks are ready, running, or not ready, and can also keep track of where application data is stored. All the above information can be used to automate resource management decisions and to implement scheduling algorithms (the more information is available, the larger the number of algorithms that can conceivably be implemented). Hundreds of WMSs have been developed over the last decades [120] and a comprehensive survey is outside the scope of this work. Our own observation of some current and popular WMSs has shown that often only baseline naive scheduling algorithms are implemented (such as arbitrary greedy algorithms). However, in some cases scheduling algorithms can be implemented by users as external plug-ins or modules (e.g., StreamFlow [24], Dask [93]). In some other open-source projects, we have determined that replacing the default scheduling algorithm would be straightforward (e.g., Parsl [7], Pegasus [30]). Consequently, in these WMSs it is feasible to implement a portfolio of algorithms. SDPS can then be implemented as a new scheduling algorithm that, whenever invoked by the WMS, either invokes the currently selected algorithm from the portfolio or selects a new algorithm and invokes it.

Information to instantiate a simulation

Instantiating a simulation of a workflow execution requires a workflow description, an execution platform specification, and the current state of the execution. As explained above, most WMSs are configured with and/or maintain such information. At a minimum, the information necessary to instantiate a simulation for SDPS includes: task dependencies; estimates of CPU and I/O work for each task; hardware specifications or benchmark results for compute network, and I/O resources; a set of completed, ongoing, and to-be-executed tasks; and locations of application data file replicas. The challenge is that this information is never perfectly accurate nor complete. Particular values may not be perfectly known and only rough estimates may be available (e.g., CPU speeds, disk bandwidths). Structural information could also be incomplete. For instance, the precise network topology of the platform is typically not known to the WMS. Instead, the WMS may only be provided with information regarding network interface bandwidth hardware specifications, or previously observed data transfer rates on that topology. In this case, a simulation will necessarily abstract away the real-world network topology, for instance as a single network link with some well-chosen

bandwidth. As we have seen in preceding chapters, automated simulation calibration is an effective approach for improving simulation accuracy in these situations. Regardless, in this chapter, we do not assume high accuracy and instead evaluate the usefulness of SDPS for a range of inaccuracy levels.

Simulator implementation

The onus of implementing and maintaining the simulator used for SDPS falls on the WMS developers as only they have all the necessary knowledge about the functional behavior of their system. The simulator can be provided to the WMS as a stand-alone program or implemented directly in the WMS using the simulation framework's API. In this chapter we use a similar simulator to WorkflowSimulator from Chapter 5, which is called SchedulingSimulator and is also implemented using WRENCH [17]. The core of SchedulingSimulator, which corresponds to what a WMS developer would have to implement, is under 300 lines of C++ code (code available on GitHub [88]). While quite similar to WorkflowSimulator, SchedulingSimulator has additional features, such as the ability to dynamically change the scheduling algorithm in use and the ability to launch speculative simulations to explore various future scheduling scenarios.

9.3.2 When to apply SDPS

SDPS necessarily executes a round of simulations at the onset of the application to pick a scheduling algorithm, but other rounds of simulations could be conducted throughout the execution. One reason for doing so is to handle dynamic behavior of the workflow or the platform, which may make different scheduling algorithms better suited at different times.

Dynamic workflow changes occur when new tasks are created at runtime based on the output generated by other tasks. Note that a preliminary study found that, even for static workflows, using more than one algorithm during the workflow execution can sometimes bring marginal benefits [19]. This is because different workflow levels can have different characteristics (e.g., different ratios of data to compute volumes) and can be better served by different scheduling algorithms. Dynamic platform changes occur for shared platforms on which resources that can be used for executing the workflow exhibit variable performance or availability (e.g., due to external load, to resources being reclaimed or released by a provider, to resources being acquired by the user). To handle these dynamic changes, rounds of simulations could be performed continuously throughout the execution (in the extreme

at each scheduling decision) provided the frequency at which they are performed is feasible given the simulation overhead (see Section 9.5.5).

Another reason for running at least one other round of simulations, even when there are no dynamic workflow and platform changes, is that simulations, and in particular those executed at the onset of the execution, are necessarily inaccurate. During the execution, it becomes possible to compare the real execution to its previously simulated counterpart, determine causes of simulation inaccuracies, and thus improve the simulation's instantiation for better future accuracy. We term this process simulation error mitigation.

9.3.3 Objective

Our objective in this chapter is to determine whether SDPS has merit in the context of WMSs. To this end, we seek to answer the following questions:

Q#1: What is the potential improvement over the traditional approach? We wish to quantify the improvement that SDPS can afford over using a single scheduling algorithm.

Q#2: What level of simulation accuracy is necessary? No simulation is perfectly accurate and we wish to determine the level of accuracy needed for SDPS to outperform or at least be comparable to the traditional approach.

Q#3: Is it useful to mitigate simulation inaccuracy at runtime? At runtime it may be possible to apply corrective measures to mitigate simulation error and apply SDPS again to select a scheduling algorithm. We wish to determine whether performing such simulation error mitigation is worthwhile.

Q#4: What is the impact of the sophistication of the simulation? As we have discussed in the previous chapter, simulators range in their implemented level of sophistication. When simulating components of the target real-world system, a developer can opt to use a naive models rather than attempt implementing and calibrating highly detailed models. We also found that some seemingly over simplistic approximations still allow for high simulator accuracy once calibrated. We wish to determine whether a less sophisticated simulator still allows SDPS to rank candidate algorithms effectively.

Q#5: Is SDPS effective for handling dynamic resource availability? Dynamic resource availability can occur in shared platforms (e.g., due to external load, to resources being reclaimed/released by a provider), in which case using different scheduling algorithms at different times could be beneficial.

Q#6: Is simulation overhead sufficiently low? For SDPS to be feasible in practice, the simulation overhead should be sufficiently low when compared to the time required for workload execution.

9.4 Experimental Methodology and Scenarios

To evaluate SDPS we consider the execution of scientific workflow applications on a *multi-cluster* platforms, i.e., platforms that comprise one or more commodity clusters (with all computes nodes within the same cluster being homogeneous) connected over a wide-area network. The scheduling objective is to minimize overall execution time, or *makespan*. In what follows we describe our experimental methodology (which relies exclusively on simulations), the platform configurations and workflow instances that we use to drive our simulations, and the scheduling algorithms we include in the portfolio.

9.4.1 Experimental Methodology

To perform a sound evaluation of SDPS we need: (i) an implementation of a WMS that executes workflows on multi-cluster platforms; and (ii) an implementation of a simulator of these executions that can be invoked at runtime by the WMS. We face two main technical difficulties. First, to answer Q#2 in Section 9.3, we need to experiment with different levels of simulation (in)accuracy, including the best-case 100% accuracy. This is not possible with a real-world implementation since a given simulator is necessarily inaccurate as seen in Chapters 5-7. Even a perfect simulator would not achieve 100% accuracy due to the nondeterministic nature of the ground-truth data. This accuracy achieved when simulating a particular scenario is typically unknown unless this scenario is part of some available ground-truth data (e.g., the ground-truth data used to calibrate the simulator). Second, we wish to evaluate SDPS on a large spectrum of workflows, platforms, and algorithms. For instance, in this work we consider a portfolio of 48 individual algorithms, plus SDPS, for $3 \times 9 = 27$ experimental scenarios (3 platform configurations, 9 workflow instances), for a total of 1,323 different application executions. Furthermore, we evaluate many different versions of SDPS and results of obtained for different random samples so that, overall, we obtain results for over 375,000 application executions. It would be prohibitive to obtain all these experimental results in a real-world setting, not only in terms of time and energy consumption, but also in terms of repeatability.

Given the above, we perform our experiments entirely in simulation. Thus SchedulingSimulator has been implemented to simulate a WMS that executes workflows on multi-cluster platforms as an analog of a production WMS implementation for which SDPS can be implemented. That is, *during its simulated execution, the WMS runs a simulation of its own future execution for each algorithm in the portfolio.* The `fork` system call is used to clone child processes that each simulate the full remaining execution of the workflow using a particular algorithm and report the observed makespan to the parent process, i.e., the WMS. The WMS then picks for future use the algorithm that achieved the lowest of these observed simulated makespans. Once the workflow execution completes the simulator outputs the workflow makespan. This is at times confusing as while there are in effect multiple simulators, the simulator performing the "real" execution and the simulators exploring future schedules, all share the same code. Where clarity is needed, we will use SchedulingSimulator to refer to the main "real" simulator and ExplorationSimulator to refer to the children processes of SchedulingSimulator (created via `fork`) that are used to explore schedules speculatively.

SchedulingSimulator and ExplorationSimulator are both implemented using the WRENCH [122] (v2.1) and SimGrid [95] (v3.32) simulation frameworks. Simulator code, raw experimental data, and scripts to analyze and plot the data are publicly available on GitHub [88]. SchedulingSimulator takes as input *a platform configuration* and a *workflow instance*, as described in the next two sections. Naturally, ExplorationSimulator takes similar inputs as well as the current execution state of SchedulingSimulator, but it inherits these inputs when it is spawned via a `fork` system call. Although we base our platform configurations and workflow instances on real-world use cases, not all required information is available. As a result, we augment the available information using reasonable assumptions, as described in the next two sections. This does not invalidate our evaluation results since our speculative simulations are simulated executions using this same augmented information. In other words, because ExplorationSimulator effectively simulates itself when invoking SchedulingSimulator, it can achieve 100% accuracy. As a result, there is no need for (automated) calibration with respect to real-world, ground-truth data for the experiments in this chapter since accuracy can be made optimal by design. In some of the experiments described in future sections, however, we inject simulation error to evaluate whether SDPS remains viable in the presence of (large) simulation error.

Table 9.1: Cluster configurations

Cluster	Ecotype	Dahu	Neowise
Number of nodes	48	32	10
Number of cores per node	10	16	48
Core speed (Gflop/sec)	3.21	4.01	6.48
Storage r/w bandwidth (Gbps)	100	100	100
Internet bandwidth (Gbps)	10	7	8
Used in platform configurations	P_1, P_2, P_3	P_2, P_3	P_3

9.4.2 Platform Configurations

We consider platforms that comprise commodity clusters with different numbers and types of compute nodes. Each cluster is homogeneous and its compute nodes are connected via a 100GbE interconnect. Each cluster is connected to the Internet via a network path with some bottleneck bandwidth. The compute nodes at each cluster have access to shared storage local to the cluster (network-attached storage, parallel file system, etc.) with some bounded aggregate I/O bandwidth. Whenever a compute node in a cluster needs to write application data, it writes it to the cluster's shared storage. Whenever a compute node in a cluster needs to read application data, it does so from the cluster's shared storage if possible. Otherwise, the data is read from a remote location (the user's machine, where all input data is located initially, or another cluster's storage) and then cached locally. We assume that storage capacity at each cluster is large enough to hold all application data if necessary. Considering storage capacity constraints would simply amount to removing some clusters from consideration when using the scheduling algorithms described in Section 9.4.4. Finally, in most of our experiments we assume that the platform is dedicated to the application's execution and delivers constant performance throughout this execution (e.g., no external load or transient behaviors). In practice, this is achieved by reserving resources in some shared platform (starting virtual machine instances in cloud platforms, submitting pilot jobs to batch-scheduled clusters, etc.).

We base our platform configurations on the specification and benchmark results of an actual hardware platform, Grid5000 [46]. Grid5000 comprises many clusters distributed over a wide-area network, and we consider three of these clusters: Ecotype, Dahu, and Neowise. Table 9.1 lists the clusters' hardware characteristics, most of which are derived

Table 9.2: Workflow instances, indicating for each the application name ("Name"), the number of tasks ("#Tasks"), the sequential compute time on a single 3.21Gflop/sec core ("Work"), the number of data files ("#Files"), the sum of all data file sizes ("Footprint"), the length of the longest path in the workflow's task graph in number of tasks ("Depth"), and the maximum number of tasks that can be executed in parallel ("Max Width").

Workflow	Name	#Tasks	Work	#Files	Footprint	Depth	Max Width
W_1	1000Genomes	328	6h02m	352	24.72GB	3	216
W_2	BLAST	303	8h45m	907	471.79KB	3	302
W_3	BWA	1004	3h44m	3012	56.79MB	3	1002
W_4	Cycles	874	5h13m	6930	6.18GB	4	652
W_5	Epigenomics	1095	5h40m	1370	8.26GB	9	542
W_6	RNA-Seq	197	0h43m	680	290.80MB	10	121
W_7	SoyKB	156	6h47m	321	2.83GB	11	121
W_8	SRA-Search	22	5h16m	48	16.51GB	3	21
W_9	Viralrecon	203	0h42m	877	270.79MB	18	52

based on advertised hardware specifications in [46]. Core speeds are from benchmark results obtained for an N-body physics simulation executed on each cluster's particular core type (Intel Xeon E5-2630L v4 for Ecotype, Intel Xeon Gold 6130 for Dahu, and AMD EPYC 7642 for Neowise). We do not have benchmark information regarding storage system bandwidths at these clusters, and we simply assume 100 Gbps for all clusters, which is typical of real-world platforms. Finally, these clusters are deployed on a wide-area network with 10Gbps end-to-end bandwidths. The bandwidths shown in Table 9.1 correspond to a particular observation on the Grid5000 platform (which provides a real-time network weather map) in which there is some background network traffic to/from the Dahu and Neowise clusters.

We consider 3 platform configurations, P_1, P_2, and P_3, where P_x corresponds to using the first x clusters from left to right in Table 9.1 (see the bottom row). These configurations correspond to different test cases for our approach in terms of platform heterogeneity and scheduling decision complexity going from P_1 (fully homogeneous with simplest scheduling decisions) to P_3 (most heterogeneous with most complex scheduling decisions).

9.4.3 Workflow Instances

We consider the execution of 9 workflow specifications based on real-world scientific applications, as listed in Table 9.2. 8 of these workflows are from the Bioinformatics domain, in which workflows (or "pipelines") are common-place. The Cycles workflow comes from the Agroecosystem domain. These workflow instances are provided by the WfCommons project [116] and were constructed based on logs from actual executions [23]. Each workflow instance defines a set of tasks, with specified execution times, and a set of files, with specified sizes. Each file can be input to and/or output from tasks, thus creating data dependencies. The metrics shown in the table show that our workflow instances are diverse, with different structures and different computation-data ratios. Overall, we expect that different scheduling algorithms will fare differently across these workflow instances [1].

WfCommons workflow instances specify task execution times in seconds, and give the specifications of the processors on which tasks where executed. None of these processors correspond to the processors of the clusters in the Grid5000 clusters described in the previous section. Furthermore, some of the platforms on which workflows were executed are heterogeneous, meaning that different tasks ran on different kinds of processors. We arbitrarily assume that the execution time for each task in the WfCommons workflow instances is for execution on a compute node of the Ecotype cluster, the cluster with the slowest nodes in our platform configurations (see Table 9.1). In our experiments, task execution time is scaled based on the cluster it is executed on. For the Dahu cluster, this scaling factor is 4.01/3.21 = 1.25 and for Neowise, the scaling factor is 6.48/3.21 = 2.02, i.e., the task execution time is inversely proportional to the core speed. Scaling the execution times by different factors would lead to different data-intensiveness of the workflows (but our workflows already span a spectrum of data-intensiveness).

WfCommons workflow instances do not include information about the execution of workflow tasks on multiple cores, but only give a single execution time t. Due to this lack of information, we assume that this time t was measured for a single-threaded task execution on a single core, and we assume an Amdahl's Law parallel speedup behavior [3]: a task that executes in time t on 1 core executes in time $\alpha \cdot t/n + (1-\alpha) \cdot t$ on n of these cores. We sample α uniformly between 0.5 and 0.9 for each workflow task. This ensures that our experiments include a broad range of parallel multi-core performance behaviors. Note that this model is

[1] This list appears similar to that used in Table 5.1. However, despite sharing some workflow names, the data is acquired for different platform and workflow configurations. Furthermore the data in this chapter is based on production executions of scientific applications with the same name, while the data used in Chapter 5 is from executions of workflow benchmarks generated to be representative of these applications.

general enough to correspond to any notion of a processing elements with a parallel speedup behaviors, such as GPUs for instance. Although our experiments are for platform configurations that comprise multi-core hosts and workflow executions on such platforms, our results are not specific to the underlying processor architecture.

9.4.4 Algorithms

As discussed in Section 9.2.3, scheduling driven by online simulations has been proposed in the context of batch scheduling for HPC clusters. In that specific context, a few classical algorithms are prevalent and lead to a natural portfolio. By contrast, in the context of scientific workflow scheduling a very large number of scheduling algorithms have been proposed over the years, as can be seen in the larger number of survey articles [63, 68, 115, 69, 82, 5, 97, 94, 48, 1]. The vast majority of algorithms proposed for makespan minimization perform list-scheduling [67] for selecting ready tasks and compute resources. A list-scheduling algorithm is invoked whenever there is at least one ready task and at least one available compute resource, and will always schedule a task (i.e., it never decides to leave a compute resource purposely idle). Given the workflow and platform configurations described in the previous sections, list-scheduling consists in:

1. Selecting a ready task using some criterion (C_1);
2. Selecting a cluster with at least one idle core using some criterion (C_2);
3. Selecting a number of cores for the task execution using some criterion (C_3);
4. Scheduling the selected task on the selected cluster using the selected number of cores.

Many options have been proposed in the literature for defining the above criteria, resulting in an enormous number of possible algorithms that could be included in a portfolio. And yet, as discussed in Section 9.3.1, most of these algorithms are not implemented in production WMSs. One of the reasons is that many proposed algorithms are difficult to implement in production systems because they rely on performance models to be designed and implemented by WMS developers. For instance the classic Heterogeneous Earliest Finish Time (HEFT) algorithm prioritizes tasks by an estimate of their earliest finish times. Implementing this algorithm thus requires that a performance model be implemented to compute this estimate. Developing accurate performance models is challenging due to the need to account for a range of complex phenomena (e.g., network contention effects, data locality). The effectiveness of sophisticated scheduling algorithms implemented based on inaccurate performance models is questionable. Furthermore, the information necessary to implement accurate performance models is not necessarily available in the first place. Given all the

above, we build a portfolio of list-scheduling algorithms that can be implemented in current production WMSs solely based on available information about the workflow specification, the platform, and the current state of the execution, without relying on any performance model. This is the same information necessary to instantiate a simulation for SDPS, as explained in Section 9.3.1. Specifically, we consider the following options for each of the above criteria:

- Criterion C_1 (task selection):
  - 0: Pick the task with the largest bottom-level (i.e., tasks on the critical path);
  - 1: Pick the task with the largest number of children tasks;
  - 2: Pick the task with the largest amount of input and output data (in total bytes);
  - 3: Pick the task with the largest computational load.
- Criterion C_2 (cluster selection):
  - 0: Pick the cluster with the fastest cores.
  - 1: Pick the cluster with the larger number of idle cores;
  - 2: Pick the cluster with the most idle compute capacity (number of idle cores multiplied by the core speed);
  - 3: Pick the cluster that holds the largest amount of task input data (in total bytes) in its shared storage;
- Criterion C_3 (number of cores selection):
  - 0: Pick as many cores as possible while ensuring that the task's parallel efficiency is above 90%;
  - 1: Pick as many cores as possible while ensuring that the task's parallel efficiency is above 50%;
  - 2: Pick as many cores as possible.

We denote each algorithm as A_x, where $x = 12 \times C_1 + 3 \times C_2 + C_3$, which gives us 48 different algorithms (A_0 to A_{47}). All above criteria, or variations thereof, have been proposed time and again in the scheduling literature. For instance, the first option for criterion C_1 corresponds to the classic idea of prioritizing tasks on the critical path [45] and the second option corresponds to the classic idea of generating parallelism as quickly as possible [50]. Although many other options could be considered, these 48 algorithms provide us with a sufficiently large and diverse algorithm portfolio (as demonstrated in Section 9.5.1).

We include these 48 algorithms in the portfolio for all experiments and for all our platform/workflow combinations, and the same portfolio is used throughout the whole execution. We note that the portfolio could be reduced for some of these combinations. For instance,

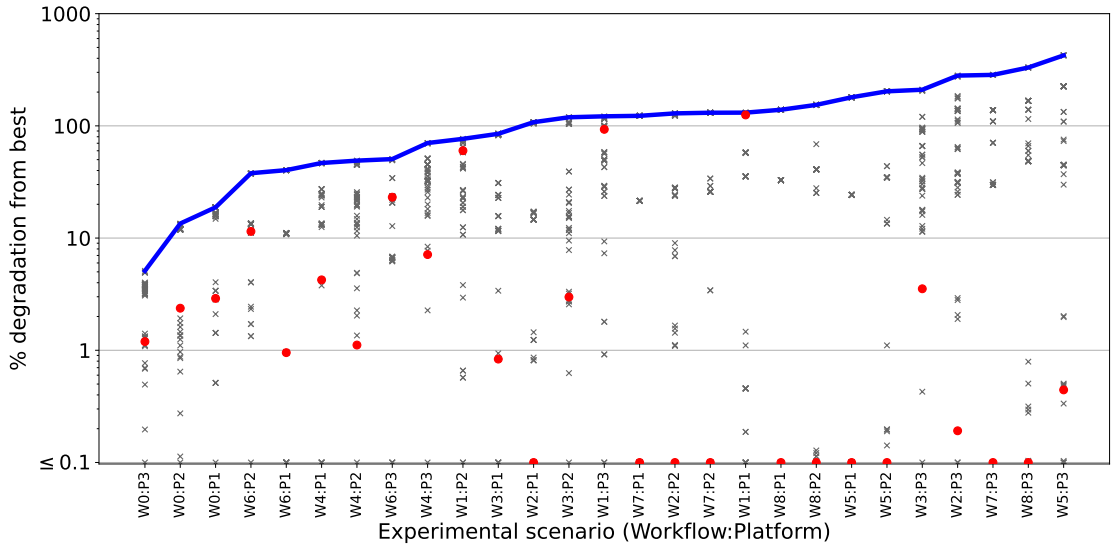


Figure 9.1: Degradation from best (*dfb*) of each algorithm in the portfolio vs. experimental scenarios sorted by increasing maximum *dfb* (shown as a blue solid line). Data points for algorithm A_8, which has the lowest average *dfb*, shown as red dots.

for platform P_1 there is no need for different cluster selection schemes (criterion C_2) since there is a single cluster. But the key idea of portfolio scheduling is that algorithms that perform poorly will simply not be used, and so we always include all 48 algorithms in the portfolio in all experiments.

9.5 Results

In this section we present experimental results so as to evaluate the potential of SDPS. We first show results that demonstrate that our algorithm portfolio is diverse. We then quantify and discuss the impact of simulation error, of simulator sophistication, of dynamic platform behavior, and of the simulation overhead. Together, these results provide answers to the research questions listed in Section 9.3.

9.5.1 Diversity of the Algorithms in the Portfolio

In Section 9.4, we claim that our experimental scenarios (workflow instances and platform configurations) would lead the algorithms in the portfolio to exhibit a range of behaviors. In this section, we verify this claim quantitatively. Figure 9.1 shows, for each experimental scenario, i.e., a workflow and platform combination, the relative difference, in percentage,

between the makespan achieved by each algorithm and that achieved by the best algorithm for this scenario. This percentage is typically termed "degradation from best" or *dfb*. In other terms, if for a particular experimental scenario each algorithm i achieves a makespan m_i, then the *dfb* of algorithm j is defined as:

$$dfb(j) = 100 \times \frac{m_j - \min_i m_i}{\min_i m_i} \ \%.$$

$dfb(j) = 0\%$ means that algorithm j achieves the lowest makespan for that experimental scenario. $dfb(j) = 100\%$ means that algorithm j achieves a makespan twice as long as the makespan achieved by the best algorithm.

In Figure 9.1, the scenarios are sorted by increasing value of the maximum *dfb* value over all algorithms. This maximum *dfb* value ranges from 5.11% to 424.80%, and is above 100% for 16 of the 27 experimental scenarios. This means that there is more than a 2x difference between the makespan achieved by the best algorithm and that achieved by the worst algorithm for that experimental scenario for more than 50% of our experimental scenarios. We conclude that our experimental scenarios are sufficiently diverse to highlight the diversity of our 48 scheduling algorithms.

Although the above results indicate diversity, one may wonder whether some (or perhaps just one?) algorithm is always best. In this case, one should just use that algorithm across the board and a portfolio of algorithms is not useful in practice. The answer to Q#1 in Section 9.3.3 would then be that SDPS has little potential improvement over the traditional one-algorithm approach. Computing each algorithm's average *dfb* over all experimental scenarios, we find that algorithm A_8 achieves the lowest average *dfb* at 12.63%. Algorithm A_8 prioritizes tasks with largest bottom-level ($C_1 = 0$), selects the cluster with most idle compute capacity ($C_2 = 2$), and uses as many cores as possible on a compute node ($C_3 = 2$). While algorithm A_8's average *dfb* is relatively low, it is not always a good choice. It happens to be the best (or within 1% of the best) choice for only 14 of our 27 scenarios. It has a *dfb* higher than 10% for 5 of the remaining 13 scenarios, and as high as 125.16% for the W_1:P_1 experimental scenario, as seen in Figure 9.1. We conclude that no single algorithm is best. Although algorithm A_8 is the best on average it can perform relatively poorly for some experimental scenarios.

In all that follows, we consider as our competitor a WMS that implements and uses algorithm A_8 as its only scheduling algorithm. Since algorithm A_8 has the best average *dfb*, it corresponds to the best choice a WMS developer could make if asked to pick one algorithm

to implement in their system, at least in the scope of our experimental settings. It is unclear how a developer would identify this algorithm, short of conducting an extensive experimental study. In fact, they could very well pick another algorithm, in which case all results hereafter would be more favorable (and often drastically so) for SDPS. For simplicity, we call this competitor OneAlg.

9.5.2 Impact of Simulation Error

With 100% accurate simulations SDPS would necessarily pick the best algorithm for each experimental scenario. In this section, to answer Q#2 in Section 9.3.3, we quantify the sensitivity of SDPS to simulation inaccuracy.

Simulation Error Injection Method

To the best of our knowledge, there is no model or characterization of the simulation error behavior of simulators of parallel and distributed computing platforms and applications. But we note that simulation error is not fully random as it typically stems from simulation models under- or over-estimating the performance delivered by hardware resources when executing application activities. That is, for each such resource, the simulation suffers from some constant bias. This bias can be due to a simulation model being inherently biased or to an incorrect instantiation of the model's configuration parameters (e.g., due to limits of the simulation calibration approach).

When SDPS performs its round of simulations we introduce simulation error by injecting random perturbations in the platform's resource speeds, i.e., each cluster's internet bandwidth, storage system bandwidth, and core compute speed. If the actual value of a speed parameter is x, in the simulation it is set to value $\mathcal{U}(\max(0, x \times (1-e)), x \times (1+e))$, where $\mathcal{U}(a, b)$ denotes the uniform random distribution on the (a, b) open interval and e denotes the magnitude of the error range. Since simulations are conducted with erroneous values of these metrics, they report erroneous makespans, based on which a scheduling algorithm is selected. Note that even small simulation errors can cause drastically different scheduling decisions (e.g., pick a different cluster on which to execute a task). As e increases there is a higher probability for SDPS to select an algorithm that is not (and is possibly much worse than) the best algorithm for the upcoming application execution.

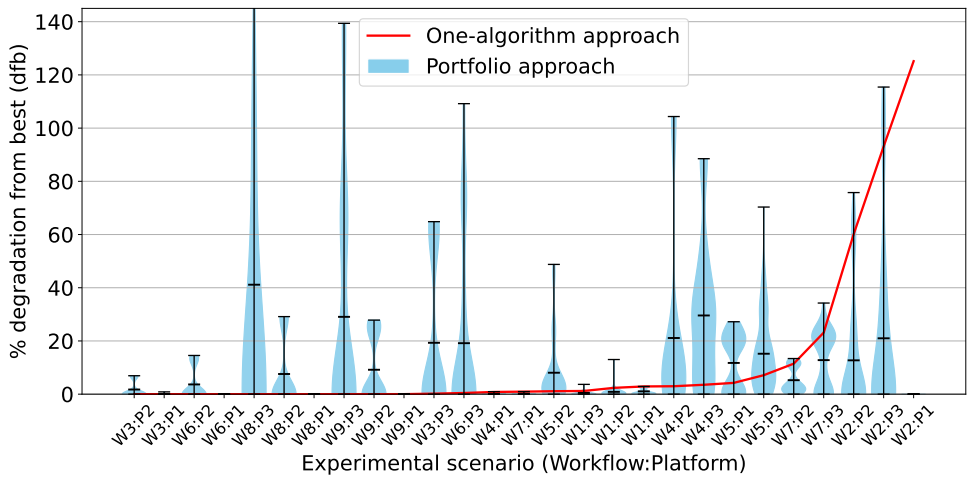


Figure 9.2: *dfb* vs. experimental scenarios, for simulation error magnitude $e = 1.0$.

Using a Single Round of Simulations

Let us first assume that algorithm selection is based on a single round of simulations at the onset of the workflow execution, and that the selected algorithm is then used throughout the execution. We conducted experiments for our 9 workflow instances and 3 platform configurations for $e \in \{0.1, 0.2, \ldots, 1.0\}$, with 100 trials (i.e., 100 different random number generator seeds) for each e value.

Figure 9.2 shows *dfb* results for $e = 1.0$ (high simulation error) for all $9 \times 3 = 27$ experimental scenarios. The scenarios are sorted by increasing *dfb* for OneAlg. Results for OneAlg are shown as a single line since its performance is not affected by simulation error (as it does not use simulation). Results for SDPS are shown as a violin plot for each experimental scenario, which depicts the distribution of 100 data points. The bottom horizontal bar of the violin plot shows the minimum value and the top horizontal bar shows the maximum value, while the middle horizontal bar shows the average. As seen in Figure 9.1, OneAlg is the best approach for many of these scenarios. However, for some scenarios, seen on the right of the horizontal axis in Figure 9.2, its *dfb* can be large.

The main observation from Figure 9.2 is that even with high simulation error SDPS still performs relatively well for many scenarios. Its average *dfb* is below 5% for 12 of the 27 experimental scenarios, and above 20% for only 5 of them. Although there are cases in which SDPS is largely outperformed by OneAlg, the converse is also true. For some scenarios the *dfb* distribution of SDPS has a high maximum value. These correspond to cases in which simulation error causes SDPS to pick an algorithm that does not perform well. We note

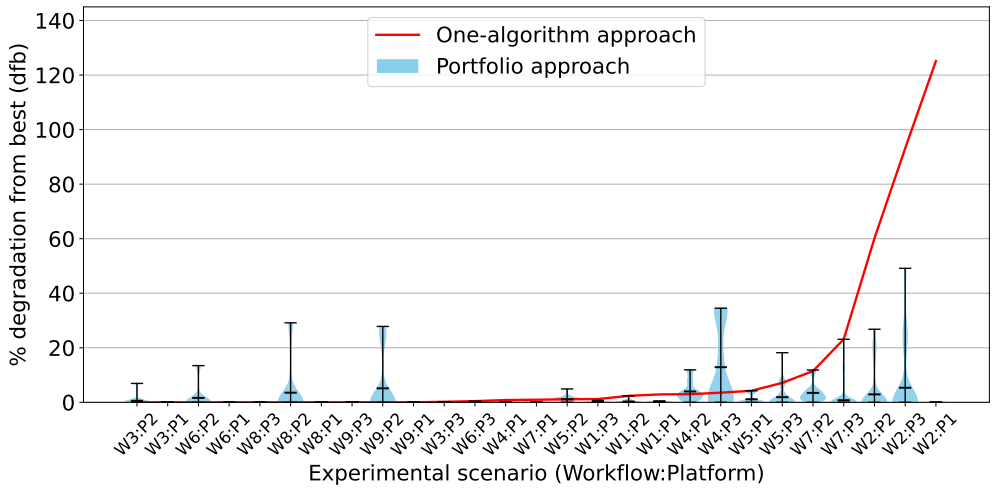


Figure 9.3: *dfb* vs. experimental scenarios, for simulation error magnitude $e = 0.3$.

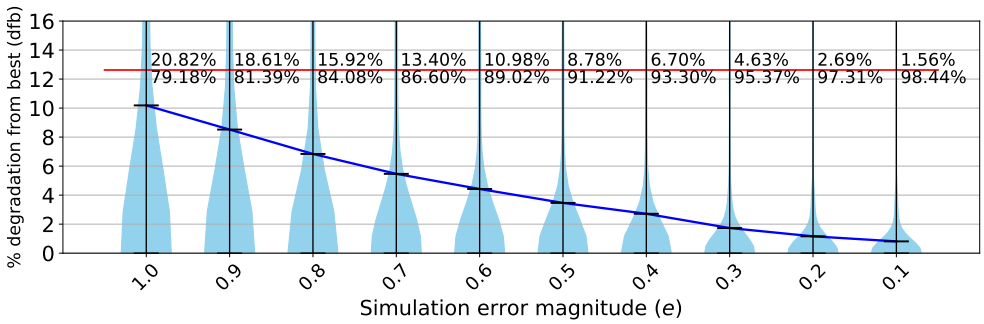


Figure 9.4: *dfb* vs. simulation error (e). Percentages denote fractions of SDPS values above/below OneAlg's value.

that these cases occur mostly for scenarios with the 3-cluster platform configuration (P_3), which has the largest scheduling decision space. Conversely, we see that for the 1-cluster platform configuration (P_1), SDPS is more tolerant to simulation error.

Figure 9.3 shows results for $e = 0.3$. Expectedly, the results are vastly improved, with SDPS's average *dfb* at most 18.54%. Overall, SDPS is equivalent with or vastly preferable to OneAlg.

Figure 9.4 shows the distribution of the *dfb* values achieved by SDPS over all experimental scenarios for all values of e. As expected, SDPS's *dfb* improves as e decreases, with the average converging to zero. The figure also depicts the *dfb* of OneAlg, which is 12.63%, as a horizontal line. We see that SDPS leads to improvements over OneAlg with relatively high probability even when simulation error is 100%. At simulation error 0.5 or lower, SDPS

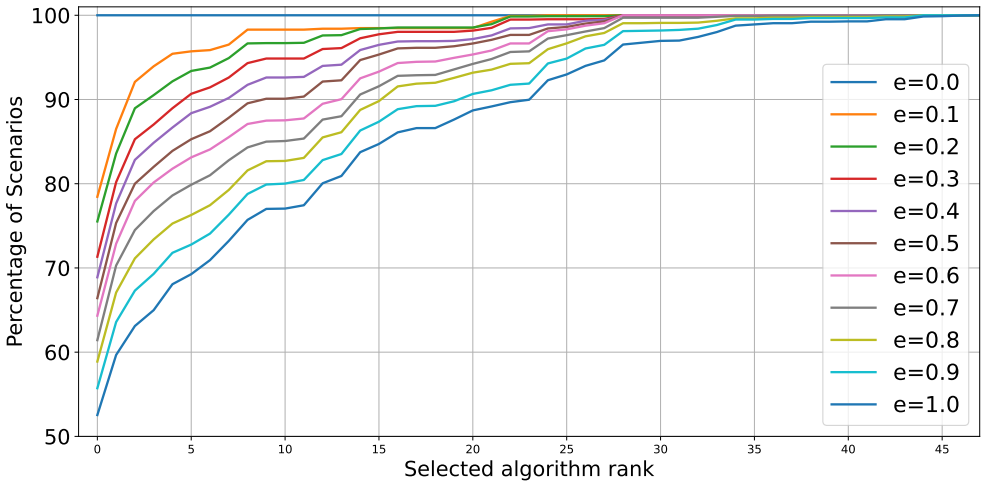


Figure 9.5: Cumulative distribution of the rank of the algorithm selected by SDPS for each simulation error magnitude value (e).

outperforms OneAlg in more than 90% of the cases.

To better explain the previous results, we compute the rank of the algorithm picked by SDPS in the portfolio (i.e., algorithm with rank 0 is the best, algorithm with rank 1 is the second-best, etc.). Figure 9.5 shows the cumulative distribution of the rank of the algorithm selected by SDPS over all experimental scenarios, for all values of e. That is, a point at coordinate (x, y) in the plot means that SDPS selected an algorithm with rank at least x for y% of the experimental scenarios. Even with high error ($e = 1.0$), SDPS selects the best algorithm in more than 50% of the cases, and one of the top 5 algorithms in about 70% of the cases. However, we see that algorithms with high rank (i.e., the worst algorithms in the portfolio) are still selected occasionally, which explains the high maxima for some of the violin plots shown in this section. Expectedly, results improve as e decreases. For instance, for $e = 0.3$, SDPS selects the best algorithm in more than 70% of the cases and a top-5 algorithm in more than 90% of the cases.

We conclude that even with high simulation error SDPS rarely picks a "bad" algorithm, and as a result compares favorably to OneAlg.

Simulation Error Mitigation via Another Round of Simulations

Chapters 5-7 demonstrate that simulators in various domains can be calibrated to ∼20% error. Others simulators have been reported to achieve simulation error well below 20% [17, 53]. Given the results in the previous section, such accuracy levels would have little negative

impact on the efficacy of SDPS. While it is certainly possible to develop low-error simulators, doing so is not a given (our review of the state of the art in Section 3.5.3 shows that many simulators are not calibrated). It is thus reasonable to expect that many simulators could exhibit relatively high error in practice.

One approach to reduce the impact of simulation error on the effectiveness of SDPS is to perform *simulation error mitigation* (Q#3). The runtime system can keep a record of the simulated execution for the algorithm that ends up being selected after a round of simulations, and then compare this execution to what actually happened in the real execution. The goal is to identify sources of simulation error, and adjust the instantiation of the simulator before the next round of simulations. For instance, comparing the real and the simulated execution could reveal that some network bandwidth has been underestimated by some factor. A new round of simulations could be conducted after applying a correction factor to that bandwidth, so as to pick a possibly different scheduling algorithm for the rest of the application execution.

To answer Q#3 in Section 9.3.3 we repeat the experiments in the previous section but assuming that simulation mitigation is performed. That is, after a task completion occurs and 10% of the application's total work has already been executed, then a second round of simulations is conducted with error $e' < e$. Assuming that the actual value of a platform's performance metric is x, in the first round the simulations use value $x' = \mathcal{U}(\max(0, x \times (1 - e)), x \times (1 + e))$. Recall that $\mathcal{U}(a, b)$ denotes the uniform random distribution on the (a, b) open interval. In the second round, a value $x'' = x + (x' - x) \times (e'/e)$ is used, which is closer to x by a factor $e'/e < 1$. In this manner, we can evaluate how SDPS fares for ranges of initial and mitigated errors.

Figure 9.6 is similar to Figures 9.2 and 9.3, but shows results for $e = 1.0$ and $e' = 0.3$. Error mitigation leads to improvements. Specifically, over the 27 experimental scenarios, the maximum *dfb* was improved for 19 scenarios and the average *dfb* was improved for 22 scenarios. In terms of average *dfb* across all scenarios, SDPS achieves a *dfb* of 10.18% with $e = 1.0$ and no mitigation, and a *dfb* of 4.48% with $e = 1.0$ mitigated down to $e' = 0.3$, while with an initial error $e = 0.3$ SDPS achieves an average *dfb* value of 1.73%.

One observation when comparing Figure 9.6 to Figure 9.2, which holds for all other e and e' values, is that error mitigation is effective only for some workflows. To illustrate this behavior, let us consider the execution of workflows W_6 and W_8 on platform P_3 (results are similar, but not as pronounced, for P_1 and P_2 because the scheduling decision space is smaller). Figure 9.7 shows average *dfb* results when SDPS is used to execute workflow W_6.

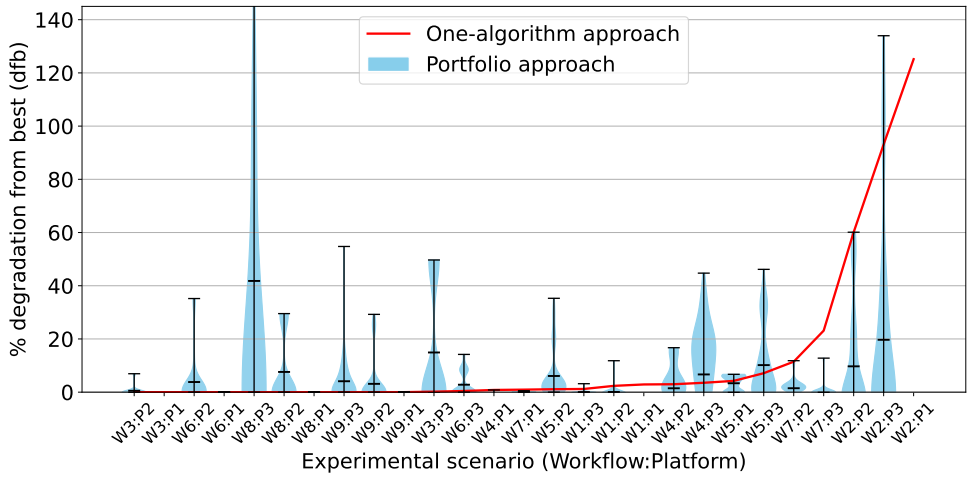


Figure 9.6: SDPS's *dfb* vs. experimental scenarios, for simulation error magnitude $e = 1.0$ and mitigated error magnitude $e' = 0.3$.

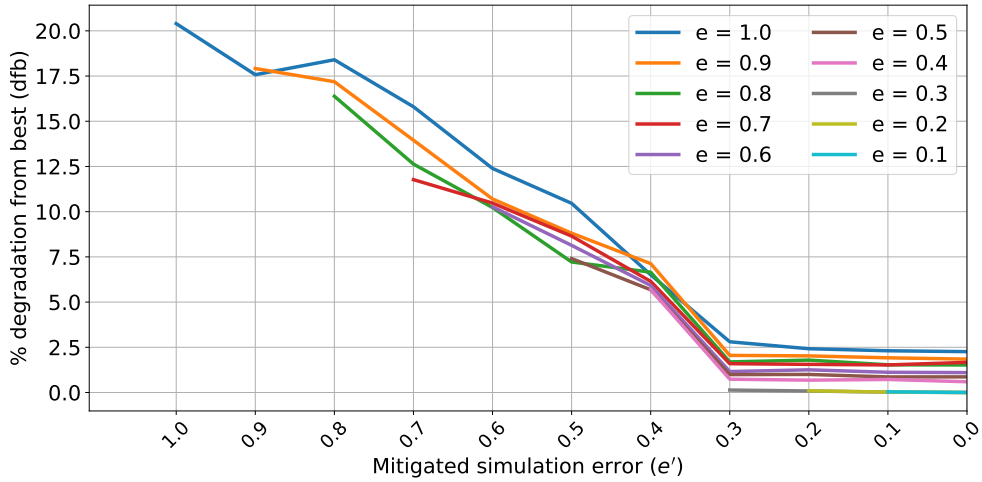


Figure 9.7: SDPS's average *dfb* vs. mitigated error (e') for each initial error magnitude (e), for workflow instance W_6.

The horizontal axis shows e' values (i.e., mitigated error) and each curve is for a different e value (i.e., initial error). Each curve shows a decreasing trend until $e' = 0.3$, at which point the curve flattens out. Furthermore, the curve for a particular e value is for the most part below the curves for higher e values. For this workflow, error mitigation is useful but, expectedly, a lower initial error value e is preferable. Trends are similar for the W_1, W_3, W_4, W_5, W_7, and W_9 workflow instances, but with different slopes and final plateaux.

Figure 9.8 shows results for the W_8 workflow (results for W_2 are similar). For this

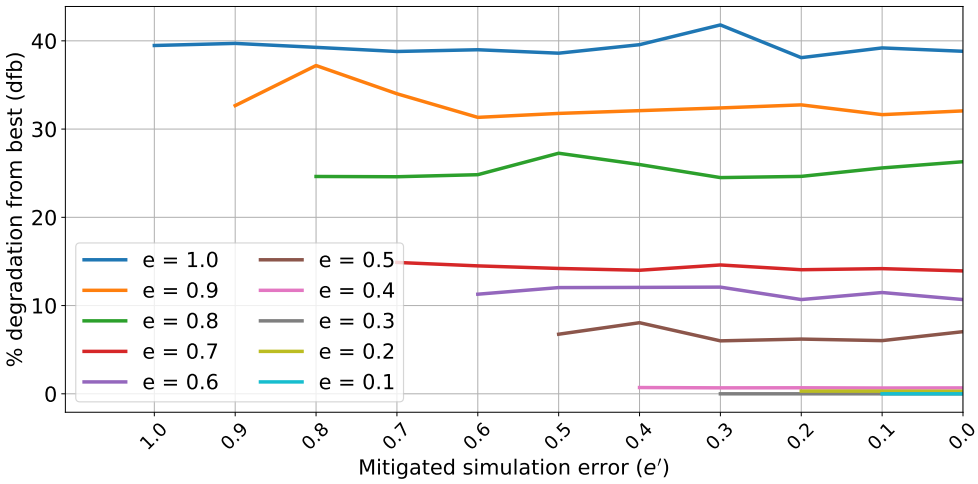


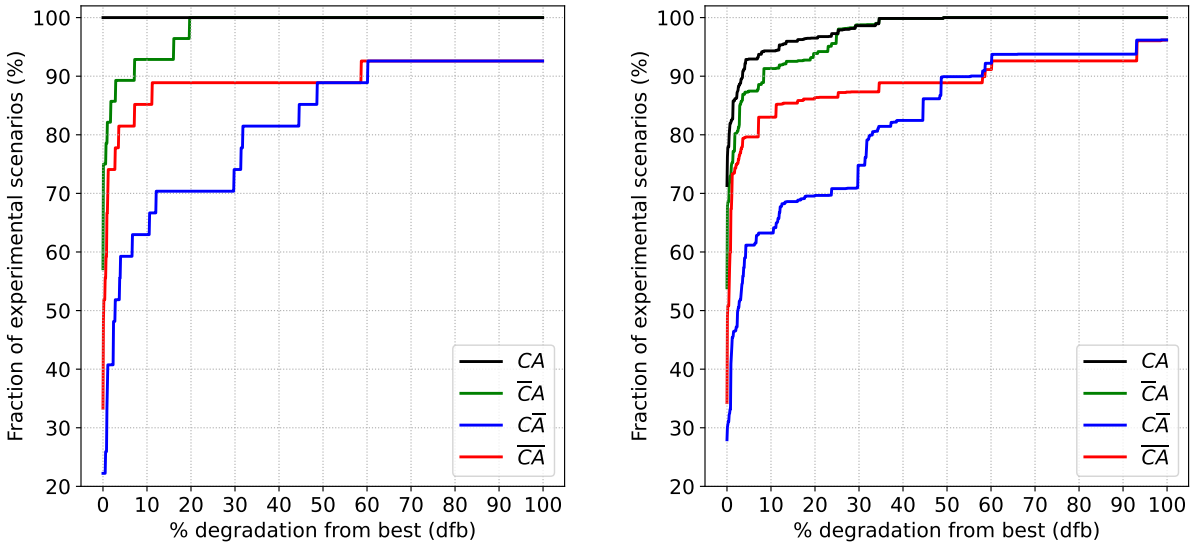
Figure 9.8: SDPS's average *dfb* vs. mitigated error magnitude (e') for each initial error (e), for workflow instance W_8.

workflow error mitigation provides little improvement. This is due to the structure of the workflow, which consists of a first level of compute-heavy tasks that account for the majority of the total work. That is, once initial scheduling decisions have been made for these tasks, scheduling decisions for subsequent tasks have almost no effect. Recall that we perform error mitigation only after a task completion occurs. The rationale is that, in real-world systems, the data required for simulation error reduction can only be gathered once a task has been completed. For this workflow, by the time the first task completion occurs, more than 95% of the total work has already been scheduled. The effectiveness of SDPS is thus entirely driven by the initial simulation error.

We conclude that simulation error mitigation is useful provided it can be performed before the bulk of the total work has begun executing.

9.5.3 Impact of Simulator Sophistication

To answer Q#4 in Section 9.3.3 we quantify the extent to which SDPS is sensitive to the sophistication of the simulator. In particular, we want to determine whether a simulator that employs simplistic, or even naive, simulation models can still be useful for SDPS. To answer this question we repeat experiments described in previous sections but we disable features of our simulator so as to reduce its level of sophistication. As seen in Chapter 8, there are many ways to increase/decrease the level of sophistication of a simulation. In this section we consider two options. First, we disable the simulation of network and I/O contention since



(a) error magnitude $e = 0.0$

(b) error magnitude $e = 0.3$

Figure 9.9: Cumulative *dfb* distribution for all experimental scenarios for two simulation error magnitude values.

many simulators neglect the simulation of contention effects on network and I/O devices. They do so because simulating contention accurately is challenging [112]. Furthermore, simulators that do not simulate contention are often used in the context of scheduling, since ignoring contention greatly simplifies the design and evaluation of scheduling algorithms [35]. Second, we disable the simulation of realistic multi-core parallel speedup. Our simulator uses an Amdahl's law parallel speedup model for estimating the execution of a workflow task on multiple cores. This model requires benchmark information about the workflow tasks, which is not always readily available. Instead, a less sophisticated option is to assume that every task has 100% parallel efficiency.

Together these 2 simplifications let us explore 4 possible simulators to use for scheduling decisions, each at a different level of sophistication, which we denote as CA, $C\overline{A}$, $\overline{C}A$, $\overline{CA}$, where C denotes contention, A denotes Amdahl's law, and a bar on the letter means that feature is disabled in the simulator. So, for instance, $\overline{C}A$ denotes our simulator with no contention simulation but with Amdahl-based parallel speedup simulation. All the results in previous sections were for the CA option.

Figure 9.9 shows the cumulative *dfb* distribution across all experimental scenarios for two simulation error magnitudes, $e = 0$ and $e = 0.3$. A point at coordinate (x, y) means that for

Table 9.3: Percentage of *dfb* values below 10% for each workflow when using each of the four simulation sophistication options, for simulation error magnitude $e = 0.3$.

Workflow	CA	$\overline{C}A$	$C\overline{A}$	$\overline{CA}$
W_1	100.00	100.00	100.00	100.00
W_2	100.00	100.00	0.00	0.00
W_3	100.00	100.00	66.67	100.00
W_4	100.00	66.67	66.67	100.00
W_5	100.00	66.67	33.33	100.00
W_6	100.00	100.00	66.67	100.00
W_7	100.00	100.00	100.00	66.67
W_8	100.00	100.00	66.67	100.00
W_9	100.00	100.00	66.67	100.00

$y\%$ of the experimental scenarios a *dfb* value better than x is achieved. That is, the faster a curve approaches the $y = 100\%$ line, the better. Simulation error magnitude $e = 0.3$ was selected as a reasonable level of error in a calibrated simulator, but results are similar for all other e values (full results available on GitHub [88]).

Expectedly, using the CA simulator leads to the best results overall. Using the $\overline{C}A$ simulator leads to results only slightly worse, which indicates that simulating contention, for our experimental settings, is not critical for SDPS to be effective. This is likely because our workflow instances do not correspond to highly data-intensive scenarios. Using simulators $C\overline{A}$ or $\overline{CA}$ leads to noticeably worse results. Interestingly, using $\overline{CA}$ often outperforms using $C\overline{A}$ despite it being less sophisticated. This is because the ranking of candidate scheduling algorithms for a particular scenario strongly depends on the communication-to-computation ratio of the execution. $\overline{CA}$, unlike $C\overline{A}$, underestimates both communication times (because it ignores contention) and computation times (because it assumes perfectly parallelizable workflow tasks). It thus ends up with a more accurate ranking of the candidate scheduling algorithms than the more sophisticated $C\overline{A}$.

The results in Figure 9.9 are aggregated over all workflows, but different workflows show different trends. Table 9.3 shows, for each workflow and for each simulator sophistication option, the percentage of experimental scenarios for which a *dfb* value under 10% is achieved, for a simulation error magnitude $e = 0.3$. The main observation is that for some workflows,

like W_1, simulator sophistication has no impact, but for others, like W_2, using more sophisticated simulators yields dramatic improvement. In the case of W_2, simulating parallel speedups accurately is critical. For some workflows, using a less sophisticated simulator can yield marginally better results than using the most sophisticated simulator (CA). This is because particular simulation errors can occasionally cause a less correct simulator to produce a more accurate ranking of the candidate scheduling algorithms. But overall, using the most sophisticated simulator yields low *dfb* values across all workflows.

Expectedly, a higher level of simulator sophistication is better. However, lower levels of simulator sophistication can still yield good results for SDPS. In the scope of our experimental settings, for instance, not simulating network and I/O contention typically leads to only marginal makespan degradation. Unfortunately, the needed level of sophistication depends on the specifics of the platform and application scenarios. In our results, a less sophisticated simulator can lead to almost no degradation for some workflows and to large degradation for others. If these specifics are known in advance, it may be possible to determine a priori the needed level of sophistication needed for SDPS to be sufficiently effective for a particular use case. But, as discussed in Chapter 8, determining the level of sophistication a priori is challenging. A better approach is to collect ground-truth execution data for a representative set of platform and application scenarios, and use our proposed automated calibration approach to pick the level of sophistication as done in that chapter.

9.5.4 Adapting to Dynamic Platform Changes

All experiments in the previous sections assume that the workflow executes on dedicated resources reserved for that execution. While this is the typical setting for production executions of workflows, there are other use cases in which the resources allocated to a workflow's execution can fluctuate (e.g., on clouds where virtual machine instances can be created and can expire dynamically, on batch-scheduled HPC platforms when using pilot jobs [110]). SDPS can then be used periodically through the execution to adapt to these changes and select different scheduling algorithms for different platform conditions.

There is no general and/or accepted model for dynamic compute resource availability in distributed computing platforms. Nevertheless, we have conducted experiments with a simple dynamic resource availability model as an attempt to answer Q#5 in Section 9.3.3. Our main finding is that while significant improvements from these adaptations can sometimes be achieved, the results are workflow-dependent (similarly to the findings in Section 9.5.2). Appendix B provides full details on these experimental results, which are necessary of limited

Table 9.4: Simulated makespan, simulation time, ratio thereof, and peak memory footprint when simulating the execution of each workflow on platform P_3 with algorithm A_8. Results obtained on a 2.80GHz core (Intel Xeon Gold 6242 CPU), averaged over 10 samples.

Workflow	Simulated Makespan (sec)	Simulation Time (sec)	Ratio	Peak Memory Footprint (MB)
W_1	10573.78	0.57	18457.90	29.82
W_2	117.16	0.55	213.39	32.83
W_3	455.60	3.67	124.11	102.06
W_4	106.53	2.92	36.43	86.80
W_5	265.60	1.66	159.79	47.40
W_6	113.49	0.26	44.96	25.92
W_7	10347.80	0.83	12,396.92	51.45
W_8	602.17	0.03	17,295.35	19.93
W_9	90.99	0.26	343.77	26.18

scope due to the particular resource availability model used to obtain them.

9.5.5 Simulation Overhead

To answer Q#6 in Section 9.3.3, we quantify simulation overhead and discuss possible approaches to mitigate it. Recall that in this work we consider that simulations are executed on the host on which the WMS itself executes. For SDPS to be practical, simulation time must be low relative to the makespan since a round of simulation needs to be executed at the onset of the execution. Furthermore, additional rounds of simulation may be useful throughout the execution, for simulation error mitigation purposes or for adapting to dynamic changes of the workflow and/or platform. In practice, simulations can be executed concurrently with the application so that the simulation overhead is fully hidden [78]. However, a high simulation overhead can still be harmful as it delays the time at which the algorithm selection decision is reached.

Table 9.4 shows results obtained when simulating the full execution of each of our workflow instances on platform configuration P_3 using algorithm A_8 (the best algorithm on average). We picked platform configuration P_3 as, among the three platform configurations considered in this chapter, it leads to the highest ExplorationSimulator execution times.

We find that all peak memory footprints are low (at most 102.06MB) and that, for most workflow instances, the ratio of simulated makespan to simulation time is large. This is because for discrete-event (as opposed to discrete-time) simulation, computational complexity depends on the number of events to simulate and not on the lengths of time being simulated. The lowest ratio is for the W_4 workflow, for which the ratio is "only" 36.43x. Overall, we find that in general the time to simulate the execution is small, and often negligible, when compared to the real execution.

The results in Table 9.4 are for the simulation of one algorithm, but SDPS runs one simulation per algorithm in the portfolio. These simulations are independent of each other and can be executed concurrently on multiple cores, which is feasible due to the low memory footprint. For instance, running 48 concurrent simulations for workflow W_3, which causes the largest memory footprint, requires less than 5GB of RAM. Running these 48 simulations concurrently on a machine with a 2.40GHz Intel Xeon Gold 6240R 48-core processors takes 10.03 seconds, while running only the slowest of these simulations (simulations take different amounts of time depending on the scheduling algorithm) takes 8.27 seconds. We conclude that although many simulations need to be executed, they can be executed concurrently on multiple cores with high parallel efficiency.

One option to reduce simulation time is to not fully simulate the execution to completion, but preliminary work [19] showed that its effectiveness is highly workflow-dependent. For some workflows simulating only a small fraction of the upcoming execution has almost no negative impact on SDPS, but for others the negative impact is large. This is because different workflows have different task- and data-dependency structures, and in some cases the impacts of initial scheduling decisions are not felt until the later phases of the execution.

If the algorithm portfolio is too large, one possibility is to prune the set of candidate algorithms. For instance, in [32] it is proposed that algorithms be placed in different categories depending on their past simulated performance, and that a bounded amount of simulation time be allocated to each category. This approach could be used in our context, with the caveat that the performance of an algorithms in earlier phases of the execution may be misleading, as discussed in Section 9.3.2.

9.6 Conclusion

In this chapter, we have assessed the potential merit of using simulation-driven portfolio scheduling (SDPS) in runtime systems that automate the execution of scientific workflow

applications on parallel and distributed computing platforms. Our results show that SDPS outperforms the one-algorithm approach, even when this approach happens to use the algorithm that performs best on average across all experimental scenarios considered in this research. Crucially, SDPS still performs well in the presence of relatively large simulation error, e.g., larger than what we achieved using automated simulated calibration in previous chapters. Furthermore, simulation error mitigation at runtime can be effective. We also found that, for some execution scenarios, even unsophisticated simulators can be used by SDPS and achieve good results. Finally, simulation overhead is sufficiently low for SDPS to be used in practice.

In our results we have compared SDPS to the best possible choice a runtime system developer could make for implementing the one-algorithm approach in the context of our experimental scenarios, i.e., pick algorithm A_8. The main motivation for this research is that it is not clear how the developer could identify this algorithm in practice (short of conducting a full experimental campaign as done in this chapter). Were the developer to pick the median algorithm, algorithm A_{22}, which has a relatively low average *dfb* at 29.38% (A_8 is at 12.63% and the worst algorithm is at 124.21%), all results presented in Section 9.5 would be improved significantly. For instance, with high simulation error $e = 1.0$ and no error mitigation, SDPS would outperform the one-algorithm approach on average for 25 of the 27 experimental scenarios (as opposed to only 9 of them as seen in Figure 9.2).

We conclude that scheduling driven portfolio scheduling is a viable method to obviate the challenge of picking a particular scheduling algorithm to implement in a WMS, and thus has the potential to resolve the disconnect between scheduling research and scheduling practice in the context of scientific workflows.

Part IV

CHAPTER 10
CONCLUSION

In this chapter, we first summarize the main contributions of this work and then outline possible directions for future work.

10.1 Summary of Contributions

The main hypothesis of this dissertation, as stated in Chapter 1, is: Automated calibration of PDC simulators is feasible, can increases simulation accuracy significantly when compared to current practice, and can unlock new simulation-driven use cases in the field. To verify this hypothesis, in the previous chapters we have examined and sought to answer four main research questions (RQs):

> **RQ1:** Can an automatic simulation calibration methodology be developed and used to increase the accuracy of PDC system simulators?
>
> **RQ2:** How much ground-truth data is required to perform sufficiently precise calibrations of PDC system simulators?
>
> **RQ3:** Does the ability to calibrate PDC system simulators make it possible to quantify the impact of the implemented level of simulation sophistication on simulator accuracy?
>
> **RQ4:** Can high-accuracy (likely due to being well-calibrated) PDC system simulators be used in to inform application scheduling decisions made by runtime systems used to run scientific workflow applications.

For each of the above RQs, we summarize our findings hereafter:

- We have found that the answer to **RQ1** is Yes. We were able to define and implement a general methodology for automated simulation calibration that, when compared to the current state-of-the art, increases simulation accuracy significantly.
- We have found that the answer to **RQ2** depends on the target use case. We investigated this question with use cases in three different domains: scientific workflows, MPI benchmarking, and high-energy physics platform. While a precise ground-truth

data selection methodology that would apply across a large number of use cases is likely impossible, the results we have obtained provide the following guidance: Ensure ground-truth data is sufficiently diverse, recognize that more ground-truth data is not necessarily better overall, and ensure that ground-truth data is easily comparable to simulator output.

- We have found that the answer to **RQ3** is Yes. We applied our automated calibration methodology to the same three case studies, but using simulators implemented at various levels of sophistication. In doing so we found that our methodology makes it possible to draw conclusions regarding which level of sophistication is sufficient for different simulated components.

- We found that the answer to **RQ4** is Yes. We investigated the use of simulation-driven scheduling in workflow management systems, by which the system simulates executions of itself at runtime to pick the most promising algorithm from a portfolio of potential scheduling algorithms. We demonstrated that this scheduling approach is feasible in practice and can provide significant performance improvements when compared to the state-of-the-art scheduling approach in workflow management systems.

In Section 3.5.3 we painted a rather bleak picture of the current state of the art of simulator calibration, which raises questions about the accuracy of simulation results in the field of PDC. With the automated methodology and open-source Simcal calibration framework presented in Chapter 4, PDC researchers and practitioners should now be able to calibrate their simulators to higher levels of accuracy. Our hope is that this work will positively impact both simulation-driven research and simulation-driven practice in the field of PDC, ultimately unlocking new application of (accurate) PDC simulation beyond those explored in Chapters 8 and 9.

10.2 Future Work

There are several potential directions for future work along two main axes: (i) improvements to the simulation calibration process and (ii) applications of automated simulation calibration. We discuss specifics for each axis hereafter.

10.2.1 Simulation Calibration Improvements

Several directions could be pursued to further improve our automated simulation calibration process. Specifically:

Optimization algorithm – While we have evaluated several optimization algorithms as part of our automated simulation calibration process, many other algorithms could be considered. Specifically, it may be worthwhile to investigate metaheuristics such as genetic algorithms, simulated annealing, particle swarm optimization, as well as generalized pattern search algorithms.

Loss function – For each of our case studies we have considered several possible user-defined loss functions. While more user-defined loss function could always be considered and analytically defined, an interesting direction is to also consider machine-learning based loss functions that are trained to quantify the loss between two datasets, either using raw datasets, or aggregating values from other loss functions.

Use of ground-truth data – Currently our automated calibration method samples all training ground-truth data for each parameter. This typically requires running the simulator once for each item in the training data, which can be time-consuming. Calibration could be greatly accelerated, and possibly improved, by running the simulator only for relevant ground-truth data. Determining what ground-truth data is relevant is of course non-trivial. That said, Bayesian optimization, which we have used as an optimization algorithm, operates by using a internal model to explore potential new areas in a parameter space and exploit known "good" areas. It may be possible to adapt this method to determine the ground-truth data with the most potential impact on a set of parameters, or even the combination of ground-truth and parameters that will reveal the most new knowledge about the calibration problem as a whole.

Synthetic calibration validation – Currently our methodology uses an arbitrary calibration for generating a synthetic dataset. There is a possibility that using different calibrations as a basis for constructing the synthetic dataset could lead to improved results. If this is the case, one would need to design a method for selecting a proper calibration as a basis for the synthetic dataset.

Simcal improvements – Several improvements could be made to the Simcal framework. For instance, Simcal currently lacks multi-machine support, meaning that the calibration can only run on a single, multi-core machine. Another improvement would be implementing tools for calibration validation in Simcal, for which it currently has only very limited support. Finally, several of the above future work directions would result in new features to be implemented as part of Simcal.

10.2.2 Applications of automated simulation calibration

The number of applications of calibrated simulators in general is potentially enormous. More specific to the contributions of this dissertation, several direct applications of our automated calibration approach would be worth investigating in the future.

SDPS on a real Workflow Management System (WMS) – In this dissertation, the contributions in the context of SDPS are limited to simulation only. A short-term future development, on which we have already embarked, is to implement a WMS that applies SDPS while running a real workflow on real hardware. This requires automated simulation calibration both for the initial simulator calibration, and also for on-the-fly error mitigation.

Further research using SDPS – A clear future work direction is the investigation of simulation forensics techniques for detecting and mitigating simulation error at runtime, which may require on-the-fly re-calibration techniques. Another future direction is using SDPS for optimizing other application execution metrics (e.g., energy consumption, monetary cost) and for addressing notoriously difficult multi-objective scheduling problems that consider multiple such metrics (e.g., satisfying multiple QoS requirements, minimizing makespan given an energy consumption budget).

Simulator Machine Learning Surrogate – Machine learning models are known to take a large amount of expensive data to train, likely more than is required to calibrate a simulator on the same problem. However, machine learning models run faster. It may be possible to build and calibrate a simulator for a use case, use that simulator to construct a larger but less expensive simulated ground-truth dataset, and then train a machine learning model on this simulated dataset. The goal is to create a model that learns the (well-calibrated) simulator's behavior and can be used as a fast approximation of the simulator.

Transferring calibrations between levels of sophistication – Given a calibration obtained for a simulator A implemented at a particular level of sophistication, we have found that it is not possible to apply that same calibration directly to a simulator B that implements a different level of sophistication. However, it may be possible to use the calibration for simulator A as a starting point to find a good calibration for simulator B, or at least reduce the time to do so.

More automated calibration case studies – Building on the lessons learned in the process of conducting our three case studies, a clear future work direction is to apply and validate our simulation calibration methodology to other simulators in the PDC domains we have already targeted as well as to simulators in other PDC domains. The objective would be to provide recommendations (loss functions, ground-truth data, optimization algorithm) that generalize to classes of simulators within each domain. Similarly, a compelling future work direction is to quantify the impact the simulator sophistication in a large number of case studies. The, perhaps overly ambitious, goal would be to find a way to predict the level of sophistication at which a simulator should be implemented given the kind of system is attempts to simulate.

CHAPTER 11
SUMMARY OF PUBLISHED WORK

[76] Jesse McDonald, Yick-Ching Wong, Kshitij Mehta, Frederic Suter, Rafael Ferreira Da Silva, Loïc Pottier, Ewa Deelman, and Henri Casanova. Determining levels of detail for simulators of parallel and distributed computing systems via automated calibration. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1452–1463, New York, NY, USA, 2025. Association for Computing Machinery

[73] Jesse McDonald, John Dobbs, Yick Wong, Rafael Ferreira da Silva, and Henri Casanova. An exploration of online-simulation-driven portfolio scheduling in workflow management systems. *Future Generation Computer Systems*, 161, 2024

[118] Yick Ching Wong, Jesse McDonald, Frederic Suter, Kshitij Mehta, and Henri Casanova. Automated calibration of a simulator of mpi application executions. In *Proceedings of the 2024 IEEE 20th International Conference on e-Science (e-Science 2024)*, Osaka, Japan, 2024

[74] Jesse McDonald, Maximilian Horzela, Frédéric Suter, and Henri Casanova. Automated calibration of parallel and distributed computing simulators: A case study. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1026–1035. IEEE, 2024

[75] Jesse McDonald, Yick-Ching Wong, Kshitij Mehta, Frederic Suter, Rafael Ferreira Da Silva, Loïc Pottier, Ewa Deelman, and Henri Casanova. On the Level of Detail of Simulators of Parallel and Distributed Computing Systems. *Future Generation Computing Systems*. [In preparation]

Appendices

APPENDIX A
GRAVITY EXAMPLE GROUND-TRUTH DATA

For the GravitySimulator in Chapter 2 and 3, we have used a simple "simulator" of projectile motion defined using $F(t, v, d) = At^2 + Bvt + Cd$. To demonstrate some aspects of calibration, the "real world" ground-truth data on which the simulator is to be calibrated is drawn from a more sophisticated model that considers quadratic drag:

$$\frac{dv}{dt} = -g - \frac{\frac{1}{2}\rho C_d A}{m} v|v|$$
$$\frac{dh}{dt} = v$$

where:

- $v(t)$ is the velocity as a function of time,
- $h(t)$ is the height as a function of time,
- g is the gravitational acceleration (set to $9.8m/s^2$),
- ρ is the air density (set to $1.225kg/m^3$),
- C_d is the drag coefficient (0.47 due to spherical projectile),
- A is the cross-sectional area of the object ($2\pi r^2$ due to spherical projectile), and
- m is the mass of the projectile.

The ground-truth data was generated for spheres with density of 2kg/m^3, 20kg/m^3, 200kg/m^3, 2000kg/m^3, 20000kg/m^3; with radius 5mm, 5cm, 50cm, and 5m; with initial elevations of 1m, 10m, 100m, and 1000m; and with initial velocity 0m/s, 1m/s, 10m/s, 100m/s, and 1000m/s. The above equations were solved numerically with `scipy.integrate` using a 0.01s time step. Note that this data represents idealized ground-truth data with no noise and perfect measurements. It also ignores the speed of sound, variations in the atmosphere, ground effect, and buoyancy.

APPENDIX B
SIMULATION-DRIVEN SCHEDULING APPLIED TO DYNAMIC PLATFORMS

Section 9.5.4 mentions an interesting use case for SDPS: the handling of dynamic platform changes. There is no general and/or accepted model for dynamic compute resource availability in distributed computing platforms. In this appendix we explore this use case for a simple model, acknowledging that our results may not generalize to other models. Designing a general model of resource availability is outside the scope of this dissertation, and likely not feasible.

We define our dynamic resource availability model based on two parameters:

- A probability p: a compute core is not available at the onset of the execution with probability p;
- A time T: an initially unavailable compute core will become available after a number of seconds sampled from a uniform distribution over the $(0, T)$ interval.

We ran experiments for:

- $p \in \{0.1, 0.2, \ldots, 0.9\}$; and
- $T \in \{\alpha \times T' | \alpha \in \{0.1, 0.5, 1.0, 2.0, 10\}\}$, where T' is the makespan achieved using algorithm A_8 when all resources are available from the get go.

We use the above scaling approach for T values because our workflows all lead to very different makespans, making picking absolute values for T problematic (e.g., for a workflow that takes 3600s a T value of 100s means that most resources are available for most of the executions, but this is not the case for a workflow that takes 600s). We conducted experiments for platform configuration P_3 (the most heterogeneous) and assuming no simulation error (to ensure that we observe the net effect of adapting to platform changes). This leads to 405 experiments (9 workflows, 9 values of p, 5 values of α), for each of which we ran 100 trials (different seeds of the random number generator used to generate the initial machine state). For each trial we compare the one-algorithm solution (algorithm A_8, a.k.a., "one-alg") with SDPS using a single round of simulation at the start ("no-adapt") and SDPS using a round of adaptation each time an additional 10% of the compute work has been completed ("adapt").

We first show and discuss results for two workflows and then discuss broad observations over all results.

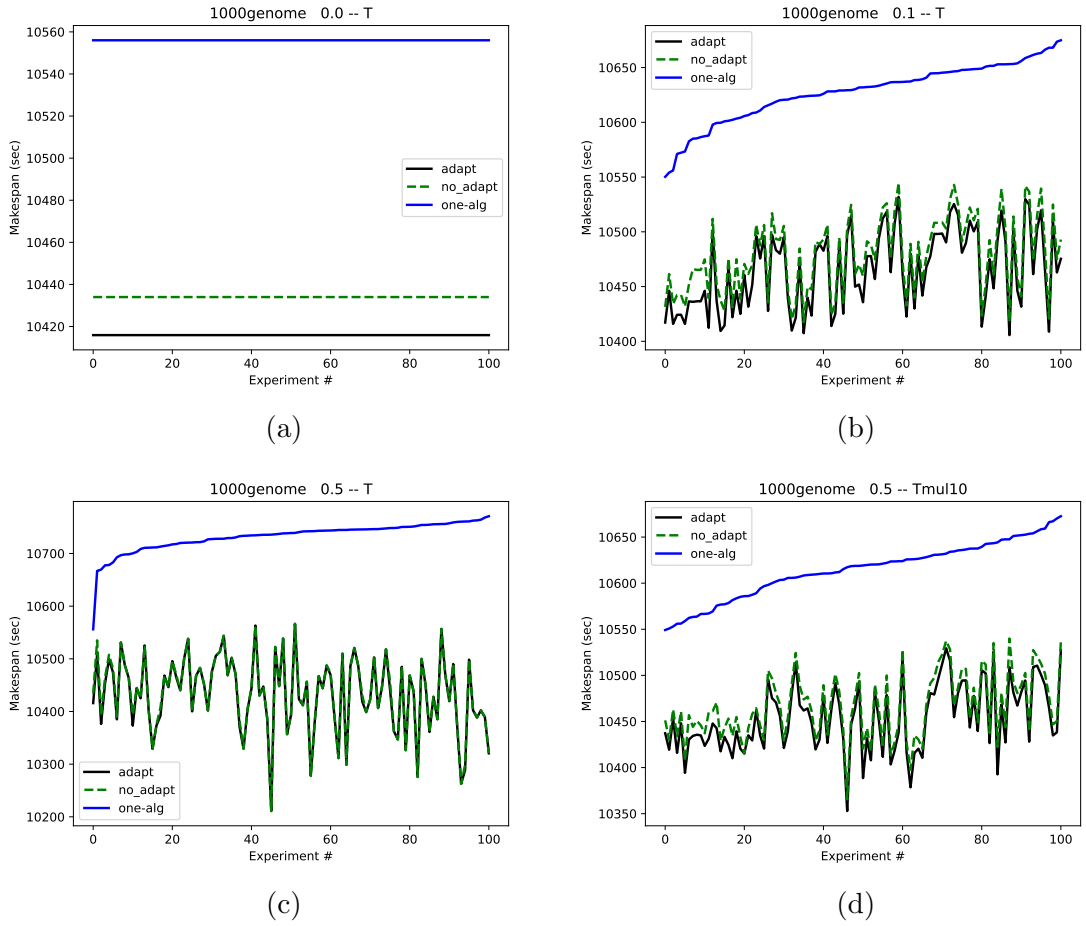


Figure B.1: Makespan results for the W_1 workflow. The first data point on the horizontal axis is always for $p = 0$. Each subsequent data point is for a different random trials. Data points are sorted by increasing makespan as achieved by algorithm A_8.

Results for the W_1 workflow – Figure B.1 shows results with the W_1 workflow for different values of p and α. Figure B.1a is a "control" experiment where all resources are always available ($p = 0$). The vertical axis is the achieved makespan, and the horizontal axis correspond to 100 random trials. In this figure, since $p = 0$, all trials are identical (all compute resources are available) hence the flat lines. We see that adaptation can bring some benefit even when everything is static, due to the workflow proceeding in different phases with different characteristics (see Section 9.3.2). Figure B.1b is for $p = 0.1$ and $\alpha = 1$. In this and other figures, data points are sorted along the horizontal axis with respect to the makespan achieved by the one-algorithm approach ("one-alg"). For reference, the leftmost data point is always for $p = 0$. We now see variations, and marginal improvements due

to running multiple rounds of adaptation. This improvement is a combination of selecting a new algorithm because the resource pool has changed and the above effect due to the workflow proceeding in different phases. Figure B.1c is for $p = 0.5$ and $\alpha = 1$, and here there are almost no improvements for "adapt" over "no-adapt". Finally, Figure B.1d shows results for $p = 0.5$ and $\alpha = 10$, with more improvement of "adapt" over "no-adapt", but still marginal. All other results for different p and α values look very similar. Overall, for this workflow, running multiple rounds of simulations does not have a large effect. This is because the structure of the workflow renders initial scheduling decisions crucial. When comparing SDPS to "one-alg" in these results, a case can be made, that the one-algorithm approach is more sensitive to the initial resource states. This is a strength of SDPS over the traditional approach, but the use of multiple rounds of simulation brings little benefit.

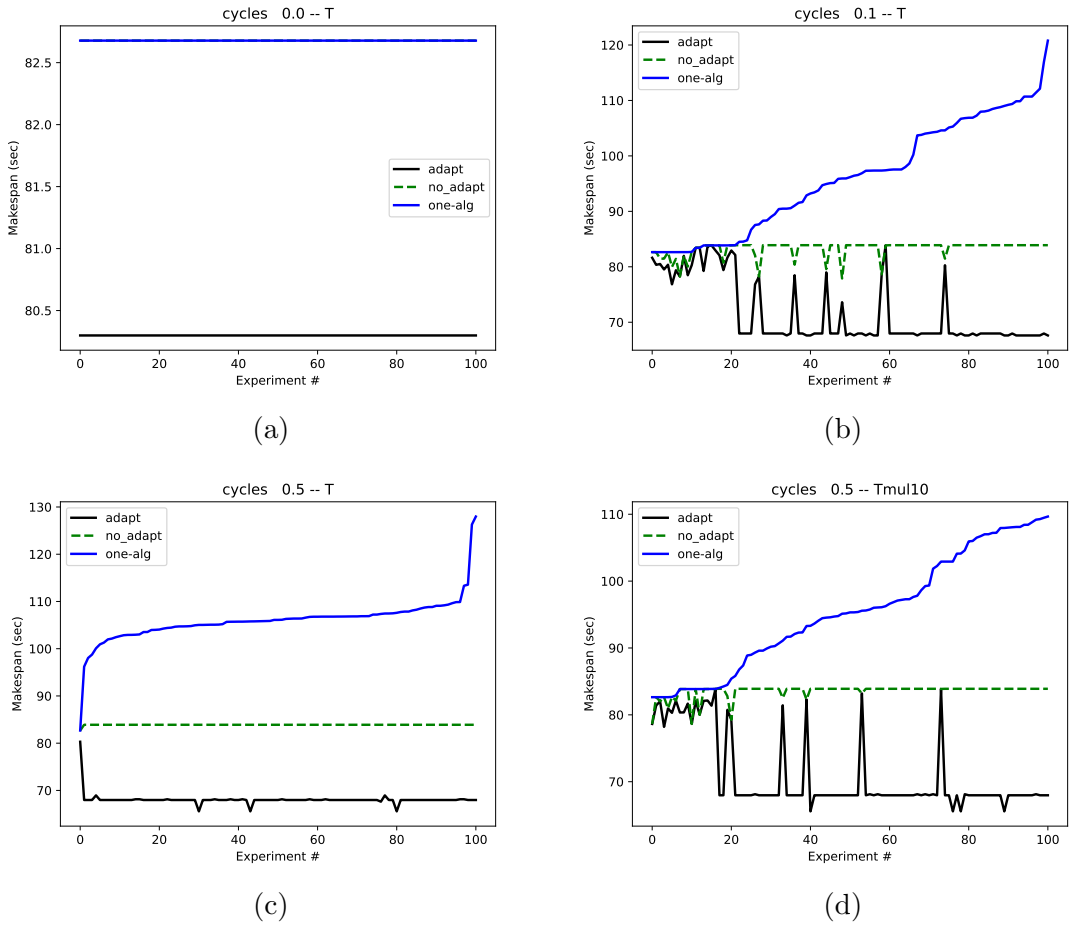


Figure B.2: Makespan results for the W_4 workflow. The first data point on the horizontal axis is always for $p = 0$. Each subsequent data point it for a different random trials. Data points are sorted by increasing makespan as achieved by algorithm A_8.

Results for the W_4 workflow – Figure B.2 is similar to Figure B.1, but for workflow W_4. For this workflow, we see that using multiple rounds of simulation with $p = 0$ does not bring any significant benefit (Figure B.2a). The other results show that, unlike for workflow W_1, using multiple rounds of simulations can bring benefits ($\sim$17% makespan reduction). Furthermore, it is clear that the performance achieved is more stable than, in particular, the traditional OneAlg approach.

Overall results – Overall, although results in Figure B.2 show that running multiple rounds of adaptation can bring some benefit, this behavior is not seen for most workflows. For instance, out of our 9 workflows, for $p = 0.5$ and $\alpha = 1$, significant improvements are seen for only 1 workflow (W_4). For $p = 0.5$ and $\alpha = 10$, significant improvements are seen for that same workflow and only minor improvements are seen for another workflow (W_5). Out of all our results, the use of multiple rounds of simulation leads to significant improvements in only a handful of cases. These results are similar to those found in Section 9.5.2. There are likely real-world workflow configurations for which the improvement could be large. In fact, we could easily craft synthetic platform configurations, initial availability patterns, and workflow configurations specifically so that improvements are large. But, within the scope of our study of 9 real-world workflow configurations and a typical real-world platform architecture, large improvements are rare at best. Regardless, if SDPS with multiple simulation rounds is implemented in a real-world WMS, provided that platform changes can be detected and supplied to the simulator, these improvements would be automatically realized with no additional effort and no downside.

BIBLIOGRAPHY

[1] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.

[2] A. Al-Haboobi and G. Kecskemeti. Developing a Workflow Management System Simulation for Capturing Internal IaaS Behavioural Knowledge. *Journal of Grid Computing*, 21(2), 2023.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[4] T. Andel and A. Yasinsac. On the Credibility of MANET Simulations. *Computer*, 39(7):48–54, 2006.

[5] L. K. Arya and A. Verma. Workflow scheduling algorithms in cloud environment - A survey. In *Proc. of Conf. on Recent Advances in Engineering and Computational Sciences*, 2014.

[6] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216–227, 2017.

[7] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive Parallel Programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019. babuji19parsl.pdf.

[8] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 659–663, 1977.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[10] Anthony Boulmier, Ioana Banicescu, Florina M. Ciorba, and Nabil Abdennadher. An Autonomic Approach for the Selection of Robust Dynamic Loop Scheduling Techniques. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 9–17, 2017.

[11] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, December 2002.

[12] Swen Böhm and Christian Engelmann. xsim: The extreme-scale simulator. In *Proc. of the Int. Conf. on High Performance Computing and Simulation*, pages 280–286, 2011.

[13] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, January 2011.

[14] Danilo Carastan-Santos and Raphael Y. de Camargo. Obtaining Dynamic Scheduling Policies with Simulation and Machine Learning. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proc. of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011. Association for Computing Machinery.

[16] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In *Proc. of the 14th ACM/IEEE/SCS Workshop of Parallel on Distributed Simulation*, pages 53–60, 2000.

[17] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, and F. Suter. Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH. *Future Generation Computer Systems*, 112:162–175, 2020.

[18] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 75(10):2899–2917, 2014.

[19] H. Casanova, Y.C. Wong, L. Pottier, and R. Ferreira da Silva. On the Feasibility of Simulation-driven Portfolio Scheduling for Cyberinfrastructure Runtime Systems. In *Proc. of the 25th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2022.

[20] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Lowering entry barriers to developing custom simulators of distributed applications and platforms with SimGrid. *Parallel Computing*, 123:103–125, 2025.

[21] Chameleon cloud. `https://www.chameleoncloud.org`.

[22] W. Chen and E. Deelman. WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments. In *Proc. of the 8th IEEE Intl. Conf. on E-Science*, pages 1–8, 2012.

[23] Taina Coleman, Henri Casanova, Loïc Pottier, Manav Kaushik, Ewa Deelman, and Rafael Ferreira da Silva. Wfcommons: A framework for enabling scientific workflow research and development. *Future Generation Comp. Sys.*, 128:16–27, 2022.

[24] Iacopo Colonnelli, Barbara Cantalupo, Ivan Merelli, and Marco Aldinucci. StreamFlow: Cross-Breeding Cloud With HPC. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1723–1737, 2021.

[25] Jason Cope, Ning Liu, Sam Lang, Phil Carns, Chris Carothers, and Robert Ross. CODES: Enabling Co-Design of Multilayer Exascale Storage Architectures. In *Proc. of the Workshop on Emerging Supercomputing Technologies*, 2011.

[26] Tom Cornebize. *High Performance Computing: Towards Better Performance Predictions and Experiments*. PhD thesis, Grenoble INP ; Université Grenoble - Alpes, 2021.

[27] Tom Cornebize, Arnaud Legrand, and F. Christian Heinrich. Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study. In *Proc. of the 2019 IEEE International Conference on Cluster Computing*, pages 1–11, 2019.

[28] Simulator and Calibrator for DCSim Case Study. `https://github.com/HEPCompSim/DCSim`, 2024.

[29] Ewa Deelman, Rafael Ferreira da Silva, Karan Vahi, Mats Rynge, Rajiv Mayani, Ryan Tanaka, Wendy Whitcup, and Miron Livny. The Pegasus workflow management system: Translational computer science in practice. *Journal of Computational Science*, 52, 2021.

[30] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a Workflow Management System for Science Automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[31] Augustin Degomme, Arnaud Legrand, George Markomanolis, Martin Quinson, Mark Stillwell, and Frédéric Suter. Simulating MPI applications: the SMPI approach. *IEEE Transactions on Parallel and Distributed Systems*, 18(8):2387–2400, 2017.

[32] Kefeng Deng, Junqiang Song, Kaijun Ren, and Alexandru Iosup. Exploring portfolio scheduling for long-term execution of scientific workloads in IaaS clouds. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[33] Kefeng Deng, Ruben Verboon, Kaijun Ren, and Alexandru Iosup. A Periodic Portfolio Scheduler for Scientific Computing in the Data Center. In *Job Scheduling Strategies for Parallel Processing*, pages 156–176, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[34] C. Engelmann. Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale. *Future Generation Computer Systems*, 30:59–65, 2014.

[35] L. Eyraud-Dubois and A. Legrand. The Influence of Platform Models on Scheduling Techniques. In Y. Robert and F. Vivien, editors, *Introduction to Scheduling*, chapter 11, pages 281–309. CRC Press, 2009.

[36] D.G. Feitelson and M. Naaman. Self-tuning systems. *IEEE Software*, 16(2):52–60, 1999.

[37] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Ilkay Altintas, Rosa M Badia, Bartosz Balis, Tainã Coleman, Frederik Coppens, Frank Di Natale, Bjoern Enders, Thomas Fahringer, Rosa Filgueira, Grigori Fursin, Daniel Garijo, Carole Goble, Dorran Howell, Shantenu Jha, Daniel S. Katz, Daniel Laney, Ulf Leser, Maciej Malawski, Kshitij Mehta, Loïc Pottier, Jonathan Ozik, J. Luc Peterson, Lavanya Ramakrishnan, Stian Soiland-Reyes, Douglas Thain, and Matthew Wolf. A community roadmap for scientific workflows research and development. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 81–90, 2021.

[38] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[39] G. Flores, M. Paredes-Farrera, E. Jammeh, M. Fleury, and M. Reed. OPNET Modeler and Ns-2: Comparing the Accuracy of Network Simulators for Packet-Level Analysis Using a Network Testbed. *WSEAS Transactions on Computers*, 2(3), 2003.

[40] R. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM*, 33(10):30–53, 1990.

[41] K. Fujiwara and H. Casanova. Speed and Accuracy of Network Simulation in the SimGrid Framework. In *Proc. of the 1st International Workshop on Network Simulation Tools*, 2007.

[42] Pablo Garrido, Manuel Malumbres, and Carlos Calafate. Ns-2 vs. OPNET: A Comparative Study of the IEEE 802.11e Technology on MANET Environments. In *Proc. of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008.

[43] Éric Gaussier, Jérôme Lelong, Valentin Reis, and Denis Trystram. Online Tuning of EASY-Backfilling using Queue Reordering Policies. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2304–2316, 2018.

[44] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources. In *2018 51st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 469–481, 2018.

[45] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In P.L. Hammer, E.L. Johnson, and B.H. Korte, editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287–326. Elsevier, 1979.

[46] The Grid'5000 Testbed. `https://www.grid5000.fr`, 2022.

[47] Ground-Truth Data for the Case Studies. `https://doi.org/10.6084/m9.figshare.30132955`, 2025.

[48] Ashish Gupta and Ritu Garg. Workflow scheduling in heterogeneous computing systems: A survey. In *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*, pages 319–326. IEEE, 2017.

[49] Alexander Hagen, Shane Jackson, James Kahn, Jan Strube, Isabel Haide, Karl Pazdernik, and Connor Hainje. Accelerated computation of a high dimensional kolmogorov–smirnov distance. *arXiv preprint*, 2021.

[50] Robert Hall, Arnold L. Rosenberg, and Arun Venkataramani. A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–9, 2007.

[51] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A. Orgerie, and M. Quinson. Predicting the Energy-Consumption of MPI Applications at Scale Using Only a Single Node. In *Proc. of 2017 IEEE International Conference on Cluster Computing*, pages 92–102, 2017.

[52] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, pages 597–604, jun 2010.

[53] M. Horzela, H. Casanova, M. Giffels, A. Gottman, G. Quast, S. Rissi Tisbeni, A. Streit, and F. Suter. Modelling Distributed Heterogeneous Computing Infrastructures for HEP Applications. In *26th International Conference on Computing in High Energy & Nuclear Physics (CHEP)*, 2023.

[54] Horzela, Maximilian, Casanova, Henri, Giffels, Manuel, Gottmann, Artur, Hofsaess, Robin, Quast, Günter, Rossi Tisbeni, Simone, Streit, Achim, and Suter, Frédéric. Modeling distributed computing infrastructures for hep applications. *EPJ Web of Conf.*, 295:04032, 2024.

[55] Ming-Yu Hsieh, Rolf Riesen, Kevin Thompson, William Song, and Arun Rodrigues. SST: A Scalable Parallel Framework for Architecture-Level Performance, Power, Area and Thermal Simulation. *The Computer Journal*, 55(2):181–191, 2012.

[56] The HTCondor Software Suite. `https://htcondor.org`, 2023.

[57] P. Hurni and T. Braun. Calibrating Wireless Sensor Network Simulation Models with Real-World Experiments. In *Proc. of the 8th International IFIP-TC 6 Networking Conference*, volume 5550 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2009.

[58] Intel MPI Benchmarks User Guide. `https://www.intel.com/content/www/us/en/docs/mpi-library/user-guide-benchmarks/2021-8/overview.html`, 2021.

[59] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A simulator for large-scale parallel architectures. *International Journal of Parallel and Distributed Systems*, 1(2):57–73, 2010.

[60] Frank J. Massey Jr. The kolmogorov–smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[61] Gabor Kecskemeti. DISSECT-CF: A Simulator to Foster Energy-Aware Scheduling in Infrastructure Clouds. *Simulation Modelling Practice and Theory*, 58:188–218, 2015.

[62] Dzmitry Kliazovich, Pascal Bouvry, Yury Audzevich, and Samee Ullah Khan. GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers. In *Proc. of the IEEE Global Telecommunications Conf.*, pages 1–5, 2010.

[63] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, dec 1999.

[64] Lawrence Livermore National Laboratory. El capitan. `https://asc.llnl.gov/exascale/el-capitan`, 2024.

[65] A. Lèbre, A. Legrand, F. Suter, and P. Veyre. Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API. In *Proc. of the 8th IEEE International Symposium on Cluster Computing and the Grid*, 2015.

[66] J. Lessmann, P. Janacik, L. Lachev, and D. Orfanus. Comparative Study of Wireless Network Simulators. In *Proc of the 7th International Conference on Networking*, pages 517–523, 2008.

[67] Joseph Y-T Leung. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, 2004.

[68] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A Survey of Data-Intensive Scientific Workflow Management. *J. Grid Comput.*, 13(4):457–493, 2015.

[69] Junwen Liu, Shiyong Lu, and Dunren Che. A Survey of Modern Scientific Workflow Scheduling Algorithms and Systems in the Era of Big Data. In *Proc. IEEE International Conference on Services Computing (SCC)*, pages 132–141, 2020.

[70] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proc. of the 13th USENIX Conference on Operating Systems Design and Implementation*, page 409–425, 2018.

[71] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin van Eyk, Laurens Versluis, Vincent van Beek, and Alexandru Iosup. OpenDC 2.0: Convenient Modeling and Simulation of Emerging Technologies in Cloud Datacenters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 455–464, 2021.

[72] John McCalpin. An Industry Perspective on Performance Characterization: Applications vs Benchmarks. In *Proc. Workshop on Workload Characterization*, 2000.

[73] Jesse McDonald, John Dobbs, Yick Wong, Rafael Ferreira da Silva, and Henri Casanova. An exploration of online-simulation-driven portfolio scheduling in workflow management systems. *Future Generation Computer Systems*, 161, 2024.

[74] Jesse McDonald, Maximilian Horzela, Frédéric Suter, and Henri Casanova. Automated calibration of parallel and distributed computing simulators: A case study. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1026–1035. IEEE, 2024.

[75] Jesse McDonald, Yick-Ching Wong, Kshitij Mehta, Frederic Suter, Rafael Ferreira Da Silva, Loïc Pottier, Ewa Deelman, and Henri Casanova. On the Level of Detail of Simulators of Parallel and Distributed Computing Systems. *Future Generation Computing Systems*. [In preparation].

[76] Jesse McDonald, Yick-Ching Wong, Kshitij Mehta, Frederic Suter, Rafael Ferreira Da Silva, Loïc Pottier, Ewa Deelman, and Henri Casanova. Determining levels of detail for simulators of parallel and distributed computing systems via automated calibration. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1452–1463, New York, NY, USA, 2025. Association for Computing Machinery.

[77] Levente Mészáros, Andras Varga, and Michael Kirsche. *INET Framework*, pages 55–106. Springer, 2019.

[78] Ali Mohammed and Florina M. Ciorba. SimAS: A simulation-assisted approach for the scheduling algorithm selection under perturbations. *Concurrency and Computation: Practice and Experience*, 32, 2019.

[79] Ali Mohammed, Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba. Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4383–4394, 2022.

[80] MPI: A Message-Passing Interface Standard - Version 5.0. `https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf`, 2025.

[81] Simulator and Calibrator for MPI Case Study. `https://github.com/wrench-project/pmbs2025_calibration_casestudy2_reproducibility`, 2025.

[82] R. Nallakumar and K.S. Sruthi Priya. A Survey on Deadline Constrained Workflow Scheduling Algorithms in Cloud Environment. *International Journal of Computer Science Trends and Technology*, 2(5):44–50, 2014.

[83] A. Nazarenki and O. Sukhoroslov. Using Simulation to Improve Workflow Scheduling in Heterogeneous Computing Systems. In *Proc. of Russian Supercomputing Days*, pages 480–490, 2017.

[84] A. Núñez, J. Vázquez-Poletti, A. Caminero, J. Carretero, and I. M. Llorente. Design of a New Cloud Computing Simulation Platform. In *Proc. of the 11th Intl. Conf. on Computational Science and its Applications*, pages 582–593, June 2011.

[85] Simon Ostermann, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. GroudSim: An Event-Based Simulation Framework for Computational Grids and Clouds. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 305–313, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[86] Eduardo Pérez. A simulation-driven online scheduling algorithm for the maintenance and operation of wind farm systems. *SIMULATION*, 98(1):47–61, jul 2021.

[87] Anton Petrunin. *Pure Metric Geometry: Introductory Lectures*. SpringerBriefs in Mathematics, Springer Nature, 2023. Includes definitions of Hausdorff and Gromov–Hausdorff distances; available online (e.g., arXiv and author's site).

[88] Reproducible Research for FGCS manuscript. `https://github.com/wrench-project/fgcs2024_manuscript_reproducible_research`, 2024.

[89] Juan A. Rico-Gallego, Juan C. Díaz-Martín, Ravi Reddy Manumachu, and Alexey L. Lastovetsky. A Survey of Communication Performance Models for High-Performance Computing. *ACM Comput. Surv.*, 51(6), 2019.

[90] George F. Riley and Thomas R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[91] A. Rizvi, T. Toha, M. Lunar, M. Adnan, and A. Alim Al Islam. Cooling Energy Integration in SimGrid. In *Proc. of the 2017 International Conference on Networking, Systems and Security (NSysS)*, pages 132–137, 2017.

[92] Stewart Robinson and Roger Brooks. Assumptions and simplifications in discrete-event simulation modelling. *Journal of Simulation*, pages 1–18, 2024.

[93] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in science conference*, pages 130–136, 2015.

[94] Maria Alejandra Rodriguez and Rajkumar Buyya. A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments. *Concurrency and Computation: Practice and Experience*, 29(8):e4041, 2017.

[95] The SimGrid Project. `http://simgrid.org/`, 2024.

[96] SimGrid Use by Others . Available at `https://simgrid.org/usages.html`, 2026.

[97] L. Singh and S. Singh. A Survey of Workflow Scheduling Algorithms and Research Issues. *International Journal of Computer Applications*, 74(15):21–28, 2013.

[98] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, USA, 2007.

[99] Configuring SMPI. `https://simgrid.org/doc/latest/Configuring_SimGrid.html`, 2024.

[100] SST/macro 14.1: User's Manual. `https://raw.githubusercontent.com/sstsimulator/sst-macro/refs/heads/master/manual-sstmacro-14.1.pdf`, 2024.

[101] L. Stanisic. *A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications*. PhD thesis, Université Grenoble Alpes, France, 2015.

[102] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. In *Proc. of the 2015 IEEE 21st International Conference on Parallel and Distributed Systems*, pages 481–490, 2015.

[103] A. Streit. The self-tuning dynP job-scheduler. In *Proc. 16th International Parallel and Distributed Processing Symposium*, 2002.

[104] Erich Strohmaier, Hans W. Meuer, Jack Dongarra, and Horst D. Simon. The top500 list and progress in high-performance computing. *Computer*, 48(11):42–49, November 2015.

[105] Nitin Sukhija, Brandon Malone, Srishti Srivastava, Ioana Banicescu, and Florina M. Ciorba. Portfolio-Based Selection of Robust Dynamic Loop Scheduling Algorithms Using Machine Learning. In *Proc. IEEE International Parallel Distributed Processing Symposium Workshops*, pages 1638–1647, 2014.

[106] Chih-Li Sung and Rui Tuo. A Review on Computer Model Calibration. *WIREs Computational Statistics*, 16(1):e1645, 2024.

[107] D. Talby and D.G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[108] The CMS Collaboration. The CMS experiment at the CERN LHC. *Journal of Instrumentation*, 3(08):S08004, 2008.

[109] Michael Tighe, Gaston Keller, Michael Bauer, and Hanan Lutfiyya. DCSim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *Proc. of the 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*, pages 385–392, 2012.

[110] M. Turilli, M. Santcroos, and S. Jha. A Comprehensive Perspective on Pilot-Job Systems. *ACM Comput. Surv.*, 51(2):43:1–43:32, 2018.

[111] P. Velho and A. Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *Proc. of the 2nd Intl. Conf. on Simulation Tools and Techniques*, 2009.

[112] P. Velho, L. Mello Schnorr, H. Casanova, and A. Legrand. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *ACM Transactions on Modeling and Computer Simulation*, 23(4), 2013.

[113] Pedro Velho, Lucas Mello Schnorr, Henri Casanova, and Arnaud Legrand. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *tomacs*, 23(4):1–26, 2013.

[114] Prateek Verma, Ashish Kumar Maurya, and Rama Shankar Yadav. A survey on energy-efficient workflow scheduling algorithms in cloud computing. *Software: Practice and Experience*, 54(5):637–682, 2024.

[115] Laurens Versluis and Alexandru Iosup. A survey of domains in workflow scheduling in computing infrastructures: Community and keyword analysis, emerging trends, and taxonomies. *Future Generation Computer Systems*, 123:156–177, 2021.

[116] WfCommons: Community Franeowrk for Enabling Scientific Workflow Research and Development. `https://wfcommons.org`, 2022.

[117] The Worldwide LHC Computing Grid. https://wlcg.web.cern.ch/, 2023.

[118] Yick Ching Wong, Jesse McDonald, Frederic Suter, Kshitij Mehta, and Henri Casanova. Automated calibration of a simulator of mpi application executions. In *Proceedings of the 2024 IEEE 20th International Conference on e-Science (e-Science 2024)*, Osaka, Japan, 2024.

[119] Simulator and Calibrator for Workflow Case Study. https://github.com/wrench-project/pmbs2025_calibration_casestudy1_reproducibility, 2025.

[120] Existing workflow systems. https://s.apache.org/existing-workflow-systems, 2022.

[121] WRENCH: workflow management system simulation workbench. http://wrench-project.org.

[122] The WRENCH Project. http://wrench-project.org/, 2025.

[123] The XRootD Project. https://xrootd.slac.stanford.edu/, 2023.

[124] Engr. Urooj Yousuf Khan, Tariq Rahim Soomro, and Muhammad Nawaz Brohi. iFogSim: A Tool for Simulating Cloud and Fog Applications. In *Proceedings of the International Conference on Cyber Resilience*, pages 01–05, 2022.