Support for Configuration and Provisioning of Intermediate Storage Systems

by

Lauro Beltrão Costa

B.Sc. Computer Science, Universidade Federal de Campina Grande, 2003

M.Sc. Computer Science, Universidade Federal de Campina Grande, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

November 2014

Abstract

This dissertation focuses on supporting the provisioning and configuration of distributed storage systems in clusters of computers that are designed to provide a high performance computing platform for batch applications. These platforms typically offer a centralized persistent backend storage system. To avoid the potential bottleneck of accessing the platform's backend storage system, intermediate storage systems aggregate resources allocated to the application to provide a shared temporary storage space dedicated to the application execution.

Configuring an intermediate storage system, however, becomes increasingly complex. As a distributed storage system, intermediate storage can employ a wide range of storage techniques that enable workload-dependent trade-offs over interrelated success metrics such as response time, throughput, storage space, and energy consumption. Because it is co-deployed with the application, it offers the user the opportunity to tailor its provisioning and configuration to extract the maximum performance from the infrastructure. For example, the user can optimize the performance by deciding the total number of nodes of an allocation, splitting these nodes, or not, between the application and the intermediate storage, and choosing the values for several configuration parameters for storage techniques with different trade-offs.

This dissertation targets the problem of supporting the configuration and provisioning of intermediate storage systems in the context of workflow-based scientific applications that communicate via files – also known as

many-task computing – as well as checkpointing applications. Specifically, this study proposes performance prediction mechanisms to estimate performance of overall application or storage operations (e.g., an application turn-around time, application's energy consumption, or response time of write operations). By relying on the target application's characteristics, the proposed mechanisms can accelerate the exploration of the configuration space. The mechanisms use monitoring information available at the application level, not requiring changes to the storage system nor specialized monitoring systems.

The effectiveness of these mechanisms is evaluated in a number of scenarios – including different system scale, hardware platforms, and configuration choices. Overall, the mechanisms provide accuracy high enough to support the user's decisions about configuration and provisioning the storage system, while being 200x to 2000x less resource-intensive than running the actual applications.

Preface

During my PhD studies, I have conducted the research presented in this dissertation and collaborated with other researchers on various projects. This preface presents the publications related to these projects. Acceptance rates are provided when available.

I was the main author of the research described in this dissertation, which is also the result of collaboration with several people, mostly from the Networked Systems Laboratory (NETSYSLAB) at the University of British Columbia (UBC). Additionally, part of the materials included in this dissertation have been either published or submitted for publication.

The list below presents, for each chapter, the corresponding publications and my role in each one. Appendix A briefly describes other research projects [26, 53, 78–81] in which I was involved during my PhD studies.

Chapter 3 "Predicting Performance for I/O Intensive Workflow Applications" has been partially published according to the list below. I was the main contributor for the research presented in this chapter, including the initial idea, system development, and evaluation. I had the collaboration of Hao Yang, Samer Al-Kiswany, Abmar Barros, and Matei Ripeanu to discuss the project, execute experiments, or make some edits in the text. The following publications present parts of the research behind Chapter 3 of this dissertation.

Supporting Storage Configuration for I/O Intensive Workflows [57]. Lauro Beltrão Costa, Samer Al-Kiswany, Hao Yang, and

Matei Ripeanu. In Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14. Pages 191–200. *Acceptance rate: 20%*. ACM, June 2014.

Predicting Intermediate Storage Performance for Workflow Applications [55]. Lauro Beltrão Costa, Samer Al-Kiswany, Abmar Barros, Hao Yang, and Matei Ripeanu. In Proceedings of the 8th Parallel Data Storage Workshop, PDSW '13, Pages 33–38. ACM, November 2013.

Energy Prediction for I/O Intensive Workflows [171] Hao Yang, Lauro Beltrão Costa, and Matei Ripeanu. In Proceedings of the 7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers, MTAGS '14. Pages 1–6. ACM, November 2014.

Chapter 4 "Using a Performance Predictor to Support Storage System Development" was partially published in the report below. I led this project including the initial idea, system development, and evaluation. I also had the collaboration of João Arthur Brunet, Lile Palma Hattori, and Matei Ripeanu to discuss the project, and edit the text.

Experience with Applying Performance Prediction during Development: a Distributed Storage System Tale [59]. Lauro Beltrão Costa, João Brunet, Lile Hattori, and Matei Ripeanu. In Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, SE-HPCCSE '14. Pages 13–19. IEEE, November 2014.

Chapter 5, "Automatically Enabling Data Deduplication", was partially published in the papers listed below. I led the work presented in this chapter including the initial idea, system development, and evaluation. I also had the collaboration of Raquel Vigolvino Lopes, Samer Al-Kiswany, and Matei Ripeanu to discuss the project, execute experiments, or make some edits in the text.

Assessing Data Deduplication Trade-offs from an Energy and Performance Perspective [54]. Lauro Beltrão Costa, Samer Al-Kiswany, Raquel Vigolvino Lopes, and Matei Ripeanu. In Proceedings of the 2011 International Green Computing Conference and Workshops, Pages 1–6. IEEE, July 2010.

Towards Automating the Configuration of a Distributed Storage System [52]. Lauro Beltrão Costa and Matei Ripeanu. In Proceedings of 11th ACM/IEEE International Conference on Grid Computing. Grid '10. Pages 201–208. *Acceptance rate: 23%*. IEEE, October 2010.

MosaStore Storage System

This research is part of a larger project related to storage systems that includes MosaStore[1], a versatile storage system that harnesses resources from network-connected machines to build a high-performance, yet low-cost, storage system. A more detailed description of MosaStore appears in Chapter 2. In MosaStore project, I have collaborated with Samer Al-Kiswany, Emalayan Vairavanathan, Abmar Barros, Hao Yang, and Matei Ripeanu.

As common in Computer Systems research, part of the research project's contribution to the community is a working software system. In this case, the MosaStore storage system is a working prototype based on a set of proposed research ideas, which is used as a research platform for the research presented in this dissertation. As a result of this project, in addition to the working software system, we have published, or submitted for publication materials that describe the research ideas implemented in MosaStore as described below.

Chapter 2, "Research Platform and Background", is based on part of my contribution to this project which includes an extended opportunity study that I led on optimizations for workflow-aware storage systems [60].

[1] http://www.mosastore.net

A Software Defined Storage for Scientific Workflow Applications. [18]. Samer Al-Kiswany, Lauro Beltrão Costa, Hao Yang, Emalayan Vairavanathan and Matei Ripeanu. Under Review. Pages 1–14.

The Case for Workflow-Aware Storage: An Opportunity Study using MosaStore [60] Lauro Beltrão Costa, Hao Yang, Emalayan Vairavanathan, Abmar Barros, Kethan Maheshwari, Gilles Fedak, Daniel Katz, Michael Wilde, Matei Ripeanu and Samer Al-Kiswany. Journal of Grid Computing. Pages 1–19. Accepted in June 2014. Available online.

A Workflow-Aware Storage System: An Opportunity Study [159] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel Katz, Michael Wilde and Matei Ripeanu. In Proceedings of the 12th International Symposium on Clusters, Cloud, and Grid Computing (CCGrid '12). Pages 326–334. *Acceptance rate: 27%*. IEEE, May 2012.

Table of Contents

List of Tables

List of Figures

Glossary

API Application Programming Interface

BLAST Basic Local Alignment Search Tool, finds regions of local similarity between DNA sequences.

CART Classification And Regression Trees

DAG Directed Acyclic Graph, a directed graph with no directed cycles.

DSS Default Storage System configuration

DVFS Dynamic Voltage and Frequency Scaling

EXT3 Third extended filesystem, a journaled file system commonly used by the Linux kernel.

FUSE Filesystem in Userspace

FIFO First In, First Out

GFS Google File System

GIS Grid Information Service

GPFS General Parallel File System

GPU Graphics Processing Unit

HDF5 Hierarchical Data Format, version 5

MD5 Message Digest Algorithm

NDBM New Database Manageris a 1986 Berkeley version of the AT&T dbm database.

NETSYSLAB Networked Systems Laboratory

NFS Network File System

NIC Network Interface Controller, also known as a Network Interface Card.

NS2 Network Simulator 2

POSIX Portable Operating System Interface

PVFS Parallel Virtual File System

QOS Quality-of-Service

QN Queuing Network-Based

RAID Redundant Array of Independent Disks

RRS Recursive Random Search

SAI System Access Interface

SHA Secure Hash Algorithm

SLA Service Level Agreement

SLO Service Level Objective

SPE Software Performance Engineering

SSH Secure Shell

SVN Apache Subversion, a software versioning and revision control system.

TDP Thermal Design Power also known as Thermal Design Point, it is the maximum amount of heat required to be dissipated.

UBC The University of British Columbia

UFCG Universidade Federal de Campina Grande - in English, Federal University of Campina Grande.

UML Unified Modeling Language, a visual language for modeling the structure of software artifacts.

VFS Virtual Filesystem

WOSS Workflow Optimized Storage System

Acknowledgments

My PhD journey has involved support from many people, and, without such support, this work would not be possible. It is hard, perhaps impossible, to acknowledge everyone properly here. My attempt to do so follows:

After thanking God, I would like to thank my advisor Matei Ripeanu who always gave me freedom to choose the – sometimes sinuous – path that I wanted to take. I am also grateful for him making time whenever I knocked his door for two quick questions, or to go upstairs for a cup of coffee and a longer discussion.

I thank the members of my examining committee. I thank Sathish, Karthik, and Ali, for the enriching feedback they provided during the conversations we had; and Mike Feeley, for the valuable points raised and pleasant discussion. I deeply appreciate the detailed feedback that Angela Demke Brown provided.

Raquel Lopes, Jacques Sauvé, and Walfredo Cirne played a mixed role of external advisors and gurus in different moments during the past few years. I would like to thank them for the technical discussions and, mainly, for the personal conversations we enjoyed.

My friends from TOTEM, Abdullah Gharaibeh and Elizeu Santos-Neto, thank you very much for making the environment fun and supportive in the NetSysLab and beyond.

To those who came as visitors and contributed to my work and lab environment, I thank you very much – especially João Brunet, Marcus Carvalho, Nigini Oliveira, and Thiago “Manel” Perreira.

To the MosaStore team – Abmar Barros, Hao Yang, Emalayan Vairavanathan, Samer Al-Kiswany, and Bo Fang – thank you for providing the base layer on which I could build my work, and for our conversations and enriching technical discussions. Scott Sallinen, thank you for your feedback on my presentations.

I thank my family, always available to give me support, words of comfort, and love in different forms throughout these years. Particularly, I deeply appreciate the never-ending patience I have received from my parents, Zé, and my wife. *Minha gratidão por vocês é inefável*.

I also would like to thank my " Vancouverite" friends, Alyssa Satterwhite and Mohammad Afrasiabi, for our brunches and dinners, Guilherme Germoglio for a great internship experience, Jean and Priscila for giving me shelter, and those who were physically far, but always close: Cássio Rodrigues, Eloi Rocha, Eulália Nogueira, Fábio Leite, and Karina Rocha. Finally, I thank my little fluffy creatures, Nori and Pipa, for the joy you show and bring every day.

Para Lile, Zé e Lourdes.

Chapter 1

Introduction

Distributed storage systems emerged to replace centralized storage solutions for large computing systems [10]. Indeed, aggregating available storage space from network-connected nodes to provide a distributed storage system has several appealing benefits: (i) low cost – it is cheaper than a dedicated storage solution; (ii) high performance – applications benefit from a wider I/O channel, obtained by distributing data across several nodes; (iii) high efficiency – it improves resource utilization; and (iv) incremental scalability – with distributed storage systems, it is possible to have a fine-grain increase in system capacity to match the demand.

These benefits, however, come at a price: decisions about resource provisioning and storage system configuration become increasingly complex [14, 27, 52, 122, 152, 156]. First, coordination of the distributed components makes managing data across several nodes more complex than in a centralized solution. Second, several distributed storage techniques present trade-offs that rarely exist in centralized solutions. Finally, different applications have different requirements, which makes these decisions about provisioning and configuration hard to implement.

To illustrate these points, consider the following trade-offs of several storage techniques employed in distributed storage systems. Data deduplication may save storage space and network bandwidth when there is high similarity across write operations, yet deduplication implies higher

computational costs to detect similar data blocks [12, 52]. Higher redundancy levels (replication and erasure codes) may accelerate data access and increase reliability [93, 164], yet they may require more complex consistency protocols, as well as additional storage space, network bandwidth and time to create the replicas or encode data [10]. Specific caching and prefetching policies may improve application's performance, yet the wrong choice for these parameters can cause extra, unnecessary, usage of resources, degrading performance [95, 134]. Different data-placement policies benefit different workloads depending on the data access pattern [14]. Large chunk sizes can reduce overhead and better use the network, yet small chunk sizes can avoid the waste of data transfers and provide more opportunities for parallel data transfers [23].

To avoid the complexity of such a large decision space, storage system designers typically opt for a design (e.g., GFS [82], NFS, and PVFS [85]) with few configuration options, which are chosen at system deployment time (e.g., , replication level, cache size, data placement policy, and chunk size). The rationale, for these designers, is to address the common use case and simplify system management, while keeping the system useful for a broad range of applications. This is known as a "one size fits all" approach [10, 14].

As an alternative to this "one size fits all" approach, previous work [10, 14] proposes a *versatile storage system* approach whose goal is to allow specialization of the storage system to each in a broad range of applications. *Versatility* in these systems means the ability to provide a set of storage techniques that can be activated or configured at compile time, deployment time, or even at runtime, depending on the application's requirements and generated workload.

This versatility provides benefits to the storage system user since it improves application execution, due to its ability to tailor the storage system to meet application-specific requirements. Thus, the user can analyze the application workload to determine the configuration that is most suitable for the workload present in her system. Tailoring versatility is especially appealing in the context of the *intermediate storage systems* approach, which is the focus of this research. This approach consists of aggregating resources

of the application's allocation to provide a shared temporary storage space co-deployed with the application execution [14, 31, 60, 112, 174]. As a result, the application prevents the potential bottleneck of accessing the platform's backend storage system, the initialization of the storage system is coupled with the application's execution, and, more importantly for this research, the storage system is dedicated to a specific application.

In this environment, the user can, for example, configure the storage system to use more nodes, use replication only for files that are anticipated to have high read demand [10, 14, 156], enable or disable data deduplication [12] to save storage space and/or writing time, or choose the most suitable data-placement policy to improve performance [95].

In practice, however, benefiting from a versatile storage system is a challenging task. Properly setting the broad range of configuration options requires a careful workload analysis and a strong understanding of storage technique trade-offs on the optimization criteria (e.g., response time, storage footprint, energy consumption).

Manual configuration is undesirable for several reasons [20, 22, 27, 152, 156]: First of all, the user may lack the necessary knowledge about the application and its generated storage workload. Also, variations in the workload, changes in the infrastructure, or new application versions can make one-time tuning meaningless. Finally, performance tuning is time-consuming, due to the application runtime and potentially large space of configurations to be considered.

The Problem. In this scenario of running applications on top of intermediate storage systems, the role of the application administrator, or user, is non-trivial: if a user wants to extract maximum performance, in addition to being in charge of running the application, the user has to make a set of important decisions to achieve the desired performance (e.g., in terms of application turnaround time, energy consumption, storage footprint, network usage, or financial cost). This involves allocating resources and configuring the storage system appropriately (e.g., chunk size, stripe width, data placement policy, and replication level).

Thus, the decision space revolves around: (i) *provisioning the allocation* –

setting the total number of nodes, and deciding on node type(s) (or node specification) for cloud environments [49]; (ii) *partitioning the allocation* – splitting these nodes, or not, between the application and the storage system; and (iii) *configuring the storage system parameters* – choosing the values for several configuration parameters for the storage system, e.g., choosing chunk size, replication level, cache/prefetching and data placement policies.

Consequently, provisioning the system entails searching a complex multi-dimensional space to determine the set of choices that delivers the user's ideal utility [152, 167] (e.g., fastest turnaround time or cost vs. time balance point). Figure 1.1 shows an example of this scenario for the Basic Local Alignment Search Tool (BLAST) application [19].

Formally,

$$\mathcal{D}_{\text{desired}} = \underset{\mathcal{D}}{\operatorname{argmax}} \mathcal{U}(\mathcal{D}) \tag{1.1}$$

where $\mathcal{U}(\mathcal{D})$ is a utility function representing the user's goal (e.g., lowest financial cost, fastest turnaround time, or some financial cost vs. performance balance), and $\mathcal{D}$ is a tuple representing the set of decisions related to (i) allocation provisioning, (ii) allocation partitioning, and (iii) storage system configuration parameters.

Research Goal. This work addresses the following high level question: *How can one support provisioning and configuration decisions for a distributed storage system with minimal human intervention?*

The end goal is to provide mechanisms to support provisioning and configuration decisions for versatile storage systems to meet application-specific requirements, delivering results for the success metrics (e.g., response time, storage footprint, energy consumption) close to the user-defined optimization criteria. This research proposes performance-prediction mechanisms in two contexts: (i) workflow applications, the main target of this work, in which the processing flow is structured as several computing tasks communicating via temporary files on a shared storage system; and (ii) data deduplication operations. Specifically, a performance-prediction mechanism in this context consists of a model, a procedure to seed the model

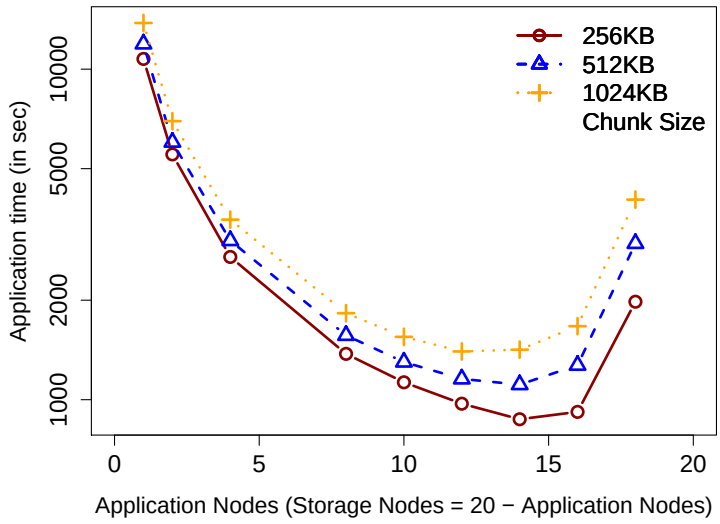


Figure 1.1: Different configurations deliver widely different application turnaround time and the choice of the optimal configuration is not intuitive. Figure shows BLAST workflow [19] execution time (log-scale on Y-axis) on top of an intermediate storage system with different configurations: different ways of partitioning the number of nodes allocated to the application among storage and applications nodes (X-axis), and different chunk sizes. As the number of nodes allocated to application increases (from left to right), the overall response time decreases until an optimal point when the I/O pressure on the storage nodes cannot increase without application performance loss. The chunk size has a high impact (up to 2x speedup) between the default configuration (1024KB) and its optimal value (256KB).

(i.e., perform system identification), and an implementation of these two, providing a software tool to predict the behaviour of intermediate storage systems.

The mechanisms are useful in various scenarios, including solutions that can automate user choices. Chapter 2 presents the target context in detail.

More concretely, the solution described in this dissertation supports the user in answering questions in different domains. For example:

Storage Configuration. What should the storage system configuration be to provide the highest application performance? Would data deduplication, for an average data similarity of 40%, reduce energy consumption on a given platform?

Resource Allocation. Given a fixed size cluster, how should the nodes be partitioned between the application and the storage, and what should be the storage system configuration to yield the most cost-efficient scenario (i.e., lowest cost per unit of performance)?

Provisioning. In a cloud or cluster environment, what is the best allocation size, and how should it be partitioned and configured to best fit my requirements?

Purchasing. What is the impact of purchasing a specific set of machines on the performance of a given application?

Development. What performance should an implementation deliver to be considered efficient enough to stop a debugging effort? What is the performance impact of implementing a specific feature (e.g., a new data placement policy)?

Chapter structure. The rest of this chapter argues why tuning storage system configurations is a difficult and time-consuming task (Section 1.1), describes the requirements for a solution to support configuration (Section 1.2), summarizes the contributions of this research (Section 1.3), and presents the structure of this dissertation (Section 1.4).

1.1 Why Tuning is Hard

A versatile storage system enables applications to harness existing storage resources more effectively by exposing several configuration parameters to facilitate per-application tuning or even per-stored object (e.g., a file) tuning [14, 95, 156]. Although versatility can improve an application's performance, it gives the user the task of deciding on the configuration of the storage system.

This task requires that the user go through a process, summarized in Figure 1.2, that involves the following steps:

1. Identify the storage techniques available and their parameters.

2. Choose the objective of the tuning operation – identify key success metrics (e.g., execution time, storage space), and possibly the desired target level (i.e., specific value) for each metric.

3. Identify the storage techniques to be enabled and, if needed, their parameters to be tuned. Enable these techniques and set their parameters.

4. Request an allocation, wait for the allocation, and run the application.

5. Measure the performance impact of the configuration changes by analyzing the system activity.

6. Repeat steps (2) to (5) until the desired performance level is obtained.

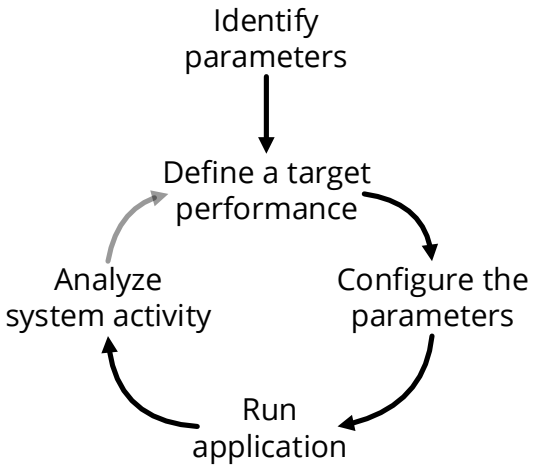


Figure 1.2: Typical configuration loop.

This tuning process is potentially complex, time-consuming, and error-prone for several reasons [20, 22, 27, 52, 54, 91, 121, 152, 156] – especially if the process is not automated.

First, the user may not be familiar with the deployment environment, the requirements of the running applications, or their workload. Such knowledge is essential to identify the most frequent and costly operations in

order to understand which storage techniques and parameters have a larger impact on the success metrics.

Second, defining the desired target values for the success metrics is complex. There are situations in which the user does not have a specific target level in mind. In fact, she may be interested in finding the configuration that can provide the fastest response time for a given combination of application and storage platform, instead of a specific value for response time. In more complex scenarios, the user has more than one target metric to consider and changing the desired level of one metric affects the others. For example, consider a case in which the user wants the shortest possible execution time, but the application produces a large amount of data, and she also has constraints on the storage space used. Enabling data deduplication can save storage space, but it introduces an additional computational overhead that may lead to a longer response time. Similarly, using more machines tends to decrease execution time while increasing the financial cost or energy consumption of running the application.

Third, the workload and platform characteristics may change. The user may go through the whole effort of configuring the system and have it perfectly tuned for a given platform, application version, and workload. However, the workload or the platform can change in an unpredictable manner. For example, the application needs to process a different dataset, or the platform changes as a result of purchasing new machines, a network upgrade, or even an operating system upgrade. In these cases, the configuration may not attain the desired objective and the user would need to restart the tuning of the system.

Finally, the storage system may have a large configuration space to be explored. In case of manual tuning, users can spend a long time in the tuning loop to evaluate different configurations and handle the aforementioned difficulties. Even in the case where automation is possible, the process can take too long, leading to a waste of resources [27, 28].

1.2 Solution Requirements

Based on the scenario described in Section 1.1, "Why Tuning is Hard", the target solution to support the configuration of a storage system should meet the following requirements:

Reduce human intervention. The proposed solution should not impose a burdensome effort to be used and should make the user's job easier. Thus, the goal of this research is to design a solution that requires minimal human intervention to enable or disable various storage techniques and to choose their configuration parameters. Specifically, any system identification (also known as model seeding) procedure needed should be simple: it should not require storage system redesign or a particular initial design to collect performance measurements, and it should be automated. Additionally, the solution should capture the behaviour of a generic object-based distributed storage design, and using it should not require in-depth knowledge of storage system protocols and architecture.

Allow identification of a satisfactory configuration. The proposed solution should provide accuracy high enough to allow the detection of a configuration that brings the performance of the system close to the user's intention. Note that the goal is not an optimal configuration, as this might be too costly to determine or simply not feasible to achieve, but a configuration that brings the system 'close' to the user-defined success criteria. For instance, if two configurations offer close performance, making a precise decision is less important as long as the proposed solution places choosing one of them as similar to choosing the other.

Have a low exploration cost. The overhead of using the proposed solution should be low. In other words, the cost required to use the proposed solution, to explore the decision space and tune the system should be small when compared to the cost of the running application or its I/O operations several times. For example, assume that the goal is to

minimize the application execution time. As the cost of discovering parameters that minimize runtime increases, the proposed solution becomes less desirable. Consequently, the proposed mechanism should scale with: (i) the system size; and (ii) the I/O intensity of I/O applications.

1.3 Contributions

This study proposes a solution to support the user in making decisions about the provisioning and configuration of intermediate storage systems, while meeting the requirements described in Section 1.2. Specifically, this dissertation shows that performance prediction mechanisms can leverage the target application's characteristics to accelerate the exploration of the provisioning and configuration space. Additionally, it demonstrates that such mechanisms can rely on a model of a generic distributed storage system architecture and on monitoring information available at the high level operations of the system, not requiring changes to the underlying storage system or specialized monitoring systems for the target context.

This research presents the following contributions:

Performance Prediction Mechanisms: Models and Seeding Procedures. This research proposes performance prediction mechanisms that rely on some characteristics of workflow and checkpointing applications to reduce the model's complexity and enable a fast exploration of the decision space for traditional performance metrics (e.g., application turnaround time). As a result, this model relies on a system-identification procedure to provide its parameters (i.e., seed the model) that has two key features: (i) it relies on application-level operations, instead of detailed intrusive measurements of the internal execution path, and (ii) it relies only on a small deployment of the system, despite the goal of predicting various scenarios of different scales.

Additionally, this dissertation demonstrates the effectiveness of the proposed solution in a broad range of scenarios, including several

different hardware platforms, scales, and possible user decisions. This also identifies a solution space where the proposed model, despite being based on a generic object-storage system architecture and coarse granularity in terms of internal behaviour of system components, can be effective in predicting the application's behaviour for the target context.

Energy Trade-offs Assessment and Extension of the Prediction Mechanisms. This dissertation presents an assessment of energy consumption, demonstrating how the differences in power proportionality of hardware platforms impact the relation between energy consumption and response time as part of the optimization criteria. Additionally, this study demonstrates that the proposed prediction mechanisms – the model and system identification procedure – can be augmented to estimate the energy consumption while keeping the characteristics of being simple, lightweight, and non-intrusive.

Storage System Development Support. This dissertation presents a successful use-case of a performance prediction mechanism, beyond the original design goal of supporting user choices, where it is incorporated as part of the development process of distributed storage systems. To this end, this research presents an experience of using its prediction mechanism to support the development process by applying it during the development of a distributed storage system at the NetSysLab.

Prediction Tool and Storage System Prototypes. This work offers a prototype tool that serves to verify the feasibility of the proposed solution, to validate it, and to evaluate it according to the requirements described in Section 1.2, using MosaStore – a distributed storage system that the NetSysLab group has developed [14, 159]. As a result of this research, MosaStore's usability, performance and stability have improved.

1.4 Dissertation Outline

This dissertation is organized as follows:

Chapter 2, "Research Platform and Background", presents the target execution environment, applications and success metrics of this research in detail. Chapter 2 briefly describes MosaStore storage system, which is used as a research platform to verify the feasibility, validate, and evaluate this work. MosaStore is also a system in which I was actively involved, leading it to several improvements [56, 60] as a consequence of the work described in this dissertation. Chapter 2 also presents an overview and evolution of past work that aimed to support the configuration of computing systems with a focus on storage systems. Related work is also presented in more detailed for each of the following three chapters.

Chapter 3, "Predicting Performance for I/O Intensive Workflow Applications", addresses the following questions: *How can one leverage the characteristics of I/O intensive workflow applications to build a prediction mechanism for traditional performance metrics* (e.g., time-to-solution, network usage)?, and *Which extensions are needed to capture energy consumption behaviour in addition to traditional performance metrics?* Chapter 3 presents the focus of this research: a low-cost performance predictor that estimates the total execution time of I/O intensive workflow applications in a given setup based on a simple seeding procedure. This chapter presents an evaluation of the prediction mechanism in a number of combinations for storage configuration parameters, execution platforms, synthetic benchmarks and real applications. Chapter 3 also presents an extension and evaluation of the performance predictor in estimating the energy consumption of I/O intensive workflow applications, from an effort led by Hao Yang [171].

Chapter 4, "Using a Performance Predictor to Support Storage System Development", sheds light on the following question: *Which, if any, are the benefits that this proposed performance predictor brings to the software*

development process of a storage system?. To this end, Chapter 4 discusses the experience of using the prediction mechanism beyond its original design goal: the use of the performance predictor to better understand and debug MosaStore, the distributed storage system that NetSysLab has developed.

Chapter 5, "Automatically Enabling Data Deduplication", focuses on a preliminary experience in this research that explored data deduplication in the context of repetitive write operations (e.g., checkpointing applications) to enable a simple predictor for an online configuration targeting the following question: *What are the challenges of an automated solution for the online configuration of an intermediate storage system?* Chapter 5 also addresses the question *Is energy consumption subject to different trade-offs than response time, or are optimizing for energy consumption and response time coupled goals?* It does so by exploring how online monitoring can enable online adaptation, and presents this research's initial experience of applying a power-profile-based approach to estimate energy consumption of a data deduplication technique – used as the basis for the energy extension presented in Chapter 3.

Chapter 6, "Concluding Remarks" summarizes the results of this research, discusses its limitations, the challenges and the lessons learned, and suggests directions for future work.

Chapter 2

Research Platform and Background

This chapter contextualizes the research presented in this dissertation. First, Section 2.1 presents the MosaStore storage system prototype used in this research. This prototype is the result of research in which I was actively involved. Then, Section 2.2 describes the target deployment scenario, applications and success metrics on which this work focuses. Finally, Section 2.3 presents an overview of past research efforts towards supporting the configuration of computer systems with a focus on storage systems, and highlights the relationship between those past research efforts and this research.

2.1 Research Platform: MosaStore Storage System

This section describes MosaStore[1] [14], an experimental distributed storage system that can be configured to enable workload-specific optimizations. Its design leverages unused storage space from network-connected machines, and offers this space as a high-performance, yet low-cost, shared storage system across these same connected machines (see Figure 2.1).

[1] http://mosastore.net

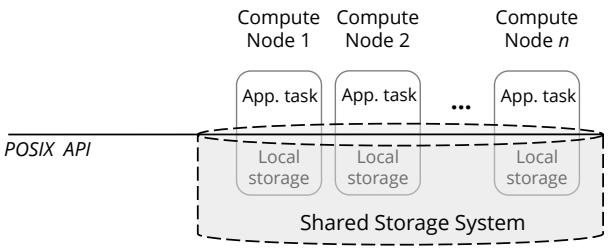


Figure 2.1: MosaStore deployment. It aggregates resources from network-connected machines to provide a high-performance, yet low-cost shared storage system across these same connected machines.

As is common in computer systems research, part of a research project's contribution to the field is a working prototype. In this case, the MosaStore storage system is a prototype based on a set of proposed research ideas initially proposed by Al-Kiswany et al. [14], and, despite being a prototype, it has been used in several different projects from multiple institutions [11, 14, 17, 52, 54, 60, 76, 115, 159, 171]. I was actively involved in the research [60, 159] that led MosaStore to enable optimization opportunities for workflow applications (described in Section 2.2.3 and the focus of Chapter 3), including an extension of an opportunity study that I led[2].

The research presented in this dissertation is part of a larger project related to storage systems that includes MosaStore. The goal is to support the configuration of a versatile storage system specifically for the environment described in Section 2.2. MosaStore is the versatile storage system used as a research platform for the evaluation of the research ideas presented in this dissertation.

The rest of this section is organized as follows: it describes the architecture of MosaStore (Section 2.1.1), the configuration versatility (Section 2.1.2), and details on its implementation (Section 2.1.3), and then it briefly compares this system with other related systems and approaches (Section

[2] *Case for Workflow-Aware Storage: An Opportunity Study using MosaStore* [60] **Lauro Beltrão Costa**, Hao Yang, Emalayan Vairavanathan, Abmar Barros, Kethan Maheshwari, Gilles Fedak, Daniel Katz, Michael Wilde, Matei Ripeanu and Samer Al-Kiswany. Journal of Grid Computing. Accepted for publication in June 2014.

2.1.4). More details about MosaStore can be found NetSysLab's and my past work [14, 17, 60, 143, 159].

2.1.1 MosaStore Architecture

MosaStore employs a widely-adopted object-based storage system architecture, similar to that adopted by the Google File System (GFS) [82], Parallel Virtual File System (PVFS) [85], and UrsaMinor [10]. To speed up data storage and retrieval, the architecture relies on data striping [88]: files are split into fixed-size chunks stored across several storage nodes. This architecture includes three main components: a centralized *metadata manager*, *storage nodes*, and a *client-side* System Access Interface (SAI) (see Figure 2.2). The list below describes the role of each component:

The metadata manager keeps track of the metadata for the entire system: the status of the storage nodes, mapping of file chunks to storage nodes, access control information, and data object attributes. The metadata service and the requests to access data chunks are completely decoupled to provide high scalability: once the client obtains the metadata from the manager, all subsequent access to the data itself is performed directly between the client and the storage nodes, decreasing the chances of a potential metadata manager bottleneck. A New Database Manager (NDBM) compatible library stores the metadata in the manager node. Additionally, the metadata manager node is responsible for running house-keeping tasks such as garbage collection, and storage node failure detection.

The storage nodes allocate part of their local storage space to the shared storage system that MosaStore builds. The storage nodes serve clients' file-chunk store/retrieve requests, and also interact with the manager by publishing their status using a soft-state registration process. Finally, the storage nodes also participate in the replication process and in the garbage collection mechanism.

Client-side SAI uses a Virtual Filesystem (VFS) via the Filesystem in Userspace (FUSE) [2] kernel module to implement a user-level file-system that provides a Portable Operating System Interface (POSIX), which is an Application Programming Interface (API), in MosaStore. Specifically, FUSE provides a set of callback functions that should be implemented by the underlying storage system. For example, consider a write operation called by the application. FUSE receives the write call and its parameters, and forwards the call to the SAI implementation. The SAI implementation contacts the metadata manager to reserve storage space and obtains a list of storage nodes to be used, then it stripes the data into chunks and connects to the storage nodes with allocated space for that operation. Once all those chunks are stored, the callback implementation returns to FUSE which in turn returns to the application. Additionally, the SAI implementation is the part of the system responsible for providing these client-side optimizations: caching, pre-fetching, and data deduplication. Client SAI implementations support data access protocols:

- *Data placement*. The default data placement generally adopted in this type of architecture is round-robin: when a new file is created on a stripe width of n nodes, the file's chunks are placed in a round-robin fashion across those nodes. Additionally, application-driven data-placement policies, which optimize for specific application access patterns, have seen increasing adoption [159, 174]. For instance, both local and co-locate data placement policies can optimize workflow applications' data access patterns (detailed in Section 2.2.3).
- *Replication*. Data replication is often used to improve reliability or access performance. However, while a higher replication level reduces contention on the node storing a popular file, it increases the file write time and the storage space consumption.

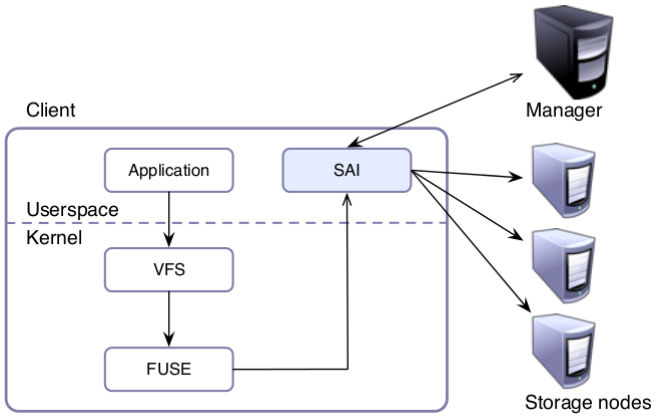


Figure 2.2: MosaStore components as described by Al-Kiswany et al. [17]

2.1.2 MosaStore Configuration

As explained in Chapter 1, distributed storage systems offer a wide set of storage techniques and trade-offs in terms of response time, throughput, energy, or cost when compared to centralized solutions. In this set-up, each application may obtain peak performance at a different configuration point as a consequence of different I/O access patterns or target metrics [10, 49, 54, 95, 152, 156].

MosaStore offers storage configuration versatility since it provides storage techniques that can be turned on or off, or with parameters that can be set at deployment time or runtime, allowing applications to improve their performance according to the workload. The configuration at deployment time is given via configuration files as typically happens in many systems. The runtime settings are provided according to a cross-layer communication solution proposed by Santos-Neto et al. [143]. My collaborators and I [60, 159] have evaluated the feasibility and assess the benefits of this approach. Al-Kiswany et al. [17] present a full design and implementation of this approach for MosaStore.

Enabling Per-File Configuration Versatility

MosaStore employs a hint-based mechanism to provide a specific optimization per file by applying different storage techniques according to the hint. The clients provide these hints via POSIX extended attributes, *tagging the files with key-value pairs*. For example, the application can inform the storage system - that is, give a hint - that a set of files will be consumed by the same client and, therefore, should be placed on the machine that will consume those files.

MosaStore has a flexible design that supports exploration and facilitates adding new custom metadata and their corresponding functionality. Because of the storage system's distribution, enabling a hint based mechanism requires support by each of the main components: the manager, client SAI, and storage nodes. The metadata manager is responsible for keeping the extended attributes; any client can set or retrieve the values of the extended attributes. Additionally, the actual functionality associated with a specific *key-value pair* is spread among the components and depends on the functionality, which may include the storage nodes. For example: data placement resides on the manager, the client SAI implements caching or prefetching, and storage nodes handle replication.

The overall flow of storage operations in MosaStore works as follows: The client SAI initiates file-related operations (e.g., create file), obtaining the file's metadata. The first time an application gets metadata of a file (e.g., open a file), it caches all the metadata information - including the application hints in the form of file's extended attributes. The SAI tags all subsequent inter-component communication related to that file with the file's extended attributes, which can trigger the corresponding callbacks at each component.

To enable the optimizations in the storage system, each component has a dispatcher following a design similar to a *strategy design pattern* [73], allowing a set of implementations to be selected on-the-fly at runtime (see Figure 2.3). Whenever a request arrives, the dispatcher checks the tags of that request and triggers the associated functionality (*strategy*) by forwarding

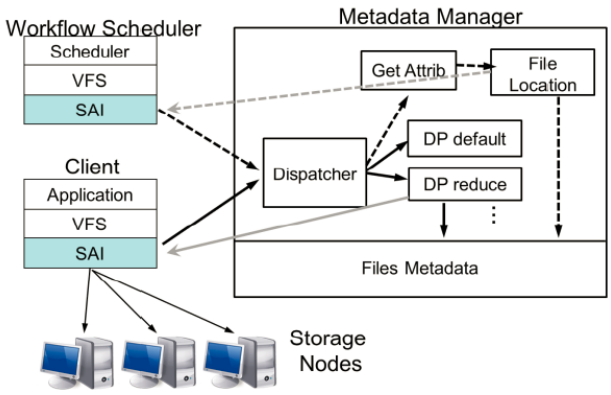


Figure 2.3: Integration of cross-layer communication with workflow execution system as described by Al-Kiswany et al. [17]: (i) the solid lines originating from the SAI represent the path followed by a client chunk allocation request: the request is processed by a pattern-specific data placement 'DP' module based on the corresponding file tags/hints, (ii) the solid lines going to storage nodes represent the data path as data is produced by the clients, and (iii) the dashed lines represent path of a request to retrieve file location information.

the request to the specific optimization module(s) associated with the hint. If no tags are provided, or there is no specific functionality associated to the given tags, the component uses a default implementation.

To add a new functionality to the system (e.g., a new optimization for a specific operation), the developer needs to decide the *key-value pair* that will specify the application hint and trigger the optimization. Then, the developer implements a callback function on the needed system components. Every callback function can access the storage component's internal information, the metadata manager, and stored blocks through an API. Al-Kiswany et al. [17] discuss the architecture's extensibility in more detail.

The tagging mechanism used to give hints about the application is a two-way cross-layer communication mechanism: (i) it allows the application to pass information to the storage system by tagging files with *key-value pairs*, and (ii) it allows the applications - including a scheduler - to retrieve the tags (the `GetAttrib` module in Figure 2.3). Section 2.2.3 describes an integration of a workflow runtime environment with the MosaStore tagging mechanism to improve workflow performance via data-aware scheduling

[60, 142].

2.1.3 Current Implementation

The MosaStore implementation follows the extensible storage system design principles as described in the previous sections. The prototype [3] is mainly implemented in C, and has approximately 10,000 lines of code. It has been used in several different projects from several institutions [11, 14, 17, 52, 54, 60, 76, 115, 159, 171]. Although to date more than fifteen developers were involved in different versions of its implementation, most of the development was carried out by Samer Al-Kiswany, Emalayan Vairavanathan, Hao Yang, and myself. MosaStore is open-source, has an Apache Subversion (SVN) repository available online[4], and has been using code reviews which are also available online[5].

The current configuration options available are described in Table 2.1. As noted in Section 2.1.2, however, adding new functionality and their parameters to optimize specific workload should be a simple process.

Currently, the MosaStore implementation has two main limitations. First, the data placement tags are effective at file creation only and, thus, changing the data placement tag for existing files does not have any impact. Second, it utilizes a centralized metadata manager, which can be a potential bottleneck at large scale, though it has not been observed [17].

2.1.4 Related Systems and Approaches

In storage systems, the term *versatility* is used to describe the ability to provide techniques that can be configured at either deployment or runtime, which allows applications to improve their performance. As with MosaStore [14], other systems provide some degree of versatility.

Other systems, however, in the past have proposed a smaller set of configuration parameters that can be tuned for specific applications. For example, Huber et al. [95] offer an API that allows the application to choose

[3] http://mosastore.net/

[4] https://subversion.assembla.com/svn/mosastore/

[5] https://codereview-netsyslab.appspot.com/

Configuration Parameter	Configuration Space	Per File	System Wide	Per Node
Number of Nodes	Integer, up to number of available nodes		✓	
Client SAI and Storage Collocation	Boolean		✓	✓
Chunk Size	Integer, typically a multiple of 64KB	✓	✓	
Cache Size	Integer, number of chunks	✓	✓	✓
Data deduplication	Two values {on, off}, can be combined with chunk size	✓	✓	
Data placement policy	Three options {local, striped, collocated}	✓	✓	
Prefetching	Integer, number of chunks	✓	✓	✓
Replication Level	Integer, typically up to 4	✓	✓	
Replication Policy	Two values {chain, parallel}		✓	
Stripe-width	Integer, up to the number of storage nodes	✓	✓	

Table 2.1: MosaStore's currently supported configuration. Note that some parameters may be affected by others. For example, a different *data placement policy* impacts on the improvement of a larger *cache size*.

data placement and data management policies. Similarly, ADIOS [113] allows the selection of an optimal set of I/O routines for a given platform. In fact, several systems proposed in the storage area in the past two decades have offered similar approaches and have been partially incorporated into systems in production (e.g., pNFS [9], PVFS [85], GPFS [144], BAD-FS [30] and Lustre [145]); this corroborates the importance of this research.

Perhaps the closest system to MosaStore is UrsaMinor [10], which is a

similar distributed storage system that also applies an object system architecture. UrsaMinor was the first system to propose the term versatility for referring to enabling specific optimization storage techniques for different applications. Examples of techniques provided include erasure coding, consistency models, and data placement policies. Compared to MosaStore, the main difference is that the optimization modules in UrsaMinor are not triggered via a POSIX API, and it has been proposed as a persistent storage, rather than an intermediate storage system. For both systems - as well as those listed earlier - the user is responsible for setting the configuration parameters.

In this context, MosaStore's approach has three main advantages over past work. The first two are interrelated: (i) application-agnostic mechanism - it requires annotating files with arbitrary *<key, value> pairs* via POSIX extended attributes, (ii) incremental - enables evolving applications and storage-systems independently while maintaining the current POSIX interface. The advantage is that MosaStore provides an extensible storage system architecture that can easily accommodate new application-specific optimizations.

These advantages of MosaStore, a wide set of configurable storage techniques (i.e., a high degree of versatility), and its focus on intermediate storage make it an excellent platform for the research goal of this dissertation. Additionally, these same advantages and the success of the performance improvement of target applications [60] have led us to extend the research project, including the proposal of a software-defined storage system approach [18].

2.2 Target Execution Environment: Platform, Metrics, and Applications

There is a wide range of applications that can leverage the optimization techniques provided by distributed storage systems on cluster platforms, and a range of metrics that can be used as optimization criteria [10, 14, 159]. This research focuses on a subset of applications and metrics to drive the

effort of supporting the configuration of versatile storage systems. This section describes the target execution platform (Section 2.2.1), applications (Section 2.2.3), and success metrics (Section 2.2.2) - discussing why each is relevant.

2.2.1 Execution Platform and Intermediate Storage Systems

This research focuses on versatile storage systems in the context of a cluster of computers (from small clusters in university labs to large parallel supercomputers), designed to deliver high performance computing for batch applications that run on a dedicated subset of allocated machines. These computing platforms have storage systems composed of a global persistent file system (e.g., Network File System (NFS) in small clusters or General Parallel File System (GPFS) [144] for large supercomputers), which is stored in just a few I/O servers and accessible from all nodes of a cluster. They also have a local file system per compute-node, which is accessed directly by tasks running on that compute node and allows better I/O performance than the global file system.

To avoid accessing the platform's backend storage system (e.g., NFS or GPFS), recent proposals [14, 169] advocate for using some of the nodes allocated to the application to deploy an *intermediate storage system*. That is, aggregating (some of) the resources of an application allocation to provide a shared storage system dedicated to (and co-deployed with) the application [14, 30, 39, 108, 159, 174] to be used as a temporary scratch space. The trade-off is the loss of computing resources by taking some compute-nodes to be used as a staging or temporary scratch area, which may pay off given the performance improvement. As a result, this intermediate storage system has a lifetime coupled to the application's lifetime, can be optimized for the application usage patterns, and is accessible from all compute nodes running the same application with a better performance than the backend storage. The approach described here as an *intermediate storage system* is also known as *burst buffers* [31, 112].

This research uses MosaStore [14] as an application-optimized stor-

age system intended to be configured and deployed together with the distributed application, making MosaStore's lifetime and performance requirements coupled with the application lifetime.

2.2.2 Success Metrics

The main metrics that this study addresses as the optimization criteria are a primary concern for many systems: traditional performance metrics. Specifically, the target success metrics traditionally are response time and data throughput for storage system operations, and turn-around time for workflow applications. This research also enables better reasoning about network bandwidth use and the resource allocation cost. Another metric is the storage footprint, since it is a typical constraint for the users and it is involved in the trade-offs of some optimization techniques (e.g., data compression and replication).

This study also targets energy consumption as part of the optimization criteria. Taking energy consumption into account when deciding on system design and configuration has received increasing attention, since energy cost is a growing concern in current supercomputers and data centers [25, 43, 71].

While the impact of traditional metrics and the existence of their trade-offs are well understood, previous studies leave a gap in energy consumption analysis. Therefore, it offers a new challenge in terms of modeling and understanding such trade-offs to determine the balance between a system's performance and its energy bill [54, 146].

2.2.3 Applications

This research focuses on supporting the storage configuration of high performance computing applications. Specifically, the target applications are workflow-based scientific applications and checkpointing applications, based on their popularity, on the characteristics of these applications, and on the focus of the NetSysLab and MosaStore research [11, 12, 14, 52, 53, 76, 143].

The rest of this section briefly describes the domain of the target applications, and how these applications can leverage an intermediate storage system.

Workflow Applications

Assembling *workflow applications* by putting together standalone binaries has become a popular approach to support large-scale science applications (e.g., BLAST [19], modFTDock [124], or Montage [32]). Many scientific applications use the workflow paradigm [6, 33, 37, 38, 147, 169], in which the processing flow is structured as several computing tasks spawned from these binaries and communicating via temporary files stored on a shared storage system.

This approach is also known as many-task computing [136], and the execution dependency of tasks in the workflow forms a Directed Acyclic Graph (DAG). In this setup, the user allocates a set of dedicated machines and runs workflow runtime engines, which are basically schedulers that build and manage a task-dependency graph based on the tasks' input/output files (e.g., Pegasus [67] and SWIFT [166]) to launch each task's execution.

In the context of many-task computing, a shared storage system abstraction brings key advantages: simplicity, support for legacy applications and support for fault-tolerance. The workflow development, deployment, and debugging processes are simpler since the workflow application can be developed on a workstation and, then, deployed on a cluster without changing the environment. New stages or binaries can be easily integrated into the workflow, since the communication via files using POSIX API allows the tasks to be loosely coupled. Finally, task faults can be tolerated by simply keeping the task's input files in the shared storage and launching a new execution of the task, potentially on a different machine.

The performance drawback of using a backend storage system as the shared file system is addressed by the intermediate storage system approach described in Section 2.2.1. Workflow applications, however, still offer opportunities for performance improvement that vary depending

on the workload and could benefit from a versatile configuration [60, 159], especially in a scenario where cross-layer communication, like that described in Section 2.1.2, is available.

Cross-layer communication enables the exchange of information between the storage system and the workflow execution engine. In this context, a workflow execution engine can, for example, guide the data placement of a set of files, and use data-location information to enable a data-aware scheduling approach. Such optimization is not possible in traditional file systems.

Workflow Data Access Patterns. A typical task in a workflow application progresses as follows: (i) each node brings the input data from the intermediate storage to memory (likely through multiple I/O operations), then (ii) the processor loads the data from the memory and processes it, and finally, (iii) the output is pushed back to the intermediate storage. Additionally, all files generated during the workflow execution are temporary, except the files containing the final results of the application.

The structure of the tasks' data-dependency graph creates a set of common data access patterns. Table 2.2 describes the main common workflow data-access patterns, lists storage configuration parameters that affect each pattern's performance, and explains how the workflow execution engine can be used to enable optimizations. These are among the most used patterns uncovered by studying over 20 scientific workflow applications by Wozniak and Wilde [169], Shibata et al. [147], and Bharathi et al. [33]. More importantly, the typical structure of a workflow application is a combination of these patterns (see Section 3.3.4 for an example).

My collaborators and I [60] have presented a more detailed description of these patterns, as well as how a workflow-aware storage system can leverage them, and have evaluated their performance when applying the optimizations described in Table 2.2.

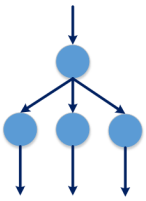
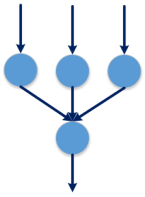
Pattern	Description	Pattern Details and Optimizations
 Pipeline	A set of compute tasks are chained in a sequence such that the output of a task is the input of the next task in the chain.	Node-local data placement (if possible), caching, and data location-aware scheduling. An optimized system stores an intermediate output file on the storage node on the same machine that executes the task that produced the file, if space is available, to increase access locality and efficiently use local caches. Ideally, the location of the data is exposed to the workflow scheduler, so that the task that consumes this data is scheduled on the same node.
 Broadcast	A single file is used by multiple tasks.	An optimized replication taking into account the data size, the fan-out and the network topology. An optimized storage system can create enough replicas of the shared file to eliminate the possibility that the node(s) storing the file become overloaded.
 Reduce	A single task uses input files produced by multiple tasks from a previous stage.	Reduce-aware data placement: co-placement (also known as co-locate) of all output files from a stage on single node, and data location-aware scheduling. An optimized storage system can place all these input files on one node, and expose their location. Then, the scheduler places the task that takes these files as input on the same node, increasing data-access locality.

Table 2.2: Common workflow patterns as described by Al-Kiswany et al. [17] and Costa et al. [60]. Circles represent computing tasks, outgoing arrows indicate that data is written to a temporary file, and an incoming arrow indicates that data is consumed from a temporary file.

Checkpointing Applications

Checkpointing is the process of persistently storing snapshots of an application's state. These snapshots (or checkpoint images) may be used to restore the application's state in case of a failure or as an aid to debugging. Checkpointing is widely adopted by long-running applications in the field of high performance computing [12].

Depending on the checkpointing technique used, the application characteristics, and the time interval between checkpoint operations, checkpointing may result in large amounts of data to be written to the storage system in bursts, and successive checkpoint images may have a high degree of similarity. A previous study by NetSysLab has collected and analyzed [12] checkpoint images produced using VM-supported checkpointing (using Xen), process-level checkpointing (using the BLCR checkpointing library [86]), and application-based checkpointing. Depending on the checkpointing technique, the time interval between checkpoints, and the similarity-detection technique used, the detected similarity between consecutive files varied between no similarity to 82% similarity (for the BLAST bioinformatics application checkpointed using BLCR at 5-min. intervals).

These bursty writes can take advantage of the intermediate storage system (also known as burst buffer) to improve performance. Moreover, the data similarity can leverage data deduplication techniques of the file system to reduce its storage footprint and speed-up writes at the cost of extra compute cycles [11, 12, 125, 135, 139]. To enable data deduplication or not is a user's decision and depends on her optimization criteria and is one of the subjects of study in this research (details in Section 2.2.4 and in Chapter 5).

2.2.4 Integrating Applications and the Storage System Cross Layer Communication

As explained in Section 2.1.2, an application can tag a file to express a hint of its data access pattern and, thus, have the storage system optimized for it.

The integration that follows between application and storage system varies in complexity.

Workflow Applications

For workflow applications, as described in this section, integrating the application with the storage system may involve more iterations than enabling a single optimization.

Currently, the MosaStore prototype is integrated with Swift - a popular language and workflow runtime system [166], and pyFlow - a similar, yet much simpler, workflow runtime that NetSysLab has developed. Such integration allows us to demonstrate the end-to-end benefits of MosaStore and the cross-layer communication approach while not requiring any modification to the application tasks. Specifically, the integration enables two main functionalities:

Location-aware scheduling. Swift and pyFlow did not support location-aware scheduling since most shared storage systems do not expose data location [159]. The modification consists of querying the metadata manager for location information of the input files of a given task, then attempting to schedule the task on the node storing the files.

Smart data placement. The runtime engine adds extra calls to explicitly indicate the data access hints (see Table 2.3). The storage system then uses this information to change the data placement policy. The solution described in Chapter 3 can also be used to evaluate different data placement policies and, when it is worthwhile, passes these hints along as part of the workflow. Alternatively, the runtime engine could receive the hints as input, or build a task-dependency graph [162], which could be used to insert these hints in an approach that does not necessarily improve performance [17, 57].

Al-Kiswany et al. [17] present a detailed description of the integration of MosaStore with workflow applications, including the limitations and a performance evaluation.

Action	API Call	Description
Broadcast hint/Replication	`set("Replication", level)`	Replicate the chunks of the file *level* times.
Enable/Disable deduplication	`set("deduplication", boolean)`	Enable or disable deduplication for a specific file.
Get Location	`get("location", null)`	Retrieves the location information of a specific file.
Manage Cache Size Per File	`set("CacheSize", size)`	Suggest a cache size for a specific file.
Manage Chunk Size Per File	`set("ChunkSize", size)`	Suggest a chunk size for a specific file.
Pipeline hint/Locality	`set("DP", "local")`	Indicates preference to allocate the file chunks on the local storage node.
Reduce hint/ Collocation	`set("DP", "collocation | groupName ")`	Preference to allocate chunks for all files within the same *groupName* on one node.
Scatter hint	`set("DP", "scatter | size")`	Place every group of contiguous *size* chunks on a storage node.

Table 2.3: Examples of MosaStore API calls used to integrate the applications and the storage systems. Most of the calls represent interactions between the workflow runtime schedulers and the storage system. Note that these calls occur on a per-file basis, and have an additional parameter to specify the target file, which is omitted to simplify the presentation. "DP" means data placement.

Figure 2.4 depicts the architecture of a Workflow Optimized Storage System (WOSS) in this context.

Checkpointing Applications and Data Deduplication

For checkpointing applications, the integration is as simple as using a tag to instruct the storage system to enable an optimization: in this case, data deduplication. Data deduplication is a method to detect and eliminate similarities in the data, which affects the time of storage operations, energy consumption and storage usage. Enabling data deduplication, or not, is a user's decision, which depends on her optimization criteria. Supporting the user in this decision is a subject of study in this research (see Chapter

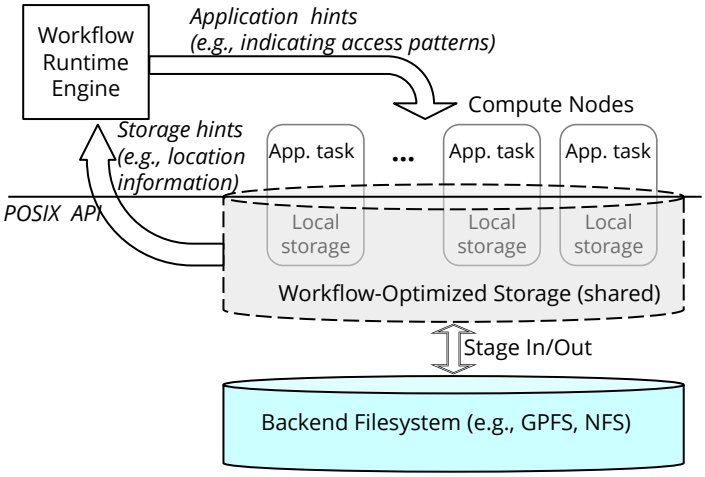


Figure 2.4: A WOSS, as described by Al-Kiswany et al. [17], aggregates the storage space of the compute nodes and is used as an intermediate file-system. Input/output data is staged in/out from the backend storage. The workflow scheduler queries WOSS for data location to preform location-aware scheduling. The scheduler submits tasks to individual compute nodes and includes hints that indicate the future data usage patterns and are used by WOSS for data placement decisions.

5). The rest of this section summarizes how data duplication works, and how its integration with the storage system affects the target metrics of this dissertation.

Briefly, deduplication works as follows: when a new file is stored, the storage system divides the file into chunks, computes identifiers for each chunk based on its content by hashing the data, compares the identifiers newly obtained with the identifiers of the chunks already stored, and persistently stores only the new chunks (i.e., those with different identifiers). Similar chunks are, hence, not stored, saving storage space, reducing the I/O load, and also reducing the network load.

Chunking. The process of detecting the data similarity (i.e., data redundancy) involves breaking the data into chunks and giving a content-based identifier for each chunk. The content-based chunk identifier is typically

based on a strong hash function, e.g., Message Digest Algorithm (MD5) or Secure Hash Algorithm (SHA), to avoid collisions, and this can be computationally costly. The deduplication process then uses such identifier to verify if there is a similar chunk already stored in the system.

The chunking strategy plays an important role in deduplication. It is the process of breaking the data into chunks and it can be performed according to two main approaches: (i) fixed-size chunking, where every chunk has the same pre-specified size and (ii) content-based boundaries, where specific patterns of data determine a chunk boundary.

The first approach, fixed-size chunking, creates a new chunk for every chunk size amount of bytes. It has the advantage of being computationally cheap since the chunking scheme does not need to analyze the data contents. It is, however, not robust for data insertions or deletions that are not multiple of the chunk size since a single byte insertion can change the detectable similarity of the data. Content-based boundaries [105], on the other hand, are more robust to data insertions and deletions since the chunk-boundary is not bound to a specific size, but to specific data patterns. The disadvantage of content-based boundaries is the high cost of analyzing all the data content in order to define the chunks.

Inline data deduplication. When a system performs inline data deduplication, the storage system deduplicates the data while the application is writing the data to the storage system. This can reduce the number of bytes sent over the network and written to storage (i.e., reduce the I/O operations). However, note that is does not necessarily render faster write operations, since the cost of creating the identifiers and chunking the data is expensive and, therefore, it is not clear that the performance cost is paid off [12, 52]. Similarly, it is not clear when energy consumption can be reduced [54].

2.3 Related Work on Supporting Provision and Configuration

Improving the efficiency of computing systems, via tuning their configuration, is a common goal of users, developers, and system administrators. Consequently, overcoming the difficulties of manually tuning computer systems has been the focus of many research efforts for many years, and has become more important as systems increase in scale complexity and maintenance costs [5, 35, 36, 43, 63, 64, 68, 137, 151, 152, 156, 158].

This section describes different approaches employed in past works to facilitate the process of tuning the performance of computer systems, and compares them with the proposed solution described in this dissertation. Chapters 3, 4 and 5 discuss in more detail related work in connection to the material presented in each chapter.

2.3.1 Measuring System Activity

The most basic way to support the user in the task of tuning a system is by measuring system activity for later analysis. In this approach, one can use code instrumentation or simple scripts to collect system activity. Some systems already have instrumentation for this in the code and provide more sophisticated monitoring tools (e.g., Stardust [158]), allowing the user to specify the type or granularity of the activity to be measured. This approach, however, can often lead to a burdensome configuration process where the user is in charge of verifying and understanding the impact of configuration changes via new measurements, and does not scale well with system size, complexity, and load - requiring specialized tools to analyze the data and support the user.

2.3.2 Predicting Performance

Knowing the level of activity in the system helps to understand the workload characteristics and the impact of configuration changes (Section 2.3.1). The user, however, still has to change the parameters and verify the impact of such changes, as described in the loop in Figure 1.2. To avoid these draw-

backs, several approaches to estimating the performance of a computing system have been proposed.

Modeling a System's behaviour

One approach is still using *direct measurements*, but reducing the actual execution cost by using just the application's kernel (a small portion of the system) and/or a representative part of the workload to infer the impact of configuration changes in the performance of the complete application and workload. Although this approach can be effective for small systems, it becomes less effective, and harder to handle, for larger and more complex systems.

As solutions move away from solely direct measurements, they rely on models of the system to capture the most important behaviour, while abstracting the details of the system's internal workings and of the target workload. The two typical approaches to model the behaviour of computing systems in order to predict performance based on some previous knowledge of the system's internals, also known as white-box, are *analytical models* and *simulations*. A third one, such as *machine learning*, relies on direct measurements to build a model of the system's behaviour, it is also known as a black box approach.

Analytical models are traditional mechanisms for predicting the performance of computing systems without requiring an actual execution [20, 117]. They are represented by mathematical formulas that have a closed-form solution providing an approximation for the capacity of system components under a different configuration or workload, without any implementation effort needed.

Analytically modeling a system, however, can be a complex task. It is difficult to capture the impact of the several configuration parameters, so it becomes necessary to make non-realistic assumptions regarding the characteristics of the workload, the system, and the platform. As a result, analytical models often include simplifications that limit their usability in realistic settings.

Simulation is another alternative for predicting system performance [8, 70, 122, 123]. Compared to analytical models, simulations allow us to capture the behaviour of more complex systems and, since a simulator can handle a near-real workload, it does not require the same non-realistic assumptions about the system, as it is the case for analytical modeling.

For example, Network Simulator 2 (NS2) [8] is a discrete event simulator for computer networks typically used to support research and provide some approximation of the actual behaviour of the system. NS2 provides wide support for the simulation of several protocols in different layers of the network stack. Also used to support research, PeerSim [123] simulates peer-to-peer systems, which commonly require simulation to scale to a large number of nodes. DiskSim [1] simulates hard disks behaviour. Other approaches [70, 122] also target the simulation of parallel file systems.

Despite its advantages over analytical models, properly capturing the necessary details of the system to result in useful output is challenging, especially in distributed environments, due to the complexity and scale of the system. As a result, the cost of implementing the simulator and executing simulations can be a drawback. PeerSim, for example, provides two operation modes that vary in simulation details: the simplest one is less accurate, but faster; the more complex one is more realistic, but may take longer, resulting in it being less scalable.

The *machine learning* approach tries to cover the gap between the need to know the systems' internal details and still providing useful information about the system behaviour. To this end, machine learning relies on direct measurements of the system to derive the expected behaviour based on different techniques (e.g., linear regressions, decisions trees, and neural networks).

For example, Crume et al. [61] use machine learning approach to define a function capturing hard disk access times. Mesnier et al. [118, 120] rank different types of machines, based on a set of benchmarks, to describe them in terms of relative performance among the different types (say 'A' and 'B'). Later, it tries to derive the behaviour of a workload in a platform 'A', given the actual behaviour on platform 'B' and a description of their

relative performance. IronModels [155] use a Classification And Regression Trees (CART) technique to derive black-box models of storage systems.

Machine learning is an attractive approach, given the promise of reducing the mismatch between performance given by the model and actual performance. Its various techniques, however, may require a large set of measurements to train their models, and typically need more data for each scenario that they try to model. Therefore, it needs data in a wide range of scenarios (e.g., a number of possible configurations), and this makes it harder to be used to reason about the behaviour of new system functionality, configuration, or deployment scenarios.

Building Prediction Mechanisms

Part of this past work (e.g., NS2 [8] and PeerSim [123]) is research oriented, having the goal of obtaining a first approximation of performance, rather than a more accurate performance in configuration-oriented scenarios.

Towards supporting configuration, some systems rely on one of the aforementioned modeling approaches, or combine several of them, to provide tools that support "*what...if...*" queries. These tools allow users to estimate the impact of configuration changes by submitting probe queries such as "*What* would be response time *if* the number of storage nodes changes from 3 to 5?".

Some database management systems (DBMS) also support "what...if..." queries in their systems (see past work [45–47, 157, 163]). Examples of supported decisions are the choice of cache size [126], and the choice of indices [44, 148].

In storage systems that provide file- or object-oriented use, Thereska et al. [156] add support to "*what...if...*" queries to aid in the configuration of the software layer for their UrsaMinor storage system [10], focusing on response time and network usage metrics. Such queries help the user to make decisions regarding data encoding (e.g., parameters for erasure codes). The prediction tool models storage nodes as the CPU, network interface, buffer cache, and disks. Each of these components is associated with a

prediction mechanism that relies on a specific technique: analytical modeling (network, and disks), simulation (buffer caches), or direct measurements (CPU). The system relies on a monitoring system [158] to feed the prediction tool with detailed measurements based on the requests' path through the system, including kernel instrumentation. Other approaches also target storage systems to automate decisions, but focusing on different metrics; for example, Keeton and Merchant [101], Gaonkar et al. [74], and Keeton et al. [102] target how configuration can impact reliability.

Chapter 3, in Section 3.5, presents additional discussion of different approaches for predicting performance for storage systems.

Automating the Configuration Choice

A naïve approach to automating configuration is to automatically vary the values of configuration parameters to perform an exhaustive search, execute the application, measure system activity, and choose the parameters that best match the optimization criteria. The drawback of this approach is the time that it takes.

Most prediction mechanisms can aid in configuration by reducing the time required to explore the configuration space, but they do not automate the choice of parameters. The user has to decide the best configuration parameters by exploring several possible configurations, which can be easily handled, or automated, if the cost to predict is low. The rest of this section gives an overview on different approaches that rely on prediction mechanisms to automate the configuration choice.

Exploring the configuration space with *direct measurements* is feasible when the cost of executing the application is low, or when it is possible to extract just the application's kernel and try different parameters. For example, Datta et al. [64] use this approach to optimize the data structure decomposition of stencil computations at compile time in order to better fit the memory bus and cache mechanism of a given computer architecture.

Analytical performance models can provide the best possible configuration for a given parameter. The drawback, however, besides the complexity of

analytically modeling the system, is that solving the necessary equations may require a closed-form expression with a well known method to calculate their roots, for example. Even if the equations are provided in this form, solving them can require long time, or even leading them to be intractable, depending on the complexity of the analytical model.

Another approach is to use *control theory* to perform online adaptation, which is effective if the system has a target value as a goal, instead of an optimization goal. This model does not provide the final value for a parameter, but it can indicate the expected behaviour of varying a given parameter. In this scenario, a feedback control loop continuously adjusts a configuration parameter until the system delivers a specific performance level. For example, Storage@desk [94] controls the network bandwidth; the user specifies a target value for bandwidth and the control loop tries to keep the bandwidth close to this value by delaying write or read requests.

Some solutions use a *model-driven approach*. For example, Oracle9i [63] and DB2 [151] use a model to predict performance before the system acts, to determine the buffer size allocated to execute different SQL queries in order to attend new requests with no abrupt impact on the performance. Mian et al. [121] use a set of heuristics to search for a cost-effective provisioning decision in the domain of data analytic workloads based on empirical response time for the target queries.

In the context of *distributed services*, dynamic provisioning [35, 36, 43, 68, 137] has also used model-driven or control theory approaches for applications that run as services in data centers. Online monitoring and historical workload models provide information about the workload characteristics. The main goal of dynamic provisioning is to reduce the cost of running the system by dynamically determining the minimum amount of resources (e.g., memory or number of web and application servers) that can meet Service Level Objective (SLO)s predefined in a Service Level Agreement (SLA).

Applying *heuristics to explore the configuration space* is one more approach towards automation, since exploring the whole configuration space can be time-consuming. In *storage systems*, this approach has been used with analytical models or simulations. For example, Strunk et al. [152] describe a

tool that helps administrators in the task of buying a cost-effective storage solution. The tool uses prediction mechanisms similar to those described by Thereska et al. [156]. It receives the datasets to be used in the future system, a utility function as optimization criteria, and explores the potential configurations.

To reduce the number of possible configurations to evaluate, such solution uses genetic algorithms to guide the configuration choice. Similar approaches (e.g., Hippodrome [21], Minerva [20], and Ergastulum [22]) target systems based on Redundant Array of Independent Disks (RAID).

2.3.3 Placing this Dissertation in Context

The work presented in this dissertation differs from past work in several ways.

The typical target context in past work on storage systems is based on the work environment of a company with an enclosed storage solution, or on predicting the performance of a small scale deployment from the perspective of a single, or few, client(s), differing from a cluster-based [14] context with tens of machines acting as storage servers.

This work proposes prediction mechanisms for different optimization techniques, with a focus on distributed storage systems - specifically intermediate storage systems in the context of workflow and checkpointing applications. As a result, it targets a wider set of configuration parameters for storage systems, and different combination of possible optimization criteria. Specifically, it targets predicting performance of a complete execution of workflow applications – focus of this dissertation – as well as performance of storage operations when employing data deduplication.

One of the main challenges for past approaches has been *seeding the model*, based on simulation or analytical modeling, so it can be used to predict performance, while still offering accuracy good enough to support configuration. One common assumption in these past approaches is that relying on human intervention is needed to seed the models or provide details about the workload. This assumption prevents some of the past

approaches from being useful in practice, since it assumes that the user knows the details about the performance of different components in the system, or has an approach to perform system identification (i.e., identify the proper values for model's parameters for a given system), as, without it, the model is useless. Other approaches rely on a fine-grain monitoring system in order to identify all of a model's parameters, which can lead to changes in the target system itself, or the underlying platform (e.g., to instrument the code or the system or the kernel), and more overhead to collect the needed measurements (e.g., Stardust [158]). Sections 1.2 and 3.2.1 present a further discussion of these problems.

This work is based on models that have a number of attractive properties: simple – by not capturing the details on the internal behaviour of each system component, uniform – by modeling each system component similarly, and generic – by relying on a high-level representation of object-storage systems.

To provide these properties, this work proposes performance mechanisms that leverage the target application's characteristics to accelerate the exploration of the provisioning and configuration space. Specifically, a performance-prediction mechanism consists of a model, a procedure to seed the model, and an implementation of these two, providing a software tool to predict the behaviour of intermediate storage systems. In this way, these predictors can be used by the user to answer her specific *"What...if..."* questions, or as building blocks for an automated configuration solution that applies some heuristic to explore the configuration space.

Additionally, these mechanisms rely on monitoring information available from application-level measurements, not requiring changes to the storage systems nor specialized monitoring systems, making it possible for the seeding information to be extracted by the simple described benchmarks. In providing these solutions, this study also identifies a solution space, for the target context, where it is possible to provide a prediction mechanism with the characteristics described above while still providing predictions with enough accuracy to support user's decisions about provisioning and configuration of storage system.

Chapter 3

Predicting Performance for I/O Intensive Workflow Applications

This chapter presents a solution to address the problem of supporting storage provisioning, allocation and configuration decisions for workflow applications in the context of many-task computing and an intermediate storage system (see Section 2.2). It focuses on the answering the following two high-level questions: *"How can one leverage the characteristics of I/O intensive workflow applications to build a prediction mechanism for traditional performance metrics (e.g., time-to-solution and network usage)"*? and *"Which extensions does the performance predictor need in order to capture energy consumption behaviour, in addition to traditional performance metrics?"*

To enable selecting a good choice in a reasonable time, the proposed approach accelerates the exploration of the configuration space using a low-cost performance predictor that estimates, among other metrics, the total execution time of a workflow application in a given setup[1]. The predictor is lightweight (200x to 2000x less resource intensive than running the application itself) and accurate (80% of the evaluated scenarios have

[1] *Predicting Intermediate Storage Performance for Workflow Applications* [55] **Lauro Beltrão Costa**, Samer Al-Kiswany, Abmar Barros, Hao Yang, and Matei Ripeanu. In Proceedings of the 8th Parallel Data Storage Workshop, PDSW '13, pages 33 – 38. ACM, November 2013.

prediction errors smaller than 10%, and in the worst case scenario the prediction error is still within 20%) [2,3].

This chapter is organized as follows: Section 3.1 presents a summary of the problem. Section 3.2 presents the requirements and the design of the proposed prediction mechanism. Section 3.3 presents experience with using this prediction mechanism when evaluated independently with synthetic benchmarks, and in the context of making configuration choices for real applications focusing on turn-around time and cost. Section 3.4 presents an extension of the initial model presented in Section 3.2.3 that takes energy predictions into consideration, and summarizes a preliminary energy evaluation led by Hao Yang[4]. Section 3.5 describes further related work. Finally, Section 3.7 summarizes the chapter, limitations, and discusses some lessons learned during the exercise of designing the performance predictor.

3.1 Motivation

As discussed in Section 2.2.3, assembling workflow applications by putting together standalone binaries has become a popular approach to support large-scale science [33, 147, 169] (e.g., modFTDock [124], Montage[32] or BLAST [19]). The processes spawned from these binaries communicate via temporary files stored on a shared storage system. In this setup, the workflow runtime engines are basically schedulers that build and manage a task-dependency graph based on the tasks' input/output files (e.g., SWIFT [166], Pegasus [67]).

To avoid accessing the platform's backend storage system (e.g., NFS or GPFS or Amazon S3), an approach using what is called an *intermediate storage*

[2] *Supporting Storage Configuration and Provisioning for I/O Intensive Workflows* [58] **Lauro Beltrão Costa**, Samer Al-Kiswany, Hao Yang, and Matei Ripeanu. Under Review.

[3] *Supporting Storage Configuration for I/O Intensive Workflows* [57] **Lauro Beltrão Costa**, Samer Al-Kiswany, Hao Yang, and Matei Ripeanu. In Proceedings of the 28th ACM International Conference on Supercomputing, ICS '14. ACM, June 2014.

[4] *Energy Prediction for I/O Intensive Workflows* [171] Hao Yang, **Lauro Beltrão Costa**, and Matei Ripeanu. Proceedings of the 7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers, MTAGS '14. pages 1 – 6. ACM, November 2014.

system has been proposed [14, 169]. This approach advocates using some of the nodes allocated to the application to provide a shared storage system. That is, aggregating (some of) the resources of an application allocation to provide a shared temporary storage system dedicated to (and co-deployed with) the application.

Aggregating node-local resources to provide the intermediate storage offers a number of advantages discussed in Chapter 1, and in more detail in Section 2.2, for example: higher performance, higher efficiency, and incremental scalability. This scenario also opens the opportunity to optimize the intermediate storage system for the target workflow application since a storage system used by a single workflow, and co-deployed on the application nodes, can be configured specifically for the I/O patterns generated by that workflow [159].

Configuring the intermediate storage system, however, becomes increasingly complex. Users have to tune the configuration of storage techniques. These techniques have trade-offs that benefit both applications and the metrics of the optimization criteria differently. Further complicating this scenario, the user faces resource allocation decisions that often entail trade-offs between cost and application turn-around time [49, 91, 100]. Typical allocation choices involve deciding on the number of nodes to be used for the application in batch-computing environments, and specifying the nodes' type in cloud-computing environments (i.e., the hardware capabilities per-node in terms of compute, memory, storage and network capabilities).

This involves allocating resources and configuring the storage system (e.g., chunk size, stripe width, data placement policy, and replication level). Thus, the decision space revolves around: *provisioning the allocation* - deciding on the total number of nodes, deciding on node type(s), or node specification, for cloud environments; *allocation partitioning* – splitting these nodes or not between the application and the intermediate storage system; and setting *storage system configuration parameters* – choosing the values for several configuration parameters, e.g., , chunk size, replication level, cache/prefetching and data placement policies, for the intermediate storage system. In this context, manually fine-tuning the storage system configura-

tion parameters and allocation decisions is undesirable for multiple reasons (see Section 1.1).

In this space, typically the user's goal is to optimize a multi-objective problem, in at least two dimensions: to maximize performance (e.g., reduce application execution time) while minimizing cost (e.g., reduce the total CPU hours or dollar amount spent). The user can commonly describe her goal in the form of a utility function [152, 167], which reduces this multi-dimensional space (e.g., performance, cost, and energy consumption) into just one dimension: the utility[5].

More concretely, the user is often interested in answering specific questions. For example:

- How should the storage system be configured to achieve the fastest execution?
- What is the allocation that can achieve the lowest total cost?
- How should I partition the allocation among application and storage nodes to achieve the highest performance?
- What is the allocation that is most cost efficient (i.e., has lowest cost per unit of performance)?
- Which configuration can lead to energy savings?
- What is the time-to-solution impact of energy-centric tuning?

3.2 The Design of a Performance Prediction Mechanism

This section presents the design of a performance prediction mechanism for object-based storage systems in the context of workflow applications by targeting the following question *"How can one leverage the characteristics of I/O intensive workflow applications to build a prediction mechanism for traditional*

[5]This dissertation does not discuss the mapping of predicted performance metrics to utility in order to support configuration, which can be found in past work [152, 167].

performance metrics (e.g., time-to-solution and network usage)"? Specifically, given a particular storage system configuration, a workload description, and a characterization of the deployment platform based on a simple system identification process (e.g., storage nodes service time, network characteristics), the mechanism predicts the total application turnaround time.

Making accurate performance predictions for distributed systems is challenging. Since purely analytical models cannot provide adequate accuracy in most cases, simulation is the most commonly adopted solution. At the one end of the design spectrum, current practice (e.g., NS2 simulator [8]) suggests that while simulating a system at low granularity (e.g., packet-level simulation in NS2) can provide high accuracy, the complexity of the model and the seeding process, as well as the number of events generated, make accurately simulating large-scale systems unfeasible, and may reduce the applicability of this approach to small systems and/or short traces. At the other end of the spectrum, coarse grained simulations (e.g., PeerSim [123] or SimGrid [42]) scale well, but at the cost of lower accuracy.

Two observations were key to enable a solution that reduces simulation complexity and increases scalability: First, as the goal is to support configuration choice for a specific workload, achieving perfect accuracy is less critical, as long as it can support the user's configuration decisions (see Section 3.3). Second, this solution takes advantage of workload characteristics generated by workflow applications: relatively large files, distinct I/O and processing phases, single-write-many-reads, and specific data access patterns. These observations enable reduction in the simulation complexity by not simulating in detail some of the control paths that do not significantly impact accuracy. For example, the chunk transfer time is dominated by the time to send the data, hence not accounting for the time of the acknowledgment messages and all metadata messages will not tangibly impact accuracy).

The proposed solution uses a queue-based storage system model. The model requires three inputs from the user: (i) the storage system configuration for the scenario to be explored, (ii) a workload description, and

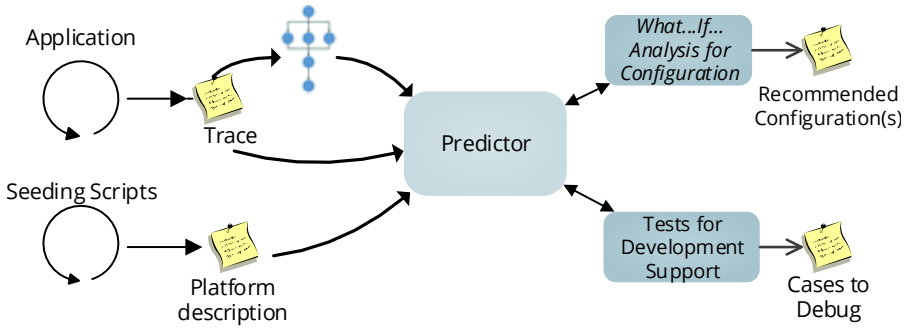


Figure 3.1: Predictor's input and possible use-cases. To predict an application's performance, the predictor (Section 3.2.3 and Section 3.2.6) receives the platform (Section 3.2.4) and workload (Section 3.2.5) descriptions. The use-cases include (i) *"What...If..." analysis* for supporting provisioning and configuration decisions - the focus of this chapter, and (ii) *performance tests* for supporting the development of a storage system, as described in Chapter 4.

(iii) the performance characteristics of each storage system component (i.e., system identification based on seeding scripts described in Section 3.2.4). The predictor instantiates the storage system model with the specific component characteristics and configuration, and simulates the application run as described by the workload description. Figure 3.1 shows how these different components interact with the predictor.

This remainder of this section discusses the requirements for a practical performance prediction mechanism (Section 3.2.1), and briefly presents the key aspects of the object-based storage system architecture modeled in this study (Section 3.2.2). Then, it focuses on the proposed solution: it presents the model (Section 3.2.3), the system identification process to seed the model (Section 3.2.4), an overview of the workload description (Section 3.2.5), and, finally, its implementation (Section 3.2.6).

3.2.1 Requirements

A practical performance prediction mechanism should meet the following, partially conflicting, requirements that bind the solution space:

Accuracy (R1). The mechanism should provide *adequate accuracy*. Although higher accuracy is always desirable, in the face of practical limitations to achieving perfect accuracy, there are decreasing incremental gains for improved accuracy. For example, to support configuration decisions, a predictor only needs to correctly estimate relative performance or trends resulting from changing a configuration parameter. Similarly, if two configurations offer near performance, perfect accuracy is less important as long as the prediction mechanism places their performance as similar. In fact, Pereira et al. [131] show that even a perfect replay of the storage system operations on the actual system, dedicated to the application, does not predict with 100% accuracy the performance of the storage system. (See evaluation presented in Sections 3.3 and 3.4.2).

Response Time and Scalability (R2). The predictor should enable exploration of the configuration space at a low cost. The mechanism should offer performance predictions quickly, have low resource usage and scale with both (i) the system size, and (ii) the I/O intensity of workflow applications. (See evaluation presented in Section 3.3.5).

Usability and Generality (R3). The predictor should not impose a burdensome effort to be used. Specifically, the bootstrapping/seeding process should be simple and it should not require storage system redesign (or a particular initial design) to collect performance measurements. Additionally, the predictor should model a generic object-based distributed storage design and using it should not require in-depth knowledge of storage system protocols and architecture. A discussion of usability is presented in the remaining part of this section as well as in Section 3.4.1.

Ability to explore "*What...if...*" scenarios (R4). A prediction mechanism should be able to support exploring hypothetical scenarios, such as scenarios that assume new or simply different platform (e.g., more machines, or faster machines). For example, "*What* would be the turnaround time of the application $\mathcal{A}$ *if* it uses 20 instead of 10 machines?" (See evaluation presented in Sections 3.3 and 3.4.2).

3.2.2 Object-based Storage Systems

The predictor focuses on the widely-adopted object-based storage system architecture (e.g., UrsaMinor [10], PVFS [85], and MosaStore [14]) described in detail in Section 2.1. This architecture includes three main components: a centralized metadata manager, storage nodes, and a client-side system access interface (SAI). The manager maintains the stored files' metadata and system state. Files are split into chunks stored across several storage nodes, and the client SAIs implement data access protocols.

Currently, the prediction mechanism assumes that the parameters provided by MosaStore (see Section 2.1.3) are configurable - e.g., chunk size, stripe width, replication level, cache size, and data placement policies. This approach can be extended to support other configuration parameters.

3.2.3 System Model

The proposed solution uses a queue-based storage system model for the system components' operations and their interactions as summarized in Figure 3.1.

All participating machines are modeled similarly, regardless of their specific role (Figure 3.2): each machine hosts a network component, and can host one or more system components - each modeled as a service with its own service time per request type and a First In, First Out (FIFO) queue.

Each system component and its infinite queue represent a specific functionality: The *manager* component is responsible for storing files' and storage nodes' metadata. The *storage* component is responsible for storing and replicating data chunks. Finally, the *client* component receives the

System Deployment	
Number of Storage Nodes	N^{sm}
Number of Client Nodes	N^{cli}
Collocation of Storage and Client Modules	$Colloc$
Performance	
Manager Service Time	μ^{ma}
Storage Module Read Service Time	μ^{smRead}
Storage Module Write Service Time	$\mu^{smWrite}$
Client Service Time	μ^{cli}
Remote Network Service Time	μ^{remNet}
Local Network Service Time	μ^{locNet}

Table 3.1: Model parameters describing the platform. To instantiate the storage system model, one needs to specify these parameters. The number of components in the system can be freely specified, the only restriction is having $N^{sm} > 0$ and $N^{cli} > 0$. The service times are part of the platform performance description and its identification is described in Section 3.2.4. For simplicity, part of the description in this chapter uses simply μ^{sm} referring to the service time of a storage operation in the storage module, which is either read or write; similarly, μ^{net} refers to μ^{remNet} or μ^{locNet}.

read and write operations from the application, implements the storage system protocols at high-level by sending control or data requests to other services, and it communicates again with the application driver once a storage operation finishes. Each of these storage system components is modeled as a service that takes requests from its queue and uses the network service to send requests and responses (see Table 3.1). The application driver can also issue requests directly to the client service queue.

The model captures the four main storage operations: open, close, read, write. As a rule, the prediction mechanism accurately models the data paths of the storage system at chunk-level granularity, and the control paths at a coarser granularity: it models only one control message to initiate a specific storage function while an implementation may have multiple rounds of control messages.

The network component and its in- and out- queues model the network-related activity of a host. Key here is to model network-related contention while avoiding modeling the details of the transport protocol (e.g., dealing with packet loss and re-transmission, connection establishment and tear-

down details). These details can improve the accuracy of the predictor, but at a cost of longer simulations. The additional accuracy is not needed to guide the configuration for the applications that this research targets (the evaluation in Section 3.3 provides evidence for this observation).

The requests in the out-queue of a network component are broken in smaller pieces that represent network frames, and sent to the in-queue of the destination host. Once the network service processes all the frames of a given request in the in-queue, it assembles the request and places it in the queue of the destination service.

In addition to the storage system modeling, the prediction mechanism captures part of the execution behaviour of the application. The predictor also performs scheduling according to the input describing the application, which includes a tasks' dependency graph (capturing workflow execution plan) used for scheduling and data placement purposes (see Figure 3.1 and Section 3.2.5 for more details).

The system components can be collocated on the same host (e.g., the client and storage components running on the same host). In this situation, requests between collocated services also go through the network, but have a faster service time than remote requests - representing a loopback data transfer (Section 3.2.4).

3.2.4 Model Seeding: System Identification

To instantiate the storage system model, one needs to specify the number of storage (N^{sm}) and client components (N^{cli}) in the system, the service times for the network (μ^{net}, it currently captures latency and bandwidth together), and the system components (storage - μ^{sm}, manager - μ^{man}, and client - μ^{cli}), and the storage system configuration. The number of components in the system and the storage system configuration (Section 2.1.3) can be freely specified and depends on the scenario currently under the *"What... if..."* analysis. The service times are part of the platform description and its identification is described in this section[6].

[6]In fact, the storage module has two main operations with different service times (μ^{smRead} and $\mu^{smWrite}$), which are seeded in a similar way. For simplicity, this section uses μ^{sm} to

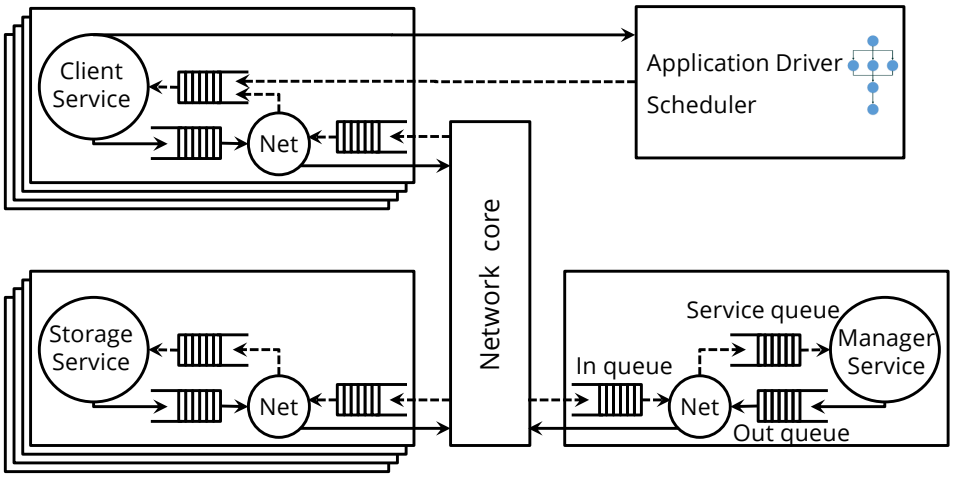


Figure 3.2: A queue-based model of a distributed storage system. Each component (manager, client component, and storage component) has a single system service that processes requests from its queue. Additionally, each host has a network component with an in- and out- queue. The network core connects and routes the messages between the different components in the system and can model network latency and contention at the aggregate network fabric level. Solid lines show the flow going out from a storage system component, while dashed lines show the in-flow path. The application driver and the scheduler are responsible for decide which client will execute a specific task and issue the operations of such task. Table 3.1 lists the model parameters describing the platform performance.

Compared to past work, this approach focuses on making the *system identification process simple, by not being intrusive as no changes are required to the storage system or kernel modules* in order to satisfy R3. Additionally, the seeding process relies on application-level calls and on a deployment of up to only three machines regardless of the size of the system simulated, to keep the process simpler and less costly.

The system identification process is automated with a script as follows. To identify the service time per chunk/request (μ^{net} - either local or remote), a script runs a network throughput measurement utility tool (e.g., iperf), to measure the throughput of both remote and local (loopback) data transfers.

describe the seeding procedure; similarly it uses μ^{net} referring to μ^{remNet} or μ^{locNet}.

Then, this script measures the time to read/write a number of files to identify client and storage service time per data chunk. To this end, the system identification script deploys one client, one storage node and the manager on different machines, and writes/reads a number of files. For each file read/write, the benchmark records the total operation time. The script computes the average read/write time (T^{tot}). The number of files read/written is set to achieve 95% confidence intervals with $\pm$5% relative errors.

The operation total time (T^{tot}) includes the client side processing time (T^{cli}), the storage node processing time (T^{sm}), the total time related to the manager operations (T^{man}), and the network transfer time (T^{net}).

$$T^{tot} = T^{cli} + T^{sm} + T^{man} + T^{net} \tag{3.1}$$

The network service time for the network (μ^{net}) uses a simple analytical model based on network throughput, and is proportional to the amount of data to be transferred in a packet.

To isolate just $T^{cli} + T^{man}$, the script runs a set of reads and writes of 0-size. This forces a request to go through the manager, but it does not touch the storage modules. Since decomposing T^{cli} and T^{man} is not possible without probes in the storage system code, the script assumes $T^{cli} = 0$ and associates the whole cost of 0-size operations with the manager to obtain μ^{man} from T^{man}. While network measurement utility tool can estimate T^{net}, the script can infer $T^{cli} + T^{man}$, and therefore

$$T^{sm} = T^{tot} - T^{net} - T^{man} \tag{3.2}$$

To obtain the service time per chunk, the times are normalized by the number of chunks. Therefore,

$$\mu^{sm} = \frac{T^{sm}}{\left(\frac{dataAmount}{chunkSize}\right)} \tag{3.3}$$

In addition to the possibility of seeding the service times based on

Item	Description
$\mathcal{A} = (u_a^s, u_a^f, \mathcal{T})$	Application $\mathcal{A}$
$\mathcal{T}$	the set of tasks in application $\mathcal{A}$
u_a^s	application start time
u_a^f	application finish time
$t_k = (v_k^i, v_k^{cli}, v_k^e, v_k^c, \mathcal{O}_k, \mathcal{D}_k)$	$t_k \in \mathcal{T}$
v_k^i	task start time
v_k^{cli}	machine that executes t_k
v_k^e	tasks' processing time
v_k^c	workflow runtime overhead for t_k
$\mathcal{O}_k$	task's storage operations (client trace)
$\mathcal{D}_k$	t_k's dependencies
o_k^j = (type, timestamp, file, size, offset)	$o_k^j \in \mathcal{O}_k$

Table 3.2: Summary of the workload description's variables.

the average obtained by the seeding scripts, the service times can also be provided by a statistical distribution. The current implementation of the seeding scripts extracts an empirical distribution.

3.2.5 Workload Description

The predictor takes as input a description of the workload. This description contains two types of information: (i) a trace of storage operations per task/client (i.e., open, read, write, close calls with timestamp, operation type, size, offset, and client id), and (ii) a task-dependency graph capturing the workflow execution plan for scheduling and data placement purposes (see Figure 3.1). Table 3.2 summarizes the information used to describe and simulate the workload.

Formally, let $\mathcal{A} = (u_a^s, u_a^f, \mathcal{T})$, where $\mathcal{T}$ is the set of tasks in application $\mathcal{A}$, u_a^s is the start time of the application $\mathcal{A}$ and of the simulation. u_a^f is the finish time of $\mathcal{A}$, it is not known at instant u_a^s, and estimating it is the main goal of the prediction mechanism.

Let $t_k \in \mathcal{T}$ and $t_k = (v_k^i, v_k^{cli}, v_k^e, v_k^c, \mathcal{O}_k, \mathcal{D}_k)$, where v_k^i is the start time of the task, v_k^{cli} is the machine that executes t_k, v_k^e is the execution time of the task, not including the storage operations, v_k^c is the workflow runtime overhead, such as scheduling operations and task execution launch

(e.g., Secure Shell (SSH) invocation) related to t_k, $\mathcal{O}_k$ is the set of storage operations performed by task k, and $\mathcal{D}_k$ is the set of tasks on which t_k depends.

Let $o_k^j \in \mathcal{O}_k$ and $o_k^j = (type, timestamp, file, size, offset)$, where o_k^j describes a storage operation of a given $type$, $size$, and $offset$ on $file$ at a specific $timestamp$ after v_k^i. v_k^i and v_k^{cli} are not known at instant u_a^s, and depend on how the simulation progresses.

v_k^e and $\mathcal{O}_k$ are part of the workload description that serves as input for the predictor. Note that $\mathcal{O}_k$ may also contain operations to specify a specific file configuration (e.g., set an extended attribute in MosaStore as described in Section 2.2.4). $\bigcup_{k=0}^{|T|} \mathcal{D}_k$ can also be represented as a DAG and is used for scheduling in the simulation.

The client traces ($\mathcal{O}_k$) are obtained by running and logging the application storage operations. The execution plan can be provided by the workflow description (as the one used by Swift [166]), by an expert user, derived from the workflow runtime and the storage system information [162], or extracted from log files ($\mathcal{D}_k$ is directed related to the input files of t_k since the tasks use files to communicate).

Currently, the workflow execution plan is obtained from pyFlow scheduler [60] and client traces from MosaStore storage logs, which require no further modification for this work. A FUSE wrapper was also developed at NetSysLab to log the storage operations, in case the storage system does not provide the needed information.

Preprocessing Storage Operations

The predictor preprocesses the logs to infer the storage operations' elapsed times, the workflow runtime overhead (v_k^c), each task's execution time (v_k^e), and inter-arrival times.

In a subsequent step, the predictor preprocesses the description of the storage operations to potentially reduce the amount of events to be simulated. Specifically, this preprocessing phase aggregates some of the storage operations issued by the client, and analyses the impact of the cache

size on the amount of data that each operation would transfer. In this way, the simulations focus on predicting the behaviour of the system when bringing the data from the storage modules to the client module, which can be an operation local to the machine.

Regarding the *aggregation*, the predictor relies on the fact that workflow applications have almost distinct phases of reads, processing and writes, focusing on a small set of files each. Relying on this characteristic, the predictor aggregates all the immediately subsequent operations of the same type on a given file. For example, suppose that an application performs 10 write operations of 256KB each; instead of passing those 10 write operations to the simulator, the preprocessing phase aggregates them, and passes just one write operation of 2.5MB to be simulated.

The amount of the data to be transferred, from storage modules to client modules or vice versa, depends on the *cache size* and on the order of the operations, but not on the network nor on the data-distribution – which still affect whether the transfers are local or remote. In this way, the preprocessing phase verifies the cache-hit ratio of the operations based on the cache size and order of the operations, and infers the actual amount of data to be transferred. It then aggregates the result of this analysis in one or few storage operations. Consequently, the simulator has to process fewer storage operations and fewer network events.

3.2.6 Model Implementation: The Simulator

The above model is implemented as a discrete-event simulator in Java. The simulator receives as input: (i) a summarized description of the application workload (Section 3.2.5), (ii) the system configuration (currently, it supports the parameters described in Section 2.1.3), (iii) the deployment parameters (number of storage nodes and clients, whether they are collocated on the same hosts - *colloc*), and (iv) a performance characterization of system components: service times for network, client, storage, and manager (Section 3.2.4).

Once the simulator instantiates the storage system, it starts the appli-

cation driver that emulates processing the application workload (Section 3.2.5). The driver functionalities can be divided into two main types: (i) functionalities related to the workflow runtime, and (ii) functionalities related to the operations of the actual task execution.

The workflow runtime functionalities focus on scheduling and currently supports two heuristics: (i) workqueue [62], and (ii) data-location aware [60, 142]. In the workqueue alternative, whenever the simulated environment has client machines available (i.e., not running any task), the driver schedules a task to that machine. For the location-aware, the driver gives priority in the allocation to the machines that already have the input files in an approach similar to the one described by Santos-Neto et al. [142] and evaluated in the context of many-task applications by my collaborators and I [60].

Once the task is scheduled (v_k^{cli} is chosen), the driver creates the events corresponding to the workflow runtime overhead, task processing, and storage operations (i.e., v_k^c, v_k^e, and $\mathcal{O}_k$) and places them in the client service queue. File-specific configuration (as proposed by UrsaMinor [10] and MosaStore [159]) is described as part of the operations in the workload description.

As in a real system, the manager component maintains the metadata of the system (i.e., implements data placement policies, and keeps track of file to chunk mapping and chunk placement).

The processing of these initial events may result in the creation of other events. To make the process clearer, consider the following example for a file write operation:

1. A client contacts the manager asking for free space, the manager replies specifying a set of storage services with free chunks.

2. The client requests each storage service to store chunks in a round-robin fashion based on the set received from the manager.

3. After processing a request to store a chunk, each storage service replies to the client acknowledging the operation success.

Operation	Description Summary
`allocateChunks`	A client may request the manager to allocate chunks on the storage systems when the previously allocated chunks are used. The manager performs such allocation according to the data-placement policy in use. It is not available to the application.
`read`	The application issues a read operation on a client, which results in several `readChunk` from the client to storage modules storing the requested chunks. A previous call to `open` may be necessary.
`readChunk`	A client requests a specific chunk from a storage module. It is result of a `read` operation. It is not available to the application.
`open`	A client requests the chunk map and other metadata information(e.g., file tags) from the manager.
`write`	The application issues a write operation on a client, which results in several `writeChunk` from the client to storage modules according to the chunks received from a previous call to `allocateChunks`. In the end of this operation, the client send an updated version of the chunk map to the manager. A previous call to `open` or to `allocateChunks` may be necessary.
`writeChunk`	A client sends a chunk to be stored by a storage module. If replication is enabled, the storage module may issue other requests of `writeChunk` to other storage modules.

Table 3.3: Summary of the main operations modeled in the simulator; these operations also have a callback associated with each one.

4. After sending all the chunks, the client sends the chunk-map to the manager.

5. Once the manager acknowledges, the client returns a notice of success to the application driver.

A write operation generates two requests to the manager and one request per chunk to the storage nodes. Table 3.3 summarizes the main operations modeled for this study and their key interactions.

The manager implements a number of data placement policies. The default policy selects, for a write operation, a "stripe-width" of storage services. To model per-file optimizations, the client can overwrite system-wide configurations by requesting the manager to use a specific data placement policy. For example, the client may require that a file is stored locally, that is, on a storage service that is located on the same host. In this case, the manager attempts to allocate space on that specific storage service for that write operation. The file-specific data placement policy request is part of the workload description.

3.2.7 Modeling Methodology Remarks

During the modeling phase, the requirements were partially conflicting: to provide adequate accuracy (R1) while keeping the model and its seeding procedure simple enough to be fast and scalable (R2), and easy to use (R3).

The focus was to be as simple as possible to satisfy R1 for the target domain, not to be a comprehensive solution for distributed storage systems nor for all possible domains of applications. Thus, the initial phase of modeling targeted only the key interactions between system components. Modeling all system subcomponents and all their interactions in detail would be too complex. Such complexity could improve prediction accuracy, but would have significant drawbacks: a significantly more complex model - as complex as the actual storage system and the underlying environment (e.g., network protocols, operating system buffers, scheduling), a complex seeding process, lower scalability, and the loss of the model's generality. Furthermore, the improvement in accuracy may not add much value (e.g., when the prediction mechanism is used to decide among system configurations).

From the initial phase, the modeling exercise followed a top-down approach: from the simplest model to adding more components or interaction details until the accuracy of all the predictions was within 10% of actual performance (and the median error was within 5%) for a set of microbenchmarks that covered approximately 30 different scenarios.

3.2.8 Summary

The predictor takes as input a description of the workload in the form of (i) a per client I/O operations trace, and (ii) the task-dependency graph for scheduling (see Figure 3.1).

Section 3.3 evaluates requirements R1 (Accuracy) and R2 (Response Time and Scalability) and discusses how they are met by this proposed prediction mechanism. Requirement R4 is a functional requirement and, besides the supported range of different scales and configurations described earlier, Section 3.3 also assesses R4 in the context of R1 and R2.

Requirement R3 (Usability and Generality) regards mainly how one can provide the input necessary to the predictor with little human effort. For R3, two main points were discussed in the beginning of Section 3.2 and in Section 3.2.4: (i) how to avoid a particular initial system design or a specialized monitoring system, and (ii) how the process of gathering the input for the performance predictor can be completely automated. Additionally, meeting R3 is also a consequence of the simplicity aspects discussed in Section 3.2.7.

Section 3.7 summarizes how some decisions made during the modeling exercise affect the requirements, and what are the limitations of this approach. Section 6.2 discusses further the main limitations and proposes how to address them as future work.

3.3 Evaluation

This section presents the evaluation of the mechanism's prediction accuracy and, more importantly, it demonstrates the mechanism's ability to support correctly identifying a quasi-optimal configuration for a specific application (Requirement R1 from Section 3.2.1). To this end, this evaluation covers a set of synthetic benchmarks and real applications. The synthetic benchmarks are designed to mimic the access patterns [159] of real workflow applications.

To understand how the prediction mechanism can be used in a real setup, two real workflow applications are used: BLAST [19] and Montage [106].

The goal is to evaluate the mechanism's ability to predict time-to-solution to support user's decisions about storage configuration and allocation.

Storage system. The deployment uses MosaStore as an intermediate storage system. The storage nodes use RAMDisks, which are frequently used to support workflow applications as intermediate storage (refer to Section 2.2.1 for a description of intermediate storage) since they have higher performance and are the only option in most supercomputers that do not have spinning disks (e.g., IBM BG/P machines).

Testbeds. To demonstrate the ability of the prediction mechanism to perform well independently of hardware deployment and scale, this section focuses on two testbeds. The first testbed (**TB20**), with 20 machines, is part of the NetSysLab cluster. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4GB RAM, and a 1Gbps Network Interface Controller (NIC) running Fedora 14 Linux OS (kernel 2.6.33.6). The second testbed (**TB101**), used for larger scale experiments, includes 101 nodes on Grid5000 'Nancy' cluster [41]. Each machine has Intel Xeon X3440 4-core, 2.53-GHz CPU, 16GB RAM, 1Gbps NIC, and 320GB SATA II.

Evaluation Metric. The evaluation focuses on prediction accuracy by comparing the predicted execution time and the actual execution time. Formally, this section reports prediction error as defined by $|1 - \frac{\mathrm{Time}_{\mathrm{pred}}}{\mathrm{Time}_{\mathrm{actual}}}|$, where $\mathrm{Time}_{\mathrm{pred}} = u_a^f$ (Section 3.2.5) and $\mathrm{Time}_{\mathrm{actual}}$ is the actual runtime of the benchmark.

Plots report the average of 10 trials. For actual performance, the figures show the average execution (turnaround) time and standard deviation (in error bars). The number of trials used to calculate the average varies and is presented in each scenario below. The sample size used is enough to guarantee a 95% confidence level according to the procedure described by Jain [98].

Deployment details. In all the experiments, one node runs the metadata manager and the workflow coordination scripts, while the other nodes run the storage nodes, the client SAI, and the application processes. The network was shared with other applications out of our control (5 other machines for

TB20, and 43 machines for TB101). The simulator is seeded according to the procedure described in Section 3.2.4.

3.3.1 Synthetic Benchmarks: Workflow Patterns

This section evaluates the accuracy of the prediction mechanism in capturing the system behaviour with multiple clients, multiple applications, and different data-placement policies designed to support workflow applications [159]. The synthetic benchmarks mimic the common data access patterns of workflow applications: pipeline, reduce, and broadcast (Figure 3.3). These are the most popular patterns uncovered by studying over 20 scientific workflow applications by Wozniak and Wilde [169], Shibata et al. [147], and Bharathi et al. [33]. Additionally, this past work shows that workflow applications are typically a combination of these studied patterns, with one pattern per stage.

The *synthetic benchmarks are designed to explore the limitations of the predictor* as they are composed exclusively of I/O operations, which generates high network and storage contention in the system.

Summary of results. The predictor has good accuracy: our approach leads to prediction errors of 5% on average, lower than 8% in 80% of the studied scenarios, and within 14% in the worst case. More importantly, the mechanism correctly differentiates between the different configurations and can support choosing the best configuration for each evaluated scenario.

Experimental setup. The label *DSS* is used for experiments with the Default Storage System configuration (DSS): client and storage modules run on all machines, client stripes data over all 19 machines of the TB20 testbed, and no optimizations are enabled for any data-access pattern. The label WOSS refers to the system configuration optimized for a specific access pattern (including data placement, stripe width or replication) [159]. All WOSS experiments assume data location-aware scheduling: for a given compute task, if the input file chunks exist on a single storage node, the task is scheduled on that node to increase access locality. For actual performance, the figures presented in this section show the average execution (turnaround)

time and the standard deviation for 15 trials. Section 2.2.3 presents a more detailed description of these patterns and their optimizations.

The goal of showing results for two different configuration choices is two-fold: (i) to demonstrate the accuracy of the predictions for two different scenarios, and (ii), more importantly, to show that the predictions correctly indicate which configuration is the most desired. To understand the impact of data size, for each benchmark, the experiments use three workloads labeled as *medium* (data size is indicated in Figure 3.3), the *small* (10x smaller than medium), and where possible (i.e., workload fits in the RAMDisks), a 10x larger, *large* workload. These sizes cover a range of different applications, as the ones presented in Sections 3.3.3 and 3.3.4 and studied by Wozniak and Wilde [169], Shibata et al. [147], and Bharathi et al. [33]. Results for a *small* workload are omitted since they exhibit a similar performance between different configurations and the predictions are inside the confidence interval. Thus, while the predictions are accurate and potentially useful, analyzing them does not make a useful scenario for this evaluation.

Pipeline benchmark. A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task in the chain (Figure 3.3). A pipeline-optimized storage system will store the intermediate pipeline files on the storage node co-located with the application. Later, the workflow scheduler places the task that consumes the file on the same node, increasing data access locality. Here, 19 application pipelines run in parallel and go through three processing stages that read input from the intermediate storage and write the output to the intermediate storage. The large workload produces too much data to fit in an in-memory intermediate storage system in the TB20 testbed. Section 3.7 discusses initial results for spinning disks.

Evaluation of the results. For the optimized configuration (WOSS), the predictor has almost perfect accuracy (Figure 3.4). For the default data placement policy (DSS), however, predicted times are 9% shorter than the actual results. For this case, all clients stripe (write) data to all machines in the system; similarly, all machines read from all others. This creates complex

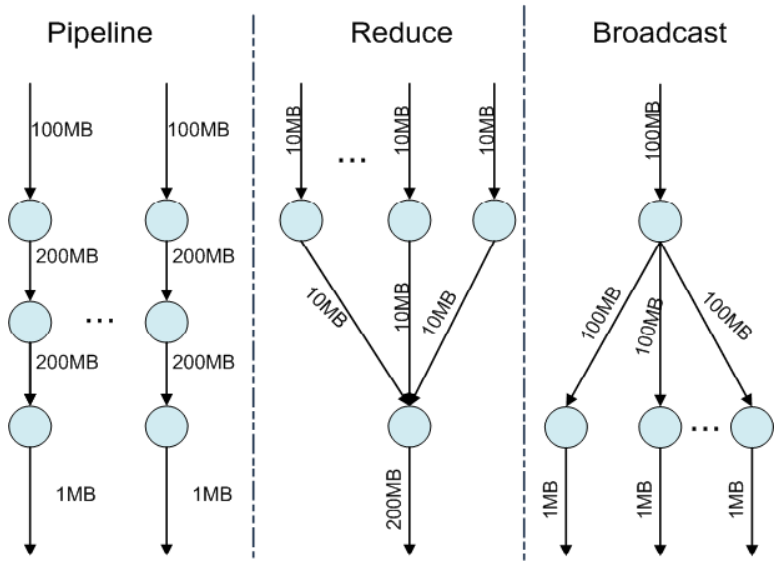


Figure 3.3: Pipeline, Reduce, and Broadcast benchmarks. Nodes represent workflow stages and arrows represent data transfers through files. The file sizes represent the *medium* workload. The part of the flow that is repeated, run on over 19 machines in this evaluation.

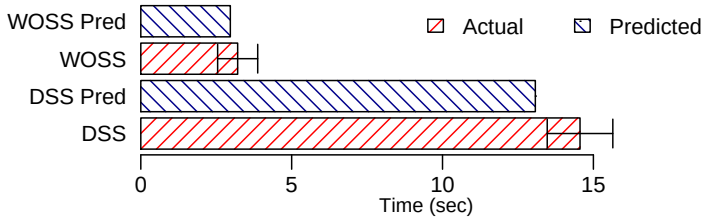


Figure 3.4: Actual and predicted average execution time for the pipeline benchmark with a medium workload.

interactions among all components in the system, leading to contention and chunk transfer retries due to connection initiation timeouts caused by network congestion, which should be the main source of prediction inaccuracies.

Reduce benchmark. In this benchmark, a single compute task uses input files produced by multiple tasks. 19 processes run in parallel on different nodes, consume an input file, and produce an intermediate file each. In the

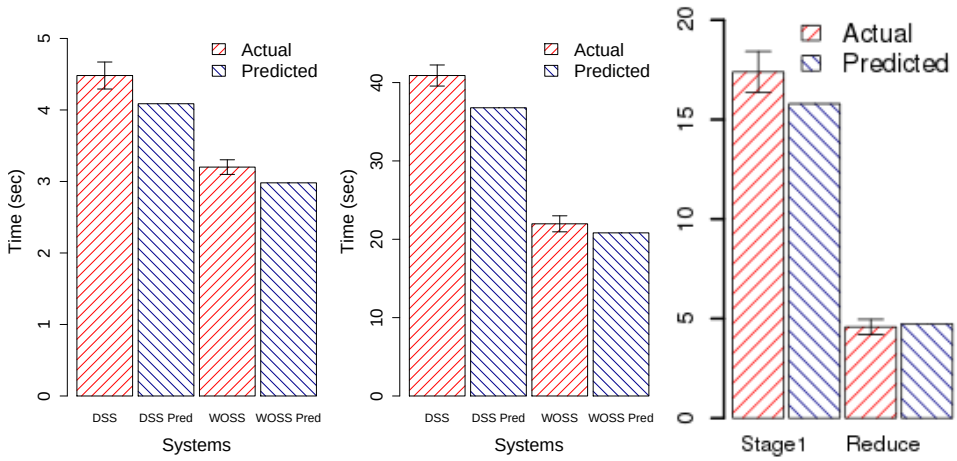
next stage of the workflow, a single process reads all intermediate files and produces the final output file. A possible data placement optimization is to use collocation - placing all input files on one node and exposing their location, which will later be used by the scheduler to place the reduce task on that machine. For WOSS configuration, the collocation optimization is enabled for the files used in the reduce stage; for the remaining files, the locality optimization is enabled.

Evaluation of the results. Similar to the pipeline benchmark, predictions for the reduce benchmark are close to the actual performance. In fact, they are within 9% of the actual average for a medium workload, and 13% of the actual performance for a large workload (Figure 3.5). More importantly, they capture the relative improvements that the pattern-specific data placement policies bring. Note that Figure 3.5b captures the behaviour of a heterogeneous scenario: to accommodate the amount of data produced, a faster machine with a larger RAMDisk runs the reduce stage. Despite this heterogeneity, the predictor captures the system performance with accuracy similar to a homogeneous system.

When the collocation and locality optimizations are not enabled, the challenge of capturing the system behaviour exactly is similar to the pipeline case: capturing the complex interactions among all machines in the system. When the specific data placement is enabled, however, the challenge is different: there is a high contention created by having several clients writing to the same storage node (the one that performs the reduce phase). Figure 3.5c shows the results per-stage for the two stages of the large workload separately to show how the predictor captures these cases.

Broadcast benchmark. Nineteen processes running in parallel on different machines consume a file that is created in an earlier stage by one task. A possible optimization for this pattern is to create replicas of the file that will be consumed by several tasks.

Evaluation of the results. Figure 3.6 shows the results for the broadcast pattern with a medium workload using a WOSS system configured with 1, 2, or 4 replicas (the large workload shows a similar trend and is omitted here). For this benchmark, all predictions matched the actual results: predictions



(a) Medium **(b)** Large **(c)** Large per Stage for WOSS

Figure 3.5: Actual and predicted average execution time for the reduce benchmark for the medium, large workloads, and per stage for large workload.

are inside the interval of average $\pm$ standard deviation, with just a 1-3% difference from the average.

This experiment highlights an interesting case for the predictor. According to the structure of the pattern and the results reported by Vairavanathan et al. [159], creating replicas could improve the performance of the broadcast pattern. The results, however, show that creating replicas does not really improve performance in this case. While creating replicas could improve performance, the results show that creating replicas does not really help here since data striping already avoids the contention on a single node. Creating replicas reduces the pressure on a given machine (as Vairavanathan et al. [159] show to be the case for this benchmark), but this benefit is cancelled by the overhead of creating a replica. The predictor captures the impact of these different configurations.

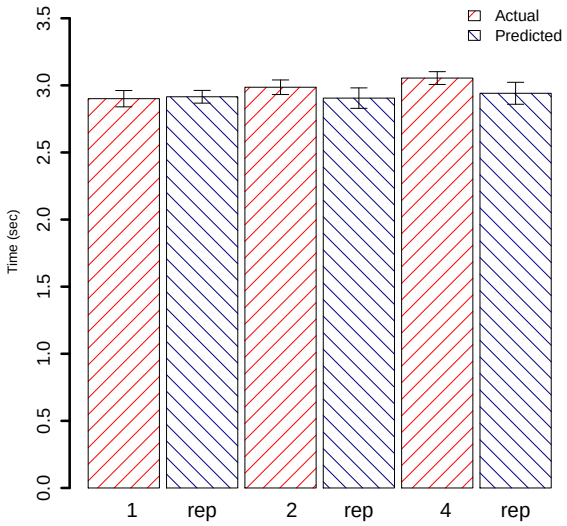


Figure 3.6: Actual and predicted performance for broadcast benchmark and medium workload. The experiment uses the WOSS system while varying the replication level.

3.3.2 The Pipeline Benchmark at Scale

This section expands the analysis of the synthetic benchmarks to answer the following questions:

- How accurate is the predictor's estimates for a different platform?
- How does the predictor capture the behaviour of larger scale systems for the synthetic benchmarks?

To answer these questions, this section presents the results for the pipeline benchmark at a larger scale on a Grid 5000 testbed with 101 machines (TB101). The pipeline benchmark is used because it is the one with the largest gap between the predicted and the actual execution, and it is the most I/O intensive - stressing the network and the metadata manager, a component well-known for being a potential bottleneck for this type of cluster-based storage systems. This section shows the results for this benchmark for a weak scaling set-up using three different scales (25, 50, and

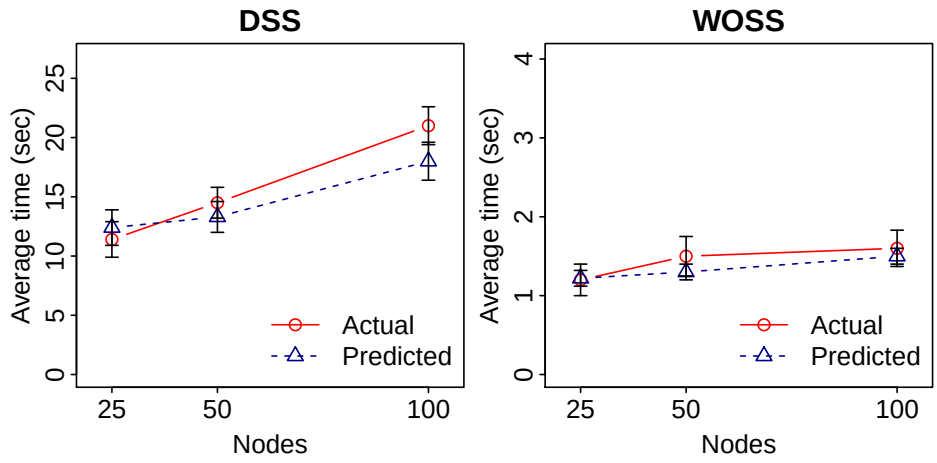


Figure 3.7: Actual and predicted average execution time for the pipeline benchmark with a medium workload for 25, 50, and 100 nodes on testbed TB101. Note the different scales for the x-axis.

100 nodes), the medium workload, and the two configurations (DSS and WOSS). For this case, the DSS option uses stripe-width of 20 instead of 19 as for TB20.

Figure 3.7 presents the results. The predictor produces estimates that differ on average by 15% from the actual time, are within 22% of the actual results for all cases, and close to the interval delimited by the standard deviation. More importantly, the predictions indicate which configurations lead to higher performance.

3.3.3 Supporting Decisions for a Real Application

Section 3.3.1 presents an evaluation of the predictor's ability to accurately estimate the turnaround time of synthetic benchmarks, and the impact of different data placement optimizations. This section targets more complex scenarios where the user has to deal with a real application, allocation decisions, as well as the choice of the storage system configuration. Further, while the previous section 3.3.1 evaluates prediction accuracy when the application and storage system are co-deployed, this section evaluates

accuracy when they are deployed on separate nodes.[7]

Specifically, this section demonstrates the predictor's ability to properly guide the user, or a search algorithm, to the desired configuration, focusing on two provisioning scenarios:

Scenario I assumes that the user has full access to a fixed-size cluster – a common set-up in several university research labs. The problem: *How should the system be partitioned between the application and the intermediate storage and what will the intermediate storage system configuration be for the best overall performance?*

Scenario II explores the provisioning problem with cost constraints (e.g., in HPC centers with limited user budget or cloud environments). The problem: *For a fixed workload, what is the cost vs. turnaround time trade-off space among the deployment options?*

Workload. The experiments explore these two scenarios with a real workflow application: BLAST [19] is a DNA search tool for finding similarities between DNA sequences. In the BLAST workflow, each node receives a set of DNA queries as input (a file for each node with 200 search queries) and all nodes search the same DNA database file stored on the intermediate storage. Each machine produces one output file, and the files are combined at the end of the application execution. The input files are transferred to the intermediate storage system prior to application execution. The traces are collected based on one execution of the application, as described in Section 3.2.5.

Deployment scenario. Among the 20 machines in the testbed TB20, one node coordinates BLAST tasks' execution and runs the storage system manager. The remaining nodes either execute tasks from the workload or act as storage nodes.

Experimental methodology. The plots report the average of at least 20 runs, leading to 95% confidence intervals for all experiments. Since

[7]Several factors may influence the decision to collocating (or not) the components (e.g., limited amount of RAM in BG/P). Past work employed both approaches (e.g., [14, 52, 156, 174]) and, thus, this evaluation explores both.

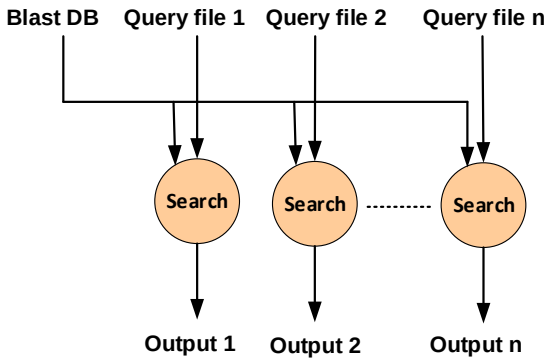


Figure 3.8: The BLAST application. The database is used by all nodes that search in parallel for different queries.

these confidence intervals are small (less than 5% of actual values), they are omitted to reduce clutter.

Scenario I: Configuring a Fixed-size Cluster

This scenario explores the following question: Given a fixed size cluster, *how should the nodes be partitioned between the application and the intermediate storage, and what is the intermediate storage system configuration that yields the highest application performance?*

Figure 3.9 shows the application execution time for different partitioning and storage system configurations. For this application, *chunk size* is the configuration parameter that has the highest impact on performance, thus, to limit the points plotted in the figure, this evaluation focuses on it only. The predictor correctly captures the lack of impact for other parameters though additional runs exploring these parameters. Additionally, by evaluating chunk size and not collocation of storage with client nodes, this section covers results for a configuration parameter not covered in Section 3.3.1.

Figure 3.9 highlights several important points. First, the performance difference between the different configurations is significant: up to a difference of 10x between the best and the worst configuration, even for the same chunk size. Second, the results show that the system achieves the fastest processing time with a partitioning of 14 application nodes and 5

storage nodes, and a chunk size of 256KB (4x smaller than the default size) – a non-obvious configuration beforehand. Third, the experiment shows that the predictor accurately captures the system performance under different partitioning strategies, and storage system configurations. Actually, the overall error of the predictions is small: up to 10% of the average, and always within the standard deviation. By comparing those to the synthetic benchmarks, they are also smaller since there is less stress on the storage system. Finally, the most important point is that the predictor can, in fact, correctly lead the user or a search algorithm to the desired configuration.

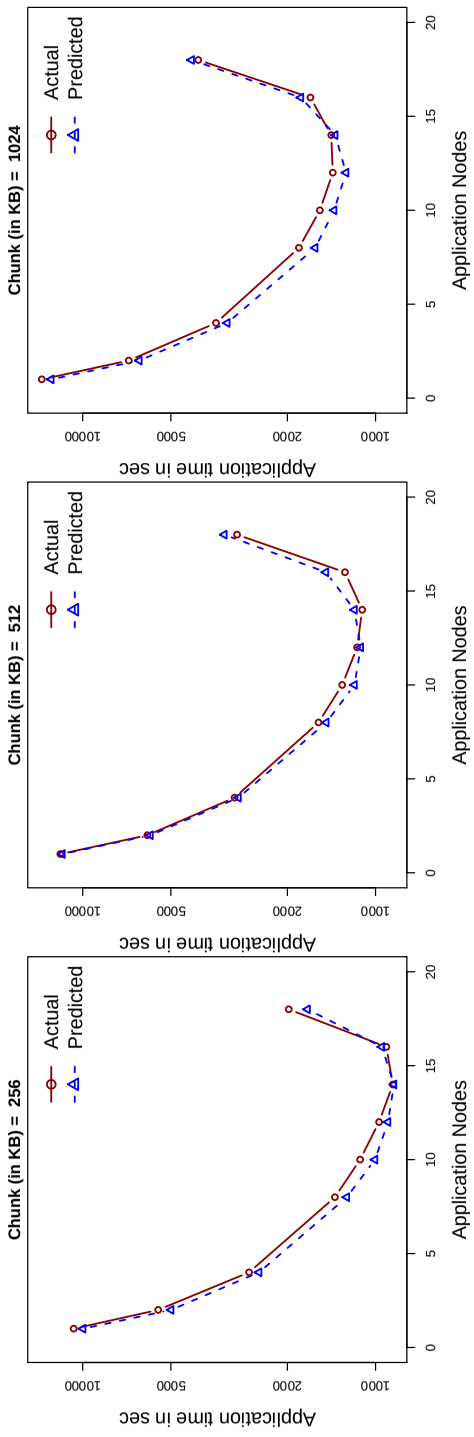


Figure 3.9: Application runtime (log-scale) for a fixed-size cluster of 20 nodes. X-axis represents number of nodes allocated for the application (storage nodes = 20 - application nodes). The three plots represent runtime for different configurations (chunk sizes).

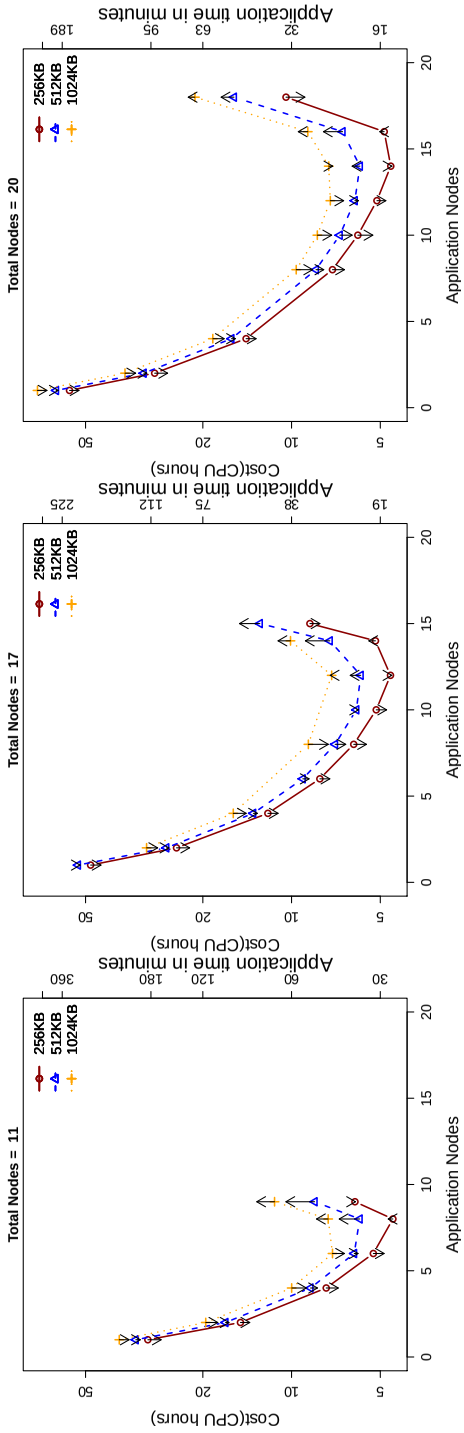


Figure 3.10: Allocation cost (total CPU-hours on the left Y axis, log-scale) and application turnaround time (right Y axis, log-scale, different scale among plots) for fixed size clusters of 11, 17, and 20 nodes while varying the chunk size. X-axis represents number of nodes allocated to the application (storage nodes = 20 - application nodes). The figures show the actual (lines) and the predicted (arrows) cost/performance.

Scenario II. Provisioning in an Elastic and Metered Environment

This scenario assumes an environment where users are charged proportional to the cumulative CPU-hours used and have a more complex trade-off between cost and time-to-solution; for example, they aim for the best application turnaround within a certain dollar budget. This scenario aims to explore the user's provisioning decisions by revealing the details of this trade-off. Specifically, this scenario helps the user to answer the following question: *What is the allocation size, and how should it be partitioned and configured to best fit the constraints and optimization criterion?*

Figure 3.10 shows the application execution time and allocation cost (measured as number of nodes x allocation time) for different cluster sizes, different partitioning configurations, and different chunk size. Similar to Scenario I, Figure 3.10 shows that the predictor captures the system performance with significant accuracy.

Figure 3.10 also shows that an allocation of 11 nodes, with partitioning of 8 application nodes, 2 storage nodes, and a chunk size of 256KB offers the lowest cost. More importantly, this figure points out an interesting case for the analysis of cost vs. time-to-solution: The user can analyze the plot to verify that an option with an allocation of 20 nodes actually offers almost 2x higher performance at a marginal 2% higher cost.

3.3.4 Increasing Workflow Complexity and Scale: Montage on TB101

This section aims to answer the following question: *"Can the predictor support user decisions for more complex scenarios?"* Specifically, the goal is to evaluate a workflow composed of more stages, tasks, and patterns, executing a more data-intensive workload at a larger scale. To answer this question, this section focuses on evaluating how accurate the estimates are for Montage [106], a complex astronomy workflow composed of 10 different stages (Figure 3.11), with varying characteristics in terms of the number of tasks, volume of data, and data access pattern. The workflow uses the 'reduce' pattern in two stages and the 'pipeline' pattern in 4 stages (as indicated by

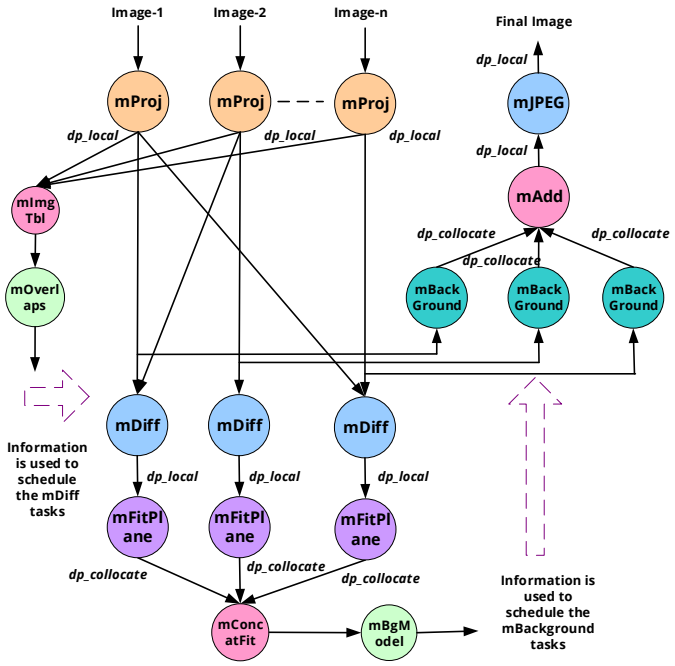


Figure 3.11: Montage workflow. Labels on the arrows represent the tags used to indicate the data usage patterns.

the labels on the arrows in Figure 3.11). Note that Montage diversity offers a rich use case for the predictor and, in fact, each stage could be seen as a different application. Zhang et al. [175] present a detailed characterization of Montage workflow.

To verify that the predictor can support configuration decisions, Montage is executed in different deployments sizes on TB101. For this application, clients are collocated with storage nodes and MosaStore is deployed on spinning disks, verifying yet another configuration parameter. Chunk size variations are omitted since it does not impact actual Montage performance (this is well captured by the predictor).

Workflow characteristics. The I/O communication intensity between workflow stages in Montage is highly variable (presented in Table 3.4 for the workload used). Overall, the workflow includes over 7,500 tasks and

Stage	Data	#Files	#Tasks	File Size	I/O Time
stageIn	1.9GB	957	1	1.7MB – 2.1MB	–
mProject	8GB	1910	955	3.3MB – 4.2MB	10%
mImgTbl	17KB	1	1	17KB	5%
mOverlaps	336KB	1	1	336KB	0.3%
mDiff	2.6GB	5640	2833	100KB – 3MB	48%
mFitPlane	5MB	1420	2833	4KB	5%
mConcatFit	150KB	1	1	150KB	0.5%
mBgModel	20KB	1	1	20KB	0.1%
mBackground	8GB	1913	955	3.3MB – 4.2MB	46%
mAdd	5.9GB	3	1	165MB – 3GB	49%
mJPEG	47MB	1	1	47MB	18 %
stageOut	3GB	2	1	47MB – 3GB	–

Table 3.4: Characteristics of Montage workflow stages. Percentage of spent on I/O operations is based on an execution on the testbed TB20, using all nodes.

generates over 10,000 files, with sizes ranging from 2KB to 3GB. In total, about 30GB of data is read from or written to storage.

Evaluation of the results. Figure 3.12 shows the actual and predicted workflow execution time, for Montage on allocations of different sizes. Plots report the average, summarizing 10 runs, for which the standard deviation is approximately 3% and omitted to reduce clutter. Figure 3.12 shows that, overall, the predictor captures the application performance well. Despite the complexity of the workflow and the scale, the predictor is accurate, the with average prediction error being 3%, the smallest is less than 1%, and the maximum prediction error is 7%).

Prediction accuracy per stage. Since each stage can be very different from others, the evaluation also considers how the predictor performs per stage for a different number of nodes. The average prediction error per stage was 5%, with the 5 stages (mProject, mDiff, mFitPlane, mAdd, and mJpeg) that account for over 70% of the time having a maximum of 4% error. The highest error was 25% for mBackground on 5 nodes, which is due to the short execution time of each task (less than 2 seconds), making it sensitive to any variation in the platform.

Time-to-solution vs. CPU cost. Similar to the BLAST evaluation (Section

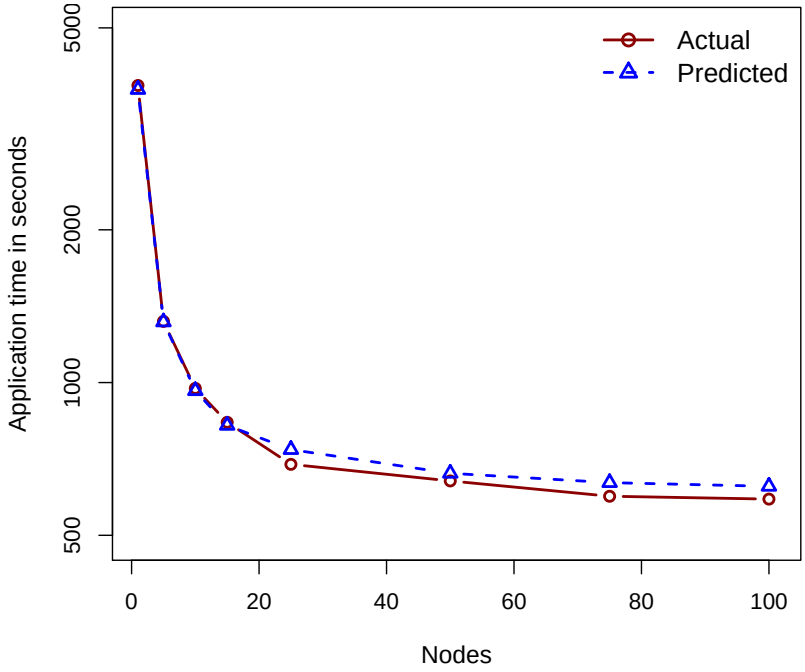


Figure 3.12: Montage time to solution (with the y-axis in log scale) for varying number of nodes deployments on TB101.

3.3.3), this section analyzes the decision of cost vs. time-to-solution. Figure 3.13 shows the workflow execution time (on the y-axis) and the total cost in CPU hours (on the x-axis), actual and predicted, for Montage using different cluster sizes (numbers of nodes indicated over line points). Note that adding more nodes increases the cost almost linearly but the performance improvement is small after 15 nodes and there is virtually no improvement after 75 nodes. This is a result of the Montage workflow structure where some stages are not able to exploit the parallelism offered by these allocations.

This scenario highlights the non-linear relation in the cost vs. time-to-solution trade-off, and how the predictor can accurately capture this relationship. The predictor can show the user the allocation options and each of their impacts on application performance and cost. When the user is interested in optimizing for just one metric, she can opt for the allocations in

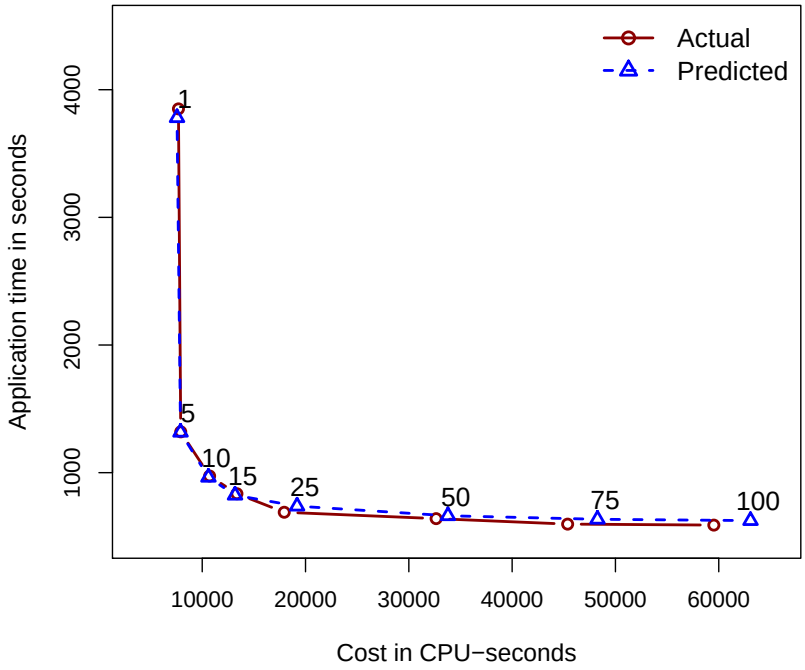


Figure 3.13: Montage running cost in CPU allocated time (x-axis) vs. time-to-solution (y-axis) for varying fixed size deployments (number of nodes shown in the text besides the data points) on TB101. Note that axes do not start at 0.

the top-most or right-most points in Figure 3.13. If the user picks the largest number of nodes available, she indeed obtains the fastest time-to-solution. However, Figure 3.13 also highlights a set of options (5 to 25 nodes) that are not so slow or expensive in comparison to the others. By analyzing the plot, she can opt for an allocation that costs 4 times less than the fastest allocation, yet it is just 20% slower.

Results on TB20. To explore a different execution platform, an additional evaluation considers TB20 with MosaStore deployed on RAMDisks [57]. Figure 3.14 summarizes the results. The workload is smaller; the workflow generates over 650 files, with sizes ranging from 1KB to over 100MB, and about 3GB of data are read from or written to the storage system. Overall, the predictions obtain "good-enough" accuracy to support

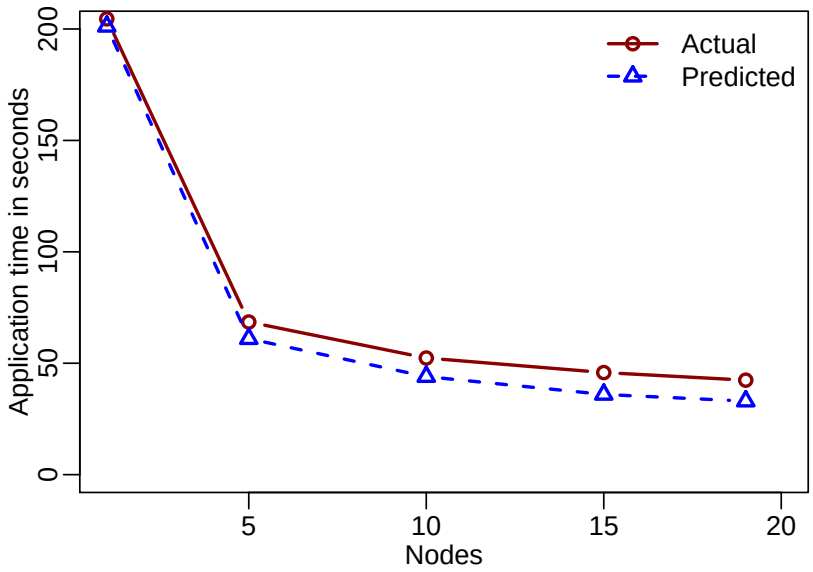


Figure 3.14: Montage time to solution for fixed-size deployments from 1 to 19 nodes on TB20 - an additional node runs the storage manager.

the user's decisions about provisioning. The average prediction error is 9%, the smallest is less than 1%, and the maximum prediction error is less 15%.

3.3.5 Predictor Response Time and Scalability

Sections 3.3.1, 3.3.3 and 3.3.4 describe the accuracy of the predictor and its ability to guide storage system configuration in the context of synthetic benchmarks and real applications (Requirement R1 from Section 3.2.1). They also demonstrate how the accuracy of the predictor behaves in a number of different scenarios (Requirement R4 from Section 3.2.1) – e.g., as the storage configuration, or the number of nodes in the system, varies in total, for clients, and for storage nodes.

The goal of this section is to demonstrate that the predictor satisfies the requirement R2 listed in Section 3.2.1: to be useful, the predictor's response time should use much less resources (machines $\times$ time) than the run of an actual application, even for large systems and I/O intensiveness. Specifically, the goal is to answer the following questions:

- How much allocated computing power (in terms of number of machines $\times$ time) does the prediction use, in comparison to the actual execution?
- How does the predictor's response time increase as the complexity of the simulated system increases?

Prediction effort. Overall, the predictor uses orders of magnitude less resources than the corresponding workflow execution: for BLAST: 200x-500x, for Montage 300x-600x, for the reduce benchmark 2000x less CPU time (all for TB20 when using all 20 nodes). For the reduce benchmark scenario, the savings can be estimated to be as high as 2×10^3 less resources.

Scalability. The remainder of this section focuses on the reduce benchmark, since, among the synthetic benchmarks, it exhibits a more complex pattern than pipeline, and similar to broadcast. Using no storage system optimizations makes this scenario more complex since it generates more events to be simulated.

In this section, each point in the plots is the average for 10 rounds, with error bars representing the standard deviation. The predictor is executed on one node of the testbed TB20 described in Section 3.3.

Figure 3.15 shows the results for increasing the amount of data while keeping the number of nodes constant at 20 as is done in the actual experiments presented in the accuracy analysis in Section 3.3 for the storage system under the default configuration. It shows evidence of how much faster the predictor can be in comparison to the actual execution presented earlier in Figure 3.5. Consider the case of a medium workload (100 $\times$ the small workload): the predictor takes an average of 41 milliseconds to predict the behaviour of the system where the actual execution takes 4.5 seconds in 20 nodes, resulting in almost 2000 $\times$ less resources in the prediction than in the actual execution (100 $\times$ less time and 20 $\times$ fewer machines). If the time to set up the storage system and launch the tasks were taken into account, 4.5 seconds would become almost two minutes, and it would increase the difference even further, leading it to be 6×10^7 times less resources. The results in the case of the large workload are even better: the predictor

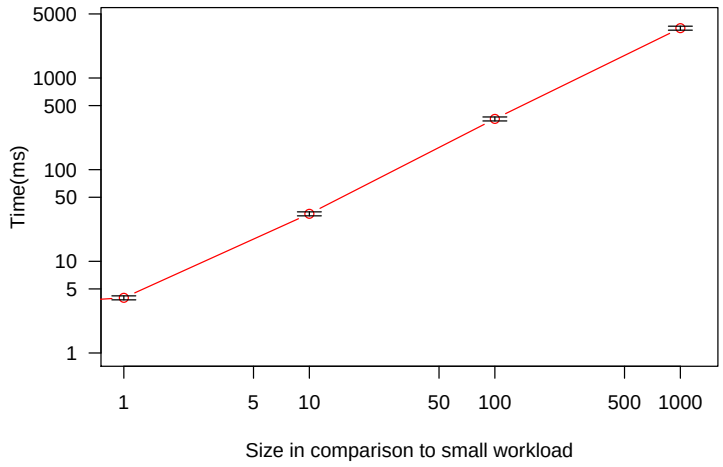


Figure 3.15: Predictor response time for 20 nodes with increasing the amount of data (varying the workload) in the system (log-scale for both axes).

takes 300 milliseconds as opposed to 39 seconds on 20 nodes for the actual benchmark, leading to a difference of 2,600 $\times$ less resources.

Figure 3.16 presents a weak scaling experiment in which the data size scales up with the number of nodes up to 1000 nodes. The idea here is to show how the response time increases as the complexity of the predicted scenarios increases for both axes: the number of nodes and the data amount. This experiment is based on the reduce benchmark for large workload sizes. Thus, there are three stages, each one writing and reading files. Stage in and intermediate stages have files of 100MB per node, stage out has N$\times$ intermediate size. In this experiment, the point for 20 nodes is exactly the prediction for the actual benchmark as shown in Section 3.3.1, taking 300 milliseconds in the simulation as opposed to 39 seconds in the actual benchmark.

Note that the response time increases linearly; the rightmost point in this plot shows the time to predict the behaviour of a system with 1000 nodes writing a total of 300,000MB (3 $stages$ $\times$ 1000 $nodes$ $\times$ 100MB) and reading the same amount of data. That is, it moves almost 600GB around. Moreover, the simulation for the 1000 nodes and 600GB scenario takes 37

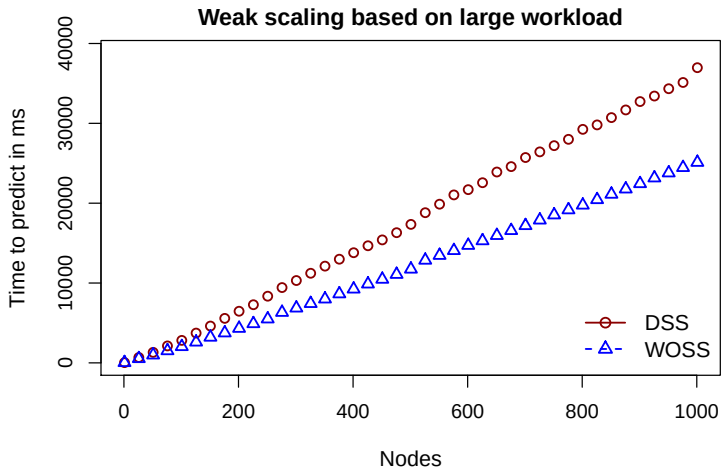


Figure 3.16: Response time in a weak scaling scenario. The data size scales up with the number of nodes up to 1000 nodes in increments of 25 nodes. Each point represents the average of 10 rounds. The standard deviation is little (smaller than 3%) when number of nodes is larger than 50 and are omitted to reduce cutter.

seconds in average for the default configuration (DSS line), which is less than the amount of time to run the actual benchmark for the much simpler scenario of 20 nodes and approximately 12GB.

The WOSS line shows the results for the system with data-placement optimizations. The scenario for 1000 nodes takes almost 25 seconds. Note that it takes less time to predict because it generates fewer events to be simulated. What happens is that data placement optimizations reduce the pressure on the network during the simulation as well. Therefore, the more local data manipulation, the fewer events and the faster prediction time.

Implementation Details

The main simulation loop has the complexity of $O(NlogM)$, where N is the total number of events to be simulated, and M is the number of events currently in a priority queue. The system tends to exhibit $N \ggg M$ as the complexity of the simulated storage system grows. Therefore, the results show a linear scalability.

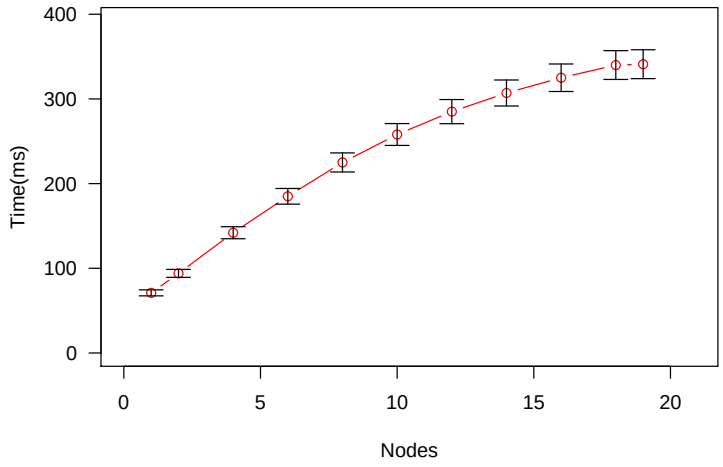


Figure 3.17: Prediction time for 20 nodes and increasing amount of data in the system. Note the log-log scale.

This logarithmic behaviour for increasing number of nodes while keeping the amount of data can also be verified for systems with small number of nodes (see Figure 3.17). This happens because the fewer nodes the system has, the more loopback data transfers it has. Therefore, it requires fewer events to simulate the network behaviour. The increase in the number of events, however, is not linear, but rather logarithmic. This is confirmed by tracking the number of simulated events.

Roughly, the simulator processes between 1250–1820 events per millisecond on one machine used in the testbed TB20 described in Section 3.3.

3.4 Predicting Energy Consumption

Section 3.2 presents the design of the prediction mechanism, including a queue-based model and a system identification procedure to seed the model, that is able to estimate traditional performance metrics, such as turn-around time, or cost of workflow applications. This section focuses on the following question: *Which extensions does the performance predictor need in order to capture energy consumption behaviour in addition to traditional performance metrics?*

Specifically, this section presents an extension of the initial model presented in Section 3.2.3 to take into consideration energy predictions – a metric that has grown in importance, due its impact on cost, or even the feasibility of building large computing infra-structure [29].

This extension relies on a coarse-grained energy consumption model associated with the energy characteristics of the underlying computing platform. My collaborators and I [54] have used this approach to estimate data deduplication energy consumption as described in detail in Chapter 5, and it has also been used by Ibtesham et al. [96]. For workflow applications, Hao Yang was the main researcher responsible for applying and evaluating this approach in targeting the energy consumption metric use-case, which is summarized in this section. Yang et al. [171] present a detailed evaluation of this approach[8].

Overall, the energy predictions obtained an average accuracy of more than 85% and a median of 90% across different scenarios, while having similar response times and resource usage to those predictions presented in Section 3.3.5.

The rest of this section is organized as follows. Section 3.4.1 describes a simple analytical energy consumption model that satisfies the requirements described in Section 3.2.1. Section 3.4.2 presents an evaluation of energy consumption predictions using synthetic benchmarks and real workflow applications.

3.4.1 Energy Model Extension

As discussed in Sections 2.2.3 and 3.2.5, a typical task in a workflow application progresses as follows: (i) the machine running the task reads the input data from the intermediate storage (likely through multiple I/O operations), (ii) the task processes the data, (iii) the machine writes the output to the intermediate storage. Although these phases are not completely separated, they have little overlap.

[8] *Energy Prediction for I/O Intensive Workflows* [171] Hao Yang, **Lauro Beltrão Costa**, and Matei Ripeanu. Proceedings of the 7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers, MTAGS '14. pages 1 – 6. ACM, November 2014.

The herein described extension of the model presented in Section 3.2.3 captures the energy consumption of a task by assuming that these phases are non-overlapping and associating each to different power profiles: (i) idle profile - power to keep the machine on (P^{idle}); (ii) application processing profile - additional power drawn to run a task on the CPU when not performing storage operations, beyond the amount already drawn in idle state (P^{App}); (iii) local storage profile - extra power to perform read and write operations on the storage node (P^{s}); and (iv) network profile - extra power to perform network data transfers (P^{net}).

With this mindset, the total energy spent during a workflow application execution can be expressed as the sum of the energy consumption of individual nodes: $E^{A}_{n} = \sum_{n}^{N} E^{tot}_{n}$, where N is the total number of nodes ($N^{cli} \cup N^{sm}$) , and E^{tot}_{n} is the total energy consumption of node n. Let

$$E^{tot}_{n} = E^{idle}_{n} + E^{App}_{n} + E^{sm}_{n} + E^{net}_{n} \tag{3.4}$$

where E^{idle}_{n} is the energy consumed to keep the node n on, given by

$$E^{idle}_{n} = P^{idle}_{n} \times T^{idle}_{n} \tag{3.5}$$

where T^{idle} is the application execution time. Similarly, the energy spent by a node in each profile is simply calculated by $E^{profile}_{n} = P^{profile}_{n} \times T^{profile}_{n}$

The rest of this section describes the extension of the seeding process used to collect the power measurements, and explains the integration of the initial performance predictor and the energy model.

Energy Model Seeding: System Identification Extension

To use the energy model, one has to provide P^{idle}_{n}, P^{App}_{n}, P^{sm}_{n}, and P^{net}_{n} in addition to the process already described in Section 3.2. The seeding script was extended to collect power profiles. Specifically, the script samples the power when the node is idle over a period (P^{idle}_{n}). For P^{sm}_{n} and P^{net}_{n}, the procedure is similar to the one described to estimate μ^{sm} and μ^{net} - the only difference is that the scripts also measure the power. Finally, the *stress* [3]

Description	Symbol	Seeding Value
Idle node power consumption	P^{idle}	$91.6W$
Additional power when stressing CPU only	P^{App}	$125.2W - P^{idle}$
Additional power for storage operations	P^{sm}	$129.1W - P^{idle}$
Additional power for network transfers	P^{net}	$127.7W - P^{idle}$
Peak power (not used for seeding)	-	$225.3W$

Table 3.5: Energy parameters and values describing one node using dual-core 2.3GHz Intel Xeon E5-2630 CPU, 32GB RAM and 10 Gbps NIC from **TB11** presented in Section 3.4.2. Additional power means power drawn in each of these states in addition to the power already drawn when the machine is idle.

utility tool is used to estimate P_n^{App} by imposing a specific load on the CPU while the scripts measure power. Table 3.5 lists the parameters of the energy model and the seeding values used in the evaluation presented in Section 3.4.2.

Implementation Extension

Figure 3.18 presents the integration of the initial performance predictor and the energy-related additions. The simulator tracks the time each node spends on the different phases of a workflow task: executing the compute intensive part of the application, writing to and reading from storage, and receiving/sending network data. The energy module added to the performance predictor receives the time from the simulator and the power seeding from the scripts, and estimates the energy consumption.

3.4.2 Energy Evaluation

This section presents the evaluation of the prediction mechanism's accuracy when it comes to energy consumption. Similarly to Section 3.3, this analysis uses synthetic benchmarks and real workflow applications with different storage configurations. Additionally, this section briefly analyzes the prediction mechanism's accuracy for predicting the impact for power

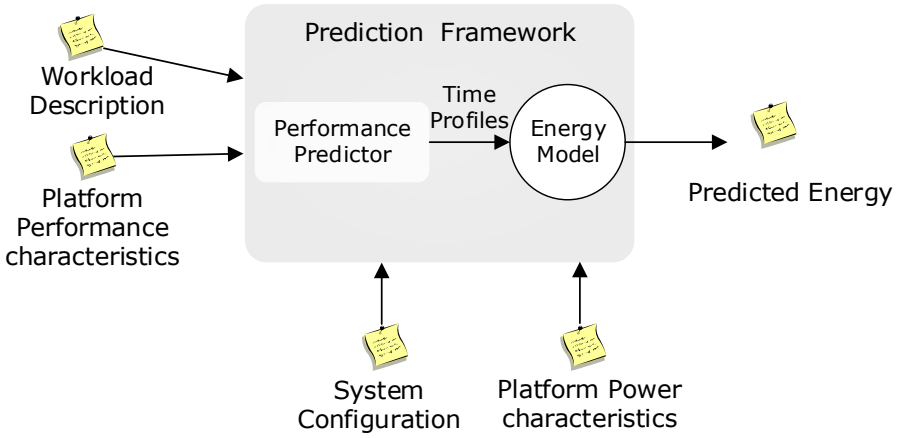


Figure 3.18: Integration of the performance predictor and the energy model as described by Yang et al. [171]. The predictor receives the same input described in Section 3.2, the information described in Table 3.5, and a breakdown of the time spent in each power profile. The energy module passes this input to the energy model to estimate the energy consumption.

tuning techniques. Yang et al. [171] present a more detailed evaluation of using this prediction approach to estimate energy consumption of workflow applications.

The experimental setup is the same as that described in Section 3.3. MosaStore is the storage system used. The label **DSS** refers to experiments using a Default Storage System configuration, and the label **WOSS** refers to experiments where the system configuration is optimized for a specific workflow pattern.

Testbed. A lack of infrastructure to measure power prevents this evaluation from using the same testbeds described earlier (TB20 and TB101). Thus, a different testbed (**TB11**) is used for the energy evaluation: it is Grid5000 'Lyon' cluster [41] where each node has two 2.3GHz Intel Xeon E5-2630 CPU, 32GB RAM, and 10 Gbps NIC. One node runs the metadata manager and the workflow coordination scripts, while the other nodes run the storage modules, the client SAI, and the application processes. This platform limits the scale of the experiments to 11 nodes.

Power measurement. Each node is connected to a SME OmegaWatt

power-meter[9], which provides 0.1W power resolution at a 1Hz sampling rate. To measure the total energy consumption, the scripts that run the experiments aggregate the power consumption for the duration of the benchmark execution. The evaluation does not take into account the energy consumed by the node that runs the metadata service and workflow scheduler, as these are not subject to the configuration changes that the prediction mechanism targets.

Evaluation Metrics. Similar to the time comparison, the evaluation focuses on prediction accuracy by comparing the predicted and actual energy consumption, and reports the prediction errors as $|1 - \frac{E_{pred}}{E_{actual}}|$.

Synthetic Benchmarks: Workflow Patterns

Figures 3.19a, 3.19b, and 3.19c present the predicted and actual energy consumption for the pipeline, reduce, and broadcast synthetic benchmarks, respectively. Overall, the energy consumption predictions have an average of 12.5% error and typically close to one standard deviation interval. For *DSS*, the pipeline benchmark has the best accuracy with 5.2% error, reduce is at 16.4%, and broadcast is at 15.9% (just one replica).

For *WOSS*, the predictions have an error of 13.4% for pipeline, 12.2% for reduce, and 16% for broadcast (using 4 replicas in the WOSS configuration). Despite the fact that the prediction mechanism is more accurate for execution time than energy, the predictions captures the energy consumption for both DSS and WOSS configurations with adequate accuracy: it accurately predicts the energy savings brought by WOSS, and can support users in making decisions about storage configurations when energy consumption is used as optimization criteria.

Predicting the Energy Consumption of Real Applications

As shown above, the predictor is able to estimate the energy consumption of synthetic benchmarks, as well as the impact of different data-placement optimizations on this metric. To understand how accurately the predictor

[9]Details can be found at https://intranet.grid5000.fr/supervision/lyon/wattmetre/

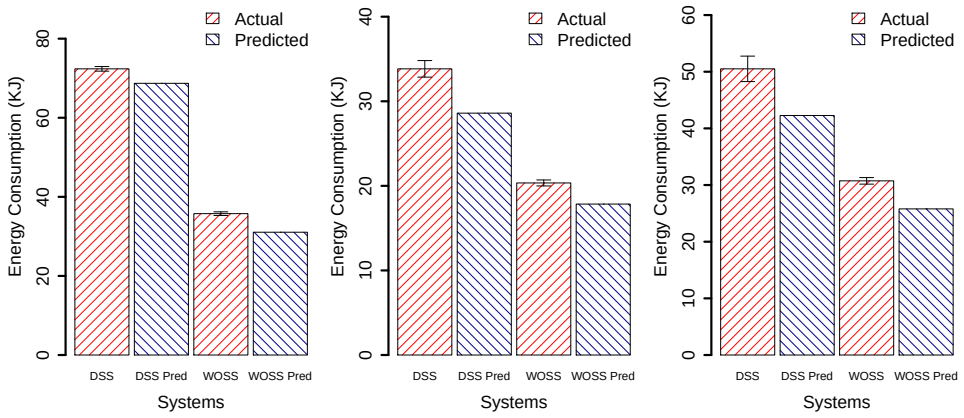
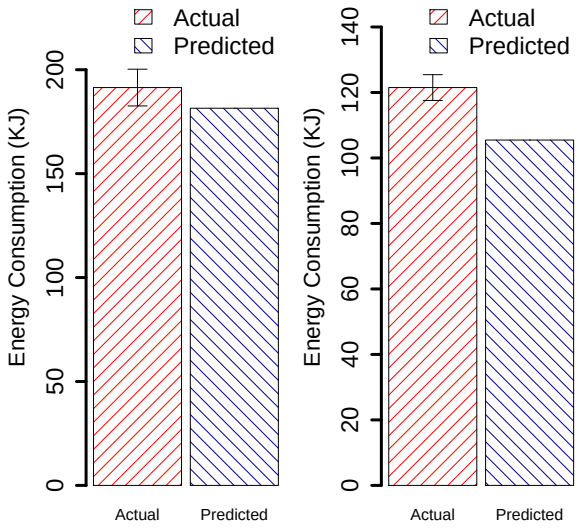


Figure 3.19: Actual and predicted average energy consumption for the *synthetic benchmarks*.

can estimate the energy consumption for real applications, Figures 3.20a and 3.20b report the energy consumption for BLAST and Montage.

The workload for both applications are scaled down because of the smaller scale of the TB11 testbed, in comparison to TB20. For *BLAST*, each node receives 8 DNA sequence queries as input to search on the same database used in Section 3.3.3. For *Montage*, the workload has approximately 2,000 tasks and is presented in Table 3.6.

Similar to the time predictions, overall, the energy predictions are more accurate for the real applications than for synthetic benchmarks since the synthetic benchmarks are designed to produce a high stress on the storage system, which results in contention and higher variance, which are harder to capture when modeling the storage system. BLAST predictions (see Figure 3.20a) exhibit an average error of 5.2%; Montage (see Figure 3.20b) exhibits a 15.9% average error.

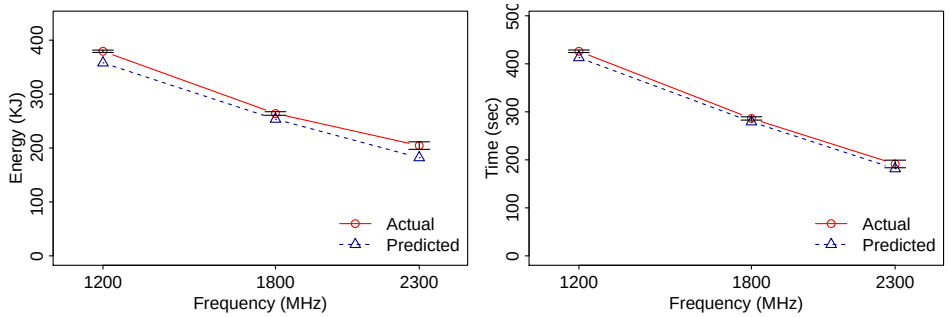


(a) BLAST **(b)** Montage

Figure 3.20: Actual and predicted average energy consumption for the *real applications*.

Stage	Data	#Files	File Size
stageIn	320MB	163	1.7MB-2.1MB
mProject	1.3GB	324	3.3MB-4.2MB
mImgTbl	50KB	1	50KB
mOverlaps	54KB	1	54KB
mDiff	409MB	895	100KB - 3MB
mFitPlane	1.8MB	449	4KB
mConcatFit	21KB	1	21KB
mBgModel	8.3KB	1	8.3KB
mBackground	1.3GB	325	3.3MB - 4.2MB
mAdd	1.3GB	2	503MB
mJPEG	15MB	1	15MB
stageOut	518MB	2	15MB-503MB

Table 3.6: Characteristics of the Montage workflow stages for the energy evaluation.



(a) Energy **(b)** Time

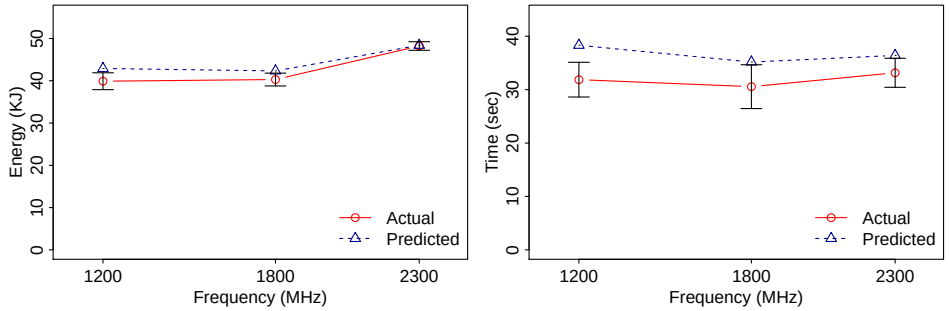
Figure 3.21: Actual and predicted average energy consumption and execution time for *BLAST* for various CPU frequencies.

Predicting the Energy Consumption Impact of DVFS Techniques

Dynamic Voltage and Frequency Scaling (DVFS) (also known as CPU throttling) is an important technique that limits the maximum frequency of processors in order to reduce power consumption, with the drawbacks of limiting the number of instructions a processor can issue in a given amount of time. Thus, although this technique can save power consumption, it is not clear whether (or when) frequency scaling can reduce the energy consumption of different applications, including workflow applications.

To evaluate the predictor's ability to predict the impact of CPU frequency scaling on energy consumption, Figures 3.21 and 3.22 present energy consumption and execution time for two benchmarks as presented by Yang et al. [171]: (i) BLAST application (Figure 3.21), and (ii) the pipeline synthetic benchmark (Figure 3.22), performing only I/O operations. DVFS is used to set the processors at the frequencies of 1200MHz, 1800MHz, and 2300MHz. The seeding procedure is performed independently for each frequency.

BLAST is more CPU intensive, so reducing the frequency increases the runtime and the energy consumed, leading to 85.5% more energy consumption between the minimum and maximum frequencies evaluated. The pipeline benchmark is not strongly affected by CPU performance and, thus, using the minimum frequency does not increase the benchmark execution



(a) Energy **(b)** Time

Figure 3.22: Actual and predicted average energy consumption and execution time for the *pipeline benchmark* for various CPU frequencies. Note that, although the average time for 2300MHz is slightly longer than for 1800Mhz, the average times are inside the confidence interval obtained for the average times at 95% of confidence and, thus, they are considered to be equivalent for this evaluation.

time significantly. More importantly, in the context of this dissertation, the predictions are useful to support configuration decisions in these cases: The predictor estimates 11% vs. 17% (actual) potential energy savings for the pipeline, and an increase of 96.5% vs. an actual 85.5% increase for energy consumption between the minimum and maximum frequencies for BLAST.

These results emphasize that it is not always clear if DVFS techniques can save energy since computational and I/O characteristics of the workflow application have an impact on the actual runtime. More importantly, the prediction mechanism was effective to predict the energy consumption with adequate accuracy, and support decisions as to when DVFS techniques should be used.

3.4.3 Energy Extension Summary

As is the case for traditional performance metrics (evaluated in Section 3.3), overall, the performance prediction mechanism is able to estimate energy consumption close to actual consumption and support configuration decisions (Requirement R1). More importantly, the extensions made to the

model and seeding procedure were minimal and do not require changes to the storage system (Requirement R2). Additionally, the time to predict the performance for the energy cases evaluated use less resources (by one to two orders of magnitude) than the actual runs, as the scenarios presented in Section 3.3.1 (Requirement R3).

Finally, the evaluation presented in Section 3.4.2 covers power-tuning techniques that target multiple scenarios and success metrics, which is key for providing more evidence that the proposed solution works properly in a number of scenarios (Requirement R4). In fact, Yang [170] has evaluated the performance prediction mechanism proposed in this dissertation, with emphasis on the energy extension, in different scenarios, including yet another testbed; Yang's results are similar to the ones presented in this chapter.

3.5 Related Work

This section describes previous work on different approaches to predicting storage system performance and tuning its configuration parameters.

Model based analysis. A number of projects use a model-based approach to estimate storage system performance with a given configuration or workload. Ergastulum [22] targets centralized storage solution based on one enclosure to recommend an initial system configuration, and Hippodrome [21] relies on Ergastulum to improve configuration, based on online monitoring of the workload. By considering a distributed system, the predictor proposed in this dissertation handles more complex interactions among system components and more configuration options – which do not exist in centralized solutions.

Simulation based systems. IMPIOUS (Imprecisely Modeling Parallel I/O is Usually Successful) [122] is a trace-driven simulator that uses an abstract storage system model designed to capture the main mechanisms of parallel file systems. The simulator is simplified to be able to simulate thousands of client and storage nodes, which has the drawback of not being accurate, producing performance estimates that under- or over- estimates

the performance by up to 60%. Another trace-driven simulator PFSsim is designed specifically for evaluating I/O scheduling algorithms in parallel file systems. PFSsim simulates the storage system at low component level, simulating the network using OMNeT++ [160] and disks using DiskSim [1]. In other work, Liu et al. [111, 112] build a simulation framework for simulating the storage system of supercomputing machines. The framework simulates all hardware components including compute and I/O nodes, the storage subsystem, and the supercomputing interconnect. Finally, FileSim [70] is a parallel file system simulation framework that simulates file system operations at low granularity, including disk operations and packet level simulation of network operations. The simulator simulates specific storage system operations, such as data placement, or locking algorithms, and can be used to validate new algorithms or metadata service.

Similar to the work presented in this chapter, Thereska et al. [156] proposed a predictor mechanism for a distributed storage system with a detailed model. To provide such information, they propose using Stardust [158] a detailed monitoring information system that requires changes to the storage system and kernel modules to add monitoring points. This approach enabled their predictor to achieve prediction within 19% of the actual predictions depending on the workload. The herein proposed predictor has achieved similar accuracy on the application's workload evaluated, however, with the advantages of a lightweight approach to seed the model, and not requiring changes to the system design or kernel modules.

Unlike some of the above-mentioned efforts, this predictor approach targets simulating a generic distributed storage system architecture in a cluster infrastructure (not special supercomputer machines) without simulating particular storage system operations. It avoids detailed low-level simulation (e.g., disk or packet level simulation), without significantly compromising accuracy, enabling the prediction mechanism to efficiently simulate large-scale deployments.

An important difference from past work on storage systems simulation is the focus on a whole workflow application and the potential interaction among the workflow's phases – rather than the average performance for a

batch (e.g., Hippodrome [21] and Ergastulum [22]) of storage operations, and on evaluating performance of the system in larger scale – in contrast to Thereska et al. [156]), for example. Additionally, this chapter targets the partitioning problem of splitting the nodes between the application and intermediate storage.

Monitoring and/or Machine Learning based tuning targeting overall application performance. Behzad et al. [27] present an auto tuning framework for Hierarchical Data Format, version 5 (HDF5) I/O library [154]; their solution intercepts the HDF5 I/O calls and injects optimized parameters into parallel I/O calls. The herein proposed solution intercepts the HDF5 I/O calls and injects optimization parameters into parallel I/O calls. Further, their framework monitors I/O performance, and explores the tuning parameter space using a genetic algorithm via actual application runs. Differently from Behzad et al. [27], ACIC's [110] machine-learning approach usesCART to build the model used to guide the optimization of parallel applications. Finally, Zhang et al. [175] have recently proposed an approach to determine the storage bottleneck for workflow applications using a set of benchmarks and actual runs of the target application.

This dissertation's approach enables an exploration of the system at a lower cost. The predictor is able to estimate performance of a scenario that adds or reduces resources and change the configuration without requiring new runs (or a larger training set) of the application for new executions of benchmarks (as e.g., Zhang et al. [175]) nor generations of the genetic algorithm (as e.g., Behzad et al. [27]). Further, unlike [27, 110] this work targets workflow applications and predicting a POSIX-based storage system performance, including the impact of scheduling decisions in combination with data placement. In fact, the proposed solution can be used though an adaptation of these machine-learning techniques, or other optimization approaches, to the target context in order to perform auto-tuning.

Finally, Elastisizer [91] and Starfish [92] target a similar problem automating allocation choices for an entire application. Their work, however, does not address the aspects of workflow applications or storage system configuration since it focuses on a different class of application: Map Reduce

jobs.

Predicting Energy Consumption. Past work has targeted modeling the power-consumption of a complete machine, based on some utilization metrics or performance counters. For example, Economou et al. [69] presents a method to capture the power consumption of a system based on the idle power CPU utilization, off-chip memory access count, and I/O rate of the network and hard disk. Gurumurthi et al. [84] proposes analytical models coupled with events relative to the system's architecture in order to simulate power consumption. In the context of parallel applications, Feng et al. [72] focuses on the characteristics of multi-threaded applications. Additionally, Pakin and Lang [129] focus on evaluating the energy savings when DVFS is enabled.

Although these approaches have achieved success, they typically need a deeper and previous knowledge of the architecture, which leads to a longer prediction time. Moreover, this chapter focuses on the distributed storage layer of workflow applications that have various patterns and are I/O intensive.

Towards a distributed system and a more I/O intensive workload, Samak et al. [141] present an approach to analyze power consumption focusing on pipeline to infer the consumption of larger infrastructures, not covering the exploration of different patterns, workload, and configuration options. Finally, some approaches like EEffSim [133] presents a simulator to evaluate energy consumption of multi-server storage systems, but lack a validation of their approaches on real testbeds.

3.6 Predictor Development

The prototype of the performance prediction mechanism is written mainly in Java with part in C. Some of the scripts used to run experiments are in Bash and Python. The source code, including the scripts used to run the experiments, aggregate data and plot the analysis, and the measurements

collected are available at an SVN repository [10]. The execution logs collected during the experiments are available at the NetSysLab cluster, since they are too large to be uploaded to the code repository.

Although, to this date, I have been the main contributor to the code repository, I have also received help from other developers, including Abmar Barros, Hao Yang, and Marcus Carvalho. We have also used automated tests and code reviews, which are also available online[11]. For the code reviews, in addition to the developers cited above, I have received help from Elizeu Santos-Neto, Samer Al-Kiswany, and Abdullah Gharaibeh.

3.7 Summary and Discussion

This chapter makes the case for a prediction mechanism to support provisioning and configuration storage choices for workflow applications. It focuses on predicting the performance of workflow applications when running on top of an intermediate object-based storage system, and presents a solution based on a queue-based model with a number of attractive properties: (i) a generic and uniform system model, (ii) a simple system identification process that does not require specialized probes nor system changes to perform the initial benchmarking, (iii) low runtime to obtain predictions, and (iv) adequate accuracy for the studied cases.

The user can also use more complex utility functions, based on the predicted performance metrics, to reach her specific goal, as showed by Strunk et al. [152] and Wilkes [167]. For these cases, the user, or an automated tool, can apply different optimization solvers to search the configuration space and propose the most desirable storage configuration and provisioning choices.

The discussion below clarifies some of the limitations of this work and the lessons learned during this designing exercise.

[10] Predictor implementation, simulation results, and its input data are available at https://subversion.assembla.com/svn/AutoConf/. Actual measurements, configuration deployment and storage system code can be found at https://subversion.assembla.com/svn/MosaStore/

[11] https://codereview-netsyslab.appspot.com/

What are the main sources of inaccuracies?

Currently, there are sources of inaccuracies at multiple levels. First, the model does not capture all the details of the storage system. For example, support services like garbage collection or storage node heartbeats or the control paths are simplified to match a generic object-based storage (MosaStore uses a FUSE-based implementation that would need more complex control path), and this model assumes that all control messages are of the same size. Second, the system identification mechanism is constrained to be simplified even further, at the cost of additional accuracy loss. Third, the model does not capture the infrastructure in detail (e.g., contention at the network fabric level or OS scheduling). Finally, so far, the application driver uses an idealized image of the workflow application; for example, all pipelines are launched in the simulation exactly at the same time, while, in the experiments on real hardware, coordination overheads make them slightly staggered).

What are the sources of inaccuracies particular to the energy consumption predictions?

In addition to the reasons summarized in Table 3.7, which include the simplicity of the model and its seeding mechanism, the error in the predictions for energy arise from some inaccuracies in the input: First, the power meters used provide 1Hz sampling rates, which excludes the consumed energy at sub-second granularity from the actual measurements. Second, the time predictor should provide an accurate breakdown of the time spent in each power profile, which is hard to validate without a more intrusive approach.

Additionally, the additive energy model is a simplification since it does not capture the lack of energy-proportionality of current platforms.

What are the trade-offs of optimizing for time, versus optimizing for cost and energy?

To optimize for time, one can use an optimized storage configuration or add more resources (e.g., add more nodes). The former approach usually

Source	Examples
Storage system	Fine granularity for the activity inside each component, detailed execution path, or maintenance services such as failure detection and garbage collection.
Infrastructure	Contention at the network fabric level, complex network topology, or detailed scheduling overhead.
Application	Tasks launched at the same time, absence of faults by crash, or machines with degraded performance.
System identification	Assumptions about client and storage service times.

Table 3.7: Summary of the limitations and main sources of inaccuracies for the prediction mechanism.

exploits data locality and location-aware scheduling, which, in addition to time, generally reduces the amount of data transfers and, thus, it reduces the energy costs as well. The idle power, however, remains a large portion of the total power consumption due to the fact that, though they are more energy proportional than their predecessors, the state-of-the-art platforms are not perfectly energy proportional (testbed TB11's idle power is 40% of the peak power). Section 5.4 extends this discussion in the context of two generations of machines with different power proportionality in the context of data deduplication storage technique.

Increasing the allocation size of an application may improve performance at the cost of money and more energy. Section 3.4.2 demonstrates, for a subclass of applications, it is additionally possible to optimize for energy only by using power-tuning techniques such as CPU throttling. These techniques, however, need to be carefully considered, as they can bring energy savings or lead to additional costs, depending on the specific application patterns. The prediction mechanism proposed in this chapter is particularly useful to support these types of decisions.

What is the impact of failures during the application execution?

Currently, the simulator does not account for failures in the task execution, either by crash or another reason that may degrade the performance and

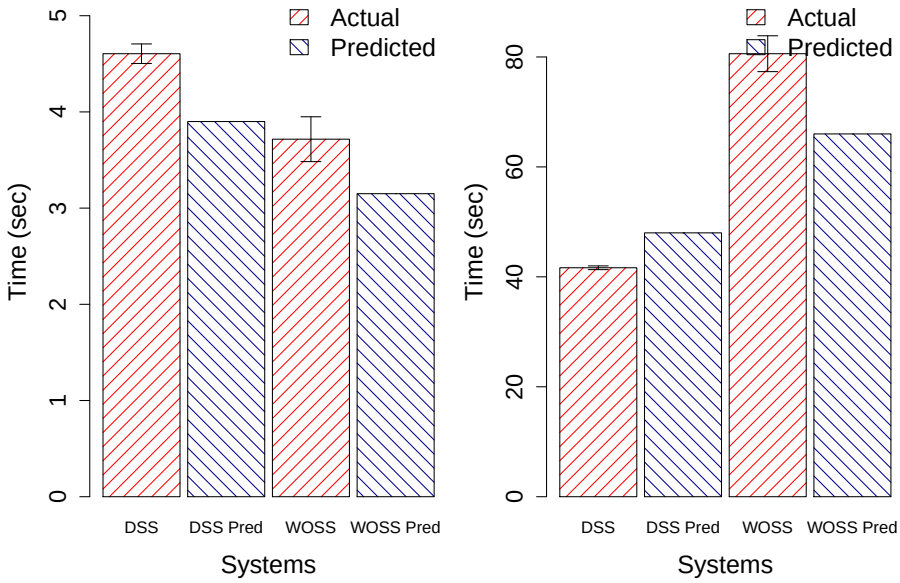
makes the execution time of some tasks longer. A common approach to address this problem is adding replication for failed tasks or tasks that are taking long to finish execution [66], which can be easily incorporated to the workqueue [62] or to the data-location-aware workqueue [142] already implemented in the simulator. Additionally, the simulator would benefit from receiving a model that captures faults for the target environment to be considered during the prediction.

How accurate is the prediction mechanism when the intermediate storage is deployed on spinning disks?

So far, the focus has been on predicting performance when the intermediate storage is deployed over RAMDisks since this is a common setup on large systems that rely on intermediate storage, as it improves performance; and some platforms do not even have spinning disks (see Section 2.2.1). The storage service does not model history-dependent behaviour, thus it is expected to achieve lower accuracy predictions when the system is deployed over spinning disks. This issue can be addressed by using a more sophisticated model or simulator of the storage device as DiskSim [1], or a machine learning approach as described by Crume et al. [61].

A preliminary evaluation, however, shows how the current (unchanged) model performs when using spinning disks, on testbed TB20, for the synthetic benchmarks described in Section 3.3.1. Figure 3.23 shows the results for the reduce pattern when using the medium and large workloads. The key observation here is that, although prediction accuracy is lower, predictions are good enough to make the correct choice between DSS and WOSS – that is, the choice of whether to use the data co-placement optimization for each of the workloads (note that this optimization is beneficial in one case, and is detrimental in the other one). Pipeline benchmark on spinning disks shows results close to those using RAMdisks (Figure 3.4).

The results when using TB101 , which has newer machines than TB20, and MosaStore deployed on spinning disks are similar to those of the RAMDisks as well. Moreover, Figure 3.12 shows results for Montage using MosaStore deployed on spinning disks. The results for TB101 show the



(a) Medium Workload **(b)** Large Workload

Figure 3.23: Actual and predicted performance for the *reduce benchmark on spinning disks*.

trend for the impact of newer disks with higher performance and larger buffers have on the predictions of these applications.

Additionally, Figure 3.24 presents the predicted run time for pipeline benchmark running on top of Ceph storage system deployed on spinning disks using TB101.

Hao Yang has been conducting an evaluation of spinning disks in the context of energy predictions.

How general is the proposed prediction mechanism? That is, can the prediction mechanism be used for other storage systems?

The prediction mechanism's goal is to model a generic object-based storage system and have a system identification process that works entirely at the application level only, to be easily portable across deployment platforms. While, so far, the predictor has been evaluated in depth for the MosaStore

system in its DSS and WOSS configurations, a preliminary experience gave encouraging results when using it to predict the relative application performance of the pipeline benchmarks on DSS in two other storage solutions, Ceph [164] and GlusterFS [65], for a limited number of scenarios.

The experiments consisted of the pipeline benchmark running on top of these two systems. The traces were the same already used for MosaStore predictions, no new traces were collected. The input not shared among the the systems was the seeding (see 3.2.4).

Figure 3.24 shows the actual and the predicted performance for the pipeline benchmark using Ceph deployed on spinning disks. This experiment also varies the number of nodes for TB101. Predictions for the average performance are within 15% of the values for the actual performance. Additionally, Ceph has a high standard deviation, which place the predictions overlapping the intervals considering the standard deviations for all scales. For GlusterFS, the predictor obtained error around 30% on top of 100 nodes.

More importantly, this accuracy is "good-enough" to compare the performance among those systems and MosaStore, capturing which one should provide better performance. Note that Al-Kiswany et al. [18] evaluate the performance of these systems and compare them to MosaStore in the context of workflow applications.

What should one do to use this solution?

The performance prediction mechanism needs to receive both: a description of the (i) workload and of the (ii) platform. The information related to the platform is coupled with the storage system and the network, requiring a new execution of the seeding procedure for a different platform – either a new network or a new set of machines.

The input related to the workload should be in the form of (i) a per client I/O operations trace and (ii) the task-dependency graph for scheduling (see Figure 3.1). Note that it is not dependent on the storage system or on a specific deployment, it is simply has to be in the format that the simulator expects to parse the input. Thus, the same input can be used

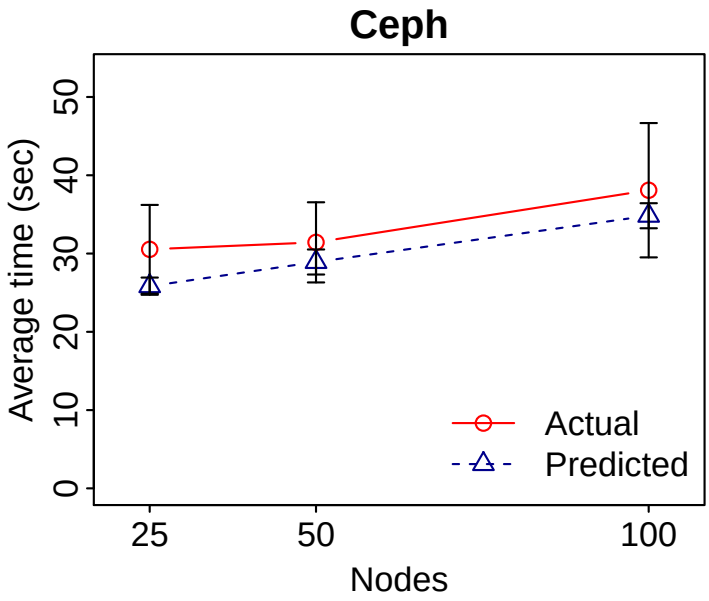


Figure 3.24: Actual and predicted performance for the pipeline benchmark while varying the number of nodes on TB101 using Ceph as storage system on spinning disks. Plot shows the average benchmark time for 20 rounds. Error bars represent standard deviation.

across different deployment or even systems. In fact, the experiments for Ceph and GlusterFS, in the discussion item above, used the same input, which was already used for MosaStore predictions.

Moreover, the platform description for Ceph and GlusterFS, used as input to the performance predictor, was also based on a deployment of just three machines as described in Section 3.2.4.

Can the same workload description be used for different versions of the same application?

Workload description is based on a trace of I/O operations. If the behaviour of the I/O operations changes, it may affect the performance on the application in a given configuration of the storage system. For example, if a new version of the application takes more advantage of data locality, the cache-hit ratio may increase, which reduces the amount of data transfers

among the nodes. This reduction decreases the I/O demand on the storage nodes and can reduce the number of storage nodes to deliver the same performance. Additionally, the new version may also change the processing time of the application, which is used to predict the overall performance of the application.

In complex applications, however, it is likely that just a portion of the binaries changes, which affects only a subset of the stages. In this case, the performance predictor would only need the trace from the stages that use the new binaries to combine with the stages from a previously collected trace. Note that it assumes that the output files from these stages would not change. If the new binaries produce file with different characteristics from the previous version, then the subsequent stages should be re-executed to produce new traces.

Does changing the data sizes requires a re-execution of the entire application? or Can one use the same traces to produce the workload description for different data sizes?

The results from this study show that the data size impact on the overall performance (e.g., total execution time), but they do not have a high impact on the overall relation among the different data-placement options. For example, the results from the reduce benchmark for medium and large workloads have different execution times, but the gains of using WOSS over DSS are proportionally similar.

For other configuration options, however, the actual best configuration choice may be different while the overall behaviour of the performance curve in a plot looks similar, just shifted. As an example of this scenario, consider the results for Montage large workload obtained from TB101 and the one obtained on TB20 in Figures 3.12 and 3.14. For both cases, the overall shape of the graph shows decreasing performance gains as the scale of the system grows. The actual values for number of nodes offering the best performance or the actual application time, however, differ significantly.

Chapter 4

Using a Performance Predictor to Support Storage System Development

This chapter presents an experience of using the prediction mechanism beyond its original design goal: using the performance prediction mechanism described in Chapter 3 to better understand system behaviour and debug MosaStore (Section 2.1). Specifically, the developers of MosaStore compare the predictions to the actual performance and decide whether they are close enough to their goal[1]. In the case in which they are not close enough, the developers proceed to debug the system, and follow this process iteratively as part of its development. The predictor is also used to evaluate the potential gains of new system features.

Overall, the predictor is useful for setting goals for the system's performance, despite the difficulties of properly seeding the model and mimicking the used benchmarks to evaluate its performance. Specifically, this approach gives confidence in the implementation of MosaStore for some scenarios and

[1] *Experience with Applying Performance Prediction during Development: a Distributed Storage System Tale* [59]. **Lauro Beltrão Costa**, João Brunet, Lile Hattori, and Matei Ripeanu. In Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, SE-HPCCSE '14. Pages 13–19. IEEE, November 2014.

points out situations that needed further improvement. The latter makes both an improvement of the system's performance by up to 30% and a decrease of 10x in the response time variance, possible.

In presenting this experience, this chapter sheds light on two higher-level questions: (i) *What are the potential benefits that the proposed performance predictor can bring to the software development process of a storage system?*, and (ii) *What are the limitations and challenges of using a performance predictor as part of the development process in the presented use-case?*.

The rest of this chapter is organized as follows: Section 4.1 presents the motivation for having a baseline for expected efficiency of distributed systems. Section 4.2 advocates for the use of a performance predictor as an approach to setting the efficiency baseline, in order to support the development of distributed systems, and presents the methodology of the use-case described in this chapter. Section 4.3 shows how the performance predictor can support the development by describing its use in the development of MosaStore. Section 4.4 summarizes the main problems faced by the performance predictor approach during this study. Section 4.5 discusses the experience of applying performance prediction and highlights the benefits and limitations. Section 4.6 presents related work specifically in the context of predicting performance to support software development. Finally, Section 4.7 presents concluding remarks regarding this study.

4.1 Motivation

Efficiency, in terms of the resource usage required to deliver 'good', or simply close to optimal performance, is an important criterion for the success of a system. In particular, for large scale computer systems, which are likely to be used by several clients and handle a variety of workloads, this non-functional requirement is crucial to achieve success and encourage their adoption. High performance must be reached while keeping the cost low, by using the least possible amount of resources, while delivering the fastest time-to-solution. Because of its importance, efficiency should be addressed from the early stages of software development [24], especially given that

maintenance and debugging costs increase over time [165].

The current state-of-the-practice in addressing performance consists of analyzing a system by employing profilers to monitor its behaviour [24]. During the analysis, these profilers should pinpoint the regions of the code where execution takes longer and, therefore, should receive more attention and be optimized. Although profiling-based optimization is undoubtedly a key part of the development of complex systems, deciding when the efficiency of a system has reached a "good enough" level, where extra effort is unlikely to render extra benefits, is still a challenge.

In this context, where optimal performance is crucial for the functioning of the system, developers should be able to rely on tools that provide an estimate of the expected performance (i.e., a baseline) and compare it to the actual performance while developing the system. This process is complex, even more complex for distributed systems, since there are more components and different interactions when compared to systems hosted on a single node (e.g., failure and retry operations to tolerate network failures). This process would be conceptually similar to the use of automated tests to check functional requirements, but with the tool being a system's performance predictor targeting a specific efficiency level. Ideally, the tool should also provide information about where the system is losing performance, similar to when automated tests identify a part of the code that is failing for a specific functional requirement.

Note that the use of a performance predictor in this scenario is different from its common use. Typically, performance predictors rely on the monitoring information from an already deployed system to support configuration decisions (see Section 2.3), capacity planning, or online provisioning adaptation during a deployment-planning or post-deployment phase. The scenario presented in this chapter advocates for the usage of a performance predictor as a tool incorporated into the software development phase in order to provide a performance baseline that guides the effort to improve performance.

4.2 Case Study

This chapter discusses the importance of using performance predictors to develop complex software systems by narrating an experience in applying a specific performance predictor in the development of a complex and error-prone distributed storage system. It shows how the performance predictor helped developers to properly handle the interaction of a potentially large number of distributed components in a storage system, while avoiding compromising efficiency due to the complex interactions of these components.

In addition to presenting the gains of applying the performance predictor in this use-case, this chapter also discusses the main problems faced during this study. To this end, we[2] proposed and followed the use of the performance prediction approach during the development of a distributed storage system.

This rest of this section presents a brief summary of both: MosaStore as the system used in this study (Section 4.2.1 and detailed in Section 2.1) and the design of the performance predictor (Section 4.2.2 and detailed in Chapter 3), while highlighting the aspects relevant to the rationale of applying the performance predictor during the development process.

4.2.1 Object System

This study focuses on MosaStore [14], a distributed storage system that can be configured to enable workload-specific optimizations. It is mainly implemented in C, and has approximately 10,000 lines of code. To this date, more than fifteen developers were involved in different versions of its implementation. Section 2.1 presents MosaStore in detail.

[2]When I use we in this chapter, I refer to the people involved in the development of MosaStore as well as two software engineering researchers, João Brunet and Lile Hattori, with whom I have discussed the goal of this study and who helped to set the goals. During this case study, none of us were actively involved in the coding of MosaStore. Specifically, the MosaStore's developers were Emalayan Vairavanathan, Samer Al-Kiswany, and Hao Yang.

MosaStore Architecture

MosaStore employs a widely-adopted object-based storage system architecture (such as that adopted by GoogleFS [82], PVFS [85], and UrsaMinor [10]). This architecture includes three main components: a centralized metadata manager, storage nodes, and a client-side SAI, which uses a VFS via the FUSE [2] kernel module to implement a user-level file-system that provides a POSIX API in MosaStore. Section 2.1.1 and Figure 2.2 present MosaStore's architecture in detail.

MosaStore Configuration Versatility

More relevant to this study, MosaStore can be configured according to a multitude of options (Section 2.1.2). These possible configurations can result in a number of different deployments of the system, where storage nodes and SAIs interact in different ways, with different overheads, and leading to different performances. In this context, the developers of MosaStore initially employed profiling analysis during its development, but in an *ad hoc* manner; there was no baseline for the best possible performance, and no indicator of when they should have started or stopped the profiling process. This scenario brought a challenge to MosaStore's developers, which is present in many other systems: for a given configuration, *what performance should an implementation deliver to be considered efficient?*

4.2.2 Performance Predictor

To support MosaStore's development in the performance-improvement phase, we decided to employ the performance predictor as part of the development process. This section describes the general reasoning behind using the performance predictor as a mechanism to provide a performance goal for the system in the context of development.

Local Deployment: A Simple Scenario

In its simplest configuration, MosaStore uses a single machine. By deploying all three components locally, the system should not face efficiency loss

related to the coordination of distributed components, nor due to contention on the network substrate.

The expected ideal performance for such a scenario, can be set by external tools that serve as benchmarks for local file systems, or even by simple analytical models. MosaStore developers already used this approach, which served as simple "sanity" check for the system's performance. For example, a scenario where a single SAI uses a manager and a storage node deployed on the same machine should provide a sequential write throughput similar to the throughput of a write to a local file system. For MosaStore, the performance baseline to be achieved was set based on the performance of a different local file system – Third extended filesystem (EXT3), in this case.

Distributing the System Components

As the number of components in the system grows, predicting performance of a distributed system accurately and in a scalable manner becomes challenging. Analytical models for the contention of the distributed environment on the storage nodes and on the manager, or the network performance that can capture different applications' workload, become increasingly more complex. Moreover, there are no tools that can be used as a baseline, and a similar system would play the role of a competitor, which would not necessarily indicate the best performance that the system can deliver in a given setup. One could indeed use a competitor system to set the goal for the performance of a system, but this approach still misses the point of informing the developers that there is more to be extracted from the system.

The predictor's approach for the distributed configurations builds on the single-machine scenario. The predictor models part of the overhead of the distributed system that is inherent from the distributed environment (e.g., data transfers or network sharing), while another part is directly related to the implementation and deployment environment (e.g., resource locking or network packet loss). To capture the inherent interactions of the distributed environment, the predictor uses a queue-based storage system model (Section 3.2.3), which is not affected by performance anomalies

caused by the implementation.

The system's performance in the single machine scenario seeds the different components of the model (SAI, storage module, and manager), capturing the performance of the system components when there is no efficiency loss caused by the distributed environment. Finally, a utility tool seeds the performance of the network part of the model.

Predictor Input/Output

The predictor requires three inputs: (i) the performance characteristics of the system's components, (ii) the storage system configuration, and (iii) a workload description. The performance characteristics of the system's components used to seed the model are based on the single-machine scenario. Not present in the single-machine case is the network substrate, which is characterized via a utility tool to measure network throughput and latency (e.g., iperf [128]). Section 3.2.4 details the procedure for seeding the predictor's model. Chapter 3 presents a complete discussion of the predictor design, its input, the model seeding process, and an evaluation for its response time, scalability, and accuracy.

Intended Usage Scenario

The predictor instantiates the storage system model with the specific component characteristics and configuration, and simulates the application run as described by the workload description. When the simulation concludes, it provides an estimate of the performance of the system for a given setup.

The developers of the system actually deploy and run the system, in the same setup used to obtain the prediction, and measure the actual performance. They then compare the estimate for the simulator to the actual performance and decide whether they are close enough to their goal. If they are not close enough, they pursue more optimizations by debugging the system, and follow this process iteratively as one phase in the development cycle (see Figure 4.1).

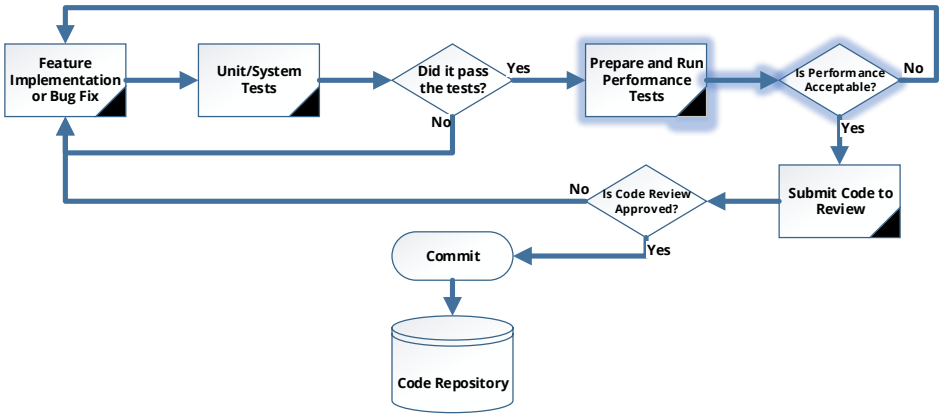


Figure 4.1: The use of the performance predictor as part of the development cycle. The developers have a new phase when they run a benchmark (i.e., a performance test) and compare to the performance predictor results to verify whether of not the performance is acceptable.

Note that with the predictor the developers can obtain the expected performance at the granularity of a module (i.e., client, storage, or manager) and main operations (e.g., read and write), which is the granularity captured by the model in the predictor (see Section 3.2.3). The developers then can track the mismatch per module, as summarized in the cases described in Section 4.3.

4.3 Experience Using the Predictor during Development

As part of the development cycle of MosaStore, synthetic benchmarks run on top of the system to mimic real workflow applications' access patterns [159] and real applications, in order to evaluate the performance of the system. These synthetic benchmarks represent worst-case scenarios in terms of the storage system's performance, as they are composed exclusively of I/O operations, which are intended to create contention and expose the overhead of the storage system.

The predictor estimates performance metrics for the same environment used to evaluate the performance of the storage system, as explained in

Section 4.2.1 as the *intended usage scenario*, and shown in Figure 4.1, the performance predictor was included in the development process. That is, the developers execute a benchmark and instantiate the predictor to mimic exactly the same set-up in terms of workload, scale, and configuration of that benchmark. After executing both, the developers compare the metrics of the predicted and actual performance.

In a number of cases, the predicted and actual performance were close, providing confidence that the implementation was efficient. There were cases however in which the actual and predicted performance differed significantly; these cases highlighted complex performance-related anomalies, leading to a debugging effort. This section presents three of these cases of performance anomalies by describing how they affected the system. Specifically, it describes how the predictor identified performance anomalies, how the predictor was useful to address them, and how the fixes impacted the system performance[3].

The cases presented here focus on two benchmarks used in Chapter 3 and summarized here:

Pipeline benchmark. A set of compute tasks are chained in a sequence such that the output of one task is the input of the next task. 19 application pipelines, one for each machine, run in parallel and go through three processing tasks.

Reduce benchmark. A single compute task uses input files produced by multiple tasks. In the benchmark, 19 processes run in parallel on different nodes, each consumes an input file, and produces an intermediate file. The next stage consists of a single task reading all intermediate files, and producing the reduce-file.

Experimental Setup. The benchmarks run on a testbed of 20 machines, each with an Intel Xeon E5345 2.33GHz CPU, 4GB RAM, and 1Gbps NIC.

[3]The accuracy of the predictor is not the focus of this chapter, which rather focuses on how the tool can be useful for supporting the development process. Chapter 3 presents a deep evaluation of the accuracy of the predictor using synthetic benchmarks and real applications, including a detailed presentation of each benchmark.

MosaStore's manager runs on one machine, while the other 19 machines run both a storage node and a client access module per machine. The plots in Figure 4.2 show the average turnaround time and standard deviation for at least 15 trials, which guarantee a 95% confidence interval, with relative error of 5%, according to the method described by Jain [98].

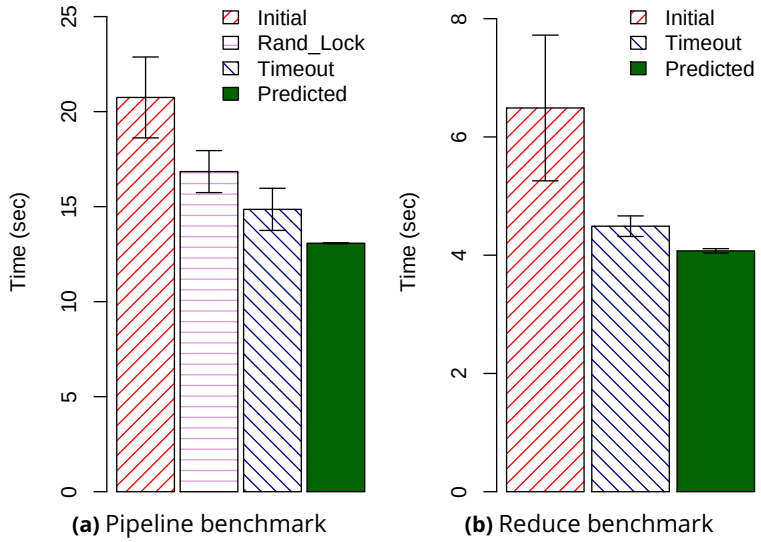
To make the impact of these improvements clear, Figure 4.2 shows the average time for the two benchmarks and different versions of the system. Figure 4.2a has four bars: one representing the time obtained from the initial version of the storage system, two showing the versions of the system after fixing the performance anomalies described in this section, and the last showing the predicted time. Figure 4.2b includes only one of the performance anomalies, since the other one does not impact this benchmark. Overall, the use of a performance predictor allowed improvements of up to 30% in the execution time of the benchmarks, and a decreased variance by almost 10x in some scenarios, by directing the developers in a debugging effort.

4.3.1 Case 1: Lack of Randomness

Context. Whenever a client creates a file, it contacts the manager to obtain a list of storage nodes that will be used during the file write operation. The number of storage nodes is determined by the stripe-width configuration parameter. The order in which the storage nodes will be used to write is determined by the manager, and should be shuffled according to a random seed.

Problem. The manager used a constant seed to shuffle the list of storage nodes returned. Therefore, when the system was set to use the maximum stripe-width, all clients obtained the list in the same order, issuing write operations to the storage nodes in the same order. This created temporal hot-spots where the first storage node received connections from all clients, while other storage nodes were idle.

Detection. When developers are interested in obtaining the response time for a given component of the system, they can simply obtain this



(a) Pipeline benchmark **(b)** Reduce benchmark

Figure 4.2: Impact of fixing performance issues. *'Rand_Lock'* bar shows results for fixing two performance anomalies together (Sections 4.3.1 and 4.3.2). Section 4.3.3 presents the case plotted as *'Timeout'* bar

.

by turning on a log option that measures the time from the reception of a request until its response at the component acting as a server for that request. In fact, this functionality helped in this case and was key in detecting the problem for the next two cases. By studying the behaviour of the system from the logs, the developers stated a hypothesis for the problem, fixed it, and executed the benchmarks again.

Fix. The fix for this problem consisted of changing the algorithm that shuffles the list of storage nodes, so it would use a different seed every time it is invoked, providing the necessary randomness for node allocation.

Impact. The performance improvement can be seen in the *Rand_Lock* bar in Figure 4.2a, which combines this and the next case.

4.3.2 Case 2: Lock Overhead

Context. Whenever a client reads or updates metadata information (e.g., opens or closes a file), it contacts the manager, which needs to lock multiple

updates over the metadata in order to avoid race conditions.

Problem. The developers of the initial version of the system chose a conservative approach, by opting to lock the large code blocks that are called during the client invocation, rather than locking only the critical regions.

Detection. The predictor shows a mismatched between the actual and predicted time for the pipeline benchmark. Based on the manager's actual service times, the developers could verify that a few requests to the manager took reasonably longer than the average. Once they spotted the problem in the manager, they started a debugging process to spot the parts of the code that took longer and detected large and unnecessary lock scopes.

Fix. Reduce the lock scope.

Impact. The performance improvement can be seen combined with the lack of randomness case in Figure 4.2b. The overall gain was around 2.5 seconds in average. Note that the variance also decreased.

4.3.3 Case 3: Connection Timeout

Context. Similar to the way in which the predictor helped us revisit assumptions about the implementation of the system, it also pointed out issues related to the middleware stack that the storage system relies upon.

If a client contacts a server to establish a TCP connection, it waits for a specific timeout before trying it again, if the original attempt fails. Note that this is to establish the connection, and not the TCP window management that occurs once the connection is established.

Problem. In cases where too many clients tried to send data to a storage node at the same time, the storage node dropped some SYN packets (packet used to establish a TCP connection) because there were too many packets in its queue. The client then waited for a timeout defined by the TCP-implementation, which is 3 seconds, consequently taking the average time to write the data in the benchmark to a much higher value than expected.

Detection. The developers also used the system logging to verify the service time of each component in this case. The predictor provides similar information, which was compared against the system's logs in order to

identify when there was a mismatch between the time predicted for the storage nodes to finish a request and the actual time. Indeed, system logs showed an actual longer waiting time.

In this case, however, obtaining service times for each request did not help much because the problem actually prevented the request from reaching the storage component in the first place. To find the gap between predicted and actual time, the developers added a new functionality that logged service time at the client side. By collecting these times and analyzing them, the developers could see that most of the requests were processed around the time predicted, and the other requests were processed three seconds later. By debugging the code, the developers found that this gap happened in the network connection phase of the request and discovered how the TCP implementation could lead to this case.

Fix. The developers decided to define their own timeout instead of relying on the system's default. To do so, they needed to change the implementation, from using blocking to non-blocking sockets during the connection phase.

Impact. The performance improvement can be seen in Figure 4.2a and Figure 4.2b. In this case, the gain affected the Reduce benchmark (Figure 4.2b) more heavily due to its data flow, where several machines write to just one.

4.4 Problems Faced

The limitations of using a separate model to capture the actual implementation behaviour are well-known, and well-captured in a sentence by G. E. P. Box: "Essentially, all models are wrong, but some are useful". Indeed, properly capturing the system's behaviour to provide useful performance estimates, or to correctly define the deployment to be simulated, can be challenging.

In some cases, the system analysis was affected by simplifications in the model, shortcomings of the seeding process, or incorrect assumptions

about the deployment platform. The following list describes the four main problems faced:

Workload description mismatch. The predictor receives a collection of I/O operations based on the log of a benchmark's execution. These benchmarks launch processes on specific nodes via `ssh` (secure shell). The actual time to launch one round of processes in the cluster varies from 0.1 to 0.3 seconds. This variance is not related to the storage system, but it affected the comparison between actual and the predicted time. After discovering this variance, we added it to the workload description to be simulated.

Platform Mismatch. During the execution of the benchmarks, we realized that one of the machines used in the experiments was reasonably faster than the others. In this case, therefore, we had specified a deployment environment that was not the one actually used. Running the seeding process on the fast machine, and specifying a more accurate deployment environment for it, properly fixed the problem.

Modeling Inaccuracy: lack of local priority for reads. In cases where there are several replicas of a chunk in the system, the simulated client selects one randomly. The actual storage system, however, gives priority to chunk replicas located in the same machine. The simulator did not capture this priority and it led to mismatched predictions in cases with higher replication levels. It happened in the initial stage of the simulator implementation, but unit tests captured the problem.

Modeling Inaccuracy: connection timeout. The connection timeout problem described in Section 4.3.3 shows a case where the implementation did not handle establishing connections properly. The implementation was changed to avoid long waits. Nevertheless, a remote machine receiving more requests to open connections than it can handle would still drop some of the requests, which may increase response time.

This situation describes a scenario where the model does not capture the actual behaviour of the deployed system. We consider this problem

to be a result of the environment. Therefore, we did not extend the model to include this behaviour and adjust the prediction since the predictor should provide accurate performance in the absence of implementation-related or environmental issues.

4.5 Discussion

This section discusses the use of a performance predictor as a tool to facilitate the development of complex systems, with the goal of calling the community's attention to the value of producing such a tool as the one presented in Section 4.2.

Can a performance predictor bring benefits to the software development process?

The case study presented in this chapter shows the potential usefulness of having a performance prediction tool that can set a performance target for a system in deployments with different configurations and scale. The predictor brings confidence in the results obtained in several scenarios, is successful in pointing out scenarios that needed improvement, and can support the improvement effort. In fact, the performance of the system improved by up to 30%, and the response time variance decreased by almost 10x in some scenarios (e.g., the case in Section 4.3.3), as a result of applying this approach.

We believe that applying a performance predictor in the development of other complex systems would also be beneficial. Similar to back-of-the-envelope calculations, the predictor indicates the bounds of expected performance for a given system. A predictor can, however, take back-of-the-envelope calculations a step further, because the model it uses provides building blocks to guarantee its usefulness in complex scenarios where back-of-the-envelope estimates are intractable or inaccurate. Not only can developers use a predictor to obtain a baseline to detect performance anomalies, but also to evaluate the potential gains of implementing new

complex optimizations, or to study the impact of a faster network and nodes on a system.

What are the limitations and challenges of using a performance predictor as part of the development process?

Overall, the problems that we faced using this approach can be split into two classes: (i) on having accurate performance predictions, and (ii) on its use during the development phase.

The **first class** covers problems related to performance predictors in general [24], such as having an accurate model and proper seeding. Section 4.4 describes the problems encountered during the case study. Additionally, the approach we apply has one main challenge: to define the baseline to set the performance goal.

Since we were interested in a distributed system, we targeted this challenge in two phases, as described in Section 4.2.2: First, we use a scenario focusing on a local deployment of all components that has the performance goal set by comparing the performance to a similar competitive system. Second, we use a distributed scenario where the performance goal is given by a queue-based model. This approach of handling the complex distribution case by building the distributed scenario on top of a single machine scenario was key for the success of the predictor use.

Other approaches, such as microbenchmarks and analytical models, can be used to define these baselines. The proper choice, however, is system-dependent as each case and approach have their own trade-offs.

The **second class** of problems is related to how developers use such a tool. In our experience, after the initial month of use, developers started skipping the prediction tool as part of the cycle. During the initial phase of applying the tool to development process, the developers could discover several problems and this led to several improvements of the system. After this initial phase, the gains of frequently running the tool and actual benchmarks decreased since most of the problems were addressed.

This problem is similar to having a suite of automated tests that take too long to execute. In these situations, the test suite tends to be split in different

suites, one to be used often by the developers and another (or several) to be used as pre-commit or during nightly builds. Hence, we advocate that the predictor tool should be used as part of a large suite of automated tests. In fact, we believe that developers should also specify a percentage of tolerated overhead over the predicted performance in order to define acceptance tests as part of this large suite of automated tests.

An additional challenge shared among all the performance prediction approaches is to define how much overhead is acceptable on the top of the baseline performance. In all cases, the developers set 10% as the minimal threshold to stop the debugging process, but they pursued a performance mismatch in all cases - even for cases that were already within the 10% threshold - and defined when to stop in an ad hoc manner after reaching at least a 10% mismatch. The proper choice remains an open question. In fact, we believe it is case-dependent, since it depends on an analysis of the trade-off of the estimated future development effort to fix the mismatch vs. the potential performance gain brought by a fix. The prediction tool, however, still serves to guide this decision.

4.6 Related Work

This section relates the approach described in this chapter with other attempts to integrate performance prediction/analysis into the software development process. The goal of this section is to summarize how different techniques are used to improve efficiency during software development. Past work (e.g., the "Performance by Design" book by Menasce et al. [116]) discusses techniques with similar goals for the design phase, before software development, which is out of the scope of this chapter.

Balsamo et al. [24] conducted a survey of model-based performance prediction at software development time. According to them, the first attempt to integrate performance analysis into the software development process was conducted by Smith [149] and was named Software Performance Engineering (SPE) methodology. SPE is a generic methodology that relies on software and system execution models to specify resource

requirements and conduct performance analysis, respectively. Our main goal, in contrast, is to have a predictor to set a goal for performance, although it can also assist in performance analysis and debugging.

Over the years, some approaches were proposed based on this SPE methodology [51, 150, 168]. Most of them analyze Unified Modeling Language (UML) diagrams (e.g., class, sequence, and deployment diagrams) to build software execution models to achieve performance prediction. For example, Williams and Smith [168] employed Class, Sequence, and Deployment diagrams enriched with a Message Sequence Chart to evaluate performance of an interactive system designed to support computer aided design activities. They described their experience in using these architectural models to verify performance requirements during software design in order to support architecture trade-off decisions. Bruseke et al. [40] use a Palladio-based component description that contains a contract for the performance. They use this contract to estimate the performance of a chain of components, and to perform blame analysis when the composition violates the contract. Cortellessa and Mirandola [51] also used UML diagrams to generate a queueing network based performance model. However, they prioritized State transition and Interaction diagrams as a source of information.

There are two main differences between these approaches and the one described in this chapter: First, our main goal is to have a predictor to set a goal for performance instead of only understanding how the performance of a given implementation will be. Second, these approaches require UML diagram analysis to build Queuing Network-Based (QN) models, while we build them from scratch with a coarser granularity (main system components).

On the one hand, the use of UML allows the developers to automate some steps of model specification, such as execution paths specification once the UML diagrams are done. On the other hand, system documentation, such as UML diagrams, may not be accurate and tends to be neglected over time[107, 132]. By focusing on the main components, the approach presented in this chapter avoids the effort of keeping the UML model updated and accurate.

Past work had proposed solutions for improving efficiency during the development cycle by detecting when the introduction of new code to add features or fix bugs negatively impacted performance. For instance, Heger et al. [89] proposed an approach to: (i) detect performance regressions during software development and (ii) isolate the root cause of such regression. The former is achieved by employing unit testing over the history of the software under analysis and comparing it to the performance results to uncover possible performance regression introductions. The latter is achieved by applying systematic performance measurements based on call-tree information extracted from the unit tests' execution. Besides controlled experiments, the authors also present their successful experience investigating a performance regression introduced by the developers of the Apache Commons Math library [4].

Similar to this work, Heger et al. [89] proposed an approach that advocates for the introduction of performance evaluation into the software development cycle at a code commit granularity. Our work, however, focuses on performance prediction, rather than detecting possible bottlenecks introduced by code changes. In summary, we are concerned with how the system will perform given a specific workload scenario before the feature development, while Heger et al. try to detect a performance regression as soon as it is introduced into the code.

4.7 Concluding Remarks

Every tool used during software development reflects a deliberate decision based on the trade-off between the cost and the benefits of employing such a tool. Hence, stakeholders need to gather information to support their choices. In this context, this chapter advocates for the use of performance predictors to develop complex software systems by narrating the experience of applying a specific performance predictor in the development of a complex and error-prone distributed storage system.

[4] http://commons.apache.org/proper/commons-math/

Overall, the approach of using a predictor was useful for setting goals for the system's performance, despite the difficulties of properly seeding the model and mimicking the benchmarks used to evaluate its performance. This chapter shows how the performance predictor helped developers to provide an efficient implementation of a distributed storage system. Specifically, this approach points out situations that needed further improvement, which render up to 30% performance improvement, and a decrease of 10x in the response time variance, for some specific scenarios. It also increases confidence in the implementation when the actual performance matched the predicted one (Chapter 3).

Based on the experience described in this chapter, we recommend the use of performance predictors during software development to help developers deal with this non-functional requirement, similar to how automated tests verify functional requirements. In particular, when developing a large scale and high performance system, in which performance is a key concern, we believe that a predictor is useful to detect potential performance problems early and, consequently, reduce the effort of required to remove performance bottlenecks.

Additionally, I suggest that the results described in this chapter can encourage researchers in the software engineering community to verify the generalization of the benefits observed in this use-case by studying a broader set of systems.

Chapter 5

Automatically Enabling Data Deduplication

To better understand the challenges entailed in automating storage system configuration at runtime, this chapter focuses on one optimization – namely, enabling online data compression through similarity detection in the context of checkpointing applications. Specifically, this chapter presents a study that uses a control-loop to automatically enable or disable data deduplication, having two key metrics forming the optimization criteria: (i) to improve writing time[1] and/or (ii) reduce energy consumption[2].

Overall, the proposed solution correctly configures the storage system to enable or disable data deduplication with small overhead – negligible extra memory and 3% extra time, and small error for the target metrics; the cost of a misprediction is as low as 5%.

The rest of this chapter is organized as follows: Section 5.1 presents a motivation for targeting data deduplication in the context of checkpointing applications. Section 5.2 presents a high-level view of the architecture

[1] *Towards Automating the Configuration of a Distributed Storage System* [52]. **Lauro Beltrão Costa**, Matei Ripeanu, 11th ACM/IEEE International Conference on Grid Computing. October 2010

[2] *Assessing Data Deduplication Trade-offs from an Energy Perspective* [54]. **Lauro Beltrão Costa**, Samer Al-Kiswany, Raquel Vigolvino Lopes, Matei Ripeanu. Workshop on Energy Consumption and Reliability of Storage Systems in conjunction with International Green Computing Conference, July 2011

used and Section 5.3 describes an instantiation of this architecture as a proposed solution. Section 5.4 describes one of this work's important contributions; it focuses on energy consumption by first studying the impact of data deduplication on energy consumption and, then, extending the proposed solution to include this metric. Section 5.5 briefly compares the work described in this chapter with past work. Finally, Section 5.6 presents a summary and a discussion of the results presented in this chapter.

5.1 Motivation

Checkpointing means persistently storing snapshots of an application's state. These snapshots (or checkpoint images) may be used to restore the application's state in case of a failure, or as an aid to debugging.

Depending on the checkpointing technique used, the application's characteristics, and the time interval between checkpoint operations, checkpointing may result in large amounts of data to be written to the storage system in bursts. These bursts provide an opportunity to the intermediate storage system approach - also known as burst buffer [112] - to avoid waiting for the backend storage system during these writes. Additionally, and more important in the context of automation, successive checkpoint images may have a high degree of similarity that data deduplication techniques can leverage to improve performance.

Data deduplication [11, 76, 109, 125, 138, 172] is a data compression storage technique that eliminates redundant data. This technique provides a trade-off between computing power and the amount of data – it consumes additional CPU cycles to detect data similarity and, in return, reduces the amount of data stored.

When the system performs inline data deduplication, i.e., the storage system deduplicates the data while the application is writing the data to the storage system, it can reduce the number of bytes sent over the network and I/O operations as well as the amount of data to be written to the storage system. However, it does not necessarily imply faster write operations or lower energy consumption, since the cost of creating the chunks' identifiers

and chunking the data is expensive and, therefore, it is not clear that the performance cost is paid off (see Section 2.2.4).

Al-Kiswany et al. [12] demonstrate that similarity can be effectively detected at the storage system level (i.e., without application support) for checkpoint applications using data deduplication and that this storage technique offers performance improvement in some cases. This past work, however, leaves a gap: it relies on the system administrator or user to configure a data deduplication technique based on her knowledge about the checkpointing technique used by the application, the application characteristics, and the checkpointing interval.

Even when the user has all necessary information, manual, static tuning can be undesirable. In a scenario where the checkpoint frequency varies (e.g., as used for debugging), the proposed solution focuses on dynamically (i.e., at runtime) enabling similarity detection only when the application produces checkpoints with high similarity and disabling similarity detection otherwise.

Similar to the workflow applications and the solution proposed in Chapter 3, this chapter presents a solution that relies on a repetitive pattern and on a prediction mechanism.

Unlike the solution proposed for workflow applications (Chapter 3), this chapter focuses on a solution predicated on one key assumption: the existence of repetitive operations throughout the *application lifetime* of a checkpointing application [12] and a small variance in the performance, or data similarity, of these operations. This characteristic gives the opportunity to compare the effects of various configuration options to estimate an optimal, or simply good enough, configuration during runtime. In this case, it satisfies the requirements described in Section 1.2 by using a control loop that relies on a simple, yet, effective performance model.

Additionally, past work on data deduplication does not analyze its impact on energy consumption, which has increasing importance in the context of high-performance computing systems (see Section 2.2.2). This study, to the best of my knowledge, is the first to study the impact of data deduplication on storage system energy consumption [54].

Goal. The high-level question that this chapter addresses is the following: *"What are the Challenges and Efficacy of an Automated Solution for the Online Configuration of an Intermediate Storage System?"*

To better understand the challenges entailed by the online automation of storage system configuration, this chapter presents a preliminary study focusing on data deduplication, in the context of checkpointing applications, using writing time, energy consumption, and storage space as optimization criteria.

The progress in the direction of enabling data deduplication not only sheds light on the challenges faced in automated tuning, but also has direct applications of practical value. Current operational practice when running checkpointing applications requires a wealth of information about the checkpointing characteristics (e.g., checkpointing technique, or frequency) in order to configure the storage system. Automating the choice of enabling or disabling data compression via deduplication allows decoupling the concerns of the application's scientist or developer from those of the storage system operator.

The measurements from this study demonstrate that the developed prototype correctly configures the storage system to enable or disable data deduplication, with minimal overhead and minimal error for the target metrics. Additionally, it shows that, in the context of data deduplication, optimizing for energy consumption and for writing time become conflicting goals as hardware becomes more power-proportional (i.e., hardware draws power proportionally to its usage load).

5.2 Architecture

A solution that automatically enables or disables inline data deduplication should meet the requirements listed in Section 1.2. In this context, they are:

- to reduce or eliminate the need for human effort to configure the storage system,
- to provide performance that is close to the user's intention,

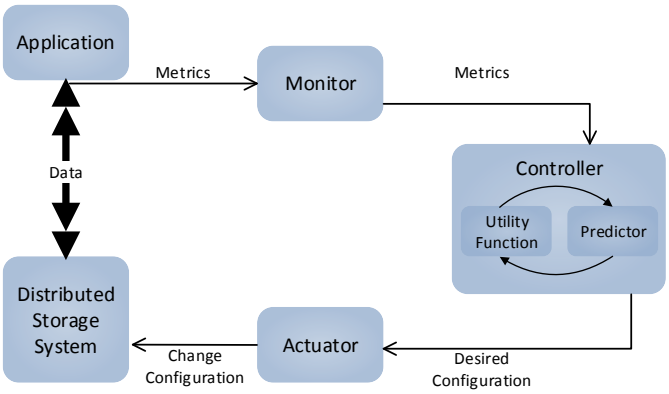


Figure 5.1: Control loop based on the monitor-control-actuate architectural pattern.

- to keep the overhead cost of automating the configuration low.

The existence of repetitive operations throughout the *application lifetime* of a checkpointing application [12] enables the use of a monitor-control-actuate loop. This approach continuously adjusts the configuration of the storage system to optimize its performance according to a user-specified optimization goal (Figure 5.1).

The closed loop consists of a *monitor*, which constantly monitors the behavior of the storage system. The actual performance metrics monitored depend on the target optimization goal and on the active configuration at the moment. For the target storage system, the monitored metrics can include: the amount of data sent through the network, the amount of data actually stored, the response time for an operation, compute or memory overhead at the client machine, and network latency.

The monitor passes these measurements to the *controller*, which analyzes them, infers the impact of its previous decisions, and may dispatch new actions to change the system configuration to the actuator. To perform its tasks, the controller needs: (i) a utility function that captures the user's intention, and (ii) a prediction mechanism that estimates the impact of its future actions on the user's goal.

The *utility function* is an equation that receives different performance metrics and produces an output according to the user's intention. The function's goal is to reduce the multidimensional space of the monitored metrics to just one dimension that guides the optimization effort. That is, the goal of the controller is to maximize the utility.

Initially, this study assumes the user focuses on a single metric to optimize upon (e.g., minimize the write time for I/O operations), which can be given by a simple function as $\mathcal{U}(\mathrm{T}^w) = -\mathrm{T}^w$, where T^w is the writing time. More complex definitions, however, are possible [152]: the user can, for example, define that she tolerates the write time using data deduplication to be up to 20% longer than when using the default configuration, if the amount of data stored is reduced by more than 30%.

The prediction mechanism estimates the impact of a specific change in configuration on the target performance metrics, given the current state of the system. For example, for a storage system that can save space by compressing data, the predictor receives estimates of the achievable compression rate, and predicts the impact of compression on the time required for write operations and the space saved.

Finally, the controller decides whether configuration changes are needed and communicates its decision to the actuator. This decision consists of estimating which configuration provides the best utility based on the current delivered utility and state, the accumulated past history of changes, and the estimates for the utility to be delivered with new configurations.

The actuator performs the configuration changes as instructed by the controller. While this architecture and a number of its components, such as the controller, the means to express utility, and performance models are generic; some of the other required components are specific for each storage system supported (e.g., the actuator, the monitor), or for each application context, as it is the case for the specific utility function and the predictor.

5.3 Control Loop for Configuring Similarity Detection

The goal of this exploration was to automate the choice between two configuration options: data deduplication on or off.

The control loop needs the following to be instantiated: the metrics to be collected, the utility function the user defines to guide the choice of a configuration, and the performance model used for the prediction. Initially, two metrics are exposed to the user in order to define the utility function: (i) the time consumed for writing a checkpoint image, and (ii) the amount of data stored. In the scenarios studied in this section, the utility function that drives optimization is a linear combination of these two metrics. For example, the user may specify that, regardless of time, the storage footprint should be minimized, or conversely, the user may specify that reducing the checkpointing time is the primary optimization criteria. Section 5.4 presents an extension of the the solution to include energy consumption as a third metric.

For each write operation, the *monitor* needs to collect the following information: (i) the operation timestamp, (ii) the total duration of the operation (note that this includes the time spent to calculate the hash value and to send data over the network), (iii) the total number of chunks received by the write function, and (iv) the number of chunks similar to older versions. From (iii) and (iv) the *controller* can extract the percentage of similarity, estimate the best configuration, and instruct the *actuator* to change the configuration.

Similar to the approach presented in Chapter 3, the controller is built on top of a performance model that estimates the impact of configuration changes on the system's performance, which is described in Section 5.3.1. Section 5.3.2 describes the implementation of each component of the loop. Section 5.3.3 evaluates the proposed solution.

5.3.1 Prediction Model for the Controller

This modeling exercise captures the behavior of the write operations under inline deduplication and captures the situations where data deduplication is

desired. The performance model needs to predict the two metrics that are exposed to the utility function: (i) total time to write (T^w) and (ii) storage space used (S^w). Additionally, there are two possible configurations: similarity detection enabled or disabled.

Symbol	Description
T^w	Time to complete a write operation
T^{io}	Time to perform the IO phase
T^{sim}	Time to detect data similarity
T^w_{on}	T^w when similarity detection is on
T^w_{off}	T^w when similarity detection is off
$t^{io}(x)$	Time to perform an IO operation for x units of storage
d	Size of data to be written (received by the write operation)
c	Size of the chunk
S^w	Amount to be stored
S^w_{on}	$S^w$when similarity detection is on
S^w_{off}	S^w when similarity detection is off
$\mathcal{Z}$	Data similarity detected
T^h	Time to calculate hash for data chunks
$t^h(x)$	Time to calculate hash for x units of storage
$\mathcal{U}$	Utility function

Table 5.1: Terms of the data deduplication prediction model.

Table 5.1 summarizes the list of terms used to formalize the prediction model. Given the terms of Table 5.1, time to perform a write operation is given by:

$$T^w = T^{sim} + T^{io} \tag{5.1}$$

Let d be the data size, and $\mathcal{Z}$ be the data similarity detected. The time to perform the actual I/O phase is:

$$T^{io} = t^{io}(d \times (1 - \mathcal{Z})) \tag{5.2}$$

When *similarity detection is disabled*, predicting the space for a write operation is simple: the amount of data received by the write operation is the amount of data persisted. Formally,

$$S^w_{off} = d \times (1 - \mathcal{Z}) = d \times (1 - 0) = d \tag{5.3}$$

The time depends on the I/O operations which typically present a higher variation and depend on several factors, including the operating system's operations, buffers, and other I/O operations. The model predicts the write time by analyzing the history of measurements obtained. It calculates a moving average of the required time to write a chunk. The moving average is based on the last N operations where N can be tuned to improve accuracy. Once the time to write a chunk is estimated, the total time for the writing operation can be obtained by multiplying the time to write a chunk by the number of chunks to write. That is,

$$T^{w}_{off} = T^{sim} + T^{io} = 0 + t^{io}(S^{w}_{off}) = t^{io}(d) = t^{io}(c) \times \frac{d}{c} \tag{5.4}$$

When *similarity detection is enabled*, the total time is the sum of the time required to calculate the hash values for every chunk and the time to write the new chunks. It ignores the time to compare hash values, as this is usually orders of magnitude smaller than the previous two. The model estimates the number of similar chunks based on the history of the last measurements. Then, the controller knows how much space is needed and can estimate the I/O operations cost using the same approach as described in Equation 5.4, when the similarity detection is disabled. Formally,

$$S^{w}_{on} = d \times (1 - \mathcal{Z}) \tag{5.5}$$

$$T^{w}_{on} = T^{sim} + T^{io} = t^{h}(d) + t^{io}(S^{w}_{on}) = \left[t^{h}(c) \times \frac{d}{c}\right] + \left[t^{io}(c) \times \frac{S^{w}_{on}}{c}\right] \tag{5.6}$$

The system can obtain a reasonable approximation of hashing overhead by calculating the hash for just one chunk, and multiplying this value by the total number of chunks received for the write operation. This can be done by writing two versions of a file of chunk-size with the same contents. The operations related to the second file consists of just hashing the data. Note that the controller can also keep a history of the time consumed to calculate the hash, and estimate the time cost of hashing one chunk based on such a

history.

Finally, the controller needs to choose the desired configuration. Currently, it uses the prediction model described in Equations 5.3, 5.4, 5.5, and 5.6 to estimate the metrics for the two possible configurations: similarity detection on or off. Then, the controller applies the utility function to the estimated metrics, verifies which configuration provides the best utility, and may request the actuator to change the current configuration, if it is not the one with the best utility. Assuming a simple utility function that just relies on writing time ($\mathcal{U}(T^w) = -T^w$), it is worth it to enable deduplication if:

$$\mathcal{U}(T^w_{on}) > \mathcal{U}(T^w_{off}) \iff T^w_{on} < T^w_{off} \tag{5.7}$$

More complex utility functions may involve different metrics and need to normalize these metrics. For example,

$$\mathcal{U}(T^w, S^w, T^w_{on}, T^w_{off}, S^w_{on}, S^w_{off}) = \left[\gamma_1 \times \frac{T^w}{max(T^w_{on}, T^w_{off})}\right] + \left[\gamma_2 \times \frac{S^w}{max(S^w_{on}, S^w_{off})}\right] \tag{5.8}$$

Equation 5.8 normalizes the value of each metric by dividing it by the maximum value among the different configuration options (e.g., $\frac{T^w}{max(T^w_{on}, T^w_{off})}$), and gives weights ($\gamma_1$ and γ_2) for the different factors. If the user specifies the weights as $\gamma_1 = \gamma_2 = -0.5$, relative savings for both metrics have the same importance in their optimization criteria.

5.3.2 Implementation

This section describes the implementation of each of the components of the control loop described in Section 5.2 as well as the effort to integrate the control loop that enables and disables similarity detection into the MosaStore prototype.

To create a file, the client application requests a new file identification from the manager and asks the manager to reserve the necessary space. The manager returns a list of storage nodes that have space. The client then starts writing the chunks of the file to those storage nodes and assembles a

chunk map that maps the chunks their storage nodes for future retrieval [12]. Note that the client only sends to the storage nodes those chunks not found in the previous-versions of chunk maps for that file, and reuses the chunks already stored by simply using their identifiers from the chunk map. Finally, when the client finishes writing, the chunk map is sent to the manager to be persistently stored. To detect similarity between the content of successive write operations, a content-addressable storage scheme is used, as described in Section 2.2.4. Other deduplication systems (e.g., Foundation [139] and Venti [135]) use similar techniques.

For each component of the control loop, a brief summary of its implementation in MosaStore follows:

Monitor. To collect measurements, MosaStore's default write operation flow was changed to provide a new one that captures the metrics previously described in this chapter, based on the design described in Section 2.1.2. The monitor also captures whether similarity detection was used or not, by saving the tag used to describe the file.

Controller. The controller receives measurements from the monitor and archives them. Once the archived history achieves a specific size (the default value in the initial version is five estimates), the controller starts to predict the amount of storage space used and time consumed by write operations, according to the performance model described in Section 5.3.1. The utility function that drives the optimization is specified by the user through a configuration file. Finally, the controller uses the utility function, and, based on the estimated metrics, determines which configuration provides the best utility, then informs the actuator if some configuration change is needed. Section 5.3.3 shows that this online monitoring approach is effective to seed the simple model regardless of the current configuration.

Actuator. Originally, MosaStore did not support configuration changes during runtime. A new feature that supports enabling/disabling similarity detection at runtime was added, following the approach

described in Section 2.1. Note that to use this feature another option of file versioning should also be enabled as described by Al-Kiswany et al. [12].

The initial configuration has the similarity detection activated since the controller needs to know the level of similarity between writes. If the controller deactivates similarity detection, the system cannot provide information regarding the similarity level. This problem is addressed by reactivating the similarity detection again after a given amount of write operations (100 by default in the initial prototype). The overhead for this approach and alternative solutions are described in Section 5.3.3.

5.3.3 Evaluation

This section presents an evaluation of this prototype according to the success criteria listed in Section 5.2: (i) effort to configure, (ii) performance delivered by the automated solution, and (iii) the configuration overhead.

Testbed: MosaStore is configured to use 10 storage nodes. The metadata manager runs on a different machine. Each machine has Intel Xeon E5345 4-core, 2.33-GHz CPU, 4GB RAM, and 1Gbps NIC, as testbed TB20 described in Section 3.3.

Workload: This study uses synthetic workloads that mimic checkpointing workloads based on a previous study from NetSysLab that collected and analyzed [12] checkpoint images; the workload generator produces files at regular time intervals and controls the similarity ratio between consecutive file versions, from 0 to 100% similarity.

Statistics: Each point in the plots is the average over all write operations performed during the application's execution. It guarantees 95% confidence intervals with 5±% accuracy, according to the procedure described by Jain [98].

Effort to Configure

The proposed solution requires minimal human effort to configure: the user only needs to specify her intention. In the prototype, this is expressed by

specifying the weights of runtime and storage footprint size in the utility function. Note that the default configuration of the utility function assumes that the user's intention is to minimize writing time.

System Performance

To analyze how satisfactory the automated configuration is, this section compares the performance of the system using automated tuning and the performance using the two manual configurations available: similarity detection always on or always off.

The first target scenario uses a synthetic workload based on past NetSysLab experience of analyzing checkpointing workload for short checkpointing intervals as used for debugging [12]. The synthetic application writes 100 files of approximately 256 MB each, with data similarity of 70% between writes, in order to analyze the case where the user is interested in minimizing writing time for the three different configurations – i.e., the lower the writing time, the higher the utility as in Equation 5.7.

The proposed automated solution is able to detect that the similarity is high enough in this case to offset the computational overheads generated by hash-based similarity detection. Indeed, the automatic solution performs as well as a configuration in which similarity detection is always activated, which is the best configuration for this case.

To analyze the system's performance under different scenarios, the same synthetic application varies the level of similarity present in the data. This is equivalent, for example, to varying the frequency of the checkpointing operations. For up to 50% similarity, similarity detection should be deactivated in this platform, as the hashing overhead is not paid off by savings in I/O operations. Figure 5.2 demonstrates that automated tuning chooses the correct configuration.

The slightly longer writing time when using automated tuning at low similarity levels is the result of the need to estimate the similarity level in the data stream before making a decision. The end of this evaluation section presents a brief discussion on the cost of automated tuning.

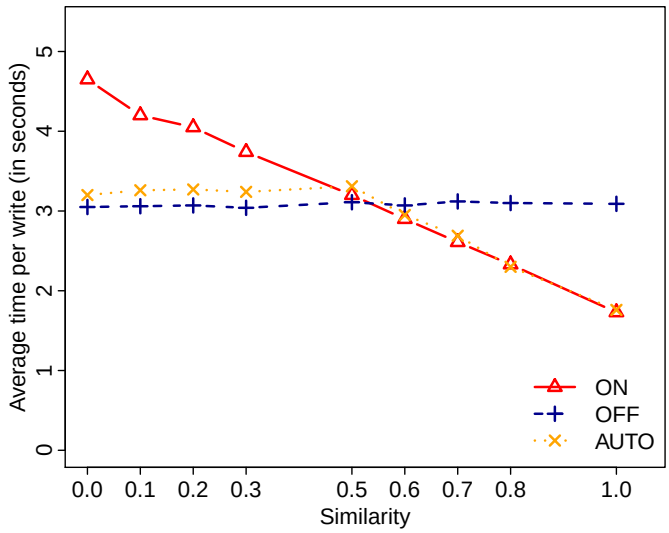


Figure 5.2: Average time to write a snapshot of 256MB. Confidence intervals were small (less than 5%) and are omitted to reduce clutter.

Experiments for writes of different sizes (32 MB, 64 MB, 128 MB, and 512 MB), while varying the similarity, exhibit similar behavior. Although the required time to complete the write operations are different from the experiments using 256 MB, this automated configuration solution always chooses the desired configuration.

To analyze a case where the user is interested in *more than one metric*, the controller uses a utility function that considers storage space and writing time as equally important by configuring the utility function with 50% weight for each of the two metrics, writing time and storage space, as in Equation 5.7.

Figure 5.3 shows the average relative utility when writing checkpointing images of 256MB, while varying the similarity rate in the data stream. The results are normalized by the value of the lowest utility among different configurations (detection always off in this case) to provide a clear comparison.

Up to 20% similarity, the best configuration for the execution platform is having the data deduplication deactivated, as the time needed to hash

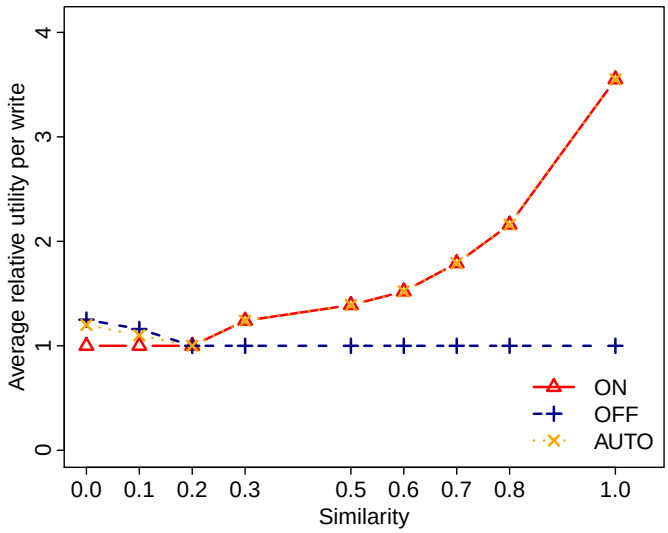


Figure 5.3: Average relative utility, when the two metrics are equally weighted, to write a snapshot of 256MB.

the data is not enough to balance the storage space saved and the savings in I/O operations. As the similarity increases, the similarity detection cost is paid off by the time saved during the I/O operations and the amount of storage space saved.

Similar to the case in which the system focuses on minimizing writing time, in this case the automated solution is able to detect that the similarity is high enough for data deduplication to provide a higher utility. Indeed, the automatic solution provides the same utility in cases where data deduplication is always activated (around 66% higher than the worst configuration for 70% similarity), which is the best configuration for this case. Overall, as the Figure 5.3 shows, the automated configuration choice always matches the optimal manual configuration.

Cost of Automated Tuning

To evaluate the configuration cost, this section analyzes the amount of extra resources used for the automated configuration: namely, CPU time and

memory.

Memory overhead is minimal: the controller only keeps a history of past measurements. Such history is short since the controller uses a moving average for the last N measurements. In the experiments, $N = 5$, which results in less than a kilobyte of extra memory. Older data can be discarded, or persistently stored on the backend for future analysis.

As the experiments presented earlier suggest (see Figure 5.2), the performance of the configuration chosen by the automated solution, including the automation overhead, is within 5% of the performance of the best configuration.

The **computing overhead** generated depends on the state of the system. If data deduplication is activated, there is no additional overhead to estimate the similarity level since similarity detection is already part of the data deduplication process. However, if data deduplication is deactivated, the system still needs to estimate the similarity level. Thus, it starts hashing the data which slows down the write operation, by up to a factor of two, for cases where there is no similarity among writes. Note that such an additional cost exists only until the automated solution detects that hashing the data is not worth it. From this moment on, this cost is amortized over the following write operations. In the current implementation, whenever similarity detection is deactivated, the system activates again after 100 writes.

Depending on the workload, however, keeping the computing overhead low can be challenging for an automated solution. This can result from a combination of two factors: (i) similarity detection is an intrusive storage technique requiring some processing over all data contents, and (ii) the similarity may vary often, requiring to be activated more often as well.

To reduce this cost, two options are available: (i) using sampling to reduce the volume of data analyzed when detecting the level of similarity, and (ii) moving similarity estimation offline, out of the critical performance path, and use an alternative approach to estimate data similarity [153]. Section 6.2 discusses these alternatives.

5.4 Data Deduplication Trade-offs from an Energy Perspective

While the impact of data deduplication on traditional metrics (e.g., data throughput, storage footprint) is well understood [12, 135, 139], previous studies leave an important gap: energy consumption analysis. Specifically, they overlook two important issues. First, while data deduplication increases the CPU load, it may reduce the load on the network and storage devices. As a result, it is unclear under what scenarios it leads to energy savings, if any. Second, the performance impact of energy-centric tuning of the storage system has been unexplored.

This section targets the case of fixed-chunk data deduplication from an energy perspective. First, to reduce the aforementioned gap in the analysis of energy consumption in data deduplication, Section 5.4.1 describes the methodology for an empirical evaluation of the energy consumption of data deduplication in the target case. Section 5.4.2 assesses energy consumption on two hardware platforms, with different characteristics in terms of power proportionality, and identifies deduplication's break-even points – the threshold that delineates when it is worth it or not to deduplicate data from an energy or time perspective, demonstrating that the break-even points for performance (in terms of writing time) and for energy efficiency are different.

Finally, Section 5.4.3 presents a simple energy consumption model that makes it possible to reason about the benefits of deduplication and offers an approximation for the energy break-even point. This model is used to extend the solution presented in Section 5.3 in order to consider energy consumption as a third optimization metric.

This study is related to a rapidly growing body of work on deploying energy efficient systems. It joins others in which the focus is to understand the energy consumption of compression techniques in different scenarios [48, 104]. To the best of my knowledge, this work is the first to study the impact of data deduplication on storage system energy consumption. Moreover, it also considers different generations of machines, to demonstrate

the impact of the new power proportional hardware (i.e., hardware whose power consumption is proportional to the utilization level) [25] on energy consumption.

The empirical evaluation and energy model suggest that, as storage systems and their components become increasingly energy proportional, the energy and time, or throughput, break-even points will shift further apart. This trend has an important consequence: optimizations for energy efficiency and performance will likely conflict. As a result, storage system designers and users will have to make conscious and informed decisions about which metric to optimize for.

5.4.1 Assessing Performance and Energy Consumption

To investigate the impact of deduplication on energy consumption, an empirical approach was chosen: monitoring MosaStore as a representative distributed storage system that adopts deduplication, and subjecting it to a checkpointing-like workload.

Workload. This section analyzes a synthetic workload that mimics checkpointing applications as the one described in Section 5.3.3. The workload generator produces files at regular time intervals and controls the similarity ratio between consecutive file versions (from 0 to 100% similarity).

Assessment Testbed

The performance evaluation and energy consumption was based on two classes of machines which are labeled *'new'* and *'old'* to make it clear they are from different generations:

'new' machines (Dell PowerEdge 610) are equipped with Intel Xeon E5540 (Nehalem) @ 2.53GHz CPU launched Q1'09 and maximum Thermal Design Power (TDP) of 80W, 48GB RAM, 1Gbps NIC, and two 500GB 7200 rpm SATA disks. Nehalem is an architecture by Intel that exhibits major improvements in power efficiency. Indeed, a machine with Nehalem, in this testbed, consumes 86W in idle mode and 290W at peak utilization.

'old' machines (Dell PowerEdge 1950) are equipped with Intel Xeon E5395 (Clovertown) @ 2.66GHz CPU launched Q4'06 and max TDP of 120W, 8GB RAM, 1Gbps NIC, and two 300GB 7200 rpm SATA disks. One of these machines consumes 188W in idle mode and 252W at peak. (Note that these are the same machines from TB20 presented in Chapter 3).

All machines run Fedora 14 Linux OS (kernel 2.6.33.6). MosaStore uses the same configuration in all experiments. As is the case for Section 3.4.2, the lack of infrastructure to measure power limits the scale of the testbed for these experiments. Thus, only two machines from each testbed are used: the storage module on one machine; and the manager, the SAI, and the workload generator on a second machine. The machines are connected by a Dell PowerConnect 6248 10Gbps switch, whose energy consumption is not reported in this study.

The machines are equipped with two WattsUP Pro [4] power meters to measure the energy consumption at the wall power socket, capturing the energy consumption for the entire system. A third machine collects the measurements from the meters via a USB interface. The meters provide 1W power resolution, a 1Hz sampling rate, and $\pm$1.5% accuracy.

Measuring Energy Consumption

For energy, since the meters give only a 1Hz maximum sampling rate, the experiments collect the measurements every second during an experimental batch, from the beginning of the first write until the completion of the last write. The power is given in watts (joules per second) for the last measurement interval, giving an estimate of energy consumed during the last measurement interval. For each machine, the sum of energy consumption estimates is used to obtain the total amount of energy consumed during the experiment batch. This section reports the average energy consumed per write, by dividing the total energy consumption by the number of checkpoint images written.

The evaluation considers the energy consumed by all storage system components: manager, storage node and client. On the client node, however,

the workload generator runs together with the storage system component. Since it is not possible to isolate the consumption only for the storage system path, the evaluation conservatively reports the energy consumed for the whole system, including the workload generator [3].

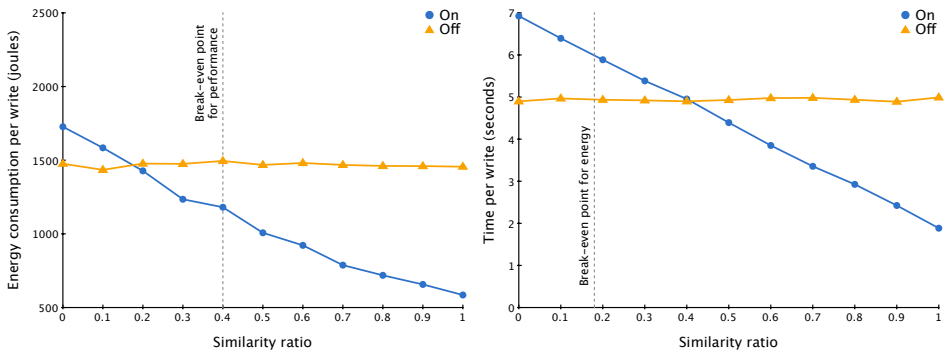
In the plots, each point presents the average value for a file write operation (for energy or time), calculated over a batch of 50 writes at a fixed similarity level. The experiments consider different data sizes (32, 64, 128, 256 and 512MB) while varying the similarity level (0% – 100%, in increments of 10%). Although the time and energy required for each operation varies, the overall relation of energy consumption, time, and similarity is the same regardless of the data sizes. Thus, the plots only show results for 256MB.

5.4.2 Evaluation of the Energy Consumption Results

The goal of this evaluation is to analyze energy consumption, and the writing time performance in writing a checkpoint image, as well as their break-even points in term of similarity level for platforms with different energy proportionality properties. Figures 5.4a and 5.4b present the energy consumption and the average write time per checkpoint on the *'new'* testbed, respectively. The main point to note is that the break-even points for energy and performance are different: for similarity lower than 18%, hashing overhead is not compensated for by the energy savings in I/O operations, thus, enabling deduplication brings benefits only when the workload has a similarity rate higher than 18%. From a time performance perspective, on the other hand, enabling deduplication makes sense only for even higher similarity levels (higher than 40%).

A second point to note is that deduplication achieves different relative gains for energy and time. For energy, the highest consumption level (at 0% similarity) is 2.9x larger than the lowest (at 100% similarity). For the time to write a checkpoint, this ratio is 3.6x. Although deduplication enables energy savings to start at lower similarity level than it enables for saving

[3] Indeed, additional experiments show the workload generator is lightweight compared to the client SAI.



(a) Average energy consumption

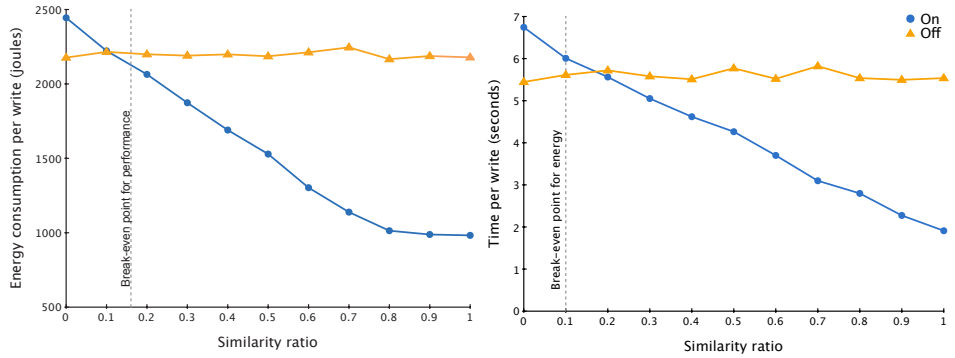
(b) Average time to write

Figure 5.4: Average energy consumed and time to write a 256MB file, with data deduplication on and off, for different similarity levels in the *'new'* testbed. Note: Y-axis does not start at 0 in Figure 5.4a.

in time (i.e., energy consumption has an earlier break-even point), at 95% of similarity, deduplication saves almost the same rate for both energy and time (around 50%).

Figures 5.5a and 5.5b present the energy consumption and the average time per checkpoint write for the *'old'* testbed, respectively. Compared to the *'new'* testbed, the break-even point for energy (at 10% similarity) is closer to the one for performance (at 16% similarity). For this testbed, there are similar differences in the relative gains that deduplication enables.

One important fact to note is that, although the two testbeds have almost the same performance profile as evidenced by the checkpoint write performance, the two testbeds have different energy profiles. The energy consumption per write with deduplication turned off is about 45-50% higher in the *'old'* testbed, even though the writing time is only 10% higher. With deduplication turned on and a high similarity rate, the differences are even more striking: about 2x higher energy consumption in the *'old'* testbed for about the same write time (Figure 5.4a). The reason for these results is that the newer generation machines with Nehalem CPUs are more power proportional and save energy by matching the level of resources enabled



(a) Average energy consumption

(b) Average time to write

Figure 5.5: Average energy consumed and time to write a 256MB file for different similarity levels in the *'old'* testbed. Note: Y-axis do not start at 0 in Figure 5.5a.

(e.g., switching cores on and off) to the offered load.

Summary

The main lesson from these experiments is the following: while for non-energy-proportional machines, performance- and energy-centric optimizations have break-even points that are relatively close, for the newer generation of energy-proportional machines, the break-even points are significantly different. An important consequence of this difference is that, with newer systems, there are higher energy inefficiencies when the system is optimized for performance. The experiments presented above quantify these inefficiencies: on the *'old'* testbed, optimizing for performance leads to up to a 5% energy inefficiency (in the 10-16% similarity rate interval). For the *'new'* testbed, optimizing for performance leads to an up to 20% energy inefficiency (in the 18-40% similarity rate interval).

5.4.3 Modelling Data Deduplication Trade-offs for Energy Consumption

Section 5.4.2 shows that deduplication can bring energy and/or performance savings if enough similarity exists in the workload. It also shows that the break-even points for energy and performance are different, and depend on the characteristics of the deployment environment. Under these conditions, system users need a mechanism to support their configuration decisions related to deduplication.

This section proposes a simple model to guide the storage system configuration. This model can be used in two ways. First, it can identify whether deduplication will lead to energy savings for a given level of similarity. Second, it can estimate the energy impact of upgrading the system; for example, by adding SSDs to improve energy efficiency and time performance, or by adding energy-efficient accelerators (e.g., GPUs) to support deduplication [11, 76].

Two guidelines direct the design of the model. First, the model should be simple to use and not require extensive machine benchmarking or the use of power meters. Alternatively, it should be seeded with power information available on technical data-sheets of various components. Second, the model should be simple and intuitive, even at the cost of lower accuracy, since such models have higher chances of being adopted and used to guide decisions in complex settings.

Consider the following variables:

- B is the total number of chunks to be written (received by a write operation), and c is the chunk size.
- P^{idle}, P^{peak}, P^{io} are used to characterize different power consumption profiles: the power consumed by the machine in idle state, peak CPU load, and peak I/O (disk and network) load, respectively.
- $E^h(c)$ is the *extra* energy consumed by the machine to compute the hash value for one data chunk. It can be roughly approximated by

$E^h(c) = (P^{peak} - P^{idle}) \times T^h(c)$, where $T^h(c)$ is the time for hashing a single chunk.

- E^{io} is the energy consumed in the transfer of one chunk on the storage path - including all system calls, sending the chunk from the client machine, receiving it at the storage node, and storing it. $E^{io} = 2 \times (P^{io} - P^{idle}) \times T^{io}(c)$. The factor 2 appears since there are both one client and one storage node involved in the storage path, and $T^{io}(c)$ is the time for sending, receiving and storing a chunk.
- $\mathcal{Z}$ is the detected similarity ratio of the data.

For each write operation, the extra energy needed to compute the hash values is $E^h(c) \times B$. The energy saved by reducing the stress on the storage path is $E^{io} \times B \times \mathcal{Z}$. Every time the energy savings are higher than the additional energy spent to compute hash values, then it is worth turning deduplication on. Note that this choice is independent of the data volume B. That is,

$$E^{io} \times B \times S > E^h(c) \times B \iff \mathcal{Z} > \frac{E^h(c)}{E^{io}} = \frac{(P^{peak} - P^{idle}) \times T^h(c)}{2 \times (P^{io} - P^{idle})} \times T^{io}(c) \tag{5.9}$$

The above modeling exercise highlights the main theme: for older, non-energy-proportional systems, optimizing for energy and for performance are similar. Thus, the decision to optimize for energy depends only on the relative runtime to hash or store a data chunk, and the similarity level present in the workload.

Power proportionality brings new factors into this equation: the relative position of power consumed when idle – under maximum I/O load, and under maximum compute load. Once a system is power proportional (that is, if P^{idle} is significantly lower than P^{peak} and/or P^{io}), and it draws different power levels at peak CPU vs. at peak I/O load ($P^{peak} \neq P^{io}$), a richer trade-off space emerges.

Seeding. The user can easily use this formula (Equation 5.9) to guide her deduplication related decisions. The parameters needed can be easily

collected by benchmarks ($T^h(c)$ and $T^{io}(c)$), or can be provided by system assemblers in technical sheets (P^{idle}, P^{peak}, and P^{io}), and estimated or extracted from the workload history ($\mathcal{Z}$).

Evaluation of the Energy Model Accuracy

To evaluate the accuracy of this simple model, the model's prediction of the energy break-even point is compared with actual measurements. In this case, the actual measurements come from the two testbeds from Section 5.4.2. To benchmark the testbed and estimate P^{idle}, P^{peak}, and P^{io}, benchmark scripts run an idle workload as well as a CPU, disk and network-intensive workload and measure the consumed power for each workload separately.

Benchmarking the testbed and plugging its characteristics into the model indicates that the energy break-even points are at 21.4% similarity for the *'new'* testbed and at 18.1% similarity for the *'old'* testbed. This is close to a perfect oracle since the actual measurements indicate 18% and 10%, respectively, for the energy break-even points.

Summary. Despite the model simplicity, it estimates the break-even point with relatively good accuracy. The model only fails to predict the proper configuration when the similarity ratio is between 18-21.4% in the new testbed, or between 10-18.1% in the old. Additionally, the cost of such misprediction is low; when the similarity is in these ranges deduplication configured using the model consumes less than 10% extra energy compared to the optimal configuration (Figures 5.4a and 5.5a).

Note that designing an accurate fine-granularity model for the storage system energy consumption is a complex task: the main reason is that it is hard to decouple the energy consumption of different components in the system, as well as the energy consumed by the application running in the system.

5.5 Related Work

This work directly relates to efforts in the design and evaluation of data deduplication solutions and energy efficient systems.

5.5.1 Data Deduplication

A number of research and commercial systems employ various forms of deduplication targeting two main goals: (i) *reducing the storage footprint* – such as Venti [135] and Foundation [139] optimized for archiving, Mirage [138], optimized for storing virtual machine images, and DEBAR [172], optimized for enterprise-scale backup services; and (ii) *improving time performance* by reducing the pressure on the persistent storage, or the volume of data transferred over the network, including low-bandwidth file systems [125], web acceleration [97], content-based caching [103], and the high performance storage system StoreGPU [11, 76].

While previous work focuses on the storage footprint and write-time-related benefits in systems of different scales, the impact of deduplication on energy consumption has not been clear and the possibility of automating the configuration is not explored. On the one hand, detecting similarity introduces computational overhead to compute the hashes of data chunks.

On the other hand, if there is detectable similarity in the workload, the computational overhead can be offset by reduced storage or network effort. This chapter explores this trade-off from an energy standpoint and proposes an initial solution to automate the decision at runtime depending on utility functions that rely on the energy consumption, time, and storage footprint of write operations.

5.5.2 Energy Optimized Systems

With non-power-proportional hardware (i.e., hardware that draws the same power regardless of the utilization level), energy efficiency [25] is tightly coupled with high resource utilization. Consequently, to increase energy efficiency, previous work recommends increasing system utilization through mechanisms such as server consolidation [87], or through runtime optimizations to reduce per-task energy consumption [146].

Hardware has, however, become increasingly power-proportional. This trend opens new opportunities for energy efficiency in the software stack; it enables shifting work from the most energy-efficient component (most

power-proportional) of a computer system to the less energy-efficient component (e.g., from disk to CPU) to reduce the total volume of energy consumed for a specific task [87].

Recently, Chen et al. [48] and Kothiyal et al. [104] have investigated the trade-off of compressing data, or not, in the context of MapReduce applications and data centers, respectively. Similar to this work, they concluded that compression data, is not always the best choice in terms of energy consumption, as it depends on the workload. Ma et al. [114] investigate deduplication performance using a commodity low-power co-processor for hash calculations. Unlike this work, they do not explore the trade-off of exchanging I/O operations with extra CPU load, and do not quantify deduplication energy savings. To the best of my knowledge, this work is the first to study the energy impact of deduplication and to put in perspective the impact of new generations of computing systems that have different energy proportionality profiles.

5.6 Summary and Discussion

This study focuses on an automated configuration solution for data compression, through similarity detection in the context of checkpointing applications. The evaluation presented in this chapter demonstrates that the developed prototype correctly configures the storage system to optimally enable or disable deduplication with a minimal error. Additionally, this study exposes the cost of delivering an automated solution and how challenging energy and similarity estimation can be.

The progress in this direction meets the main goal of shedding light on the challenges and accuracy of automated tuning, in the context of data deduplication. It also has direct practical value, as automating the choice of enabling or disabling data compression deduplication allows one to decouple the concerns of the application's scientist and developer from those of a storage system operator.

This chapter also analyzes *energy consumption* in the target scenario. Energy efficiency has become a pervasive theme when designing, configuring

and operating computer systems nowadays. While for non-energy proportional computer systems, energy and performance centric optimizations do not conflict, the recent trend towards increasingly energy-proportional systems opens new trade-offs that makes the design space significantly more complex, as optimizations for these two criteria may conflict.

The energy analysis focuses on the energy consumption and response time of write operations using data deduplication in storage systems on two different generations of machines. This evaluation supports and quantifies the above intuition: the more power proportional a system, the greater the opportunities to trade among different resources, and the larger the gap between performance- and energy-centric optimizations.

Additionally, this chapter proposes an energy consumption model that highlights the same issues and, in spite of its simplicity, can be used to reason about the energy and performance break-even points when configuring a storage system.

The discussion below aims to clarify some of the limitations and the lessons of this preliminary study.

What is the impact of using other similarity detection mechanisms (e.g., content-based chunking) instead of fixed-size chunking?

As Section 2.2.4 presents, fixed-size chunking creates a new chunk for every certain amount of bytes. It has the advantage of being computationally cheaper than content-based chunking, since the chunking scheme does not need to analyze the data contents. It is, however, not robust for data insertions or deletions that are not a multiple of the chunk size. Content-based chunk boundaries (content-based chunking) [125], on the other hand, are more robust to data insertions and deletions since the chunk boundary is not bound to specific data patterns, and is not a specific size. The trade-off is the high computation cost of analyzing the data content versus the higher degree of data similarity detected [12].

After verifying that content-based chunking had a high cost and that the extra similarity detected would not pay off the overhead cost of chunking the data (i.e., the time cost of chunking the data would always be higher than

sending it over the network), the initial solution described in this chapter focused on fixed-size chunking. Al-Kiswany et al. [12] describes a similar conclusion.

What is the impact of hardware of accelerators on speeding up chunking operations and the proposed approaches?

Past work has proposed the use of GPUs to offload hash computations (for instance, my collaborators and I [53]) as well as some systems to use these offloaded hash computations to speed up data deduplication operations (e.g., StoreGPU [11, 76] and ShedderGPU [34]). These systems leverage the extra computational power of GPUs and make fixed-size chunking overhead almost negligible when compared to the I/O cost. As a consequence, in terms of writing time, enabling data deduplication is almost always the desired approach.

Additionally, these systems overcome the problem introduced by the computationally-intensive operations required for content-based chunking in the sense that, depending on the degree of similarity in the data, content-based chunking can actually speed up write operations for inline data deduplication – which, according to Al-Kiswany et al. [15], is not provided by current CPU-based implementations.

In this case, the decision space shifts; using content-based chunking can improve performance, depending on the similarity of the data. This makes the decision of using GPUs and content-based chunking similar to that of using CPUs and fixed-size chunking, as described in this chapter. Thus, the solution proposed in this study can serve as an initial step for this new strategy.

Can this solution be applied to data deduplication in other workload?

The target scenario of this chapter is inline data deduplication, using fixed-size chunking for checkpointing applications due to its potential data similarity. Other applications can offer similarity as high as checkpointing and, therefore, could use the proposed approach to optimize response

time or energy consumption. As examples, Jayaram et al. [99] present an empirical analysis of virtual machine repositories, and Park and Lilja [130] analyze datasets of backup applications; storage space savings can be as high as 60% for a generic archival workload [135], or 95% for a virtual machine image repository [109].

How difficult is it to predict similarity for different data deduplication parameters based on a history of write operations?

Data deduplication may offer several parameters to be configured that affect the degree of detected data similarity. Content-based chunking, for instance, is based on a sliding window to detect chunk boundaries and receives the window size, the minimum chunk size, and the amount of bits advanced at each step, among other parameters. As Tangwongsan et al. [153] show, estimating how the detectable degree of similarity changes as the parameters change is difficult, if not impossible. Instead, they propose a method to estimate an optimal expected chunk size, which can be obtained as a result of the parameters listed above, for the same data.

This lack of stability (and predictability) in the similarity of the data flow, makes it difficult to perform an on-the-fly adaptation for inline data deduplication. Moreover, it increases the cost of estimating similarity, since it requires more probing activations of similarity detection.

Does energy consumption impose extra challenges over time and storage as part of the optimization criteria?

Capturing the behavior of this new metric in the model requires the extension described in Section 5.4. This preliminary evaluation did reveal that considering energy consumption poses a challenge to monitoring. In contrast to what happens for measuring time and storage space, the machines are not commonly equipped with a means to monitor power consumption easily via software. Even when the machines have power meters, the granularity is coarse, with sampling rates of 1Hz or 2Hz at their best. Thus, estimating the energy consumed by storage system operations requires analyzing a batch of operations, and reduces the precision of

quantifying the consumption of a single operation. Moreover, accessing the measurements can be troublesome since it is common for them to require different user permissions, or a specialized API.

Due to the difficulties of monitoring energy, a predictor for energy consumption has one additional requirement: it should not depend on fine grain measurements from power meters during runtime. Ideally, the predictor should rely on a model that could be seeded only with benchmarks based on coarse grain measurements. Alternatively, the seeding procedure could use power information available on the technical data sheets for various hardware components.

Additionally, it is not possible to isolate the consumption of the storage system only, or any software system for that matter; this study's evaluation conservatively reports the energy consumed for the whole system. Isolating energy consumption of just a part of a system, as it is possible to perform for time measurements, remains a challenge for future research.

How should the results obtained change for new hardware?

Part of this discussion is related to the use of GPUs, or faster processors in general, as presented earlier in this section. This chapter presents an empirical analysis indicating that, for storage systems that use power-proportional computing components, the energy and throughput break-even points are further apart than less power-proportional systems. Additionally, the proposed energy model suggests that, as the trend toward more power-proportional systems continues, optimizations for energy efficiency and performance will likely conflict more often.

A direct consequence of this conflict is that storage system designers, administrators and users will have to decide, via conscious and informed decisions, about the metric to optimize for.

Chapter 6

Concluding Remarks

Clusters of computers, including parallel supercomputers, that are designed to provide a high performance computing platform for batch applications, typically provide a centralized persistent backend storage system. To avoid the potential bottleneck of accessing the platform's backend storage system, *intermediate storage systems* aggregate resources of the application's allocation to provide a shared, temporary storage space dedicated to the application's execution [14, 31, 60, 112].

Indeed, distributed storage systems have emerged to replace centralized storage solutions for large computing systems, as the distributed solution has appealing benefits over the centralized one, including lower cost, higher efficiency and performance, and incremental scalability. As a drawback, making decisions about resource provisioning, and storage system configuration are more costly in distributed systems. This happens because managing data across several nodes requires more complex coordination than in a centralized solution, and distributed storage techniques expose trade-offs that rarely exist in centralized solutions and are strongly dependent on the applications.

Instead of offering a "one size fits all" solution via fixed parameters for these decisions, some systems offer *versatile solutions*. These versatile systems target improved performance of the application by supporting the ability to 'morph' the storage system to best match the application's

demands. To this end, versatile storage systems significantly extend the flexibility to extend the storage system at deployment- or run-time. This flexibility, however, introduces a new problem: a much larger, and potentially dynamic, configuration space to be explored.

In this scenario, system provisioning, resource allocation, and configuration decisions for I/O-intensive batch applications are complex even for expert users. Users face choices at multiple levels: choosing the amount of resources, allocating these resources to individual sub-systems (e.g., the application layer, or the storage layer) as well as configuring each of these optimally (e.g., replication level, chunk size, caching policies in case of storage) – all having a large impact on the overall performance of the application.

Consequently, using a versatile storage system entails searching a complex multi-dimensional configuration space to determine the user's ideal cost vs. performance balance point, making manual configuration an undesirable task.

The research presented in this dissertation addresses these problems in the context of intermediate storage systems. In particular, this work targets workflow applications – which communicate via files and are the focus of this study – as well as checkpointing applications.

The traditional performance metrics that this research addresses as the optimization criteria are a primary concern for many systems: application turnaround time, allocation cost of resources, and response time and data throughput of storage system operations.

Additionally, this research addresses energy consumption as part of the optimization criteria. First, it presents an analysis of the energy consumption in target scenarios to reduce the gap in past research when it comes to modeling and understanding the trade-offs of optimization for energy consumption and other traditional performance metrics. Second, it extends the initial solution proposed, for the prediction of an application's performance according to traditional performance metrics, to take into account energy consumption as well.

The end goal of this research is to provide support for provisioning and configuration decisions for intermediate storage systems to deliver the success metrics (e.g., response time, storage footprint, energy consumption) close to user-defined optimization criteria. Specifically, this dissertation proposes performance prediction mechanisms that leverage the target application's characteristics to accelerate the exploration of the provisioning and configuration space. The mechanisms rely on monitoring information available at application level, not requiring changes to the storage systems, nor specialized monitoring systems. Additionally, these mechanisms can be used as building blocks for a solution that automates the configuration decisions.

The proposed solution meets the following requirements: First, they reduce human intervention – as they require minimal human intervention to enable or disable various optimization techniques and to choose their configuration parameters. Second, they produce a satisfactory configuration – as they enable configuration choices that bring the performance of the system close to the user's intention. Third, they have a low exploration cost – as the overhead of using the proposed solution are low when compared to the cost of running the application or I/O operations several times.

6.1 Contributions and Impact

This section presents a summary of the contributions that this dissertation provides, as well as a discussion of their impact.

6.1.1 Performance Prediction Mechanisms: Models and Seeding Procedures

"Essentially, all models are wrong, but some are useful." – G.E.P. Box (1976)

To provide performance predictors that satisfy that the requirements listed above and detailed in Section 1.2, this study focuses on the following questions.

How can one leverage the characteristics of I/O intensive workflow applications to build a prediction mechanism for traditional performance metrics (e.g., time-to-solution and network usage)?

Chapter 3 shows that the complexity of a prediction mechanism can be reduced by relying on some of the characteristics of workflow applications: relatively large files, almost distinct I/O and processing phases, many-reads-single-write, and specific data access patterns. Moreover, as the goal is to support configuration choice for a specific workload, achieving perfect accuracy - at the cost of extra complexity - is less critical, as long as the mechanism can support configuration decisions.

By reducing the complexity of the prediction mechanism, this work is able to provide a system identification procedure to provide the model's parameters (i.e., seed the model) that has two key features: (i) it relies on application-level operations (e.g., read and write) instead of detailed measurements of the execution path of these operations through the storage system, and (ii) it relies on a small deployment of the system, despite the goal of predicting the performance of a larger-scale deployment of the system. These two key features allow the mechanism to be simple, lightweight, and non-intrusive, since it does not require storage system or kernel changes to collect monitoring information, while still being effective at predicting the target performance metrics.

The impact of such an approach is that it enables the selection of a good configuration choice in a reasonable amount of time. Specifically, for a wide range of scenarios explored, the predictor is lightweight – being 200x to 2000x less resource-intensive than running the application itself, and accurate – given that 80% of the evaluated scenarios have predictions within 10% error and, in the worst case scenario, the prediction is still within 21%. This accuracy is similar to or higher than past work that predicted the behavior of distributed storage system or distributed applications. For example, Thereska et al. [156] target different workloads in the storage system domain while using a more detailed model and monitoring information [158], Herodotou et al. [91] focus on map-reduce jobs, and Kaviani et al. [100] provide a solution for support placement of software functions in a

cloud environment.

What are the Challenges and Efficacy of an Automated Solution for the Online Configuration of an Intermediate Storage System?

Chapter 5, "Automatically Enabling Data Deduplication", focuses on the experience of providing a simple performance predictor for an online configuration by exploring data deduplication in the context of repetitive write operations (e.g., checkpointing applications).

Chapter 5 presents a solution that relies on a repetitive pattern and on a prediction mechanism, similar to the solution for workflow applications in Chapter 3. This solution for data deduplication, however, is predicated on the key assumption that repetitive similar operations occur throughout the *application lifetime* of a checkpointing application [12]. This characteristic provides the opportunity to compare the effects of various configuration options in order to detect a "good enough" configuration during runtime. In this case, it uses a control loop that relies on a simple, yet effective, performance model.

The progress in this direction not only sheds light on the challenges related to automated online configuration, but is also directly applicable in practice since automating the choice of enabling or disabling deduplication allows the application's scientist/user to decouple her concerns about the checkpointing characteristics (e.g., checkpointing technology, frequency) in order to configure the storage system.

Chapter 5 also points out that, depending on the workload, having a low computing overhead can be challenging for an automated solution because (i) similarity detection is an intrusive storage technique, and (ii) the similarity may vary frequently, requiring similarity detection to be activated more often as well. Section 6.2.4 discusses this case in more detail.

6.1.2 Energy Trade-offs Assessment and Extension of the Prediction Mechanisms

The energy study presented in this dissertation addresses two main questions:

- *Is energy consumption subjected to different trade-offs than response time, or are optimizing for energy consumption and for response time coupled goals?*
- *Which extensions does the performance predictor need in order to capture energy consumption behavior, in addition to traditional performance metrics?*

Chapter 3 shows that workflow applications may have different trade-offs between optimizing for time-to-solution and energy consumption, depending on the characteristics of the application. To incorporate energy consumption as a possible part of the optimization criteria, Chapter 3 presents an extension of the performance predictor to estimate the energy consumption of I/O-intensive workflow applications based on the idea of associating power profiles, while satisfying the same requirements that the performance predictor should meet. Yang et al. [171] present and evaluate this extension in detail, and demonstrate that the proposed prediction mechanism can estimate energy consumption with accuracy similar to that of traditional performance metrics.

Chapter 5 explores how online monitoring can enable online adaptation, and the initial experience of applying a power profile-based approach to estimate the energy consumption of a data deduplication technique. Chapter 5 demonstrates the effectiveness of the prediction mechanism extension, in terms of accuracy, for energy consumption. Moreover, it demonstrates that, in the context of data deduplication, hardware platforms with different characteristics in terms of energy-proportionality present different trade-offs between optimizing for time-to-solution and energy consumption.

6.1.3 Supporting Development

The prediction mechanism proposed in this dissertation can also be used to support distributed system development. To assess this case, this study

demonstrates the potential gains of using the prediction mechanism, beyond the original design goal, by applying the performance predictor to better understand and debug a distributed storage system that the NetSysLab group has been developing. For this use-case, this dissertation sheds light on the following questions: *Which, if any, are the potential benefits that the proposed performance predictor brings to the software development process of a storage system?* and *What are the limitations and challenges of using a performance predictor as part of the development process?*

The developers compare this mechanism's predictions to the actual performance and decide whether they are close enough to their goal. In the case that they are not, they proceed to debug the system, and follow this process iteratively as part of the development. The predictor was also used to evaluate the potential gains of new system features.

This study provides evidence that the use of performance predictors during software development helps developers deal with this non-functional requirement of meeting a specific efficiency, similar to how automated tests verify functional requirements. Using the predictor helps the developers to set goals and improve the system's performance. Specifically, this approach improved confidence in the implementation of MosaStore for some scenarios, and pointed out situations that needed further improvement. The latter made possible an improvement of the system's performance by up to 30% and a decrease of 10x in the response time variance for some of the target benchmarks.

Using the predictor as part of the software development process presents some of the challenges inherent to any performance predictor: difficulties in properly seeding the model and mimicking the benchmarks used (i.e., the workload) to evaluate the performance of its implementation. It also entails the difficulties inherent to any additional step in the development process: providing a tool easy to be used that brings perceptive value to the developers.

An important challenge that this approach brings is defining what is "close enough" to the performance goal. Although the predictor can set an objective ideal target to stop debugging, the developers are still in charge

of defining how much variation in the target value is tolerated, since some overhead brought by the distributed system is expected.

6.1.4 Storage System and Prediction Tool Prototypes

Chapters 3, 4, and 5 present evaluations that rely on the prototype implementations of the MosaStore storage system and the proposed prediction mechanisms. These prototypes are available in the code repositories for MosaStore[1] and the predictors[2]. These repositories also contain scripts and configuration data used to execute the experiments for the evaluation described in this dissertation, as well as the scripts and a summary of the results of the experiments[3] used in the data analysis of this study.

In addition to the research study's results and impact presented above, as is typical in Computer Systems research, part of a research project's contribution is a working software system. In this case, the MosaStore storage system is the working prototype based on, and/or serving as a research platform for the evaluation of, a number of research ideas [11–15, 17, 75–77, 115, 159, 171] from several institutions, including this dissertation [52, 54, 55, 57, 58, 60].

The implementation of these prototypes follow a development process that includes automated unit and acceptance tests, as well as a code review process[4]. Having a working prototype available that relies on structured testing and code reviewing processes, and is used by various projects also increases confidence in the evidence that this research provides.

In addition to improving MosaStore's usability, by providing support for its configuration, it is worthwhile to point out that MosaStore's performance and stability improved as a direct consequence of the research described in this dissertation (see Chapter 4).

[1] https://subversion.assembla.com/svn/MosaStore/.

[2] https://subversion.assembla.com/svn/AutoConf/.

[3] Complete logs are available at NetSysLab cluster, since they are too large to be upload to these repositories.

[4] Code reviews are available at https://codereview-netsyslab.appspot.com/

6.2 Limitations and Future Work

In addition to the results described in Section 6.1, the research presented in this dissertation identifies some limitations of the proposed solutions which are summarized in this section. Additionally, this section describes possible extensions to this research, including alternatives to overcome the identified limitations.

6.2.1 Complete Automation of User Decisions

The proposed prediction mechanisms accelerate the process of evaluating different decisions for the provision and configuration of storage systems. One natural next step is to use the prediction mechanisms to provide a completely automated solution that receives a utility function capturing the user's intention and produces a configuration that maximizes the utility.

In the past, several approaches have been proposed to search for parameters that maximize a given function. For example, in the context of a computer system's configuration, the Recursive Random Search (RRS) algorithm has been applied for large-scale network parameter optimization [173] and for map-reduce jobs [90, 92]. The RRS algorithm uses a random sampling, with adjusted sample space, to tolerate random noise and irrelevant or trivial parameters, which is relevant to the domain of this research. Alternatively, approaches based on genetic algorithms could also provide complete automation by using the proposed prediction mechanisms. For example, Strunk et al. [152] propose a set of utility functions for storage systems and a tool based on genetic algorithms to find the input that maximizes these functions. Behzad et al. [27] also use genetic algorithms to optimize the runtime of parallel applications by sampling the configuration space via actual application runs. These, or similar approaches, can use the solution described in this dissertation as building blocks for a tool that automates the user's decisions about provisioning and configuration of intermediate storage systems. For example, a genetic algorithm could query a performance predictor using different values for the configuration parameters instead of running the actual application.

Additionally, note that the user may not be interested in a single configuration that optimizes a utility function, or she may not have the utility function. In this case, the user would be interested in seeing a sample of possible configurations with different trade-offs in order to pick one. An example of this is the case of performance vs. allocation cost (Sections 3.3.3 and 3.3.4), where the user is interested in what she believes to be a cost/performance balance point. For these cases, a complete automation solution could use a Pareto sampling approach to provide the user with a subset of most desired configuration choices for a given optimization criteria (i.e., Pareto frontier) [16].

6.2.2 Prediction Mechanism Support for Heterogeneous Environments

Some cluster and cloud environments are not completely homogeneous across nodes. This heterogeneity breaks one of the assumptions held by the seeding procedure, described in Chapter 3 – that is, relying on a minimal deployment to be simple, and low cost.

Chapter 3 argues that the storage system simulation is able to support the heterogeneity and, in fact, presents a heterogeneous scenario for the evaluation of the reduce benchmark. In this case, the seeding procedure should be performed on the different types of machines and provided to the prediction mechanism as part of the setup to be evaluated. For real applications, however, the computing part of the workflow tasks are hard to incorporate in the prediction mechanism in the way it has been proposed in this study. The main problem is that it would require re-executions of the tasks for different machine types, which may take long – breaking the prediction mechanism requirements of being low cost.

Two alternatives could overcome this problem. First, a *task sampling* approach could enable the predictor to execute just part of the tasks, per stage and to then infer the behavior of the other tasks in the same stage [91, 140]. In this case, at least one complete execution of the application should be executed in order to guarantee that the input files for all tasks will be available for later sampling in heterogeneous machines, as an incomplete

execution would prevent the execution of the next stage. In fact, for cases where partial execution of a stage can still produce useful results, this approach can already be applied to a homogeneous environment [91]. Based on Montage and BLAST cases in this study (Section 3.3), a sample as small as 20% of the tasks in a stage can be effective for inferring the behavior of other tasks, and it would only lead to an additional 2-6% prediction error.

Second, a *relative performance* approach [119, 120] proposes the use of basic operations to benchmark different types of machines, and then describe them in terms of relative performance among the different types. In this case, one complete execution of the task is still needed, but then the predictor could infer the performance of each task in a different type of machine based on the relative performance description. A hybrid approach that combines task sampling and relative performance may also be effective.

6.2.3 Support for Virtual Machines and More Complex Network and Storage Device Interactions

Using dedicated nodes (i.e., a space-shared platform) is the typical set-up for clusters and supercomputers, and they are also available in cloud platforms. Virtual Machines, however, have become widely available as an alternative for these platforms. One of the consequences of the use of virtual machines is the performance interference of the physical machine. Although performance insulation are available and effective for some of the machine components (e.g., CPU), it is difficult to provide insulation for storage and network devices [83, 161].

This lack of perfect performance insulation can lead to larger errors by the performance prediction mechanism and, hence, ineffective support for storage configuration. In this case, the following extension could be useful: First, an evaluation would provide evidence whether the performance interference affect the storage to a point that prevents the performance prediction mechanisms from supporting the user making her decisions about the storage system. Second, depending on the results of the evaluation, the prediction mechanism can be extended to incorporate more detailed model of the platform components (e.g., the use of NS2 [8] to capture

more complex network interactions and DiskSim [1]). Alternatively, this problem can be addressed though the evolution of performance insulation mechanisms and Quality-of-Service (QoS) agreements [127].

6.2.4 Support for GPU and Content-Based Chunking for Data Deduplication

The work on an automated solution for data deduplication, presented in Chapter 5, was an initial study that had the goal of assessing how online monitoring can enable online adaptation and the initial experience of applying a power profile-based approach to estimate energy consumption of a data deduplication technique.

As discussed in Section 5.6, two extensions can enrich the automated solution proposed for data deduplication. First, the use of GPUs to offload hash computations can speed up data deduplication operations. Based on the small variance of the results per chunk, as evaluated in the StoreGPU project [11, 13, 15, 76, 77], the approach proposed in Chapter 5 should be effective in predicting response time. An extended evaluation is needed to assess the impact of this approach on predicting energy consumption.

The second extension is related to the configuration of several parameters that affect can the degree of detected data similarity. Tangwongsan et al. [153] propose an approach that can be an initial step towards this direction; their method estimates an "optimal" expected chunk size, which can be obtained as a result of the parameters to be configured over the same data. Tangwongsan et al. [153], however, also discuss that estimating how the detectable degree of similarity changes is difficult, even for the same data (e.g., same virtual machine image). Such estimation should be harder for cases over different datasets (e.g., different checkpoint images), as the target of this dissertation.

6.2.5 Study on the Use of Performance Predictors to Support Development

The use-case discussed in this dissertation should be extended to evaluate a larger set of distributed systems and developers to verify the generalization of the observations presented in Chapter 4, and to understand what information would provide more benefits to the developers (e.g., profiling aid).

In addition to understanding what extra information can aid the developers in the profiling and debugging process, it is necessary to assess when and how to use a predictor in the development process. Specifically, the community would benefit from understanding the trade-offs related to its use and the comparison with simpler alternatives such as back-of-the-envelope calculations, and the trade-offs between the extra effort of pursing a given performance level provided by a predictor versus the benefits of obtaining such performance.

Bibliography

[1] DiskSim. http://www.pdl.cmu.edu/DiskSim/. → pages 36, 94, 100, 167

[2] Fuse: Filesystem in userspace. http://fuse.sourceforge.net/. → pages 17, 109

[3] Stress. http://people.seas.harvard.edu/~apw/stress/. → pages 85

[4] Watts up? pro product details. https://www.wattsupmeters.com/. → pages 143

[5] A conversation with Jim Gray. *Queue*, 1(4):8–17, June 2003. ISSN 1542-7730. doi:10.1145/864056.864078. URL http://doi.acm.org/10.1145/864056.864078. → pages 34

[6] Overview of DOCK6. http://dock.compbio.ucsf.edu/Overview_of_DOCK/index.htm, June 2011. → pages 26

[7] *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012, April 16-20, 2012, Asilomar Conference Grounds, Pacific Grove, CA, USA*, 2012. IEEE. → pages 172, 183

[8] The network simulator ns2. http://www.isi.edu/nsnam/ns/, 2012. → pages 36, 37, 46, 166

[9] pNFS. http://www.pnfs.com, 2012. → pages 22

[10] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the Conference on File and Storage Technologies*, December 2005. → pages 1, 2, 3, 16, 18, 22, 23, 37, 49, 57, 109

[11] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-997-5. doi:10.1145/1383422.1383443. URL http://doi.acm.org/10.1145/1383422.1383443. → pages 15, 21, 25, 29, 126, 147, 150, 153, 163, 167

[12] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. In *International Conference on Distributed Computing Systems*, ICDCS, pages 613–624, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3172-4. → pages 2, 3, 25, 29, 33, 127, 129, 135, 136, 137, 141, 152, 153, 160

[13] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, and M. Ripeanu. On GPU's viability as a middleware accelerator. *Cluster Computing*, 12(2): 123–140, June 2009. ISSN 1386-7857. doi:10.1007/s10586-009-0076-0. URL http://dx.doi.org/10.1007/s10586-009-0076-0. → pages 167

[14] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. The case for a versatile storage system. *ACM SIGOPS Operating Systems Review*, 44:10–14, March 2010. ISSN 0163-5980. → pages 1, 2, 3, 6, 11, 14, 15, 16, 21, 23, 24, 25, 40, 44, 49, 69, 108, 156

[15] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu. GPUs as storage system accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1556–1566, 2013. ISSN 1045-9219. doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.239. → pages 153, 163, 167

[16] S. Al-Kiswany, H. Hacıgümüş, Z. Liu, and J. Sankaranarayanan. Cost exploration of data sharings in the cloud. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 601–612, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1597-5. doi:10.1145/2452376.2452447. URL http://doi.acm.org/10.1145/2452376.2452447. → pages 165

[17] S. Al-Kiswany, E. Vairavanathan, L. B. Costa, H. Yang, and M. Ripeanu. The case for cross-layer optimizations in storage: A workflow-optimized storage system. *CoRR*, abs/1301.6195, 2013. → pages 15, 16, 18, 20, 21, 28, 30, 32, 163

[18] S. Al-Kiswany, L. B. Costa, E. Vairavanathan, H. Yang, and M. Ripeanu. A software defined storage for scientific workflow applications. Under Review, September 2014. → pages vii, 23, 102

[19] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990. ISSN 0022-2836. → pages 4, 5, 26, 43, 60, 69

[20] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions Computer Systems*, 19(4):483–518, November 2001. ISSN 0734-2071. doi:10.1145/502912.502915. URL http://doi.acm.org/10.1145/502912.502915. → pages 3, 7, 35, 40

[21] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *In Proceedings of the Conference on File and Storage Technologies*, pages 175–188. USENIX Association, 2002. → pages 40, 93, 95

[22] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions Computer Systems*, 23(4):337–374, November 2005. ISSN 0734-2071. → pages 3, 7, 40, 93, 95

[23] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems (TOCS)*, 14(1):41–79, 1996. → pages 2

[24] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004. → pages 106, 107, 120, 121

[25] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007. ISSN 0018-9162. → pages 25, 142, 150

[26] S. Basu, L. B. Costa, F. Brasileiro, S. Banerjee, P. Sharma, and S.-J. Lee. Nodewiz: Fault-tolerant grid information service. *Journal of Peer-to-Peer Networking and Applications*, 2(4):348–366, 2009. ISSN 1936-6442. doi:10.1007/s12083-009-0030-1. URL http://dx.doi.org/10.1007/s12083-009-0030-1. → pages iv, 195

[27] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 68:1–68:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi:10.1145/2503210.2503278. URL http://doi.acm.org/10.1145/2503210.2503278. → pages 1, 3, 7, 8, 95, 164

[28] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir. Improving parallel i/o autotuning with performance modeling. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 253–256, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi:10.1145/2600212.2600708. URL http://doi.acm.org/10.1145/2600212.2600708. → pages 8

[29] C. L. Belady. In the Data Center, Power and Cooling Costs more than the IT Equipment it Supports. February 2007. http://www.electronics-cooling.com/2007/02/in-the-data-center-power-and-cooling-costs-more-than-the-it-equipment-it-supports/. → pages 84

[30] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 27–38, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251175.1251202. → pages 22, 24

[31] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring. Jitter-free co-processing on a prototype exascale storage stack. In *MSST* DBL [7], pages 1–5. → pages 3, 24, 156

[32] G. B. Berriman, J. C. Good, D. Curkendall, J. Jacob, D. S. Katz, T. A. Prince, and R. Williams. Montage: An on-demand image mosaic service for the nvo. *Astronomical Data Analysis Software and Systems XII ASP Conference Series*, 295:343, 2003. → pages 26, 43

[33] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, 2008. → pages 26, 27, 43, 62, 63

[34] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-accelerated incremental storage and computation. In *10th USENIX Conference on File and Storage Tech. (FAST '12)*, FAST, page 14, February 2012. → pages 153

[35] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855533.1855545. → pages 34, 39

[36] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-585-7. doi:10.1145/1555271.1555273. URL http://doi.acm.org/10.1145/1555271.1555273. → pages 34, 39

[37] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1151–1158, May 2011. doi:10.1109/IPDPS.2011.281. → pages 26

[38] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855745. → pages 26

[39] D. Bradley, O. Gutsche, K. Hahn, B. Holzman, S. Padhi, H. Pi, D. Spiga, I. Sfiligoi, E. Vaandering, F. Würthwein, et al. Use of glide-ins in cms for production and analysis. *Journal of Physics: Conference Series*, 219 (7):072013, 2010. → pages 24

[40] F. Brüseke, G. Engels, and S. Becker. Decision support via automated metric comparison for the palladio-based performance blame analysis. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 77–88, New York, NY, USA, 2013.

ACM. ISBN 978-1-4503-1636-1. doi:10.1145/2479871.2479886. URL http://doi.acm.org/10.1145/2479871.2479886. → pages 122

[41] F. Cappello, E. Caron, M. Daydé, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 8 pp.+, 2005. → pages 61, 87

[42] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3114-4. → pages 46

[43] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 103–116, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi:10.1145/502034.502045. URL http://doi.acm.org/10.1145/502034.502045. → pages 25, 34, 39

[44] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 367–378, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5. doi:10.1145/276304.276337. URL http://doi.acm.org/10.1145/276304.276337. → pages 37

[45] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325856. → pages 37

[46] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3. URL http://dl.acm.org/citation.cfm?id=645926.671696. → pages

[47] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1265–1265. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164259. → pages 37

[48] Y. Chen, A. Ganapathi, and R. H. Katz. To compress or not to compress - compute vs. IO tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, Green Networking '10, pages 23–28, 2010. → pages 141, 151

[49] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002. → pages 4, 18, 44

[50] W. Cirne, F. Brasileiro, N. Andrade, L. B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006. → pages 195

[51] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Proceedings of the 2nd international workshop on Software and performance*, pages 58–70. ACM, 2000. → pages 122

[52] L. B. Costa and M. Ripeanu. Towards Automating the Configuration of a Distributed Storage System. In *Proceedings of the 11th ACM/IEEE International Conference on Grid Computing*, Grid'2010, October 2010. → pages vi, 1, 2, 7, 15, 21, 25, 33, 69, 125, 163

[53] L. B. Costa, S. Al-Kiswany, and M. Ripeanu. GPU Support for Batch Oriented Workloads. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, pages 231–238, December 2009. doi:10.1109/PCCC.2009.5403809. → pages iv, 25, 153, 195

[54] L. B. Costa, S. Al-Kiswany, R. V. Lopes, and M. Ripeanu. Assessing data deduplication trade-offs from an energy and performance perspective. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, pages 1–6. IEEE, 2011. doi:10.1109/IGCC.2011.6008567. → pages vi, 7, 15, 18, 21, 25, 33, 84, 125, 127, 163

[55] L. B. Costa, S. Al-Kiswany, A. Barros, H. Yang, and M. Ripeanu. Predicting intermediate storage performance for workflow applications. In *Proceedings of the 8th Parallel Data Storage Workshop*,

PDSW '13, pages 33–38, New York, NY, USA, November 2013. ACM. ISBN 978-1-4503-2505-9. doi:10.1145/2538542.2538560. URL http://doi.acm.org/10.1145/2538542.2538560. → pages v, 42, 163

[56] L. B. Costa, J. Brunet, L. Hattori, and M. Ripeanu. Experience on applying performance prediction during development: a distributed storage system tale. Technical report, UBC/ECE/NetSysLab, September 2013. http://www.ece.ubc.ca/~lauroc/tr/tech2.pdf. → pages 12

[57] L. B. Costa, S. Al-Kiswany, H. Yang, and M. Ripeanu. Supporting storage configuration for I/O intensive workflows. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS'14, pages 191–200. ACM, June 2014. ISBN 978-1-4503-2642-1/14/06. doi:10.1145/2597652.2597679. URL http://dx.doi.org/10.1145/2597652.2597679. → pages iv, 30, 43, 78, 163

[58] L. B. Costa, S. Al-Kiswany, H. Yang, and M. Ripeanu. Supporting Storage Configuration and Provisioning for I/O Intensive Workflows. Under Review, 2014. → pages 43, 163

[59] L. B. Costa, J. Brunet, L. Hattori, and M. Ripeanu. Experience on applying performance prediction during development: a distributed storage system tale. In *Proceedings of 2nd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, SE-HPCCSE '14, pages 13–19, Piscataway, NJ, USA, November 2014. IEEE Press. ISBN 978-1-4799-7035-3. doi:10.1109/SE-HPCCSE.2014.6. URL http://dx.doi.org/10.1109/SE-HPCCSE.2014.6. Previously "International Workshop on Software Engineering for Computational Science and Engineering" and "International Workshop on Software Engineering for High Performance Computing Applications". → pages v, 105

[60] L. B. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, and S. Al-Kiswany. The case for workflow-aware storage: An opportunity study. *Journal of Grid Computing*, pages 1–19, 2014. ISSN 1570-7873. doi:10.1007/s10723-014-9307-6. URL http://dx.doi.org/10.1007/s10723-014-9307-6. Published online in July 2014. → pages vi, vii, 3, 12, 15, 16, 18, 21, 23, 27, 28, 55, 57, 156, 163

[61] A. Crume, C. Maltzahn, L. Ward, T. Kroeger, M. Curry, and R. Oldfield. Fourier-assisted machine learning of hard disk drive access time

models. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW'13, pages 45–51, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2505-9. doi:10.1145/2538542.2538561. URL http://doi.acm.org/10.1145/2538542.2538561. → pages 36, 100

[62] D. P. Da Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proceedings of the 9th International European Conference on Parallel Processing*, Euro-Par'03, pages 169–180. Springer, 2003. → pages 57, 100

[63] B. Dageville and M. Zait. Sql memory management in oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 962–973. VLDB Endowment, 2002. URL http://dl.acm.org/citation.cfm?id=1287369.1287454. → pages 34, 39

[64] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, November 2008. doi:10.1109/SC.2008.5222004. → pages 34, 38

[65] A. Davies and A. Orsaria. Scale out with glusterfs. *Linux Journal*, 2013 (235), Nov. 2013. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2555789.2555790. → pages 102

[66] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI '04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264. → pages 100

[67] E. Deelman, G. Singh, M. hui Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005. → pages 26, 43

[68] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies*

and Systems, USITS'03, pages 5–19, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251460.1251465. → pages 34, 39

[69] D. Economou, S. Rivoire, and C. Kozyrakis. Full-system power analysis and modeling for server environments. In *Proceedings of Workshop on Modeling Benchmarking and Simulation*, MOBS '06, 2006. → pages 96

[70] M. A. Erazo, T. Li, J. Liu, and S. Eidenbenz. Toward comprehensive and accurate simulation performance prediction of parallel file systems. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Network*, DSN'12, pages 1–12. IEEE, 2012. → pages 36, 94

[71] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi:10.1145/1250662.1250665. URL http://doi.acm.org/10.1145/1250662.1250665. → pages 25

[72] X. Feng, R. Ge, and K. W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 34–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2312-9. doi:10.1109/IPDPS.2005.346. URL http://dx.doi.org/10.1109/IPDPS.2005.346. → pages 96

[73] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*, chapter 5: Behavioral Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2. → pages 19

[74] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders. Designing dependable storage solutions for shared application environments. *IEEE Transactions on Dependable and Secure Computing*, 7(4):366–380, October 2010. ISSN 1545-5971. doi:10.1109/TDSC.2008.38. URL http://dx.doi.org/10.1109/TDSC.2008.38. → pages 38

[75] A. Gharaibeh and M. Ripeanu. Exploring data reliability tradeoffs in replicated storage systems. In *Proceedings of the 18th ACM International*

Symposium on High Performance Distributed Computing, HPDC '09, pages 217–226, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-587-1. doi:10.1145/1551609.1551643. URL http://doi.acm.org/10.1145/1551609.1551643. → pages 163

[76] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 167–178, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi:10.1145/1851476.1851497. → pages 15, 21, 25, 126, 147, 150, 153, 167

[77] A. Gharaibeh, S. Al-Kiswany, and M. Ripeanu. ThriftStore: Finessing reliability trade-offs in replicated storage systems. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):910–923, June 2011. ISSN 1045-9219. doi:10.1109/TPDS.2010.157. → pages 163, 167

[78] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi:10.1145/2370816.2370866. URL http://doi.acm.org/10.1145/2370816.2370866. → pages iv, 194

[79] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 851–862, May 2013. doi:10.1109/IPDPS.2013.37. → pages 194

[80] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu. The energy case for graph processing on hybrid CPU and GPU systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA^3 '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2503-5. doi:10.1145/2535753.2535755. URL http://doi.acm.org/10.1145/2535753.2535755. → pages 194

[81] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu. Efficient large-scale graph processing on hybrid cpu and gpu systems. *Journal Submission*, 2014. Under Review. → pages iv, 194

[82] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, New York, NY, USA, 2003. ACM Press. ISBN 1581137575. → pages 2, 16, 109

[83] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924974. → pages 166

[84] S. Gurumurthi, A. Sivasubramaniam, M. Irwin, N. Vijaykrishnan, and M. Kandemir. Using complete machine simulation for software power estimation: the softwatt approach. pages 141–150, Feb. 2002. → pages 96

[85] I. F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux Journal*, 2000(80es), Nov. 2000. ISSN 1075-3583. → pages 2, 16, 22, 49, 109

[86] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006. URL http://stacks.iop.org/1742-6596/46/i=1/a=067. → pages 29

[87] S. Harizopoulos, M. A. Shah, J. Meza, and P. Ranganathan. Energy efficiency: The new holy grail of data management systems research. In *CIDR*, 2009. → pages 150, 151

[88] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *ACM Transactions Computer Systems*, 13(3):274–310, 1995. → pages 16

[89] C. Heger, J. Happe, and R. Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the ACM/SPEC international conference on International conference on performance engineering*, pages 27–38. ACM, 2013. → pages 123

[90] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011. → pages 164

[91] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings*

of the 2nd ACM Symposium on Cloud Computing, SoCC'11, pages 18:1–18:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi:10.1145/2038916.2038934. URL http://doi.acm.org/10.1145/2038916.2038934. → pages 7, 44, 95, 159, 165, 166

[92] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2011. URL http://research.microsoft.com/apps/pubs/default.aspx?id=168754. → pages 95, 164

[93] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988. → pages 2

[94] H. Huang and A. Grimshaw. Automated performance control in a virtual distributed storage system. In *Proceeding of the 9th IEEE/ACM International Conference on Grid Computing*, Grid '08, pages 242–249, Sept 2008. doi:10.1109/GRID.2008.4662805. → pages 39

[95] J. V. Huber, Jr., A. A. Chien, C. L. Elford, D. S. Blumenthal, and D. A. Reed. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 385–394, New York, NY, USA, 1995. ACM. ISBN 0-89791-728-6. doi:10.1145/224538.224638. URL http://doi.acm.org/10.1145/224538.224638. → pages 2, 3, 6, 18, 21

[96] D. Ibtesham, D. DeBonis, K. B. Ferreira, and D. Arnold. Coarse-grained energy modeling of rollback/recovery mechanisms. In proceedings the 4th Fault Tolerance for HPC at eXtreme Scale (FTXS) 2014, held in association with The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014), 2014. → pages 84

[97] S. Ihm, K. Park, and V. S. Pai. Wide-area network acceleration for the developing world. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC '10, Berkeley, CA, USA, 2010. USENIX Association. → pages 150

[98] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Interscience. John Wiley & Sons, Inc., New York, NY, first edition, 1991. → pages 61, 114, 136

[99] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An empirical analysis of similarity in virtual machine images. In *Proceedings of the Middleware 2011 Industry Track Workshop*, Middleware '11, pages 6:1–6:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1074-1. doi:10.1145/2090181.2090187. URL http://doi.acm.org/10.1145/2090181.2090187. → pages 154

[100] N. Kaviani, E. Wohlstadter, and R. Lea. Manticore: A framework for partitioning software services for hybrid cloud. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, CloudCom 2012*, pages 333–340. IEEE, December 2012. ISBN 978-1-4673-4511-8. → pages 44, 159

[101] K. Keeton and A. Merchant. A framework for evaluating storage system dependability. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '04, pages 877–886, June 2004. doi:10.1109/DSN.2004.1311958. → pages 38

[102] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: Restoring data after disasters. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, pages 235–248, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. doi:10.1145/1217935.1217958. URL http://doi.acm.org/10.1145/1217935.1217958. → pages 38

[103] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '10, 2010. → pages 150

[104] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *International Systems and Storage Conference*, SYSTOR, pages 4:1–4:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-623-6. → pages 141, 151

[105] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 239–252, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855511.1855529. → pages 33

[106] A. C. Laity, N. Anagnostou, G. B. Berriman, J. C. Good, J. C. Jacob, D. S. Katz, and T. Prince. Montage: An Astronomical Image Mosaic

Service for the NVO. In P. Shopbell, M. Britton, and R. Ebert, editors, *Astronomical Data Analysis Software and Systems XIV*, volume 347 of *Astronomical Society of the Pacific Conference Series*, page 34, December 2005. → pages 60, 74

[107] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6): 35–39, 2003. → pages 122

[108] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14, pages 6:1–6:15, New York, NY, USA, November 2014. ACM. ISBN 978-1-4503-3252-1. doi:10.1145/2670979.2670985. URL http://doi.acm.org/10.1145/2670979.2670985. → pages 24

[109] A. Liguori and E. V. Hensbergen. Experiences with content addressable storage and virtual disks. In M. Ben-Yehuda, A. L. Cox, and S. Rixner, editors, *Workshop on I/O Virtualization*. USENIX Association, 2008. → pages 126, 154

[110] M. Liu, Y. Jin, J. Zhai, Y. Zhai, Q. Shi, X. Ma, and W. Chen. ACIC: Automatic Cloud I/O Configurator for HPC Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 38:1–38:12, New York, NY, USA, November 2013. ACM. ISBN 978-1-4503-2378-9. doi:10.1145/2503210.2503216. URL http://doi.acm.org/10.1145/2503210.2503216. → pages 95

[111] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn. Modeling a leadership scale storage system. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Vol. Part I*, PPAM'11, pages 10–19, 2012. ISBN 978-3-642-31463-6. URL http://dx.doi.org/10.1007/978-3-642-31464-3_2. → pages 94

[112] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Proceedings IEEE 28th Symposium on Mass Storage Systems and Technologies* DBL [7], pages 1–12. → pages 3, 24, 94, 126, 156

[113] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible I/O and integration for scientific codes through the adaptable I/O system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-156-9. doi:10.1145/1383529.1383533. URL http://doi.acm.org/10.1145/1383529.1383533. → pages 22

[114] L. Ma, C. Zhen, B. Zhao, J. Ma, G. Wang, and X. Liu. Towards fast de-duplication using low energy coprocessor. In *International Conference on Networking, Architecture, and Storage*, pages 395–402, Los Alamitos, CA, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4134-1. → pages 151

[115] K. Maheshwari, J. M. Wozniak, H. Yang, D. S. Katz, M. Ripeanu, V. Zavala, and M. Wilde. Evaluating storage systems for scientific data in the cloud. In *Proceedings of the 5th ACM Workshop on Scientific Cloud Computing*, ScienceCloud '14, pages 33–40, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2911-8. doi:10.1145/2608029.2608034. URL http://doi.acm.org/10.1145/2608029.2608034. → pages 15, 21, 163

[116] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0130906735. → pages 121

[117] A. Merchant and P. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, 45(3):367–373, March 1996. ISSN 0018-9340. doi:10.1109/12.485575. → pages 35

[118] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 44–51, May 2004. doi:10.1109/ICAC.2004.1301346. → pages 36

[119] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 37–48, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-639-4. doi:10.1145/1254882.1254887. URL http://doi.acm.org/10.1145/1254882.1254887. → pages 166

[120] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger. Relative fitness modeling. *Communications of ACM*, 52(4): 91–96, Apr. 2009. ISSN 0001-0782. doi:10.1145/1498765.1498789. URL http://doi.acm.org/10.1145/1498765.1498789. → pages 36, 166

[121] R. Mian, P. Martin, and J. L. Vazquez-Poletti. Provisioning data analytic workloads in a cloud. *Journal of Future Generation Computer Systems*, 29(6):1452 – 1458, 2013. ISSN 0167-739X. URL http://www.sciencedirect.com/science/article/pii/S0167739X12000209. → pages 7, 39

[122] E. Molina-Estolano, C. Maltzahn, J. Bent, and S. Brandt. Building a parallel file system simulator. In *Journal of Physics: Conference Series*, volume 180, page 012050. IOP Publishing, 2009. → pages 1, 36, 93

[123] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, Sep 2009. → pages 36, 37, 46

[124] G. Moont. 3d-dock suite. http://www.sbg.bio.ic.ac.uk/docking/, 2012. → pages 26, 43

[125] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. → pages 29, 126, 150, 152

[126] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting dbms. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '05, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2458-3. doi:10.1109/MASCOT.2005.21. URL http://dx.doi.org/10.1109/MASCOT.2005.21. → pages 37

[127] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi:10.1145/1755913.1755938. URL http://doi.acm.org/10.1145/1755913.1755938. → pages 167

[128] NLANR/DAST. Iperf. http://iperf.sourceforge.net/. → pages 111

[129] S. Pakin and M. Lang. Energy Modeling of Supercomputers and Large-Scale Scientific Applications. In *Proceedings for the 4th International Green Computing Conference*, IGCC '13, June 2013. → pages 96

[130] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC, pages 1–10, December 2010. doi: 10.1109/IISWC.2010.5650369. → pages 154

[131] T. E. Pereira, L. Sampaio, and F. V. Brasileiro. On the accuracy of trace replay methods for file system evaluation. In *Proceedings of the IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '13, pages 380–383. IEEE, 2013. → pages 48

[132] M. Petre. Uml in practice. In *35th International Conference on Software Engineering (ICSE 2013)*, May 2013. → pages 122

[133] R. Prabhakar, E. Kruus, G. Lu, and C. Ungureanu. Eeffsim: A discrete event simulator for energy efficiency in large-scale storage systems. In *Energy Aware Computing (ICEAC), 2011 International Conference on*, pages 1–6. IEEE, 2011. → pages 96

[134] D. Qin, A. D. Brown, and A. Goel. Reliable writeback for client-side flash caches. In *Proceedings of USENIX Annual Technical Conference*, ATC '14, pages 451–462, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/qin. → pages 2

[135] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association. → pages 29, 135, 141, 150, 154

[136] I. Raicu, I. T. Foster, and Y. Zhao. Special section on many-task computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6): 897–898, 2011. ISSN 1045-9219. doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.138. → pages 26

[137] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-driven server migration for internet data centers. In *Proceedings of the 10th IEEE International Workshop on Quality of Service*, pages 3–12. IEEE, 2002. → pages 34, 39

[138] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *ACM International Conference on Virtual execution environments*, VEE, pages 111–120, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. → pages 126, 150

[139] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the USENIX 2008 Annual Technical Conference*, USENIX ATC '08, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association. → pages 29, 135, 141, 150

[140] C. A. D. Rose, T. Ferreto, R. N. Calheiros, W. Cirne, L. B. Costa, and D. Fireman. Allocation strategies for utilization of space-shared resources in bag-of-tasks grids. *Future Generation Computer Systems*, 24(5):331 – 341, 2008. ISSN 0167-739X. doi:http://dx.doi.org/10.1016/j.future.2007.05.005. URL http://www.sciencedirect.com/science/article/pii/S0167739X07000945. → pages 165

[141] T. Samak, C. Morin, and D. Bailey. Energy Consumption Models and Predictions for Large-Scale Systems. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 899–906, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4979-8. doi:10.1109/IPDPSW.2013.228. URL http://dx.doi.org/10.1109/IPDPSW.2013.228. → pages 96

[142] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP '04, pages 210–232, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25330-0, 978-3-540-25330-3. doi:10.1007/11407522_12. URL http://dx.doi.org/10.1007/11407522_12. → pages 21, 57, 100

[143] E. Santos-Neto, S. Al-Kiswany, N. Andrade, S. Gopalakrishnan, and M. Ripeanu. Enabling cross-layer optimizations in storage systems with custom metadata. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 213–216, New York, NY, USA, 2008. ACM. ISBN 9781595939975. → pages 16, 18, 25

[144] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 16–16, Berkeley, CA, USA, 2002. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973333.1973349. → pages 22, 24

[145] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003, 2003. → pages 22

[146] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST '10, pages 253–266, Berkeley, CA, USA, 2010. USENIX Association. → pages 25, 150

[147] T. Shibata, S. Choi, and K. Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 746–755, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. → pages 26, 27, 43, 62, 63

[148] A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 101–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0506-6. URL http://dl.acm.org/citation.cfm?id=846219.847390. → pages 37

[149] C. U. Smith. Performance engineering ofsoftware systems. *Addison-Wesley*, 1:990, 1990. → pages 121

[150] C. U. Smith and L. G. Williams. Performance engineering evaluation of object-oriented systems with spe· ed tm. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 135–154. Springer, 1997. → pages 122

[151] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164220. → pages 34, 39

[152] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 313–328, 2008. → pages 1, 3, 4, 7, 18, 34, 39, 45, 97, 130, 164

[153] K. Tangwongsan, H. Pucha, D. G. Andersen, and M. Kaminsky. Efficient similarity estimation for systems exploiting data redundancy. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 1487–1495, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-4244-5836-3. URL http://dl.acm.org/citation.cfm?id=1833515.1833727. → pages 140, 154, 167

[154] The HDF Group. Hierarchical Data Format, version 5, 1997-2014. http://www.hdfgroup.org/HDF5/. → pages 95

[155] E. Thereska and G. R. Ganger. IRONModel: robust performance models in the wild. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 253–264, 2008. → pages 37

[156] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *Proceedings of the 3rd International Conference on Autonomic Computing*, pages 187–198, 2006. → pages 1, 3, 6, 7, 18, 34, 37, 40, 69, 94, 95, 159

[157] E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask "what-if"? *Knowledge Engineering Review*, 21(3):261–267, September 2006. ISSN 0269-8889. doi:10.1017/S0269888906000920. URL http://dx.doi.org/10.1017/S0269888906000920. → pages 37

[158] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 3–14, New York, NY, USA, 2006. ACM. ISBN 1-59593-319-0. doi:10.1145/1140277.1140280. URL http://doi.acm.org/10.1145/1140277.1140280. → pages 34, 38, 41, 94, 159

[159] E. Vairavanathan, S. Al-Kiswany, L. B. Costa, Z. Zhang, D. S. Katz, M. Wilde, and M. Ripeanu. A workflow-aware storage system:

An opportunity study. In *Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '12, pages 326–334, Los Alamitos, CA, USA, June 2012. IEEE Computer Society. ISBN 978-0-7695-4691-9. → pages vii, 11, 15, 16, 17, 18, 21, 23, 24, 27, 30, 44, 57, 60, 62, 66, 112, 163

[160] A. Varga. Using the OMNeT++ Discrete Event Simulation System in Education. *IEEE Transactions on Education*, 42(4), 1999. ISSN 0018-9359. → pages 94

[161] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 5–20, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267903.1267908. → pages 166

[162] Y. Wang and P. Lu. Dataflow detection and applications to workflow scheduling. *Concurrency and Computation: Practice and Experience*, 23 (11):1261–1283, 2011. → pages 30, 55

[163] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 20–31. VLDB Endowment, 2002. URL http://dl.acm.org/citation.cfm?id=1287369.1287373. → pages 37

[164] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485. → pages 2, 102

[165] G. Wiederhold. What is your software worth? *Communications of ACM*, 49(9):65–75, September 2006. ISSN 0001-0782. doi:10.1145/1151030.1151031. URL http://doi.acm.org/10.1145/1151030.1151031. → pages 107

[166] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. T. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011. → pages 26, 30, 43, 55

[167] J. Wilkes. *Market Oriented Grid and Utility Computing*, chapter 4: Utility functions, prices, and negotiation. John Wiley & Sons, Inc., October 2008. ISBN 978-0-470-28768-2. → pages 4, 45, 97

[168] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *Proceedings of the 1st international workshop on Software and performance*, pages 164–177. ACM, 1998. → pages 122

[169] J. M. Wozniak and M. Wilde. Case studies in storage access by loosely coupled petascale applications. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 16–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-883-4. → pages 24, 26, 27, 43, 44, 62, 63

[170] H. Yang. Energy Prediction for I/O Intensive Workflow Applications. Master's thesis, The University of British Columbia, September 2014. URL http://hdl.handle.net/2429/50405. → pages 93

[171] H. Yang, L. B. Costa, and M. Ripeanu. Energy prediction for I/O intensive workflows. In *Proceedings of the 7th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers*, MTAGS '14. ACM, November 2014. → pages v, 12, 15, 21, 43, 84, 87, 91, 161, 163

[172] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance deduplication storage system for backup and archiving. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1 –12, April 2010. doi:10.1109/IPDPS.2010.5470468. → pages 126, 150

[173] T. Ye and S. Kalyanaraman. A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 196–205, New York, NY, USA, 2003. ACM. ISBN 1-58113-664-1. doi:10.1145/781027.781052. URL http://doi.acm.org/10.1145/781027.781052. → pages 164

[174] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 85:1–85:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. → pages 3, 17, 24, 69

[175] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. T. Foster. MTC Envelope: Defining the Capability of Large Scale Computers in the Context of Parallel Scripting Applications. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing - HPDC'13*, pages 37–48, June 2013. → pages 75, 95

Appendix A

Research Collaborations

During my PhD studies, I have conducted the research presented in this dissertation and collaborated with other researchers on various projects. The Preface presents the publications directly related to the research described in this dissertation. I have also collaborated with other researchers in different projects. Below, I briefly describe each of these projects and list the resulted publications.

TOTEM

TOTEM is a graph-processing solution for many graph-based applications, such as social networks and web analysis. As the memory footprint required to represent the graphs grows, they become more challenging to be processed efficiently. TOTEM's goal is to leverage commodity hybrid platforms (i.e., platforms built with processors optimized for sequential processing and accelerators optimized for massively-parallel processing as Graphics Processing Unit (GPU)) to accelerate large-scale graph processing at low cost. The long-term goal is to leverage the different characteristics of these two types of processors (sequential processors and parallel accelerators) to provide a platform that is neither expensive (e.g., supercomputers) nor inefficient (e.g., commodity clusters).

Abdullah Gharaibeh leads this project, which also includes the collabora-

tion of Elizeu Santos-Neto, and Matei Ripeanu. The results of this research have been published in the following:

Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems [81]. Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. Journal Submission. Pages 1–14. Under Review. Submitted in January 2014.

The Energy Case for Graph Processing on Hybrid CPU and GPU Systems [80]. Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. In Proceedings of the Workshop on Irregular Applications: Architectures & Algorithms (IA^3) in conjunction with SuperComputing '13. ACM, November 2013.

On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest [79]. Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto and Matei Ripeanu. 27th IEEE International Parallel & Distributed Processing Systems (IPDPS 2013). Pages 851–862. *Acceptance rate: 21%*. IEEE, May 2013

A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing [78]. Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto and Matei Ripeanu. In Proceedings of the IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT 2012). Pages 345–354. *Acceptance rate: 19%*. ACM, September 2012

GPU Support for Batch Oriented Workloads

I led this project during 2009, and it was conducted in collaboration with Samer Al-Kiswany and Matei Ripeanu. This project explores the ability to use GPUs as co-processors to harness the inherent parallelism of batch operations. Specifically, we have chosen Bloom filters (space-efficient data structures that support the probabilistic representation of set membership) as the queries these data structures support are often performed in batches. We

implemented BloomGPU, an open-source library that supports offloading Bloom filter support to the GPU, which outperforms an optimized CPU implementation of the Bloom filter for large workloads. The results of this work were published in:

GPU Support for Batch Oriented Workloads [53]. Lauro Beltrão Costa, Samer Al-Kiswany, and Matei Ripeanu. In Proceedings of the IEEE 28th International Performance Computing and Communications Conference (IPCCC). IPCCC '09. Pages 231–238. *Acceptance rate: 29.7%*. IEEE, December 2009.

NodeWiz Grid Information Service

Large scale grid computing systems provide multitudinous services, from different providers, whose quality of service varies. These services are deployed and undeployed in the grid with no central coordination, requiring a Grid Information Service (GIS) to help other grid entities to find the most suitable set of resources on which to deploy their services. NodeWiz is a GIS that allows multi-attribute range queries to be performed efficiently in a distributed manner, while maintaining load balance and resilience to failures.

This work is a result of collaboration with Hewlett-Packard Labs and the Distributed Systems Lab at the Universidade Federal de Campina Grande (UFCG). Most of the work related to this project was carried out before I joined UBC, including a working version of the system that was used in production with OurGrid [50]. Since starting my PhD studies at UBC, it resulted in the following publication:

Nodewiz: Fault-tolerant grid information service [26]. Sujoy Basu, Lauro Beltrão Costa, Francisco Brasileiro, Sujata Banerjee, Puneet Sharma, and S-J Lee. Journal of Peer-to-Peer Networking and Applications, 2(4):348–366. Springer, December 2009.