Understanding and Improving the Performance Consistency of Distributed Computing Systems

Mahmut Nezih Yiğitbaşı

Understanding and Improving the Performance Consistency of Distributed Computing Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 4 december 2012 om 12:30 uur

door **Mahmut Nezih Yiğitbaşı**

Master of Science in Computer Engineering, Istanbul Technical University, Turkey
geboren te Istanbul, Turkey.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. D.H.J. Epema

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr.ir. D.H.J. Epema	Technische Universiteit Delft, promotor
Prof.dr. A. van Deursen	Technische Universiteit Delft
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft
Dr.ir. A. Iosup	Technische Universiteit Delft
Prof.dr. H.A.G. Wijshoff	Universiteit Leiden
Dr.-Ing. Habil. Th. Kielmann	Vrije Universiteit Amsterdam
Th. L. Willke, EngScD	Intel Corporation, USA
Prof.dr. K.G. Langendoen	Technische Universiteit Delft, reservelid

This work has been carried out in the ASCI graduate school. ASCI dissertation series number 264.

This work has been done in the context of the Guaranteed Delivery in Grids (Guard-g) project, funded by NWO.

Part of this work has been done in collaboration with Intel Research Labs, USA.

ISBN: 978-94-6186-071-2

To my wife, with love.

Acknowledgments

A PhD thesis is really not something that I could have written alone. Along this long journey, I have been supported and encouraged by numerous people, to whom I am deeply grateful.

First and foremost, it has been a great pleasure for me to work with Prof. Dick Epema during my PhD. I am truly impressed by Dick's wisdom, attention to details, and high-quality work. I owe Dick many thanks for his support and encouragement for me to do internships at Intel Research Labs, which have paved the way to many opportunities. Dick's patience, insightful guidance, and encouragement have made this research possible, thank you Dick!

Alex, you have been a great mentor and a friend throughout my PhD. I have learned and enjoyed a lot from our collaboration, and our collaboration really helped me improve my research skills, thanks a lot!

I am grateful to Prof. Henk Sips for always being kind and willing to help. I would also like to express my appreciation to the rest of my committee members for allocating their valuable time to assess my thesis. Your insightful comments have improved this thesis a lot.

I would like to offer my special thanks to Intel Research Labs for providing me with the research opportunities as an intern. I have learned a lot during my internships at Intel. I have enjoyed working with various smart people at Intel Research Labs during 2010, 2011, and 2012. Thank you Babu Pillai and Lily Mummert for being great mentors and opening up new opportunities for me. You were always very friendly during my internship. I also want to thank Ted Willke for being an extraordinary mentor during my internships in 2011 and 2012. Ted, you have always been a great colleague to work with. You are the busiest people I know, and still you have allocated your valuable time to serve in my PhD committee, and I really appreciate that. I enjoyed both our technical and non-technical discussions a lot. Especially our discussions on "polluted models" and "state machines" were fantastic, thank you very much for everything!

I want to thank my Turkish friends, Ozan, Zeki, Zulkuf, and Gorkem, who helped me enjoy my time at TUDelft. I enjoyed being a roommate with Ozan for around two years, and I really enjoyed our collaboration. You have been a great friend and you have always been there whenever I needed help, thanks a lot! I also enjoyed spending time with Zulkuf and Zeki, guys I wish you success with your careers. Finally, thank you Gorkem for being a great friend, I really enjoyed our chats on research, politics, academia, etc.

Siqi and Yong, thanks for being great roommates. It was great knowing you guys, I wish you success in your PhDs, and I am sure you will do wonders.

I also want to thank the members of the PDS group, you have made this group a great place to work, thank you guys.

I am grateful for the assistance given by Paulo, Munire, and Stephen. You were always kind and friendly, thank you! I also want to thank the secretaries of the PDS group for making life easier for me.

Finally, I would like to thank my parents for their love, support, encouragement, and patience. Moreover, I am indebted to my beloved wife. We have been through many ups and downs during my PhD. Without her love and support, this thesis will not be possible.

Contents

Chapter 1

Introduction

The history of distributed computing systems goes back to ARPANET, which was created in the late 1960s, and which is known as the predecessor of the Internet [215]. The first successful distributed application utilizing the ARPANET infrastructure was the e-mail application created by Ray Tomlinson in the early 1970s [75]. With the growing interest in distributed computing systems, the field of distributed computing became an important branch of computer science in the late 1970s and 1980s. Since then, the field has attracted significant attention from both academia and industry, and we have seen many innovations along the way, such as clusters, grids, and recently, clouds.

Around the late 1970s, client workloads have started pushing the limits of single machines with their increasing complexity and processing requirements motivating the need for server *clusters*, which comprise multiple machines that are connected by a local area network and provide a single system image to its users [166]. The first commercial cluster was ARCnet, which was created by Datapoint in 1977 [214]. However, cluster computing was not really adopted until DEC released its VAXcluster product in 1984, which was built from general purpose off-the-shelf hardware and its general purpose VAX/VMS operating system [126, 214].

Later, in the mid 1990s, the term *grid* was used to describe the technologies that enable users to have access to a large amount of resources on-demand [81]. With grid computing, resources from different administrative domains in different countries are opened up transparently to scientists [80], leading to *e-science* that enables world-wide collaboration among scientists for solving complex research problems [153, 94]. Various grid infrastructures have been deployed all around the world: the European Grid Infrastructure (EGI) in Europe [71], the Distributed ASCI Supercomputer (DAS) in the Netherlands [21], the e-Science grid in the U.K. [204], the Grid’5000 grid in France [35], and the Open Science Grid (OSG) [159] and TeraGrid [200] in the United States, to name just a few.

Recently, *cloud computing* has been emerging as a new distributed computing paradigm where infrastructures, services, and platforms are provided to the users on demand. Clouds now enable everyone to have access to an “infinite” amount of resources with their credit cards. The common characteristics of clouds are the pay-per-use billing model, the illusion of an infinite amount of resources, elastic resource management (grow/shrink resources on demand), and virtualized resources [81]. Currently some of the popular cloud computing vendors are Amazon with their Elastic Compute Cloud (EC2) [12], Google with their App Engine [88], Microsoft with their Azure cloud [19], Rackspace [171], and GoGrid [87].

With the increasing and widespread adoption of distributed computing systems in both academia and industry, both scientific and business requirements motivate the users to demand more from these systems in terms of their compute and storage performance. For example, in the scientific domain the Large Hadron Collider (LHC) generates roughly 15 PB/year [38], and the high energy physics community has already been dealing with petabytes of data produced as a result of their experiments [26]. Similarly, the need to perform realistic simulations of complex systems also motivates scientists to have access to powerful resources; researchers have successfully simulated earthquakes on the Jaguar supercomputer of NCCS (National Center for Computational Sciences) [57], and the human heart has been realistically simulated on the T2K Open Supercomputer in Tokyo [97]. The industry is also pushing the limits of distributed computing systems—companies such as Google and Facebook now serve hundreds of millions of users around the world. Moreover, the decreasing cost of data acquisition and storage technologies enable companies to store massive amounts of data to drive their business innovation, and they have already deployed very large-scale distributed infrastructures to process these *big data*. For example, Google has reported processing 100TB of data per day in 2004 [61] and 20PB of data per day in 2008 [62], which is a 200-fold increase in only four years. Similarly, Facebook has reported having roughly 30PB of data in one of their MapReduce clusters as of 2011 [1].

Furthermore, with this increasing adoption users now also depend on distributed infrastructures for latency and throughput sensitive applications, such as interactive perception applications and MapReduce applications, which make the performance of these systems more important than before. Besides, distributed systems are also serving various mission critical services, such as banking, air traffic control, naval command and control systems, and telecommunications. Therefore, users expect *consistent performance* from these systems, that is, they expect the system to provide a similar level of performance *at all times*, such as having an acceptable performance variability even under system overload and failures, or having a consistent processing latency of less than 200 ms for their interactive applications while at the same time minimizing the number of latency spikes (transient high variability in latency) for a crisp user experience.

In this thesis, we provide an understanding of the performance consistency of state-of-the-art distributed computing systems, and using various resource manage-

ment and scheduling techniques we show how we can improve the performance consistency in diverse distributed systems, such as clusters and multi-cluster grids. We particularly focus on *distributed computing systems* as an important class of distributed systems to make it explicit that we do not consider many other types of distributed systems, such as web server systems or distributed database systems. Rather, we focus on various important distributed computing systems such as multi-cluster grids (Chapters 2 and 3), clusters (Chapter 4), and clouds (Chapters 5 and 6). Therefore, in the rest of this thesis we use the term distributed systems to refer to distributed computing systems.

The rest of this chapter is organized as follows. Section 1.1 presents the research problem we address in this thesis. Then, Section 1.2 presents the challenges that make this problem non-trivial. Finally, Section 1.3 concludes the chapter with our research contributions and the outline of this thesis.

1.1 What is the Research Problem?

Users expect consistent performance from distributed systems, that is, a system is expected to deliver roughly the same level of performance at all times–if an application usually takes 10s to complete, it will be annoying when sometimes the same application takes significantly more than that. Besides leading to user dissatisfaction and confusion, inconsistent performance can have various undesirable consequences. First and foremost, systems with high performance variability are inherently unpredictable, and therefore, hard to manage and debug. Secondly, performance inconsistency is a serious obstacle to productivity and efficiency. Because, high performance variability results in less work being done, lost compute cycles due to jobs being killed by the resource manager, which may be due to hard limits on job runtimes such as the 15 minute limit in DAS-4 [59], and less effective scheduling decisions [190]. In addition, highly variable performance makes it very difficult to reason about system behavior.

The consequences of inconsistent performance can be even more serious in production systems for both the users and the service providers. For the users, in a system that uses a pay-per-use billing model such as clouds, highly variable performance makes the costs unpredictable, and makes it very difficult for the users to properly provision resources for their workloads. Similarly, for the service providers, high variability may cause significant loss of revenue. For example, Amazon has reported that even small (100 ms) delays for web page generation will cause a significant (1%) drop in sales [132]. Likewise, Google has reported that an extra 0.5s in the search time causes a traffic drop of around 20% [132].

Before we can start to improve the performance consistency of distributed systems, an understanding of the performance of these systems is of crucial importance, but unfortunately, even understanding the performance behavior of these systems is non-trivial, primarily because of system complexity. Although traditional system engineers usually decompose a system into its components and try to understand these components to understand the complete system, this bottom up approach fails when systems get more complex, because they can behave in unexpected ways due to *emergent behavior* [149]; emergent behavior cannot be predicted with analysis at any level that is simpler than the complete system itself [70]. For example, it is not uncommon that systems fail badly when moving them from test to production environments as production systems may have significantly different workload characteristics, which can uncover corner cases. Similarly, systems usually behave completely unpredictable or they may even crash under overload. Another example is the characteristics of failures in distributed systems. Failures in real distributed systems have completely different characteristics than what have been assumed in traditional models; while failures were assumed independent in those models they are actually correlated in real systems as processes in a distributed system have complex interactions and dependencies between them.

In this thesis, we provide an understanding of the performance consistency of state-of-the-art distributed systems, and we explore resource management and scheduling techniques to improve the performance consistency in these systems. For this purpose, this thesis takes an empirical approach and explores this problem across diverse distributed systems, such as clusters, multi-cluster grids, and clouds, and across different types of workloads, such as bags-of-tasks (BoTs), interactive perception applications, and scientific workloads. Besides, since failures are shown to be an important source of significant performance inconsistency [116, 181, 65, 117, 33, 216, 22, 139], this thesis also provides a fundamental understanding of failure characteristics in distributed systems, which is necessary to design systems that can mitigate the impact of failures on performance consistency. In particular, we aim to address the following research questions in this thesis:

Can overprovisioning help to provide consistent performance in multi-cluster grids? We define overprovisioning as increasing the capacity of a system by adding more nodes (scaling out) to better handle the fluctuations in the workload, and provide consistent performance to users. Overprovisioning has been successfully used in telecommunication systems [168] and modern data centers for performance and reliability concerns [14, 24]. We investigate whether overprovisioning can also help to provide consistent performance in multi-cluster grids through realistic simulations.

How can we improve the performance of multi-cluster grids under overload? When large applications are submitted concurrently to grid head-nodes, they can get overloaded leading to degraded performance and responsiveness, and eventually noticeable performance inconsistencies. Various overload control techniques have been proposed in the literature [113, 48, 205, 185] primarily for web servers; among them, throttling, that is, controlling the rate at which workloads are pushed through the system, is a relatively simple technique that can deliver good performance. However, few of these techniques have been adapted for and investigated in the context of multi-cluster grids. Therefore, we address this question by exploring the performance of various static and dynamic throttling-based overload control techniques, including our adaptive throttling technique, in multi-cluster grids using BoT workloads; BoTs are the dominant application type in grids as they account for over 75% of all submitted tasks and are responsible for over 90% of the total CPU-time consumption [101].

How can we schedule interactive perception applications to minimize their latency subject to migration cost constraints? Interactive perception applications (e.g., controlling a TV with gestures) are a relatively new class of applications structured as data flow graphs. These applications usually comprise compute-intensive computer vision and machine learning algorithms, many of which exhibit coarse-grained task and data parallelism that can be exploited across multiple machines. To provide a responsive user experience, interactive applications need to ensure a consistent end-to-end latency, which is usually less than 100–200 ms for each processed data item (i.e., video frame). Moreover, it is also desirable for these applications to reduce the latency spikes as much as possible; frequent migrations of the application components can introduce such spikes, which reduces the quality of the user experience. We address this research question by devising algorithms that can *automatically* and *incrementally* place and schedule these applications on a cluster of machines to minimize the latency while keeping the migration cost in bounds, and by evaluating these algorithms with both simulations and real system experiments using two applications on the Open Cirrus testbed [17].

Is the performance of clouds sufficient for scientific computing? Cloud computing holds great promise for the performance-hungry scientific computing community as clouds can be a cheap alternative to supercomputers and specialized clusters, a more reliable platform than grids, and a much more scalable platform than the largest of commodity clusters. However, fundamental differences in the system size, the performance demand, and the job execution model between scientific computing workloads and the initial target workload of clouds raise the question of whether the performance of clouds is really sufficient for scientific computing. We address this question with an in-depth performance evaluation of four public clouds, GoGrid, ElasticHosts, Mosso, and Amazon EC2, which is one of the largest commercial clouds currently in production.

How variable is the performance of production cloud services, and what is the impact of the performance variability on distributed applications? An important hurdle to cloud adoption is trusting that cloud services are dependable, for example that their performance is stable over long time periods. However, service providers do not disclose information regarding their infrastructures or how they evolve, and these providers operate their physical resources in time-shared mode, which may cause significant performance variability. We address this research question with a comprehensive investigation of the long-term performance variability of ten production cloud services provided by Amazon Web Services and Google App Engine. We also explore through realistic trace-based simulations the impact of the performance variability on three large-scale applications. Our study is the first long-term study on the variability of performance as exhibited by popular production cloud services of two popular cloud service providers, Amazon and Google.

What are the characteristics of failures in distributed systems? Failures are an important source of performance inconsistency in distributed systems. With this research question we aim to provide a fundamental understanding of failure characteristics in distributed systems. First and foremost, understanding failure characteristics can help to design systems that can mitigate the impact of failures on performance consistency. For example, using good failure models, system architects can design schedulers that predict when a failure may occur and the number of machines that will fail, and then use this information to migrate workloads so that the performance remains unaffected. Moreover, understanding failures is also crucial for developing and assessing new fault tolerance mechanisms. Many of the previous studies have assumed that failures are independent and identically distributed [92, 234, 147]. Only a few studies [198, 34, 103, 199] have so far investigated the bursty arrival and correlations of failures for distributed systems. However, the findings in these studies are based on data collected from single systems– until the recent creation of online repositories such as Failure Trace Archive [123] and Computer Failure Data Repository [183], failure data for distributed systems were largely inaccessible to the researchers in this area. To address this research question we perform a detailed investigation using various data sets in the Failure Trace Archive, which are collected from diverse large-scale distributed systems including grids, P2P systems, DNS servers, web servers, desktop grids, and HPC clusters. Our study is one of the first failure studies at a very large scale; the data sets that we have used in our analysis comprise more than $100K$ hosts and more than $1M$ failure events, and span over 15 years of system operation in total.

1.2 Why is the Problem Challenging?

We identify five main challenges that make our research problem difficult, which we describe in turn.

1. **Distributed systems are complex.** Real-world distributed systems are asynchronous and non-deterministic by nature, and they comprise a large number of machines that have complex interactions between them over an unreliable network. For example, Google has an estimated data center size of around 1M servers [122], while Amazon EC2 and Microsoft data centers are estimated to contain around half a million servers [11] and tens or hundreds of thousands of servers [110], re-

spectively. Even research testbeds such as DAS-3 [58] and DAS-4 [59] comprise hundreds of servers. At such a scale, failures become inevitable, which introduces additional complexity to distributed systems. Given this size and complexity, understanding and reasoning about system behavior, and improving the performance consistency of these systems is non-trivial as complexity leads to emergent behavior, which is inherently unpredictable [70]. Moreover, these systems are usually very dynamic and heterogeneous, complicating the problem even further; resources come and go due to failures and elasticity of the resources, and these systems comprise multiple generations of hardware due to replacement of failed machines and due to infrastructure upgrades.

2. **Resources in a distributed system are shared by multiple users.** The shared nature of distributed systems makes it non-trivial to provide consistent performance to the users. For example, in multi-cluster grids a large user base shares the same compute, storage, and network resources. Similarly, cloud servers host multiple virtual machines to serve different users on the same physical machine, which complicates the problem of providing consistent performance as user workloads may interact in complicated and unpredictable ways. For example, a user's virtual machine can easily saturate the network, degrading the network performance of other tenants on the same physical machine.

3. **Distributed applications may have different requirements.** While some users run batch workloads, such as BoTs and MapReduce applications, other users may run interactive perception applications on the same cluster. The requirements of these applications are significantly different; batch workloads usually have high throughput requirements while interactive perception applications have high data rate and tight response time requirements. The diversity in application requirements makes it challenging to provide the required level of performance consistency to each application.

4. **Workloads processed by distributed systems are complex.** Users execute workloads of complex structures such as parallel applications, BoTs, workflows, interactive perception applications, and MapReduce applications. These workloads can be very large relative to the system in terms of number of tasks, runtime, and I/O requirements [77], and they may have significantly different performance requirements. This workload complexity makes it very difficult to understand their execution and reason about their performance, and in the end, makes it very difficult to reduce the performance variability.

5. **Failures in distributed systems are the norm rather than the exception.** Finally, we already know that the scale and complexity of distributed systems make the

occurrence of failures the norm rather than the exception [83, 225], and that failures are a serious hurdle to providing consistent performance [234, 143, 116, 181] as they cause noticeable variability and degradation in performance.

1.3 Research Contributions and Thesis Outline

In this thesis we address the problem of understanding and improving the performance consistency of distributed computing systems. To this end, we address the research problems presented in Section 1.1. We now present our research contributions and the outline of this thesis.

Overprovisioning strategies for performance consistency in grids (Chapter 2). We investigate overprovisioning to provide consistent performance to multi-cluster grid users. Overprovisioning can be defined as increasing the system capacity through adding more nodes (scaling out), by a factor that we define as the overprovisioning factor, to better handle the workload fluctuations, and provide consistent performance even under unexpected user demands. Through simulations, we present a realistic evaluation of various overprovisioning strategies with different overprovisioning factors and different scheduling policies. We show that beyond a certain value for the overprovisioning factor there is only slight improvement in performance consistency with significant additional costs. We also show that by dynamically tuning the overprovisioning factor, we can significantly (as high as 67%) increase the number of BoTs that have a makespan within a user specified range, thus improving the performance consistency. The content of this chapter is based on our research published in CCGRID'10 [222] and GRID'10 [223].

The performance of overload control in multi-cluster grids (Chapter 3). We investigate the performance of throttling-based overload control techniques in multi-cluster grids, motivated by our DAS-3 multi-cluster grid, where running hundreds of tasks concurrently leads to severe overloads and performance variability. Notably, we show that throttling leads to a decrease (in most cases) or at least to a preservation of the makespan of bursty workloads, while significantly improving the extreme performance (95^{th} and 99^{th} percentiles) for application tasks, which reduces the overload of the cluster head-nodes, and also leads to more consistent performance. In particular, our adaptive throttling technique improves the application performance by as much as 50% while also improving the system responsiveness by up to 80%, when compared with the hand-tuned multi-cluster system without throttling. The content of this chapter is based on our research published in GRID'11 [224].

Incremental placement of interactive perception applications (Chapter 4). We investigate the problem of incremental placement of perception applications, which are structured as data flow graphs, on clusters of machines to minimize the makespan subject to migration cost constraints. These applications require both low latency and, if possible, no latency spikes at all, which reduce the quality of the user experience. The vertices of such applications are coarse-grained sequential processing steps called stages, and the edges are connectors that reflect data dependencies between the stages. We propose four incremental placement heuristics that cover a broad range of trade-offs of computational complexity, churn in the placement, and ultimate improvement in the latency. A broad range of simulations with different perturbation scenarios (perturbing a random stage, perturbing a random processor, or adding a new stage instance to the application graph) show up to 50% performance improvement over the schedule without adjustment, that is, we let the application run after a perturbation and do not re-place the stages to other processors. Similarly, our experiments using two applications on the Open Cirrus testbed demonstrate 18% (10%) and 36% (38%) improvements in median (maximum) latency over the unadjusted schedule, respectively. In addition, we show that our heuristics can approach the improvements achieved by completely rerunning a static placement algorithm, but with lower migration costs and churn. The content of this chapter is based on our joint work with the Intel Science and Technology Center for cloud computing, previously published in ACM HPDC'11 [228] and Open Cirrus Summit'11 [235].

Performance evaluation of public clouds (Chapter 5). We investigate using various well-known benchmarks, such as LMbench [142], Bonnie [37], CacheBench [150], and the HPC Challenge Benchmark (HPCC) [136], the performance of four public compute clouds, including Amazon EC2. Notably, we find that the compute performance of the tested clouds is low. In addition, we also perform a preliminary assessment of the performance consistency of these clouds, and we find that noticeable performance variability exists for some of the cloud resource types we have explored. Our preliminary assessment only considers performance consistency over short periods of time and with low-level operations, such as floating point additions or memory read/writes, thus motivating us to explore the performance variability in depth in Chapter 6. Finally, we compare the performance and cost of clouds with those of scientific computing alternatives, such as grids and parallel production infrastructures. We find that, while current cloud computing services are insufficient for scientific computing at large, they may still be a good alternative for the scientists who need resources instantly and temporarily. The content of this chapter is based on our research published in CCGRID'09 [226], CloudComp'09 [162], and IEEE Transactions on Parallel and Distributed Systems [105].

Performance variability of production cloud services (Chapter 6). We investigate the performance variability of production cloud services using year-long traces that we have collected from the CloudStatus website [2]. These traces comprise performance data for two popular cloud services: Amazon Web Services (AWS) and Google App Engine (GAE). Our analysis reveals that the performance of the investigated services exhibits on the one hand yearly and daily patterns, and on the other hand periods of stable performance. We also find that many of these services exhibit high variation in the monthly median values, which indicates large performance changes over time. Moreover, we find that the impact of the performance variability varies significantly across different types of applications. For example, we demonstrate that the service of running applications on GAE, which exhibits high performance variability and a three-months period of low variability and improved performance, has a negligible impact for running grid and parallel production workloads. On the other hand, we show that the GAE database service, which exhibits a similar period of better performance as the GAE running service, outperforms the AWS database service for a social gaming application. The content of this chapter is based on our research published in CCGRID'11 [109].

Space-correlated failures in large-scale distributed systems (Chapter 7). We develop a statistical model for space-correlated failures, that is, for failures that occur within a short time frame across distinct components of the system using fifteen data sets in the Failure Trace Archive [123]. Our model considers three aspects of failure events, the group arrival process, the group size, and the downtime caused by the group of failures. We find that the best models for these three aspects are mainly based on the lognormal distribution. Notably, we find that for seven out of the fifteen traces we investigate, a majority of the system downtime is caused by space-correlated failures. Thus, these seven traces are better represented by our model than by traditional models, which assume that the failures of the individual components of the system are independent and identically distributed. The content of this chapter is based on our research published in Euro-Par'10 [83].

Time-correlated failures in large-scale distributed systems (Chapter 8). We investigate the time-varying behavior of failures in large-scale distributed systems using nineteen data sets in the Failure Trace Archive [123]. We find that for most of the studied systems the failure rates are highly variable, and the failures exhibit strong periodic behavior and time correlations. In addition, to characterize the peaks in the failure rate we propose a model that considers four parameters: the peak duration, the failure inter-arrival time during peaks, the time between peaks, and the failure duration during peaks. Remarkably, we find that the peak failure periods explained by our model are responsible for on average over 50% and up to 95% of the system downtime suggesting that failure peaks deserve special attention when designing fault-tolerant distributed systems. The content of this chapter is based on our research published in GRID'10 [225].

Finally, **Chapter 9** presents a summary of this thesis, presents the major conclusions, and describes several future research directions.

Chapter 2

Overprovisioning strategies for performance consistency in grids[*]

Users expect consistent performance from computer systems—when some interaction with an interactive application always finishes within 1 second, they are annoyed when suddenly the response time jumps to say 10 seconds. Likewise, when a certain Bag-of-Tasks (BoT) submitted to a grid has a response time of 5 hours, then the user will be surprised when a BoT with twice as many tasks (of a similar type as in the first BoT) takes say 24 hours. However, preventing such situations and providing consistent performance in grids is a difficult problem due to the specific characteristics of grids like the lack of support for advance reservations in many Local Resource Managers (LRMs), highly variable workloads, dynamic availability and heterogeneity of resources, and variable background loads of local users. In this chapter we investigate overprovisioning for solving the performance inconsistency problem in grids.

Overprovisioning can be defined as increasing the capacity, by a factor that we call the *overprovisioning factor*, of a system to better handle the fluctuations in the workload, and provide consistent performance even under unexpected user demands. Although overprovisioning is a simple solution for consistent performance and it obviates the need for complex algorithms, it is not cost effective and it may cause systems to be underutilized most of the time. Despite these disadvantages, overprovisioning has been successfully used in telecommunication systems [168] and modern data centers for performance and reliability concerns. Studies have shown that typical data center utilization is no more than 15-50% [14, 24], and telecommunication systems have roughly 30% [13] utilization on average.

A large body of work on providing predictable performance [66, 193, 218], and Ser-

---

[*] This chapter is based on previous work published in the *IEEE/ACM International Conference on Grid Computing* (Grid'10) [223] and the *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGRID'10) [222].

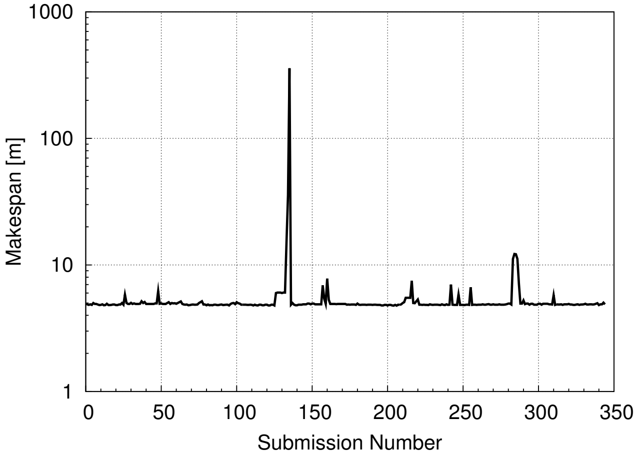


Figure 2.1: Evidence of the performance inconsistency in grids. The vertical axis has a logarithmic scale.

vice Level Agreements [130, 124, 8] already exists. What is missing so far from this research is a detailed realistic investigation of how we can achieve consistent performance in grids. In this chapter we fill this gap by performing a realistic investigation of both static and dynamic overprovisioning strategies for achieving performance consistency in grids. To this end, we propose several overprovisioning strategies for multi-cluster grids, and we classify these strategies as static or dynamic based on when the resources are provisioned. Then, we assess the performance and the cost of these strategies with realistic simulations. In our simulations we model the DAS-3 [58] multi-cluster grid and we use various synthetic workloads consisting of BoTs, which constitute the dominant application type in grids [99, 101]. Moreover, our model includes the actual background load of other users, which is one of the causes of performance inconsistency. We also approach the performance inconsistency problem from the user's perspective, and we design and evaluate a feedback-controlled system that exploits the elasticity of computing clouds to give performance guarantees to grid users. Our system overprovisions a grid dynamically using the user specified performance requirements and the measured system performance.

The rest of the chapter is organized as follows. Section 2.1 presents the motivation for the performance consistency problem. Section 2.2 and Section 2.3 describe the overprovisioning strategies and the system model that we evaluate in this chapter, respectively. Section 2.4 presents the experimental setup, and Section 2.5 presents the results of our performance evaluation. Section 2.6 describes the feedback-controlled system that dynamically determines the overprovisioning factor based on the specified user performance requirements. Finally, Section 2.7 reviews the related work on overprovisioning in grids, and Section 2.8 summarizes the chapter.

2.1 Motivation

Grid users may observe highly variable performance when they submit similar workloads at different times depending on the system state. From the users' point of view, any variability in performance should only be caused by their own applications (due to modifications of the applications or inputs) and not by the system or by load due to other users. Hence, inconsistent performance is usually undesirable, and it leads to user dissatisfaction and confusion.

Figure 2.1 shows evidence of the performance inconsistency in grids. In this experiment, we submit the same BoT consisting of 128 tasks periodically every 15 minutes to our multi-cluster grid DAS-3, which is usually underutilized. The graph shows the makespan in minutes for each submitted BoT. Since the system is mostly empty, we do not observe high variability in makespan for the first 130 submissions. However, we observe a significant variability between the 130th and 140th submissions, which is due to the background load created by other users, causing some tasks of the BoTs to be significantly delayed. The ratio of the maximum to the minimum makespan in this experiment is roughly 70! This result shows that even for a grid like DAS-3, which is a research grid, and hence usually underutilized, we may observe very strong performance inconsistencies.

It is a challenge to develop efficient solutions for providing consistent performance in grids due to their high degree of heterogeneity and the dynamic nature of grid workloads. It is possible to address this problem at two levels: at the (global) scheduler level, and at the resource level which consists of the computing nodes in the grid. To solve this problem at the scheduler level, we need to design appropriate *mechanisms*, e.g., admission control, and (scheduling) *policies*. In this chapter we take the latter approach and focus on the resource level, and we investigate overprovisioning to solve this performance inconsistency problem.

2.2 Overprovisioning Strategies

We define overprovisioning as increasing the capacity of a system to provide better, and in particular, consistent performance even under variable workloads and unexpected demands. We define the *overprovisioning factor* κ as the ratio of the size of an overprovisioned system to the size of the initial system. Overprovisioning is a simple solution that obviates the need for complex algorithms. However, there are also some disadvantages of this solution. First, overprovisioning is of course a cost-ineffective solution. Second, overprovisioning may cause the system to be underutilized since resources may stay idle most of the time; however, the industry is used to low utilization in data centers where the utilization is in the range 15-50% [14, 24], and in telecommunication systems where the average utilization is roughly 30% [13].

To overprovision grids we propose various strategies, and we classify them as static or dynamic based on when the resources are provisioned. We summarize these strategies below:

- **Static Overprovisioning:** The resources are provisioned statically at system deployment time, hence before the workload arrives at the system. We distinguish:
    - **Overprovision the Largest Cluster (`Largest`):** Only the largest cluster of the grid in terms of the number of processors is overprovisioned in this strategy.
    - **Overprovision All Clusters (`All`):** All of the clusters of the grid are overprovisioned equally.
    - **Overprovision Number of Clusters (`Number`):** The number of clusters of the grid is overprovisioned. The number of processors to deploy to the newly added clusters are determined according to the overprovisioning factor.
- **Dynamic Overprovisioning (`Dynamic`):** Since fluctuations are common in grid workloads, static resource provisioning may not always be optimal. Therefore, we also consider a dynamic strategy where the resources are acquired/released in an on-demand fashion from a compute cloud. We use low and high load thresholds specified by the system administrator for acquiring/releasing resources from/to the cloud, which is also known as *auto-scaling* [12]. We continuously monitor the system and determine the load of the system periodically, where the period is also specified by the administrator. If the load exceeds the high threshold we acquire a new resource, and if the load falls below the low threshold we release a resource to the cloud.

The number of processors to be deployed to a specific cluster is determined by the overprovisioning factor κ and the overprovisioning strategy. For example, assume that a grid has *N* clusters where cluster i has C_i processors, and that we use the `All` strategy for overprovisioning. Assume also that *C* is the size of the initial system, so $C = \sum_{i=1}^{N} C_i$. We want the size of the overprovisioned system $C' = \kappa C$, hence we set C'_i, the size of the overprovisioned cluster i, as $C'_i = \kappa C_i$. Thus, $C' = \sum_{i=1}^{N} C'_i = \sum_{i=1}^{N} \kappa C_i = \kappa C$. For the other strategies, the number of processors to deploy to attain a certain value of κ is derived similarly.

2.3 System Model

2.3.1 System Model

In our simulations we model our multi-cluster grid DAS-3 [58] which is a research grid located in the Netherlands. It comprises 272 dual-processor AMD Opteron compute nodes,

Cluster	Nodes	Speed [GHz]
Vrije University	85	2.4
U. of Amsterdam	41	2.2
Delft University	68	2.4
MultimediaN	46	2.4
Leiden University	32	2.6

Table 2.1: Properties of the DAS-3 clusters.

and it consists of five homogeneous clusters; although the processors have different performance across different clusters, they are identical in the same cluster. The cluster properties are shown in Table 2.1.

We assume that there is a Global Resource Manager (GRM) in the system interacting with the LRMs which are responsible for managing the cluster resources. The jobs are queued in the GRM's queue upon their arrival, and then dispatched to the LRMs where they wait for cluster resources. Once started, jobs run to completion, so we do not consider preemption or migration during execution.

When evaluating the `Dynamic` strategy, we assume that there is overhead for acquiring/releasing resources from/to the compute cloud. We have performed 20 successive resource acquisition/release experiments in the Amazon EC2 cloud with the `m1.small` instance type to determine the resource acquisition/release overheads [226]. We found that the minimum/maximum values for the resource acquisition and release overheads are 69/126 seconds and 18/23 seconds, respectively. We assume that the acquisition/release overhead for a single processor is uniformly distributed between these minimum and maximum values.

2.3.2 Scheduling Model

As the application type we use BoTs, which are the dominant application type in grids [99]. To model the application execution time, we employ the SPEC CPU benchmarks [195]: the time it takes to finish a task is inversely proportional to the performance of the processor it runs on. We consider the following BoT scheduling *policies*, which differ by the system information they use:

- **Static Scheduling:** This policy does not use of any system information. Each BoT is statically partitioned across the clusters where number of tasks sent to each cluster is proportional to the size of the cluster.
- **Dynamic Scheduling:** This policy takes the current state of the system (e.g., the load) into account when taking decisions. We consider two variants of dynamic scheduling:
  - **Dynamic Per Task Scheduling:** In this policy, a separate scheduling decision

	Bag-of-Tasks		Task
	Inter-Arrival Time	Size	Average Runtime
	W(4.25,7.86)	W(1.76,2.11)	N(2.73,6.1)
Average	124.6 s	6.1	7859.7 s

Table 2.2: The distributions and the values for their parameters for the BoT workload model described in [106]. N(μ,σ^2) and W(λ,k) stand for the Normal and Weibull distributions, respectively.

is made for each task of each BoT, and the task is sent to the cluster with the lowest load, where we define the load of a cluster as the fraction of used processors.

  - **Dynamic Per BoT Scheduling:** In this policy, a separate scheduling decision is made for each BoT, and the whole BoT is sent to the least loaded cluster.

- **Prediction-based Scheduling** We consider only one such policy:
  - **Earliest Completion Time (ECT):** This policy uses historical data to predict the task runtimes. With this policy each task is submitted to the cluster which is predicted to lead to the earliest completion time taking into account the clusters' queues. To predict the runtime of a task, we use the average of the runtimes of the previous two tasks [203], since this method is known to perform well in multi-cluster grids [193].

2.4 Experimental Setup

In this section we introduce our experimental setup. First, we describe the workload that we use in our simulations. Then, we describe our methodology and the metrics for assessing the performance and cost of the overprovisioning strategies. In our simulations, we model the DAS-3 multi-cluster grid (see Section 2.3.1) using our event-based grid simulator DGSim [108]. We extended DGSim with the scheduling policies described in Section 2.3.2, and we made extensions for performing simulations with compute clouds.

2.4.1 Workload

We have performed experiments with BoT workloads that we generate using the realistic BoT model described in [106]. The values for the important workload attributes are summarized in Table 2.2. These parameters are determined after a base-two logarithmic transformation is applied to the empirical data. In addition, in [106] the authors assume that the minimum BoT size is two, whereas we assume that single tasks are also BoTs with size one.

In our simulations we impose a background load together with the BoT workload in

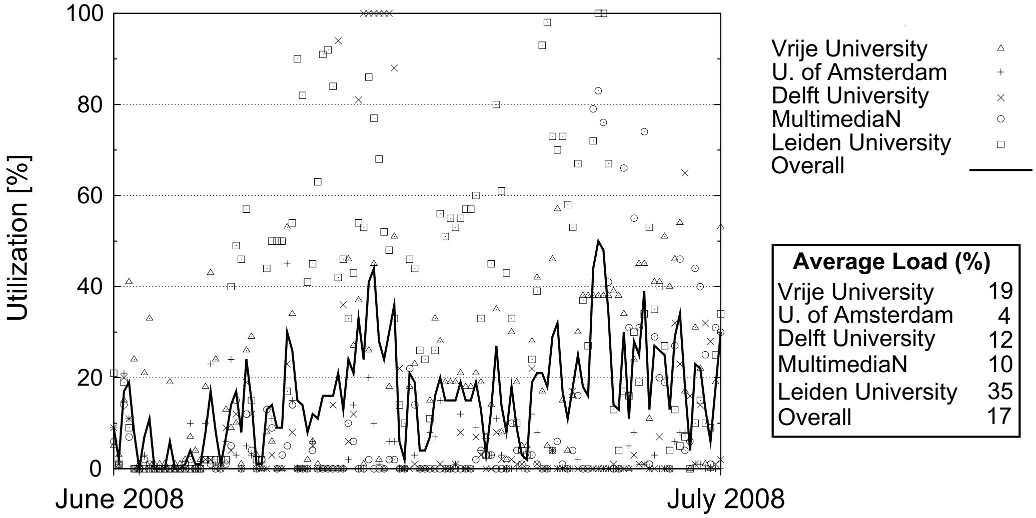


Figure 2.2: The overall utilization as well as the utilization in the individual clusters due to the background load, which consists of the jobs submitted to the DAS-3 system during June 2008.

order to attain realistic scenarios. The background load consists of the jobs submitted to DAS-3 during June 2008, and the corresponding workload trace is obtained from the Grid Workloads Archive [104]. Figure 2.2 shows the utilization of the background load. During the simulations, the background tasks are submitted to the LRMs of their original execution sites.

For our experiments, we have generated ten workloads that load the initial system to 80% on average, which we think is representative for a system that will be overprovisioned. Each workload contains approximately 1650 BoTs, and 10K tasks, and the duration of each trace is roughly between 1 day and 1 week.

2.4.2 Methodology

For assessing the static overprovisioning strategies, first, we evaluate the system with the aforementioned workloads, then we overprovision the system according to the strategy under consideration, and we use the same workload to assess the impact of the overprovisioning strategy. For the `Dynamic` strategy, a criterion has to be defined which determines when the system should acquire/release resources from/to the compute cloud. To this end, for the simulations with the `Dynamic` strategy, where the BoT workload imposes 80% load on the system, we use a high threshold of 70% and a low threshold of 60% for deciding when to acquire and release additional resources, respectively. When using the `Dynamic` strategy, κ varies over time. Hence, in order to obtain comparable results in our simulations with the `Dynamic` strategy, we keep the average value of κ

always in the ± 10% range of the specified value. For example, for $\kappa = 2.0$, when acquiring resources we do not exceed $\kappa = 2.2$, and when releasing resources we do not fall below $\kappa = 1.8$.

Finally, to obtain comparable results we assume that cloud resources have the same performance as the slowest grid cluster.

2.4.3 Performance Metrics

To evaluate the performance of the strategies, we use the makespan and the normalized schedule length as performance metrics. The makespan (MS) of a BoT is defined as the difference between the earliest submission time of any of its tasks, and the latest completion time of any of its tasks. The Normalized Schedule Length (NSL) of a BoT is defined as the ratio of its makespan to the sum of the runtimes of its tasks on a reference processor. Lower NSL values are better, in particular, NSL values below 1 (which indicates speedup) are desirable.

We also define and use two consistency metrics to assess different strategies. We define consistency in two dimensions: across BoTs of different sizes, and across BoTs of the same size. For assessing the consistency across BoTs of different sizes, we define

$$C_d = \max_{k,l} \frac{\bar{N}_k}{\bar{N}_l},$$

where N_k (N_l) is the stochastic variable representing the NSL of BoTs of size k (l).

To assess the consistency across BoTs of the same size, we define

$$C_s = \max_{k} CoV(N_k),$$

where $CoV(N_k)$ is the coefficient of variation of N_k. The system gets more consistent as C_d gets closer to 1, and C_s gets closer to 0. We also interpret a tighter range of the NSL as a sign of better consistency.

To evaluate the accuracy of the task runtime predictions when using the ECT policy, we use the accuracy, defined as in [203].

Finally, when evaluating the cost of the strategies, we use the *CPU-hours* metric which we define as the time in hours a processor is used. We believe that this metric is a fair indicator of cost independent of the underlying details like the billing model. When calculating the CPU-hours, we round up the partial instance-hours to one hour similar to the Amazon EC2 on-demand instances pricing model [12]. Although there are other costs like administration and maintenance costs of the resources, we neglect these costs, and we only focus on the resource usage.

Overprovisioning Strategy	$\kappa = 1.0$ (NO)		$\kappa = 1.5$		$\kappa = 2.0$		$\kappa = 2.5$		$\kappa = 3.0$	
	C_d	C_s	C_d	C_s	C_d	C_s	C_d	C_s	C_d	C_s
`All`	29.59	12.05	15.13	10.54	4.72	9.33	2.64	7.36	2.62	5.38
`Largest`	29.59	12.05	16.88	11.57	3.67	9.27	2.63	7.38	2.63	5.58
`Number`	29.59	12.05	17.71	10.61	3.75	9.12	2.70	6.90	2.42	5.67
`Dynamic`	29.59	12.05	14.65	10.27	3.50	8.64	2.42	6.36	2.10	4.60

Table 2.3: Summary of consistency values for all strategies and for different overprovisioning factors (κ).

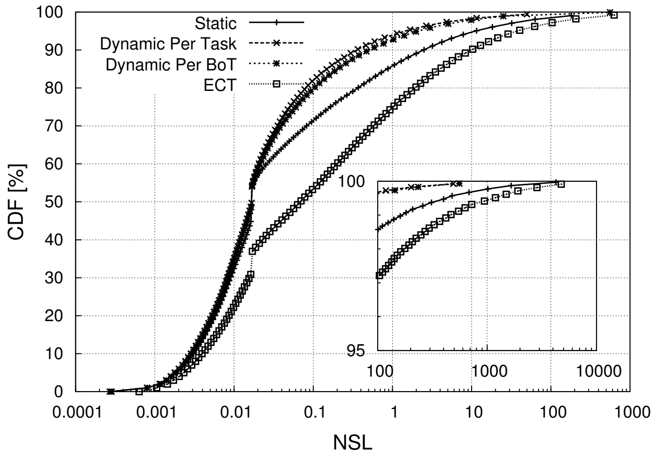


Figure 2.3: The Cumulative Distribution Function (CDF) of the Normalized Schedule Length (NSL) for the various scheduling policies. The horizontal axis has a logarithmic scale.

2.5 Experimental Results

In this section, we present the evaluation of the performance (Section 2.5.1) and cost (Section 2.5.2) of the overprovisioning strategies.

2.5.1 Performance Evaluation

Impact of the scheduling policy on performance Figure 2.3 shows the NSL distribution for all policies when no overprovisioning is applied. Although the Dynamic Per Task and the Dynamic Per BoT policies have similar performance, the Dynamic Per Task policy performs slightly better. The ECT policy has the worst performance by far compared to other policies. When using the ECT policy, the prediction accuracy is around 40%, which is low since all tasks in a BoT arrive within a short time interval, and hence the same prediction error is made for all tasks. This low prediction accuracy leads to scheduling decisions that cause some BoTs to suffer high response times with the ECT policy.

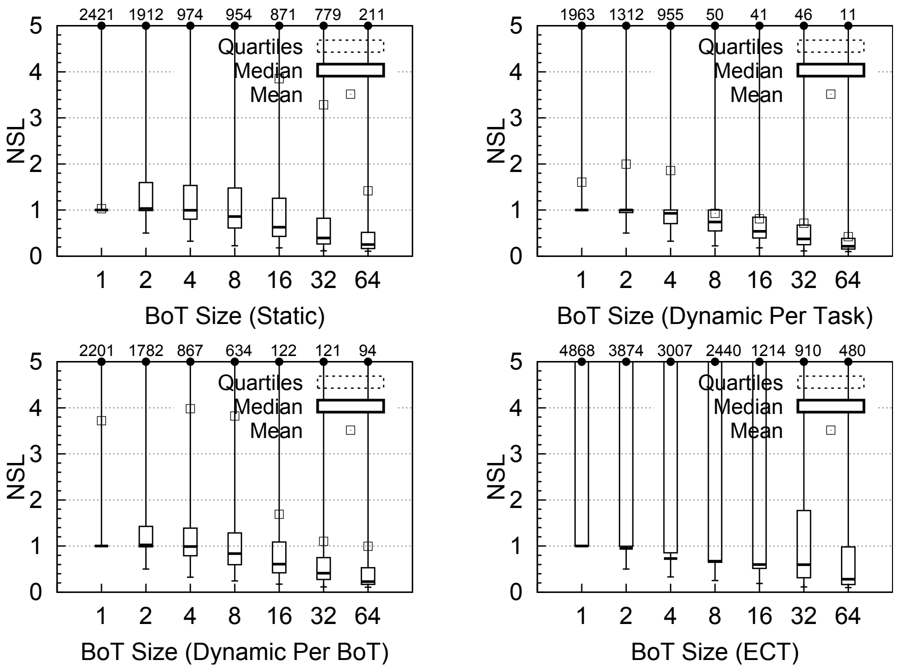


Figure 2.4: The impact of the scheduling policy on Normalized Schedule Length (NSL) when using the `All` strategy and $\kappa = 2.0$. The mean, and for ECT the third quartile is not always visible.

Figure 2.4 shows the impact of the scheduling policy on the NSL when we use the `All` overprovisioning strategy and κ is 2.0. In this section, for the box-whisker plots, the values at the top of the graphs are the maximum values observed, which are probably outliers, so what we are really interested in are the mean/median values and the quartiles. We observe that as the policy uses more recent system information, the NSL improves (lower interquartile range), hence the NSL of the Dynamic Per Task and Dynamic Per BoT policies is better than that of the other policies.

Since the Dynamic Per Task policy has the best performance among the policies, we use this policy in the rest of our evaluation.

Performance and consistency of the overprovisioning strategies The NSL distributions for the static strategies are shown in Figure 2.5 and for the `Dynamic` strategy it is shown in the upper-right graph of Figure 2.7 when κ is 2.0. Corresponding consistency metric values are shown in column 3 of Table 2.3, where the first column ($\kappa = 1.0$) shows the consistency values for the initial system (NO). Clearly, the consistency obtained with different strategies is much better than the initial system due to increased system capacity. We observe that the `Dynamic` strategy provides better consistency compared to static strategies (Table 2.3) since this strategy is able to handle the spikes in the workload that the static strategies can not handle. The static strategies have similar performance, so when overprovisioning a grid statically what really matters is the overprovisioning factor.

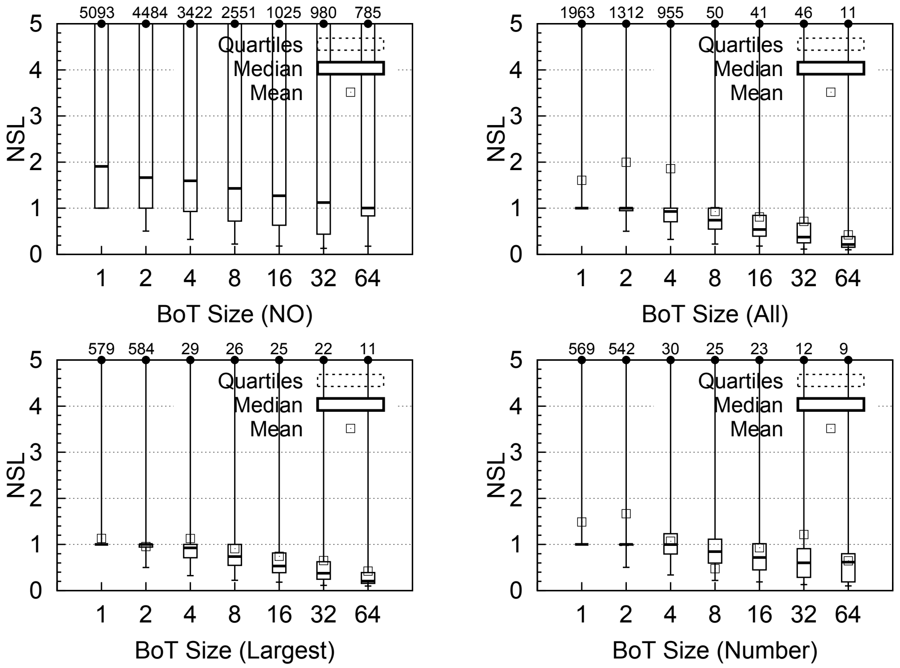


Figure 2.5: The Normalized Schedule Length (NSL) distributions for the static strategies (κ = 2.0). The third quartile is not visible for the initial system (NO).

However, since `Number` increases the number of clusters in the grid, hence increasing the administration costs, `All` and `Largest` are the viable candidates among the static strategies.

Impact of the overprovisioning factor κ on consistency Figure 2.6 and Figure 2.7 show the effect of κ on consistency with the `All` strategy and the `Dynamic` strategy, respectively. Corresponding consistency metric values are shown in Table 2.3. As expected, we observe significant improvements in the overall consistency of the system with increasing overprovisioning factors. The outliers that we observe with smaller overprovisioning factors disappear with increasing overprovisioning factors since the overprovisioned system can handle these spikes. In particular, going from $\kappa = 2.0$ to 2.5 dramatically reduces the outliers. Also, the outliers are much smaller for the `Dynamic` strategy than the `All` strategy.

However, we observe minor improvements in consistency as κ increases beyond $\kappa = 2.5$: *the overprovisioned system with* $\kappa = 2.5$ *can already handle the variability in the workload*. Hence overprovisioning beyond a certain value of κ (in our case for $\kappa = 2.5$), which we call the *critical value*, incurs significant costs but does not improve consistency significantly. Therefore, to maximize the benefit of overprovisioning it is important to determine the critical value of the overprovisioning factor.

Finally, the consistency metrics converge to similar values as κ approaches 3.0 (see Table 2.3). Although the system is overprovisioned significantly when $\kappa = 3.0$, there is

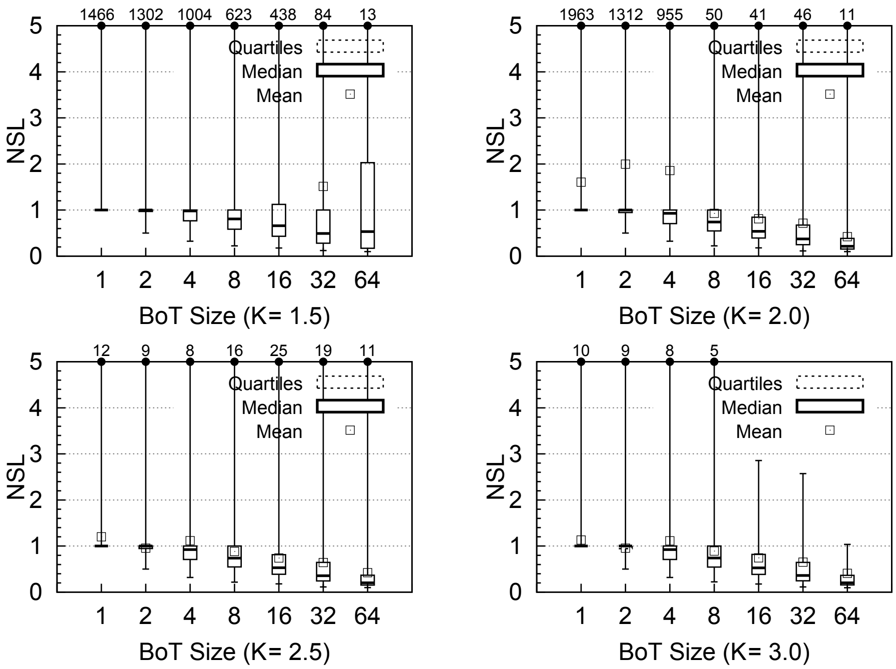


Figure 2.6: The effect of the overprovisioning factor κ on the Normalized Schedule Length (NSL) distribution with the `All` strategy for $\kappa = 1.5$ (upper left), $\kappa = 2.0$ (upper right), $\kappa = 2.5$ (lower left) and $\kappa = 3.0$ (lower right), respectively. Some of the mean values are not visible for the $\kappa = 1.5$ case.

still some variability in the performance which is probably due to the variability inherent in the workload.

2.5.2 Cost Evaluation

Due to the dynamic nature of grid workloads, static strategies may cause underutilization and hence increase the costs. The on-demand resource provisioning approach used with the `Dynamic` strategy overcomes these problems. In this section we evaluate the cost of the strategies for various overprovisioning factors to understand how much we can gain in terms of cost when using the `Dynamic` strategy. We use the CPU-hours metric described in Section 2.4.3 to assess the cost of the strategies.

Table 2.4 shows the cost of the `All` and `Dynamic` strategies for different overprovisioning factors. In this table, we only report the results for the `All` strategy since the cost is the same for different static strategies for the same overprovisioning factor. Although the cost increases proportionally with κ, we do not observe proportional performance improvement as we already show in Section 2.5.1. This situation is due to the underutilization of resources caused by static allocation. When using the `Dynamic` strategy, there is a significant reduction, as high as 42%, in cost since the resources are only ac-

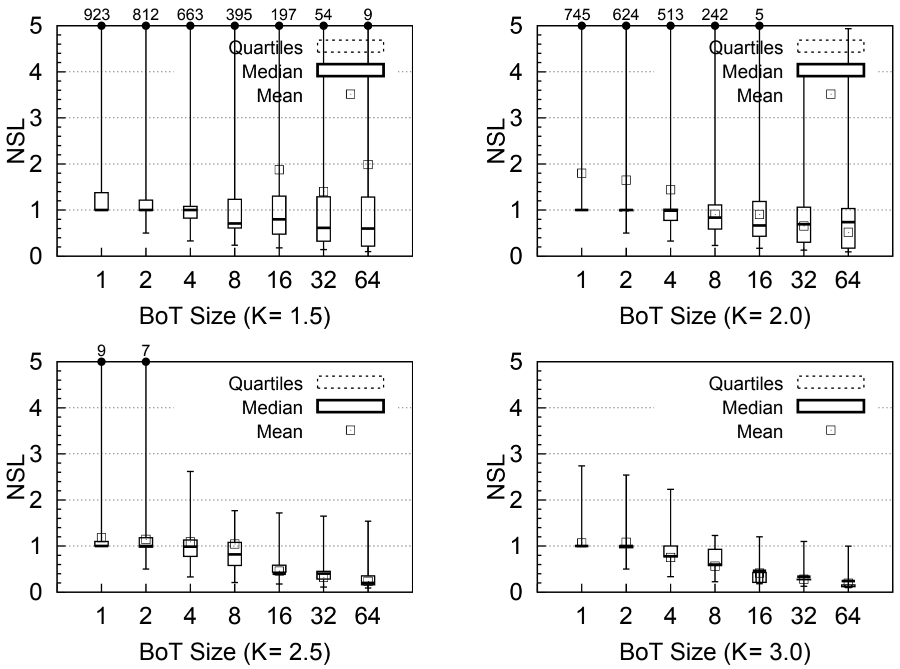


Figure 2.7: The effect of the overprovisioning factor κ on the Normalized Schedule Length (NSL) distribution with the `Dynamic` strategy for $\kappa = 1.5$ (upper left), $\kappa = 2.0$ (upper right), $\kappa = 2.5$ (lower left) and $\kappa = 3.0$ (lower right), respectively. Some of the mean values are not visible for the $\kappa = 1.5$ case.

κ	`All`	`Dynamic`	Reduction (%)
1.5	56655	32446	42.7
2.0	75540	49427	34.5
2.5	94425	69572	26.3
3.0	113310	85484	24.5

Table 2.4: Cost of the `All` and `Dynamic` strategies in terms of CPU-hours.

quired on-demand, and they are not allowed to stay idle as with static overprovisioning. As κ increases, the number of idle resources in the cloud also increases, hence decreasing the cost reduction. As a result, we conclude that the `Dynamic` strategy provides better consistency with lower costs compared to static strategies.

2.6 Dynamically Determining the Overprovisioning Factor

Up to this point, we evaluated the performance and cost of various strategies from the *system's perspective* with different overprovisioning factors and scheduling policies. In particular, our goal was to improve the *system's performance consistency*. We now approach our problem from the *user's perspective*, and we answer the question of how can we dynamically determine the overprovisioning factor to give performance guarantees to users. As our aim is to determine the overprovisioning factor and deploy additional processors *dynamically* to meet *user specified performance requirements*, in this section we only use the `Dynamic` strategy. Towards this end, we design a feedback-controlled system which exploits the elasticity of clouds to dynamically determine κ for specified performance requirements. Instead of a control-theoretical method, we follow an approach inspired by the controllers in the SEDA architecture [212]. Although control theory provides a theoretical framework to analyze and design feedback-controlled systems, the complexity and non-linear nature of grids make it very difficult to create a realistic model. In addition, due to the dynamic nature of grids, the parameters of a control-theoretical model will definitely change over time.

The controller uses various parameters shown in Table 2.5 for its operation. The `Window` parameter determines the number of BoTs that should be completed before the controller activation, hence, it determines how frequently the controller is activated and how fast it reacts to changes in the system performance. The `TargetMakespan` parameter determines the makespan target that the controller has to meet, and the `ReleaseThreshold` parameter determines the makespan threshold the controller uses to release cloud resources. The aim of the controller is to meet the `TargetMakespan` while at the same time avoid wasting resources when unneeded using the `ReleaseThreshold`. When specifying the `TargetMakespan` and `ReleaseThreshold` parameters, we use the 90th percentile of the makespan. This metric has two advantages compared to other metrics like the average or maximum: it better characterizes the makespan distribution, and we also believe that it reflects the user-perceived performance of the system better. To determine the sensitivity of the controller to the parameters of Table 2.5, we have performed various simulations with different parameter values except for the `Window` parameter, for which we use the value of ten BoTs.

In our architecture, the controller treats the system as a black box, and it measures the performance of the system at each activation using the historical performance data of the most recently completed BoTs. At each activation, if the measured performance exceeds the `TargetMakespan` value, the controller instructs the acquisition of a resource from

Parameter	**Description**
`Window`	Number of BoTs completed before controller activation
`TargetMakespan`	The target makespan
`ReleaseThreshold`	The makespan threshold used to release cloud resources

Table 2.5: The controller parameters with their corresponding descriptions.

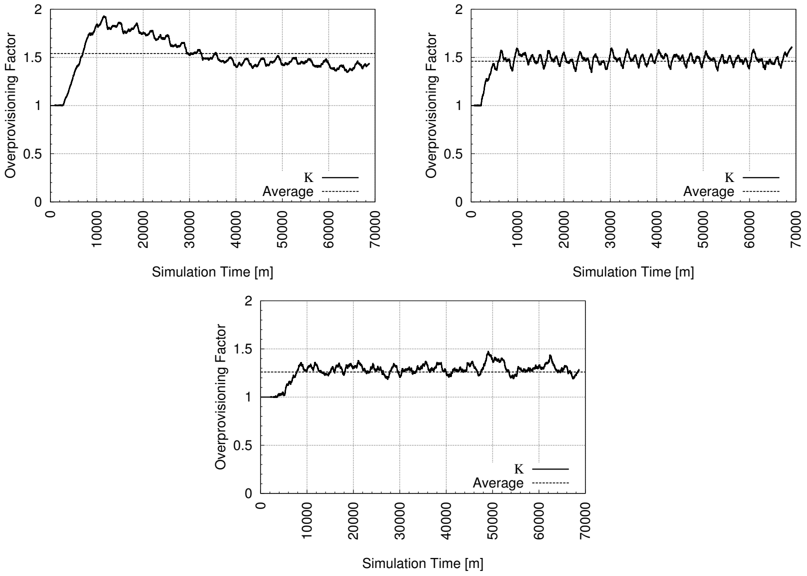


Figure 2.8: Overprovisioning factor over time and the average overprovisioning factor for the [`ReleaseThreshold`-`TargetMakespan`] values of [250m-300m] (left), [700m-750m] (center), and [1000m-1250m] (right).

the cloud. Similarly, if the measured performance falls below the `ReleaseThreshold` value, the controller instructs the release of a resource to the cloud. The provisioning of resources are performed one by one, and we leave as future work to determine the optimal number of resources to provision simultaneously.

To evaluate our design, we simulate the DAS-3 grid and we use the Dynamic Per Task scheduling policy without any background load. For these simulations, to empirically show that the controller stabilizes, we use an approximately one and a half month long workload consisting of 32860 BoTs, and the average BoT makespan for the workload in the initial system (without the controller) is roughly 3120 minutes (m). In our simulations we evaluate three different scenarios for loose and tight performance requirements. To this end, we use the [`ReleaseThreshold`-`TargetMakespan`] values of [250m-300m], [700m-750m], and [1000m-1250m] from tight to loose makespan performance requirements, respectively.

[`ReleaseThreshold`-`TargetMakespan`]	**w/o Controller**	**w Controller**	**Improvement**
[250m-300m]	4732	26849	67%
[700m-750m]	5900	21870	48%
[1000m-1250m]	6959	20219	40%

Table 2.6: Number of BoTs (out of 32860) that meet the specified performance requirements without (w/o) and with (w) the controller, and the resulting improvement (% of 32860) over the system without the controller.

Figure 2.8 shows the overprovisioning factor over time for the different performance requirements. Initially, there are no resources used from the cloud, hence $\kappa = 1$. The controller uses fewer cloud resources as the performance requirements get looser, resulting in lower overprovisioning factors compared to tight performance requirements. In addition, the average overprovisioning factor is smaller for loose performance requirements. It is also remarkable to note that when the performance requirements get tighter, there is only a rather small increase in the overprovisioning factor.

Table 2.6 shows the number of BoTs that meet the specified performance requirements (having a makespan less than the `TargetMakespan` value) without and with the controller, and the improvement (%) in the number of BoTs with the controller over the system without the controller. There is a significant improvement as high as 67% when the performance requirements are tight. The improvement gets smaller as the performance requirements get looser, as expected, since the system without the controller is already able to meet such loose performance requirements.

2.7 Related Work

We classify the previous work into three categories where the primary focus is either on predictable performance, Service Level Agreements (SLA) or overprovisioning. Although an extensive body of research focused on these research problems, there is no detailed investigation of how we can achieve consistent performance in grids. In [222], we took the first step towards filling this gap and we evaluated the performance of static overprovisioning strategies. In this chapter we extend our previous work by evaluating the performance and cost of both static and dynamic overprovisioning strategies with realistic simulations. We summarize the related work below.

Related work on predictable performance Various studies investigated advance reservations [191, 39, 156, 187] to provide the requested resources exactly when needed, therefore increasing the predictability of a system. We believe that advance reservations can also be used for providing consistent performance as the reserved resources are guaranteed to be available when needed (assuming no failures occur). However, designing scheduling policies that support advance reservations are shown to be difficult, and scalability is known as a major challenge in this design process [40].

As another solution for providing predictable performance, several studies explored prediction methods to predict various parameters like the job runtime and queue wait time. These predictions have been successfully used for scheduling and admission control decisions in grids [66, 193, 218].

The primary focus of this body of work is on providing *predictable* performance with advance reservations and predictions. In contrast, our work focuses mainly on providing *consistent* performance with overprovisioning.

Related work on SLAs Leff et al. [130] propose a dynamic offload infrastructure similar to our feedback-controlled system to meet the SLAs in a commercial grid deployment. This infrastructure hosts a resource pool comprising preconfigured servers that stay idle, and to meet the specified performance requirements additional servers are acquired from this pool. In contrast, we use on-demand resources from a compute cloud instead of keeping a pool of idle resources within our system, which is definitely a waste of capacity. In this work, the authors perform experiments with web workloads while we use realistic grid workloads to perform a more detailed investigation than theirs to investigate various tight and loose performance requirements.

Kounev et al. [124] present an approach for QoS aware resource management in grids using online performance models. The authors show that the negotiated SLAs can be satisfied with the proposed approach. Unlike this work we use overprovisioning to meet the negotiated performance requirements. Moreover, our work targets multi-cluster grids while their work targets service-based grids on which services are deployed and the workload consists of HTTP requests.

Menascé et al. [144] present a resource allocation framework that finds the optimal resource allocation to minimize the total cost and meet the execution time specified by the SLAs. The workload used in this work comprise applications with dependent tasks while in our work we use BoTs as the workload, and we use overprovisioning instead of optimization methods to provide consistent performance. Moreover, compared to this work our performance evaluation is more in depth with diverse grid workloads.

Finally, Al-Ali et al. [8] present a prototype QoS system for real-time grid applications. Similar to [124], this work targets service-based grids. In this work the authors use advance reservations to meet the SLAs, in contrast we focus on providing consistent performance with static and dynamic overprovisioning techniques.

Related work on overprovisioning In [60], De Assuncao et al. explore six scheduling policies to extend a cluster's capacity with cloud resources. In particular, they investigate the performance and cost trade off with simulations, and they show that request backfilling and redirection provides a good balance between the performance and the cost. In this work authors focus on improving the performance of a single cluster using only dynamic overprovisioning. However, we focus on providing consistent performance in multi-cluster grids using both static and dynamic overprovisioning strategies. In addition, we also design and evaluate a feedback-controlled system to determine the amount of resources to provision (κ) for specified performance requirements.

2.8 Summary

Providing consistent performance in grids is a difficult research problem. In this chapter we have investigated overprovisioning to solve this problem, and we have performed a realistic evaluation of overprovisioning in multi-cluster grids. Although our main focus is on grids, we believe that the main ideas are also applicable to other large-scale distributed systems.

We have presented a realistic evaluation of various overprovisioning strategies with different overprovisioning factors (κ) and scheduling policies. We found that beyond a certain value for the overprovisioning factor (in our case $\kappa = 2.5$) there is only slight improvement in consistency with significant additional costs. In addition, the `Dynamic` strategy provides better consistency with lower costs compared to static strategies. Finally, to dynamically determine the overprovisioning factor to give performance guarantees to users, we have designed and evaluated a feedback-controlled system exploiting the elasticity of clouds. Through various simulations for loose and tight makespan performance requirements, we have shown that our system provides significant improvements over the initial system, as high as 67%, in the number of BoTs that meet the specified performance requirements.

Chapter 3

The performance of overload control in multi-cluster grids[*]

Many scientists rely on the execution of applications on multi-cluster grids, that is, of large-scale distributed systems comprised of heterogeneous clusters. Multi-cluster grids such as the DAS-3 in the Netherlands, the EGEE grid in Europe, and the Open Science Grid in the US provide efficient execution infrastructures for applications with a loosely coupled structure, such as bags-of-tasks (BoTs) and workflows. When executing such applications, the system may become *overloaded*, that is, the system resources shared by running applications may become bottlenecks—the disks of the cluster file systems may become saturated, the grid communication protocols may break down due to thousands of concurrent submissions, etc. Since overloads can degrade the performance and even cause systems to crash, many overload control techniques have been designed [113, 48, 205, 185]; among them, *throttling*, that is, controlling the rate at which workloads are pushed through the system, is a relatively simple technique that can deliver good performance. However, few of these techniques have been adapted for and investigated in the context of multi-cluster grids. In this chapter we present a dynamic throttling technique along with an extensive performance evaluation of throttling-based overload control techniques for multi-cluster grids.

The consequences of overload can be severe, such as increased backlogs at shared resources, and decreased performance and responsiveness leading to unpredictable system behavior and user dissatisfaction. In multi-cluster grids overloads can lead to task wait times that are often in excess of several hours [99]. The situation is even worse in production systems where overloads can cause significant loss of revenue to service providers. For example, Amazon reported that even small (100 ms) delays for web page generation will cause a significant (1%) drop in sales [132]. Similarly, Google reports that an extra

---

[*] This chapter is based on previous work published in the *IEEE/ACM International Conference on Grid Computing* (Grid’11) [224].

0.5s in search time causes a traffic drop of 20% [132].

In multi-cluster grids there are two primary causes of overload. First, grid workloads may be very bursty or even difficult to predict, at both short and long time scales [193]. To illustrate this, Figure 3.1 shows the number of tasks submitted to the DAS-3, the SHARCNET, and the GRID3 multi-cluster grids, and to a multi-thousand node production MapReduce cluster of an online social networking company. Second, the applications submitted to multi-cluster grids can be large relative to the system in terms of number of tasks, runtime, and I/O requirements [77].

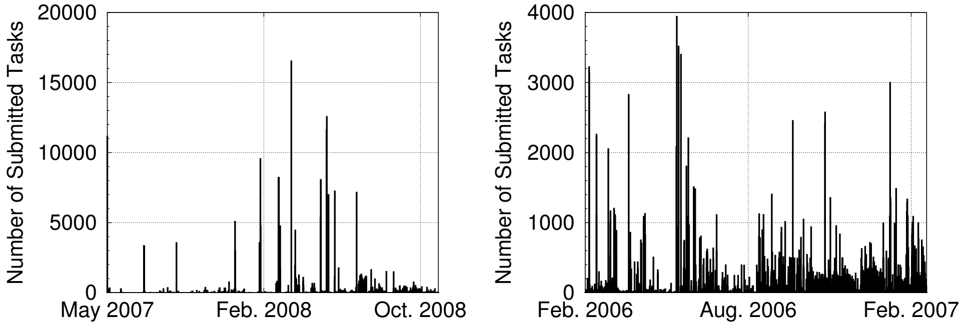
The overload control problem has been studied extensively and several techniques for alleviating overloads, such as congestion and admission control [48], control theoretic approaches [212], scheduling [185], and overprovisioning [205], have been proposed. However, they have not been investigated in the context of multi-cluster grids, which differ significantly from these other systems in both structure and workload. Structurally, multi-cluster grids are comprised of heterogeneous clusters distributed over a wide-area network. The typical workload of a multi-cluster grid consists of scientific applications with BoT, workflow, and parallel HPC structure [99, 101]. Among these application types, BoTs are the dominant application type in grids, as they account for over 75% of all submitted tasks and are responsible for over 90% of the total CPU-time consumption [101].

In this chapter, we first adapt a dynamic throttling technique to control overload in multi-cluster grids under bursty workloads. Then, we investigate the performance of three throttling techniques, including our technique, with extensive experiments using diverse workloads in our DAS-3 multi-cluster grid. Our performance evaluation leads to two main observations. First, we find that throttling can significantly improve both application performance and system responsiveness in multi-cluster grids, even under bursty workloads. Second, we find that, for multi-cluster grids, the dynamic throttling-based overload control technique can replace the static (hand-tuned). The latter result is particularly significant in multi-cluster grid settings, where hand-tuning is slow and costly due to the number of clusters, and difficult due to workload burstiness.

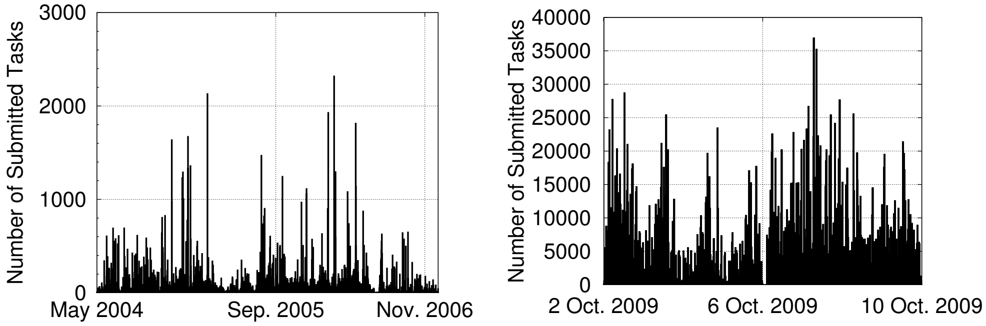
The rest of the chapter is organized as follows. Section 3.1 presents the multi-cluster grid model used throughout this chapter. Section 3.2 describes the throttling-based overload control techniques that we evaluate in this chapter. Section 3.3 describes the experimental setup, and then Section 3.4 presents our performance evaluation results. Finally, Section 3.5 reviews the related work on overload control in diverse computer systems, and Section 3.6 summarizes the chapter.

3.1 Multi-Cluster Grid Model

In this chapter we focus on multi-cluster grids comprising heterogeneous clusters. Such systems usually include a head-node for each cluster, which is a central node that users



(a) DAS-3 (b) SHARCNET



(c) GRID3 (d) MapReduce Cluster

Figure 3.1: Number of tasks submitted to three multi-cluster systems and a multi-thousand node production MapReduce cluster within five minute intervals. All systems have periods of burst submissions.

connect and which uses middleware to interact with the rest of the system. The middleware operates in each cluster and is responsible for managing the compute resources (worker nodes). Tasks that are submitted to the middleware are initially placed into the middleware queue until there are enough resources to execute the tasks. After the submission, the middleware dispatches the tasks to the assigned nodes and manages the task execution. This model fits many production multi-cluster systems, including the worldwide LCG, Grid5000, TeraGrid, and our DAS-3 system. Our model also fits other multi-cluster systems, and in particular the numerous deployed systems using Globus, which is arguably the most used middleware [86], the Grid Engine, and PBS/Torque. Our model does not exactly fit the systems based on loosely-integrated resources, such as the systems based on Condor; however, while other configurations are possible, many Condor pools use in practice a single Negotiator, which effectively plays the role of the cluster head-node in our model.

As an example, our DAS-3 system employs two primary components: a *runner* (application-level scheduler) deployed on a head-node which is responsible for a single application submission, and an *execution service* deployed on each head-node which is responsible for interacting with the middleware and performing protocol conversion between the middleware and the runners. These two components may communicate over the local area network or the wide area network. In each of the clusters, the head-node communicates with the worker nodes for task execution management and with the distributed file system for the file transfers.

3.2 Overload Control Techniques

In this section we describe the throttling-based overload control techniques that we investigate in this chapter.

To detect overload we use the head-node CPU load and the disk utilization metrics which we think are good indicators of overload based on our experience with multi-cluster grids and their workloads. For each metric we set a *threshold* and a *maximum* value. During workload execution depending on the measured values of these metrics and the threshold and maximum values a cluster may be in one of two states, overloaded or underloaded. An underloaded cluster transitions to the overloaded state when *either* the head-node's CPU load or the disk utilization exceeds its maximum value, or when *both* metrics exceed their threshold values. Similarly, an overloaded cluster transitions to the underloaded state when *either* of these metrics falls below its threshold value. After an overload is detected, the throttling technique reacts by enforcing a *concurrency limit*–the maximum number of concurrently running tasks–for every application in the system. We describe in the following our throttling-based overload control techniques, in turn.

- **Static throttling (`Static`):** This technique uses a static concurrency limit for throttling. With `Static` it is possible to underutilize the system with a low concurrency limit, and overload the system with a high concurrency limit. Thus, it is crucial to determine the best concurrency limit for a particular system and workload with `Static`. For our experiments we have manually tuned the concurrency limit to the value that gives the best performance over many experiments, so in our evaluation (Section 3.4) `Static` provides the best performance for our system and workloads.
- **Bang Bang Control (`BBC`) [96]:** With `BBC`, the execution service notifies the runner to stop submitting tasks when the head-node transitions to the overloaded state. When a head-node transitions back to underloaded state, the execution service notifies the runner to resume its task submission. `BBC` lets the runner to temporarily overload a cluster, as too many tasks may be submitted when that cluster recovers from overload and before the execution service can detect and react to the new overload.

  In heterogeneous multi-cluster grids `BBC` may perform poorly: it is possible that all but the fastest cluster may get overloaded and only the fastest cluster may be underloaded. Such a situation causes the fastest cluster to receive all the tasks while the other clusters are recovering from their overloads causing the queueing times at the local resource manager to increase noticeably. To solve this problem we adapt the original BBC algorithm by introducing a maximum concurrency limit (`C_LIMIT_MAX`) for each cluster so that when the number of tasks that are running concurrently in a cluster exceeds `C_LIMIT_MAX` the cluster transitions to the overloaded state.

- **Adaptive throttling (`Adaptive`):** To address the inflexibility of `Static` and the problem of temporarily overloading the clusters of `BBC`, we propose an Additive Increase Multiplicative Decrease (AIMD) based controller that dynamically adjusts the concurrency limit. It has been shown that AIMD-based control is a provably convergent control rule [49]. However, to design a controller that gives additional guarantees control theory can also be used [93].

  `Adaptive` operates in each cluster independently. It uses the following constants as inputs: the number of nodes in the cluster (`N_NODES`), the threshold and maximum values for the CPU load and disk utilization, and three parameters that are explained in the following (α, β, and `C_LIMIT_MAX`). `Adaptive` tunes the concurrency limit (`c_limit`) as follows. Initially, `c_limit` is set to `N_NODES`. Periodically, `Adaptive` measures the head-node CPU load and the disk utilization, and it checks whether the cluster is overloaded using the corresponding threshold and max values. If the cluster is overloaded, `c_limit` is decreased by being set to $\alpha \cdot$ `c_limit`, with $0 < \alpha < 1$. If the cluster is not overloaded, then `c_limit` is increased by being set to `c_limit` $+ \beta \cdot$ `n_finished`, with $0 < \beta \leq 1$ being used to gradually increase `c_limit` to avoid overshooting and `n_finished` tasks have finished since the last control period. To prevent clusters from being severely overloaded even temporarily, `c_limit` is not allowed to exceed the maximum concurrency limit `C_LIMIT_MAX`. We describe in Section 3.3.4 how we set the values of these parameters in our experiments.

We have implemented the throttling techniques presented in this section as a part of the runner and the execution service presented in Section 3.1. In Section 3.4, we evaluate the performance of these throttling techniques in our multi-cluster grid described in the next section.

3.3 Experimental Setup

In this section we first describe our multi-cluster grid DAS-3 in which we evaluate the performance of the throttling techniques presented in the previous section. Then we describe the workloads that we use and the performance metrics that we report as a result of

Cluster	# of Nodes	Node CPU Speed [GHz]	# of Cores on the Head-Node
C1	22	2.6	8
C2	29	2.2	8
C3	60	2.4	8

Table 3.1: The processing capability of our multi-cluster grid.

our evaluation in Section 3.4.

3.3.1 Multi-Cluster Testbed

We perform our experiments on three clusters of our DAS-3 testbed. Table 3.1 shows the processing capability of our testbed. Each cluster has a separate distributed file system and on each cluster the Grid Engine (GE) middleware operates as the local resource manager. GE has been configured to run tasks on the nodes exclusively (in space-shared mode). We have deployed the execution service on each cluster's head-node, and the runner has been deployed on the head-node of the C3 cluster; execution service and runner are described in Section 3.1.

3.3.2 Workloads

We evaluate our throttling techniques (Section 3.2) using BoTs, which are the dominant application type in multi-cluster grids. We summarize in Table 3.2 the characteristics of the workloads used in our experiments. All tasks of a BoT are submitted to the system at the same time, so our workloads represent the worst-case overload scenario. The W-Base workload comprises 1,000 tasks, each with a runtime of 60 seconds and performing 100 MB I/O. To understand the impact of the workload characteristics, we perform the evaluation across three dimensions: starting from W-Base we increase, in turn, the number of tasks (W-Task), the task runtimes (W-Run), and the task I/O requirements (W-IO) of the BoT. Although each workload is homogeneous, together they cover a wide range of scenarios, from compute-intensive to communication-intensive, and from small-scale to large-scale applications. Their tasks have similar runtimes and I/O requirements to the tasks observed in real multi-cluster grid workloads [99].

3.3.3 The Performance Metrics

In our evaluation we use several metrics that we categorize as system or user metrics. System metrics quantify the performance of the system components while user metrics quantify the performance perceived by the user.

Workload	Number of Tasks	Task Runtime [s]	Total I/O Per Task [MB]
W-Base	1,000	60	100
W-Task	5,000	60	100
W-Run	1,000	300	100
W-IO	1,000	60	200

Table 3.2: Workloads used in our experiments.

- **System Metrics:**
    - **CPU Usage [%]:** The fraction of time a process keeps the CPU busy as reported by the Linux `top` utility. We use this metric to assess the overhead of our scheduler in Section 3.4.1.
    - **CPU Load:** The number of processes which are in the processor run queue or waiting for I/O. We report the average CPU load calculated over one minute intervals as reported by the kernel. When the CPU load is high, the head-nodes cannot respond to connection requests, so we use this metric to quantify the system responsiveness. It is better if this metric is close to the number of cores of a head-node.
    - **Disk Utilization [%]:** Fraction of time the disk is busy as reported by the Linux `iostat` utility. We report the average utilization calculated over five second intervals.
    - **Cluster Utilization [%]:** Fraction of available nodes that are used.
- **User Metrics:**
    - **I/O Service Time [ms]:** The time it takes for the disk to serve an I/O request. We report the average service time calculated over five second intervals.
    - **Task Execution Time [s]:** The time it takes for a task to complete its execution.
    - **Makespan [s] (of a BoT):** The difference between the earliest time of submission of any of its tasks and the latest time of completion of any of its tasks.

3.3.4 Parameters for the Overload Control Techniques

Table 3.3 summarizes the parameters for the throttling techniques with their values that we use in our experiments. Since the best values for these parameters depend on a particular system and workload we have performed several experiments to determine the best values for our system.

We use a control period of 30s which is smaller than the shortest task in our workloads. Hence, the throttling techniques react fast enough to the changes in the monitored

Parameter	**Value(s)**
Control Period	30 s
CPU Load Threshold	7
Max. CPU Load	10
Disk Utilization Threshold	40%
Max. Disk Utilization	60%
α	0.5
β	0.5 and 1.0
`C_LIMIT_MAX`	Number of nodes (see Table 3.1)

Table 3.3: The parameters for the overload control techniques and their values used in our experiments.

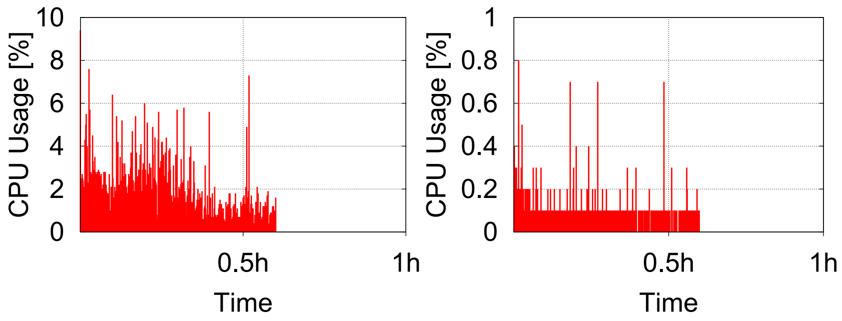
metrics. Since all cluster head-nodes are 8-core machines, we use a CPU load threshold of 7 (corresponding roughly to 90% utilization) and a maximum CPU load of 10 (letting a head-node to be overloaded up to 125%). When our system is empty, the average disk utilization is less than 20%. Therefore, for this metric we use 40% as the threshold and 60% as the maximum value.

For `Adaptive` the value of the α parameter should be set to provide a balance between the throughput and the speed of overload recovery. We use $\alpha = 0.5$ in our experiments. Small values of α may degrade the throughput while larger α values may cause the system to recover from overload slowly. Experiments with larger values, such as 0.7 and 0.8, did not lead to substantial differences in the observed performance. For the β parameter we use a value of either 0.5 or 1.0. Unless otherwise specified, we use $\beta = 0.5$ in our experiments. For small values of β the throughput may degrade while for larger β values the runner may temporarily overload a cluster as the concurrency limit will be increased quickly.

Finally, for the maximum concurrency limit parameter (`C_LIMIT_MAX`), with `Static` we use 30 tasks which we found through several experiments to perform well for our system, and we set the value of this parameter to the number of available nodes on each cluster for `BBC` and `Adaptive` to prevent the tasks from getting queued in the local resource managers.

3.4 Experimental Results

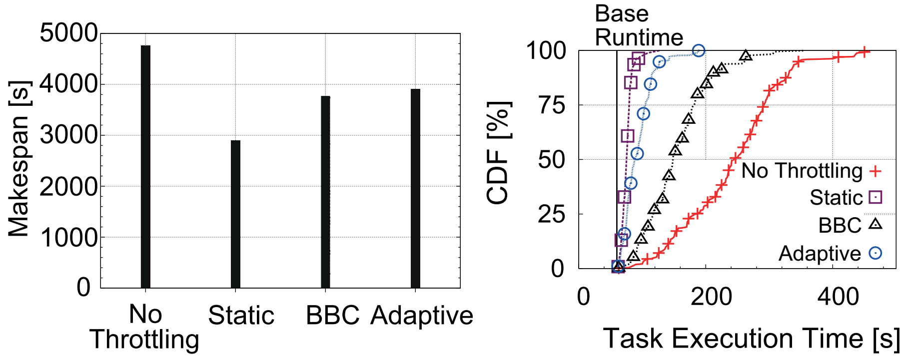
In this section we assess the performance of the throttling techniques described in Section 3.2 and of the system without throttling (`No Throttling`). We first validate the assumption that our system's scheduling middleware is not a bottleneck (Section 3.4.1). Then, we perform two sets of experiments, one in a single cluster and the other on three heterogeneous clusters.



(a) Runner.

(b) Execution service.

Figure 3.2: **Single-Cluster Experiments [W-Base]**: The CPU usage [%] of the runner (left) and the execution service (right).



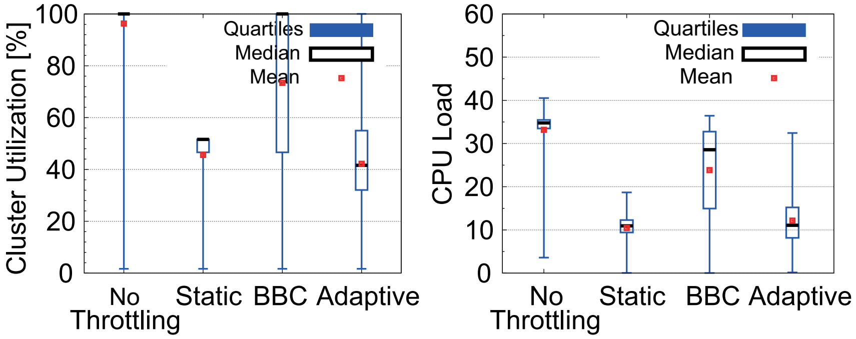
(a) The makespan.

(b) The distribution of the task execution times.

Figure 3.3: **Single-Cluster Experiments [W-Base]**: Application performance. CDF denotes cumulative distribution function.

3.4.1 Scheduling Overhead

We assess the overhead of the runner and the execution service after tuning our system to make sure that these components do not contribute to system overload. To this end, we run the W-Base workload on a single cluster (C3) without using throttling. Figure 3.2 shows the CPU usage of the runner and the execution service during this experiment. The runner and the execution service maximum CPU usage is well below 100%, with the runner having a maximum CPU usage of 10% and the execution service having a maximum CPU usage of 1%. This confirms that these components have relatively low overhead, and therefore they do not contribute to the system overload in the experiments.



(a) The basic statistical properties of the C3 utilization.
(b) The basic statistical properties of the CPU load of the C3 head-node.

Figure 3.4: **Single-Cluster Experiments [W-Base]**: System load. Experiments in the C3 cluster.

3.4.2 Results for Single-Cluster Experiments

In this section we investigate the performance of the throttling techniques presented in Section 3.2 with experiments on the C3 cluster using the W-Base workload.

We analyze the application performance and show the results in Figure 3.3. We observe that throttling improves the makespan over the system without throttling; the improvement is 40% with `Static`, 20% with `BBC`, and 18% with `Adaptive` (Figure 3.3(a)). The reason for the makespan improvements is that, without throttling, the cluster becomes fully utilized during the workload execution (see the values for `No Throttling` in Figure 3.4(a)). So, the tasks running in parallel congest the shared distributed file system and the intra-cluster network, which leads to an increase in the individual task runtimes and further to an increase in the makespan. With throttling, fewer tasks run in parallel as the runner delays the task submissions taking into account the concurrency limits, but the resulting delay is smaller than the overheads of running many tasks simultaneously. Throttling also helps individual tasks: the median task execution time is reduced by 70% with `Static`, 65% with `Adaptive`, and 40% with `BBC` over `No Throttling` (Figure 3.3(b)). Furthermore, with throttling the task execution time distribution has a shorter tail than that of `No Throttling`; at 95th percentile we observe significant improvements: 75% with `Static`, 25% with `BBC`, and 63% with `Adaptive` (see Table 3.4). Although throttling introduces additional delay for individual tasks the resulting makespan is much better than without throttling. Makespan-wise `Static` performing the best, with `BBC` and `Adaptive` having similar performance. Moreover, with `Static` and `Adaptive`, the resulting task execution performance is more consistent (has a shorter distribution tail) than without throttling.

We analyze the performance of the system and show the basic statistical properties of the C3 cluster utilization in Figure 3.4(a). We observe that `Static` and `Adaptive` reduce the median cluster utilization by 50% versus the system without throttling. However, similarly to `No Throttling`, for `BBC` the median and maximum cluster utilization are 100%, significantly higher than for `Static` and `Adaptive`; for the latter, the lower utilization is due to the fewer tasks running concurrently in the system. Although the cluster is lowly utilized with throttling, which may not be desired by system administrators, the resulting application performance is significantly better (Figure 3.3).

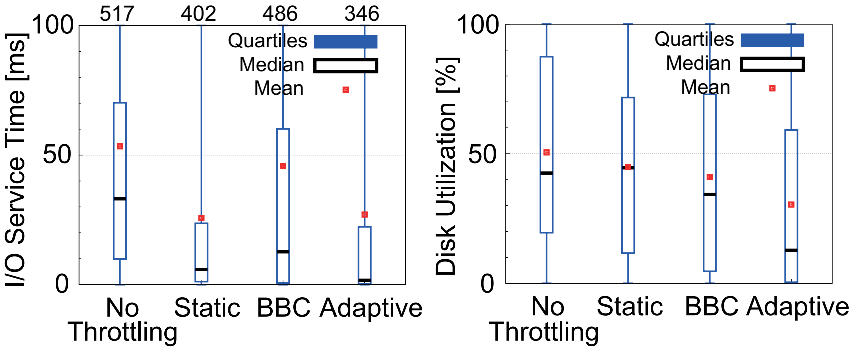
We assess the basic statistical properties of the CPU load of the C3 head-node and show the results in Figure 3.4(b)[1]. Throttling improves the median CPU load, hence the system responsiveness, substantially: 70% with `Static`, 20% with BBC, and 68% with `Adaptive`. Without throttling, the CPU load is constantly high with a median load of 35 causing the system to be unresponsive to user requests. With `Static`, the CPU load is constantly low with a median load of 10. Among the techniques, `BBC` performs the worst in terms of CPU load as the runner overloads the cluster temporarily several times during the workload execution. Nevertheless, it still performs better than `No Throttling`, with an improvement of 20% in median CPU load. `Adaptive` performs similarly to `Static`, and it performs significantly better than `BBC`. With `Static` and `Adaptive` throttling, the CPU load is much lower compared to `No Throttling`: throttling also improves the system responsiveness substantially.

We investigate the I/O performance and show the basic statistical properties of the I/O service time and the disk utilization in Figure 3.5. All techniques improve the median I/O service time over `No Throttling`: `Static` by 80%, `Adaptive` by 93%, and `BBC` by 63% (Figure 3.5(a)). Since `BBC` lets the runner temporarily overload the cluster, the maximum I/O service time with `BBC` is close to that of `No Throttling`. Finally, the disk has a lower utilization with throttling than `No Throttling`; the median disk utilization decreases by up to 70% with `Adaptive` (Figure 3.5(b)). With `No Throttling` and `BBC`, the I/O service time is highly variable, while with `Static` and `Adaptive` the I/O service time has lower variability. We conclude that, in addition to significant improvements in task execution performance, throttling also improves the I/O performance substantially.

The quality of the service offered by a system to its users (Service Level Agreement, SLA) is often quantified by the service performed on a large fraction of the work requests, such as the 95^{th} or the 99^{th} percentiles of the task execution time; we call this quantifier the *extreme performance* of the system. We compare in Table 3.4 the 95^{th} and the 99^{th} percentiles of three performance metrics–task execution time, CPU load, and I/O service time–with and without throttling; the row “Ideal Case” additionally presents the metric values for the system without overload. As expected, the overloaded system has much lower extreme performance than the ideal case. However, among the techniques the use of either of the `Static`, `BBC`, and `Adaptive` techniques leads to significant improvements

---

[1] As the C3 cluster’s head-node has an 8-core CPU, it is better if the CPU load is close to 8.



(a) The basic statistical properties of the I/O service times.

(b) The basic statistical properties of the disk utilization.

Figure 3.5: **Single-Cluster Experiments [W-Base]**: I/O performance. The values at the top of graph (a) are the maximum values observed.

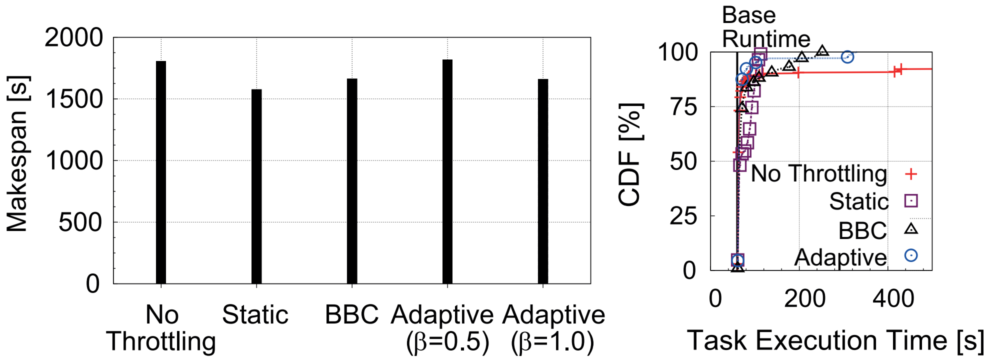
	Task Execution Time [s]		CPU Load		I/O Service Time [ms]	
	95th	**99th**	**95th**	**99th**	**95th**	**99th**
No Throttling	346	443	37	38	178	318
Static	89	109	14	17	124	262
BBC	262	309	35	36	214	354
Adaptive	127	187	22	31	147	256
Ideal Case	60		8		6	

Table 3.4: **Single-Cluster Experiments [W-Base]**: The 95th and the 99th percentiles for the task execution time, CPU load, and I/O service time metrics.

in one or more of the metrics, especially the task execution time and the I/O service time. Thus, throttling is to be preferred to `No Throttling` when extreme performance guarantees are part of the SLA. Furthermore, `BBC` delivers consistently worse extreme performance than the other techniques; the differences between `Static` and `Adaptive` illustrate the time-performance trade-offs offered by manual and automatic-and-dynamic system tuning, respectively.

3.4.3 Results for Multi-Cluster Experiments

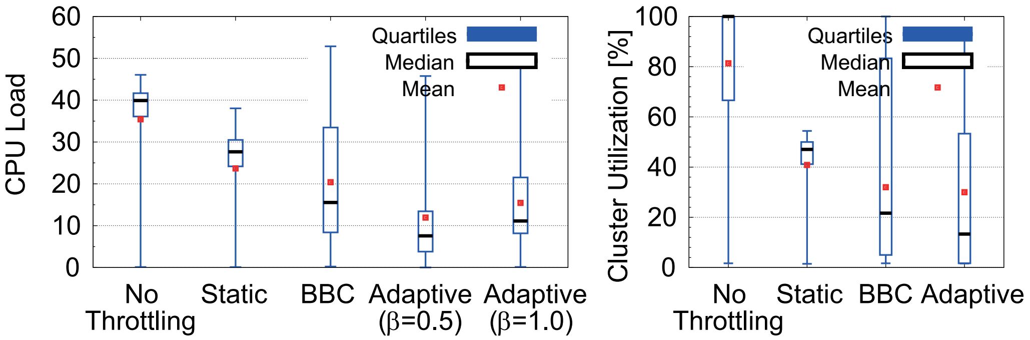
We now evaluate the performance of the throttling techniques in a multi-cluster setting. The three clusters we use (Section 3.3) are heterogeneous in terms of size and network bandwidth. We first perform experiments with our baseline workload, W-Base, using all the techniques. Then, we use, in turn, a workload with increased number of tasks (W-Task), task runtimes (W-Run), and task I/O requirements (W-IO); we assess with these



(a) The makespan.

(b) The distribution of the task execution time.

Figure 3.6: **Multi-Cluster Experiments [W-Base]**: Application performance.



(a) The basic statistical properties of the CPU load of the head-node.

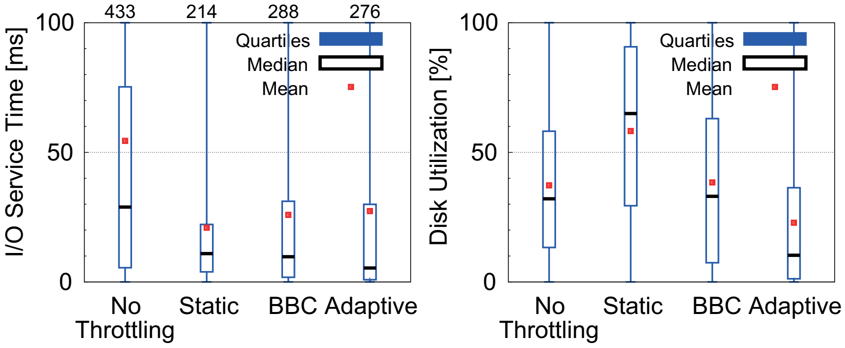
(b) The basic statistical properties of cluster utilization.

Figure 3.7: **Multi-Cluster Experiments [W-Base]**: System load for the C3 cluster.

workloads the performance of the `BBC` and `Adaptive` techniques.

We analyze the application performance and show the results in Figure 3.6. As more resources are used during this experiment, the makespan here is lower than for the single-cluster experiments (compare Figure 3.3(a) with Figure 3.6(a)). Similarly to the results obtained for single-cluster experiments, throttling noticeably improves the application performance (Figure 3.6(a)). `Static` and `BBC` improve the makespan by 13% and 8% over `No Throttling`, respectively. `Adaptive` with an average adaptation rate ($\beta = 0.5$, see Sections 3.2 and 3.3.4) provides roughly the same makespan as `No Throttling`. However, `Adaptive` with $\beta = 1.0$ provides a makespan of 1,600 ms (similar to `BBC`) and improves the application performance by 10% over `No Throttling`. All techniques improve significantly the application performance and the extreme performance of the task execution (shorter distribution tail in Figure 3.6(b)).

We investigate the performance of the system and show the basic statistical proper-



(a) The basic statistical properties of the I/O service time.

(b) The basic statistical properties of the disk utilization

Figure 3.8: **Multi-Cluster Experiments [W-Base]**: I/O performance for the C3 cluster. The values at the top of graph (a) are the maximum values observed.

ties of the cluster utilization of the C3 cluster and the CPU load of the C3 head-node in Figure 3.7. All techniques reduce the CPU load leading to better system responsiveness; `Static` by 33%, `BBC` by 40%, and `Adaptive` by 80% (Figure 3.7(a)). This improvement adds to the improvements observed for the application performance (Figure 3.6). Compared with `No Throttling`, `Adaptive` with $\beta = 0.5$ preserves the application performance (Figure 3.6(a)) while using less resources (Figure 3.7(b)), and improving the system responsiveness (Figure 3.7(a)). Moreover, with $\beta = 1.0$, although `Adaptive` yields a better performance (Figure 3.6(a)), it leads to a 50% higher CPU load over $\beta = 0.5$ (Figure 3.7(a)). Because with $\beta = 1.0$, `Adaptive` increases the concurrency limit faster than with $\beta = 0.5$, thus letting the runner overload the head-nodes. Our results show that a trade-off between the application performance and system responsiveness exists. As a consequence, while determining the values of the parameters of the throttling techniques, this trade-off should be taken into account.

We investigate the I/O performance and show the basic statistical properties of the I/O service time and the disk utilization for the C3 cluster in Figure 3.8. Throttling also helps in reducing the I/O service times: the median I/O service time is reduced by 62% with `Static`, 65% with `BBC`, and 81% with `Adaptive` over `No Throttling` (Figure 3.8(a)). Finally, in terms of the disk utilization `BBC` performs similar to `No Throttling` while `Adaptive` performs slightly better decreasing the disk utilization by 66%. Due to the heterogeneity of our testbed, although `Static` has a higher disk utilization than `No Throttling` (Figure 3.8(b)), it improves significantly the CPU load (Figure 3.7(a)) and the I/O service time over `No Throttling` (Figure 3.8(a)).

From now on, we continue our evaluation with dynamic techniques, because with `Static` the concurrency limit has to be tuned for all clusters and workloads making it an impractical solution, and we have already shown that `Adaptive` has comparable

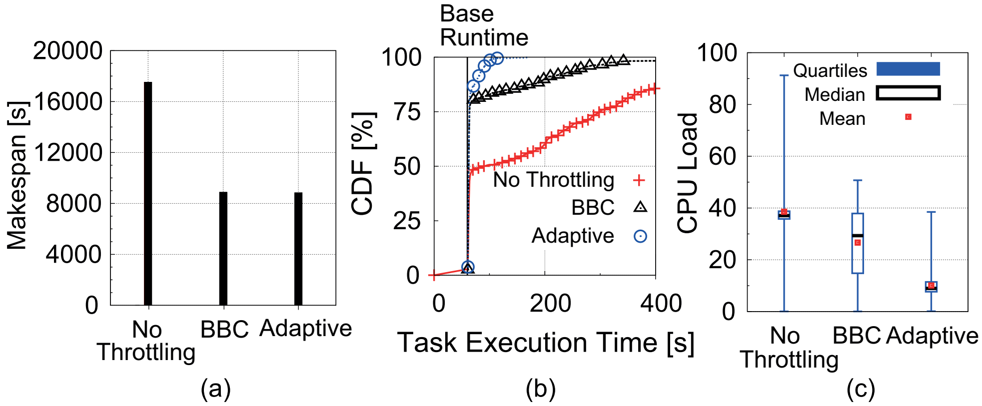
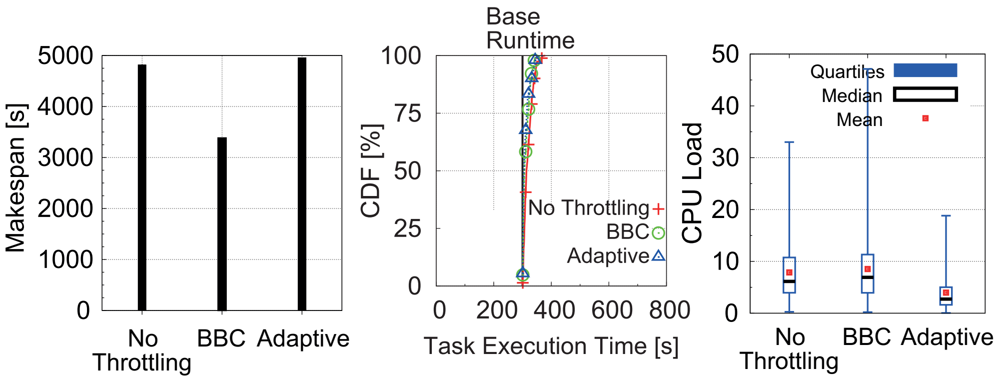


Figure 3.9: **Multi-Cluster Experiments [W-Task]**: Makespan (a), the distribution of the task execution time (b), and the basic statistical properties of the CPU load of the C3 head-node (c).

performance with `Static`. We assess the performance of the `BBC` and `Adaptive` techniques with the W-Task workload and show the results in Figure 3.9. As W-Task contains more tasks than W-Base, the overloads in all clusters are more severe. As a result, throttling improves drastically the application performance: `Adaptive` and `BBC` improve the makespan by 50% (Figure 3.9(a)) while improving the median CPU load by 22% and 78% (Figure 3.9(c)), respectively. The reasons for such a difference are the increased number of parallel I/O operations, and the increased number of simultaneous inter-cluster file transfers that put more load on the shared resources. Compared to the other experiments, with the W-Task workload the improvements in the application performance and the CPU load is higher resulting in a more responsive system. Moreover, throttling also improves the extreme performance of the task execution time (Figure 3.9(b)) leading to better performance consistency than without throttling.

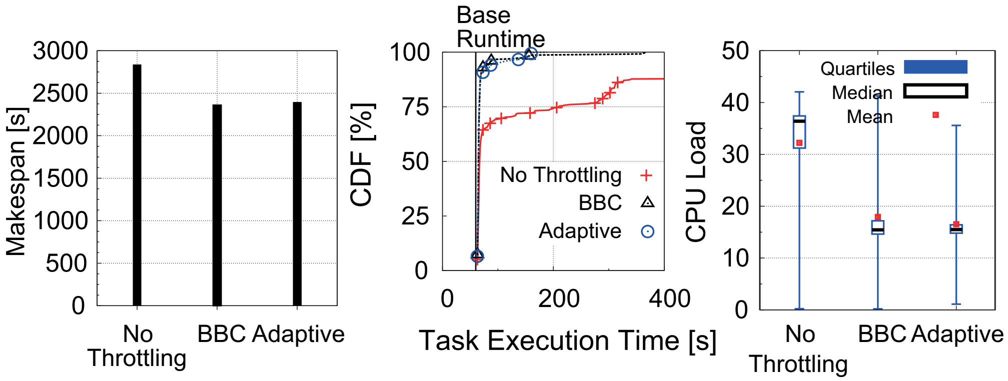
We evaluate the performance of the `BBC` and `Adaptive` techniques with the W-Run workload and show the results in Figure 3.10. Unlike the results for the W-Task workload, with `Adaptive` the makespan is roughly the same as the makespan without throttling (Figure 3.10(a)). Although `Adaptive` and `BBC` have similar task execution performance (Figure 3.10(b)), the makespan is smaller with `BBC` with a 30% improvement over `No Throttling` as tasks have higher wait times (throttling delay + queuing delay) with `Adaptive` than with `BBC`, leading to a higher makespan. Similar to the results for the W-Task workload, `Adaptive` improves the CPU load by 60% over `No Throttling` leading to better system responsiveness than `BBC` (Figure 3.10(c)). With the W-Run workload `Adaptive` leads to a similar makespan with `No Throttling` while `BBC` results in a better makespan but with a higher CPU load.

Using the W-IO workload we investigate the performance of the `BBC` and `Adaptive`



(a) (b) (c)

Figure 3.10: **Multi-Cluster Experiments [W-Run]**: Makespan (a), the distribution of the task execution time (b), and the basic statistical properties of the CPU load of the C3 head-node (c).



(a) (b) (c)

Figure 3.11: **Multi-Cluster Experiments [W-IO]**: Makespan (a), the distribution of the task execution time (b), and the basic statistical properties of the CPU load of the C3 head-node (c).

techniques and present the results in Figure 3.11. Both techniques lead to similar makespan, with an improvement of 15% over `No Throttling` (Figure 3.11(a)). Since the workload is I/O intensive, the less powerful cluster (C3) gets overloaded quickly, causing a large number of tasks to be submitted to faster clusters with both techniques yielding a similar makespan. Similarly to the results of the W-Task and W-Run workloads, both techniques improve the extreme performance of the task execution time (Figure 3.11(b)). Finally, both techniques result in similar CPU load due to the large file transfers with this workload (Figure 3.11(c)). Nevertheless, the system responsiveness is improved substantially as both techniques reduce the CPU load by 55%.

3.5 Related Work

In this section we survey prior research exploring the following overload control techniques: congestion control, admission control, scheduling, overprovisioning, and throttling.

Congestion Control is a well-researched technique for network traffic engineering; we refer to [213] for a survey of a wide range of TCP congestion control mechanisms.

Admission Control is a technique under which the amount of work accepted to a system is policy controlled. Admission control has been used in web servers to mitigate flash crowds [212], and in e-commerce systems [74] and multi-tier distributed systems [146] for overload control. Although effective, admission control can help stave off degrading response times under overload but cannot prevent it completely.

Scheduling has also been investigated as a solution to the overload control problem. In [185] authors address the transient overload problem of web servers by using the shortest remaining processing time scheduling policy. A similar study shows significant response time improvements by favoring short connections [56]. Although these studies showed improvements to the response times under transient overload, they do not evaluate the policies under permanent overloads. Previous studies [74, 192] also show that scheduling can prevent overload only to a certain extent.

Overprovisioning is a technique for handling workload fluctuations that may cause temporary overloads at bottleneck resources. Overprovisioning can solve the overload problem only to a certain extent [121], even when using overflow pools to handle transient overload [82] or dynamic overprovisioning [205]. Overprovisioning is difficult to employ for highly variable workloads—at one extreme, a system that is overprovisioned for the peak load incurs high costs, at the other, a system overprovisioned for the mean load cannot handle severe overloads.

Throttling is a technique under which the rate at which workloads are pushed through the system is controlled depending on the system load. Throttling has been used in diverse computer systems to control overload; it has been used in distributed file systems [6], resource management systems like Grid Engine and Condor [55], networks of SIP servers [96], and in cycle stealing systems for efficiently enforcing resource limits on I/O subsystems [174].

Closest to our work are the studies in networks of SIP servers [96], in cycle stealing systems [174], and in Condor DAGMan [55]. Our study is different from [96] since the workload characteristics of multi-cluster grids are significantly different from multimedia workloads [101]. In contrast to [55], we perform a more extensive evaluation; we investigate both static and adaptive throttling techniques where they only focus on static throttling, and moreover, we evaluate these techniques in a real multi-cluster grid.

3.6 Summary

Due to highly demanding and bursty workloads, overloads are inevitable in multi-cluster grids, leading to decreased system performance and responsiveness. Further motivated by our DAS multi-cluster grid, where running hundreds of tasks concurrently leads to severe overloads, in this chapter we have investigated the performance of throttling-based overload control techniques in multi-cluster grids.

Our results show strong evidence that throttling can be used for effective overload control in multi-cluster grids. In general, we have shown that throttling leads to a decrease (in most cases) or at least to a preservation of the makespan of bursty workloads, while significantly improving the extreme performance (95^{th} and 99^{th} percentiles) for application tasks leading to more consistent performance and reducing the overload of cluster head-nodes. In particular, our adaptive technique improves the application performance by as much as 50% while also improving the system responsiveness by up to 80%, when compared with the tuned multi-cluster system without throttling. Our results further indicate that our adaptive throttling technique performs similarly to static throttling, which is based on the manual tuning of our system that provides the best observed performance, and better overall than the other adaptive throttling technique investigated in this chapter.

Chapter 4

Incremental placement of interactive perception applications[*]

Multimedia recording and playback capability has been long established in the computer industry, and has become commonplace with the availability of low-cost digital cameras and recording hardware. Until recently, applications making use of audio and video data have largely been limited to recording, compression, streaming, and playback for human consumption. Applications that can directly make use of video streams, for example as a medium for sensing the environment, detecting activities, or as a mode of input from human users, are now active areas of research and development [46, 54, 140, 196]. In particular, a new class of *interactive perception applications* that uses video and other high-data rate sensing for interactive gaming, natural gesture-based interfaces, and visually controlled robotic actuation is becoming increasingly important.

Interactive perception applications pose some unique challenges to their effective implementation in real systems. First, the data rates associated with video streams are high, making it challenging to process, store, and transmit the data without loss. This problem is compounded by the ever-improving resolution and frame rates of low-cost cameras. Second, the state-of-the-art computer vision and machine learning techniques employed by interactive perception applications are often compute intensive. For example, Scale Invariant Feature Transform (SIFT) feature extraction [133], a commonly-used algorithm for finding and describing features in an image, can take over a second to run for each frame of a standard definition video on a modern processor, over 30× too slow to keep up with the video stream. Furthermore, the computation load of these algorithms is highly variable, and depends on scene content factors such as background clutter, motion in the scene, and the number of people in view. Finally, these applications have tight response

---

[*] This chapter is based on previous work published in the *ACM International Symposium on High-Performance Parallel and Distributed Computing* (HPDC'11) [228] and the *International Open Cirrus Summit 2011* [235].

time requirements. To provide a crisp, responsive user experience, interactive applications may need to ensure that the latency, from when sensor data arrive to when outputs for actuation or updates of a display are generated, is limited to 100–200 ms for each data item (i.e., video frame).

On the other hand, interactive perception applications provide ample opportunities to exploit parallelism. They are often structured as data flow graphs of processing stages, which are amenable to various parallelization techniques. Recent work [167] demonstrates that it is possible to run such applications in an interactive setting by exploiting the coarse-grained parallelism inherent to these applications, and carefully distributing their execution across multiple processors and machines. The use of parallel resources can reduce the time to execute these algorithms, but additional overheads for data transfer and coordination are introduced. The effectiveness of this approach hinges on careful allocation and scheduling of processing stages on different processors such that the latency for the distributed data flow to process each item, including processing and data transfer time, i.e., the makespan, is minimized. However, given the variability in perception workloads, it is difficult to determine a good placement of the processing stages a priori.

In this chapter we devise algorithms to *automatically* and *incrementally* place and schedule stages of an application on a set of processing nodes to minimize latency (makespan). Our system continuously monitors performance of the running application stages, and, as conditions change, adjusts the placement by migrating stages between processing nodes. We develop and implement four heuristics that perform incremental placement to minimize latency while bounding migration cost. We demonstrate the benefits of these heuristics through detailed simulations and execution on a prototype system using two real interactive perception applications. In particular, with our experiments using these applications we show that the heuristics improve the median latency by up to 36%.

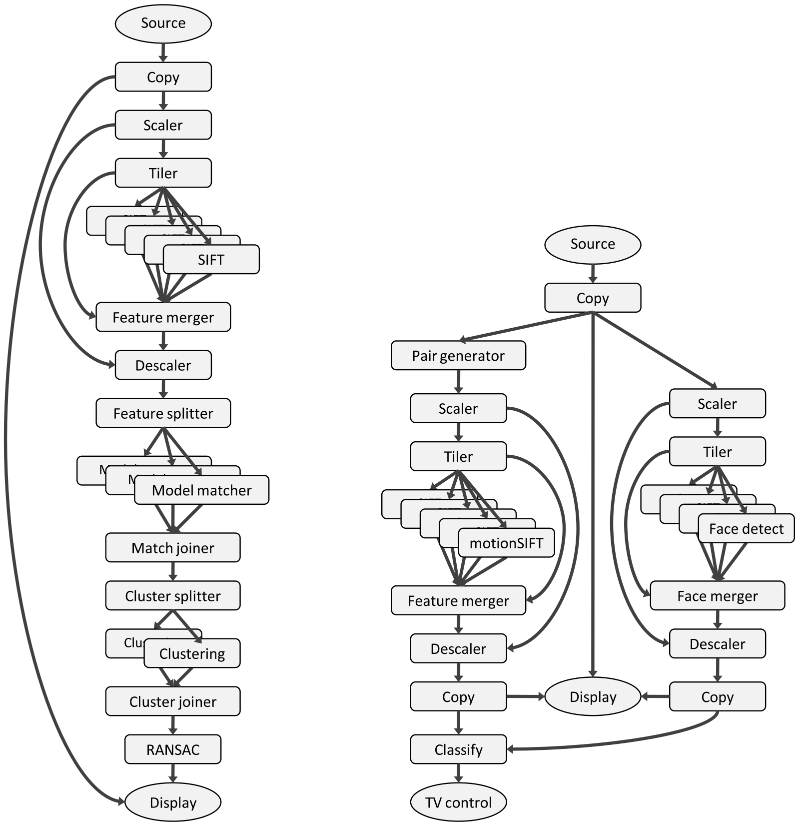
The rest of the chapter is organized as follows. Section 4.1 introduces the two interactive perception applications that we study in this chapter. Section 4.2 describes the Heterogeneous Earliest Finish Time (HEFT) scheduling heuristic on which our incremental heuristics are based. Section 4.3 presents the incremental placement problem, and Section 4.4 describes our incremental placement heuristics. Section 4.5 describes the implementation details of our system. Section 4.6 describes the experimental setup. Section 4.7 and Section 4.8 present our performance evaluation results with simulations and with experiments on a real system, respectively. Finally, Section 4.9 reviews the related work and Section 4.10 summarizes the chapter.

4.1 Interactive Perception Applications

We consider parallel interactive perception applications structured as data flow graphs. These applications usually comprise compute-intensive computer vision and machine learning algorithms, many of which exhibit coarse-grained task and data parallelism that can be exploited across machines. The vertices of such applications are coarse-grained sequential processing steps called *stages*, and the edges are connectors which reflect data dependencies between stages. The stages interact only through connectors, and share no state otherwise. Source stages provide the input data to the application, for example, as a video stream from a camera consisting of a sequence of frames. This sequence of frames flows through and is transformed by multiple processing stages, which, for example, may implement a computer vision algorithm to detect when the user performs a particular gesture. Finally, the processed data is consumed by sink stages, which then control some actuator or display information to the user. The data flow model is particularly well suited for perception, computer vision, and multimedia processing tasks because it mirrors the high-level structure of these applications, which typically apply a series of processing steps to a stream of video or audio data. In this data flow model, concurrency is explicit – stages within an application can execute in parallel, constrained only by data dependencies and available processors.

We use an application-independent runtime system [167] to distribute and execute applications in parallel on a compute cluster. The system provides mechanisms to migrate stages and set tunable parameters, including the degree of parallelism (e.g., number of data-parallel operators). Migrating a stage includes making the necessary RPCs to activate the stage on the remote node and transferring the stage state. Setting the tunable parameters of an application enables changing the application fidelity by updating the algorithm parameters, and changing the graph structure by adding new data parallel stage instances. The system also monitors application performance, and provides interfaces for extracting stage latency data. Our work extends this runtime system by adding automatic initial placement of stages, as well as the ongoing incremental adjustment of placement to maintain low application makespan as conditions change.

We study two applications in this paper. The first application, pose detection, is an implementation of an algorithm for object instance recognition and pose registration used in robotics [54]. As shown in the data flow of Figure 4.1(a), each image (frame from a single camera) first passes through a proportional down-scaler. SIFT features are then extracted from the image and matched against a set of previously constructed 3D models for the objects of interest. The features for each object are then clustered by position to separate distinct instances. A random sample consensus (RANSAC) algorithm with a non-linear optimization is used to recognize each instance and estimate its 6D pose. As this application is intended for visual servoing of a robot arm, it requires low processing



(a) Pose detection.

(b) Gesture-based TV control.

Figure 4.1: Data flow for our applications.

latency for each frame, with a goal of 50 ms.

The second application, TV control, provides an interface to control a television via gestures [45]. Each video frame is sent to two separate tasks, face detection and motion extraction, shown in Figure 4.1(b). The latter accumulates frame pairs, and then extracts SIFT-like features that encode motion in addition to appearance. These features, filtered by the positions of detected faces, are aggregated over a window of frames using a previously generated codebook to create a histogram of occurrence frequencies. The histogram is treated as an input vector to a classifier trained for the control gestures. For this application, low latency, on the order of 100 ms, for processing each frame is needed to achieve a responsive user interface. We note that in both of these applications, the processing times of the vision algorithms are the primary contributors to latency, though other sources, such as network transfer overheads, are also non-negligible.

4.2 The HEFT Algorithm

To dynamically and incrementally place perception applications structured as data flow graphs on compute nodes, we propose four heuristics which use the Heterogeneous Earliest Finish Time (HEFT) algorithm [202] as a building block. HEFT is a well-studied list scheduling heuristic for multiprocessor scheduling of an application task graph. It has low complexity, is easy to implement, and performs well compared with many heuristics in the literature. We describe the steps of the algorithm in turn:

- **Vertex/Edge Weight Assignment:** Initially, HEFT sets the computation costs of stages and the communication costs of edges with mean values computed over all processors and data links in the system.
- **Task Prioritization:** Then, HEFT assigns each stage v_i an upward rank value $rank_u(v_i)$, which is the length of the critical path from stage v_i to the exit stage including v_i's computation cost. The stages are sorted by decreasing order of upward rank, with ties broken randomly.
- **Processor Selection:** Finally, HEFT traverses the list of stages in decreasing order of upward rank and *places* stage v_i on processor p_k that minimizes the stage's Earliest Finish Time EFT(v_i,p_k), and *schedules* each stage using an insertion-based policy. With this policy a stage may be inserted in a slot of the schedule of the processor between two already scheduled stages on this processor if the length of the slot is long enough for the new stage.

The complexity of HEFT is $\mathrm{O}(ep)$ where e is the number of edges in the graph and p is the number of processors. For dense graphs, since $e = \mathrm{O}(v^2)$, the complexity of HEFT is $\mathrm{O}(v^2 p)$, where v is the number of vertices in the graph.

4.3 Problem Formulation

In this section we first describe the initial placement problem, and then we describe the incremental placement problem which is the main focus of this study.

4.3.1 The Initial Placement Problem

We represent an interactive perception application as a data flow graph $G = (V, E)$, comprising a set of processing stages $V = \{v_1, ..., v_n\}$ and a set of data dependencies $E = \{e_{ij} | v_j \text{ requires data from } v_i\}$. The application runs on a possibly heterogeneous cluster with m processors $n_j, j = 1, ..., m$ with capacities (number of cores) $c_j, j = 1, ..., m$.

Stage v_i has weight w_i representing its execution time, and edge e_{ij} has weight w_{ij} representing the latency of sending output data from stage i to stage j. If stage i and

j are running on the same processor, we set $w_{ij} = 0$. Each stage v_i has a processing demand (number of cores) d_i. Let $P = \{p_{ij}\}$ be a placement where $p_{ij} = 1$ if stage v_i is placed on processor n_j and $p_{ij} = 0$ otherwise. The *initial placement problem* is to find a placement P that minimizes the makespan (latency) of the application subject to the capacity constraints $\sum_{i=1}^{n} p_{ij} d_i < c_j, j = 1, ..., m$.

4.3.2 The Incremental Placement Problem

The performance of a running application can be affected by a variety of factors. For example, the runtime of a stage may change due to a change in the input data or the values of tunable parameters, stages on a processor may slow down due to the arrival of background load, and data parallel instances of stages may be added or removed from the application graph. These perturbations can affect latency enough to warrant revising the initial placement.

Re-placing stage i involves migrating it from its current processor k to another processor l with non-zero migration cost m_{ikl}. Given an initial placement P, the *incremental placement problem* is to find a placement P' that minimizes the makespan of the application subject to the migration cost constraint $\sum m_{ikl} < M$, where M is the migration cost bound, and the capacity constraints $\sum_{i=1}^{n} p_{ik} d_i < c_k, k = 1, ..., m$. The reason to bound the migration cost is that the migration of a stage can be noticeable by the user as a transient increase in latency.

We now describe the requirements for an incremental placement algorithm for interactive perception applications which make this problem non-trivial:

- Stage execution times can vary due to changes in the input data, the values of tunable parameters, or the arrival of background load. Stage input and output data amounts may also vary. The incremental placement algorithm must accept changes to node and edge weights.
- The incremental placement algorithm will be executed repeatedly to revise the existing placement, therefore it should be efficient.
- When modifying an existing placement by migration, the incremental placement algorithm must keep the disruption (churn) in the existing placement within a given migration cost bound.
- The degree of parallelism in an application graph can be changed dynamically by setting tunable parameters (see Section 4.1), which changes the number of stages in the graph. The incremental placement algorithm must deal with these structural changes.
- Stages may have constraints on where they can be placed. For example, a stage may require a specific resource (e.g., a camera), or mutual exclusion (e.g., due to non-

thread-safe libraries). The incremental placement algorithm must take into account such constraints.

In the general case, assigning tasks to processors subject to precedence constraints is known to be NP-complete [84]. Therefore, either optimal enumerative search based methods or approximate heuristics can be used. In practice, enumerative techniques are very expensive even for very small graphs on a few processors, and intractable for modest systems (e.g. 25 stages on 10 processors). Thus, we take the latter approach and propose four heuristics which we present in the next section for solving the incremental placement problem.

4.4 Incremental Placement Heuristics

To perform incremental adjustment to placement, we have developed four heuristics all of which use HEFT as their primary building block. To this end, we modify the original HEFT algorithm such that it can accept as input a partial placement of stages. If a stage is already placed, the algorithm does not do any placement but only schedules it. Therefore, the runtime of HEFT for the case when there are already placed stages will be less than for the case when none of the stages are placed. We use this modified HEFT algorithm to build our incremental heuristics.

HEFT-MS (One Move/Swap): We define a move as the migration of a stage from its current processor to another processor and we define a swap of a pair of stages s_i and s_j as the migration of s_i to the processor that s_j is currently running on and vice versa. The main characteristic of this greedy algorithm is that it tends to minimize the migration cost by limiting the number of migrations either to a single stage or a pair of stages (swap). This algorithm finds the best single move or the best single swap operation by searching the whole search space of possible single moves and swaps. Single moves are explored before single swaps since the migration cost of a single move will likely be less than that of a single swap. At the end, the algorithm updates the current placement by applying the operation (either a single move or a single swap) with the smaller makespan. The algorithm consists of two steps: try_one_moves and try_one_swaps. So the complexity of the algorithm is the maximum of the complexities of these two steps. The complexity of the try_one_moves step is $\mathrm{O}(vep^2)$ for sparse graphs and $\mathrm{O}(v^3p^2)$ for dense graphs. The try_one_swaps has a complexity of $\mathrm{O}(v^2ep)$ for sparse graphs and $\mathrm{O}(v^4p)$ for dense graphs. Therefore, the complexity of HEFT-MS is $\mathrm{O}(\max(vep^2, v^2ep))$ for sparse graphs and $\mathrm{O}(\max(v^3p^2, v^4p))$ for dense graphs.

HEFT-Iter: The goal of this algorithm is to improve the quality of the schedule (makespan) as much as possible within the specified migration cost bound. Therefore, this algorithm may result in higher migration costs compared to HEFT-MS which is good at minimizing the migration cost. This algorithm is an iterative greedy algorithm which runs HEFT-MS iteratively until the migration cost bound is exceeded or no further improvement to the current schedule is possible. As a result, this algorithm may perform multiple moves and swaps, compared to HEFT-MS which is limited to a single move or swap. Assuming that HEFT-Iter runs HEFT-MS T times, the complexity of this algorithm is $T \cdot \mathrm{O}(\max(vep^2, v^2ep))$ for sparse graphs and $T \cdot \mathrm{O}(\max(v^3p^2, v^4p))$ for dense graphs. For the worst case when the migration cost bound is infinite, T is $\mathrm{O}(v)$, hence the complexity of this algorithm is $\mathrm{O}(\max(v^2ep^2, v^3ep))$ for sparse graphs and $\mathrm{O}(\max(v^4p^2, v^5p))$ for dense graphs.

HEFT-Relax: The main goal of this greedy algorithm is to provide a lightweight alternative to the other heuristics. At each iteration, each stage is examined by relaxing its placement and letting HEFT place and schedule the stage. The relaxation that produces the smallest makespan is selected. If the resulting migration cost is less than the migration cost bound and the resulting makespan improves the minimum makespan found so far, then the placement of the relaxed stage is updated, and the algorithm proceeds with the next iteration. The algorithm terminates if the total migration cost exceeds the given bound or if the minimum makespan found so far can not be improved. In the worst case when the migration cost bound is infinite, the algorithm terminates after relaxing all stages. Therefore, the complexity of the algorithm is $\mathrm{O}(v^2ep)$ for sparse graphs and $\mathrm{O}(v^4p)$ for dense graphs.

HEFT-DRelax (Dual Relax): This algorithm is an iterative greedy algorithm similar to HEFT-Relax. Until the migration cost bound is exceeded, at each step HEFT-DRelax finds the best pair of stages, stages that improve the minimum makespan found so far if relaxed, and relaxes the placement of each pair and lets HEFT place and schedule these stages. If a feasible relaxation can be found, that is the resulting migration cost is less than the migration cost bound and the resulting makespan improves the minimum makespan found so far, then the placements of the relaxed stages are updated, and the algorithm proceeds with the next iteration. The algorithm terminates if the total migration cost exceeds the given bound or if the minimum makespan found so far can not be improved. In contrast to HEFT-Relax, this algorithm relaxes a pair of stages at each step. Therefore, there is more room for the HEFT algorithm to improve the current schedule while the resulting migration cost of this algorithm may be higher. In the worst case when the migration cost bound is infinite, the algorithm terminates after finding for each stage the best stage to relax in pair, therefore the complexity of the algorithm is $\mathrm{O}(v^3ep)$ for sparse graphs and $\mathrm{O}(v^5p)$ for dense graphs.

4.5 Implementation Details

We implemented placement functionality as an extension of the system described in Section 4.1. A placement manager runs within a management process that configures and launches applications. As such, the execution time of the placement algorithms is not directly noticeable by the user.

The placement manager runs periodically, where the interval is a configurable parameter with a default of two seconds. During each iteration, the placement manager first retrieves performance data for each stage from the servers running the application. This data is logged by the underlying runtime system, and includes information such as stage execution times and input and output data amounts. This data is used to construct a model of the running application. The placement manager then runs HEFT to determine if the placement can be improved, and if so, whether the necessary changes fall within the migration bound. If not, it then runs HEFT-Iter.

The resulting changes are effected using the stage migration functionality provided by the underlying runtime system. Migrating a stage involves activating a new instance of the stage on its destination server, creating connections to its sources and sinks, synchronizing the transfer of the connections and stage state to the new instance, and teardown of the old instance. Except for state transfer, which occurs between stage executions, these operations run in the background while stages continue to execute. Thus, only the cost of state transfer is manifest to the application.

Although the state transferred and cost is generally small, an improved underlying runtime system employing live state migration with triggered change propagation can hide this latency as well. Implementing such a system is beyond the scope of this paper. In both the hypothetical and actual systems, it is important to minimize churn, and mitigate the extra resource consumption of the migrating stages.

4.6 Experimental Setup

In this section we first describe the workloads that we use in our experiments, and then we describe the performance metrics that we report as a result of our experiments.

4.6.1 Workloads

We evaluate the incremental placement heuristics described in Section 4.4 via simulations and real experiments. For the experiments, we use the applications presented in Section 4.1. For the simulations, we create a generic model for these applications to do an extensive performance evaluation with a diverse set of synthesized workloads. To ensure these are realistic, we first model the stage runtime, the size of the stage output and the

	Pose detection					TV Control				
	Model	**mean**	**std-dev**	**D**	A^2	**Model**	**mean**	**std-dev**	**D**	A^2
Stage Runtime [s]	LN(-5.55,2.52)	0.09	2.25	0.09	50.35	LN(-7.49,2.38)	0.009	0.16	0.07	24.54
Stage State Size [bytes]	U(40.52,62.11)	51.31	6.23	0.27	1117.40	N(53.75,6.13)	53.75	6.13	0.15	78.05
Stage Output [bytes]	W(2.03E+5,0.26)	3.66E+6	26.85E+6	0.20	288.38	W(9353.5,0.19)	1.64E+6	30E+6	0.24	496.97

Table 4.1: **Model Parameters**: The parameters for the best fitting distributions with corresponding mean and standard deviation values, and D and A^2 statistics that show how well the model fits the empirical data with the KS and AD goodness of fit tests. LN(μ,σ^2), U(a,b), N(μ,σ^2), and W(λ,k) stand for the LogNormal, Uniform, Normal, and Weibull distributions, respectively.

size of the stage state data (for modeling the migration cost) for all stages of the pose detection and the TV control applications (Section 4.1). We use the maximum likelihood estimation method to fit the well-known Log-normal, Pareto, Weibull, Beta, Gamma, Uniform, Normal, and Exponential distributions to empirical data collected while running the applications. Table 4.1 shows the parameters for the best fitting distributions, their corresponding means and standard deviation values, and D and A^2 statistics that show how well the model fits, as assessed with the Kolmogorov-Smirnov (KS) and Anderson-Darling (AD) tests with a significance level of 0.05. Smaller D and A^2 values denote a better fit. As the two models are similar, for simplicity, we only use the pose detection model parameters in our simulations. Finally, to model execution on different processors, we scale the stage runtime by the ratio of the processor clock rates.

We then synthesize application graphs composed of a desired number of the realistically-modeled stages using a custom graph generator. We start generating a random graph with a single root stage and continue with the following operations until the graph has the specified number of stages:

- **Vertical split operation** adds a single child to a stage. A uniform random value is generated in the range [0,1], and a stage is split vertically if the generated value is less than the vertical split probability parameter, v.
- **Horizontal split operation** adds several children to a stage, and a stage that is a child of all these children stages. A uniform random value is generated in the range [0,1], and a stage is split horizontally if the generated value is less than the horizontal split probability parameter, h. The number of children added is a uniform random value in the range [1,f], where f is the maximum horizontal split fanout parameter.

After vertical and horizontal split operations are performed, the stages without children are terminated with a single exit stage. The reason that we use horizontal and vertical split operations instead of well-known random graph models like Erdös-Rényi [76] is to create subgraphs that model data parallel instances in real applications well. In our performance evaluation, we use random graphs comprising 20, 50, 100 or 200 stages, with

25 instances of each size. Unless otherwise stated, we use $v = 0.3$ and $h = 0.3$, but we also evaluate the impact of the parameters v and h on the performance of the heuristics in Section 4.7.1. Finally, for the maximum horizontal split fanout parameter we use a value of 5 which we think is representative of our real perception applications. We note that our synthetic workloads have stage characteristics close to those of the real applications, but have generally much larger and more varied application graphs, which results in much longer makespans than the two real applications.

4.6.2 Performance Metrics

We evaluate both the quality of the schedules calculated by the heuristics and the runtime of these heuristics. To this end, we use the following metrics:

- **Runtime [μs/ms]:** The time it takes for a heuristic to schedule a given application graph.
- **Makespan [ms]:** Total time from the start of the root stage to the end of the exit stage of the graph for a single frame. Makespan is also called latency or schedule length.
- **Migration Cost [ms]:** The time it takes to migrate a stage from its current processor to a new processor. The migration cost includes the time for RPC and the time it takes to transfer the stage's state to the new processor.

In our experiments, we report the averages for these performance metrics calculated over all input graph instances.

4.6.3 Testbed

We run both the simulations and the real system experiments on the Open Cirrus testbed [17]. For the simulations (Section 4.7), we use a machine with 8 GB main memory and with an Intel® Xeon® 8-core CPU running at 2.8 GHz. We simulate cluster topologies having a specified number of processors each with a single core and with a random clock frequency in the range [1.6-3.6] Ghz to model the heterogeneity. In addition, to model the cluster that we use in the real system experiments, we simulate a cluster where machines are connected with 1 Gbps Ethernet links. For the real system experiments (Section 4.8), we use a cluster of 15 processing nodes in the Open Cirrus testbed connected via a 1 Gbps Ethernet switch. Each node in the cluster has 8 GB main memory and an Intel® Xeon® 8-core CPU running at 2.8 GHz. Note that we use physical machines, not virtual machine instances that are typically used on Open Cirrus.

4.7 Simulation Results

In this section we investigate the performance of the incremental heuristics of Section 4.4 with simulations. First, we evaluate the makespans of schedules produced by the incremental heuristics using diverse workloads and under different perturbation scenarios. Then, we evaluate the runtimes and scalability of the HEFT algorithm and our incremental heuristics. As we will see, the makespans obtained with our heuristics are very similar to those obtained with plain HEFT, but the migration cost is very much lower as desired, which comes at the price of much longer runtimes of the heuristics compared to HEFT.

4.7.1 Application Latency

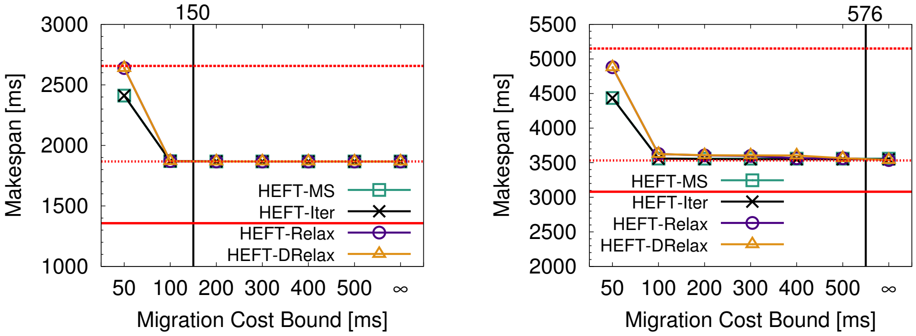
We first evaluate the quality of the schedules, in terms of makespan, the incremental heuristics calculate. To this end, after the initial placement is done with HEFT, we evaluate the incremental placement heuristics in the three perturbation scenarios described in Section 4.3.2, which we model in the following way:

- **Perturb a random stage:** We increase the runtime of a random stage by a random factor which is uniformly distributed between 1 and 10. This uniform random factor is observed in real perception applications where the runtime of stages may change significantly during execution due to changes of the input data or the values of tunable parameters.
- **Perturb a random processor:** We perturb all stages on a random processor, and we increase the execution time of all stages on that processor by a random factor which is uniformly distributed between 1 and 10. This perturbation corresponds to the introduction of background load on the processor.
- **Add a new data parallel stage instance to the graph:** We add a new data parallel instance of a stage to an application during execution. This change occurs as a result of setting a tunable parameter.

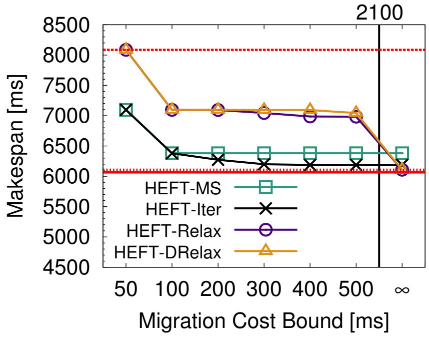
To assess the quality of the schedules that the incremental algorithms calculate, we compare the makespan of these schedules with a *baseline* makespan, which we calculate by rerunning the HEFT algorithm on the updated graph. For our simulations we use general random graphs created using the generator described in Section 4.6.1 which may have significantly different sizes and structures than those of real perception applications. Therefore, our goal with the simulations is to assess the quality of the heuristics with diverse abstract graphs rather than meeting the latency objectives stated in Section 4.1.

Perturb a random stage: small-scale settings

Figure 4.2 shows the results of the simulations with relatively small-scale settings: graphs having 20, 50 or 100 stages, and a cluster of 16 processors. There are three horizontal



(a) 20 Stages (b) 50 Stages



(c) 100 Stages

Figure 4.2: Makespan for the simulations with various graph sizes and 16 processors. Horizontal lines: makespan without adjustment (top), with rerunning HEFT (middle), and the initial makespan (bottom). Vertical lines: migration costs of rerunning HEFT – 150 ms (a), 576 ms (b), and 2100 ms (c).

lines in the graph. The top line is the makespan without adjustment, that is, we let the application run after a perturbation and do not re-place the stages to other processors. The middle line is the makespan obtained by rerunning the HEFT algorithm on the updated graph (the baseline). Finally, the bottom horizontal line is the initial makespan, which is the makespan before perturbation.

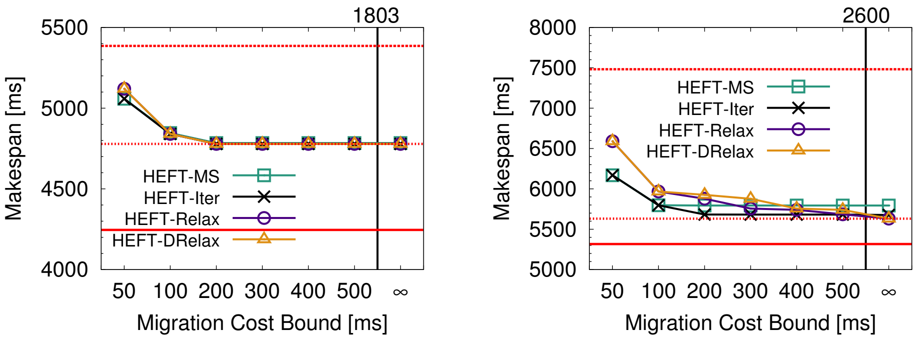
For 20 stages (Figure 4.2 (a)), the incremental heuristics perform similarly; they converge to the baseline relatively fast when the migration cost bound is 100 ms. We observe 30% improvement with the incremental heuristics (doing adjustment) compared to the case without adjustment: *it is worth adjusting the schedule*. For this experiment, rerunning HEFT has an average migration cost of 150 ms, and all heuristics converge to the baseline solution with a migration cost of 100 ms, therefore, having less migration (churn) compared to rerunning HEFT.

For 50 stages (Figure 4.2 (b)), similar to the results with 20 stages, we observe 31% improvement with adjustment over the case without adjusting the schedule. For this experiment, none of the heuristics converge to the baseline solution, however all heuristics are less than 1% off the baseline solution when migration cost bound is 500 ms causing less churn than rerunning HEFT, which has a cost of 576 ms. When the migration cost bound is 100 ms, all heuristic are less than 2% off the baseline solution: incremental heuristics provide roughly the baseline makespan with significantly lower (80%) migration costs than rerunning HEFT. This result shows how well the incremental heuristics address the trade-off between the cost of migration and the resulting makespan.

For 100 stages (Figure 4.2 (c)), although HEFT-Relax and HEFT-DRelax both converge to the baseline solution, HEFT-DRelax performs slightly better when the migration cost bound is larger than 200 ms. HEFT-Iter performs slightly (%3) better than HEFT-MS, and they both do not converge to the baseline solution. The reason why these heuristics do not converge to the baseline solution is that all heuristics make locally optimum decisions, and they do not consider all permutations of moves, therefore they may not be able to find the global optimum solution. So, the resulting schedules can be far from the global optimum schedule. Nevertheless, HEFT-Iter and HEFT-MS are slightly off the baseline when the migration cost bound is infinite; by 1% and 4%, respectively. Although HEFT-Iter is slightly off the baseline when migration cost bound is infinite, it performs the best for other migration cost bounds. For this experiment, the average migration cost for rerunning the HEFT algorithm is around 2100 ms. HEFT-Relax and HEFT-DRelax, which converge to the baseline solution, have a migration cost of 2014 ms and 2420 ms, respectively. However, HEFT-Iter is only 1% off the baseline solution with a migration cost of only 400 ms causing significantly less churn in the system.

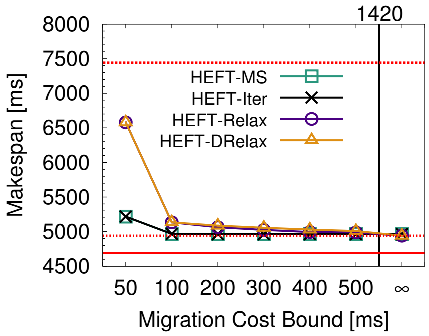
Perturb a random stage: large-scale settings

Figure 4.3 shows the results of the simulations with relatively large-scale settings; graphs having 100 stages and for various cluster sizes. For 32 processors (Figure 4.3 (a)), incremental heuristics perform similarly; all heuristics converge to the baseline solution when the migration cost bound is 200 ms. For this experiment, adjusting the schedule improves the makespan around 12% over the case without adjustment. Furthermore, average migration cost of rerunning HEFT is 1803 ms while HEFT-MS is less than 1% off the baseline with only 80 ms of migration. Similarly, HEFT-Relax and HEFT-DRelax converge to the baseline solution with roughly 150 ms of migration, therefore significantly reducing the churn in the system. However, HEFT-Iter converges to the baseline solution with 250 ms of migration which is larger than the other heuristics. The reason is that, HEFT-Iter may perform many migrations for very slight makespan improvements increasing the resulting migration cost. For 64 processors (Figure 4.3 (b)), rerunning HEFT has a migration cost of 2600 ms while HEFT-Relax and HEFT-DRelax provide the same makespan with a migration cost of less than 600 ms. Similar to other experiments, it is worth adjusting



(a) 32 processors

(b) 64 processors



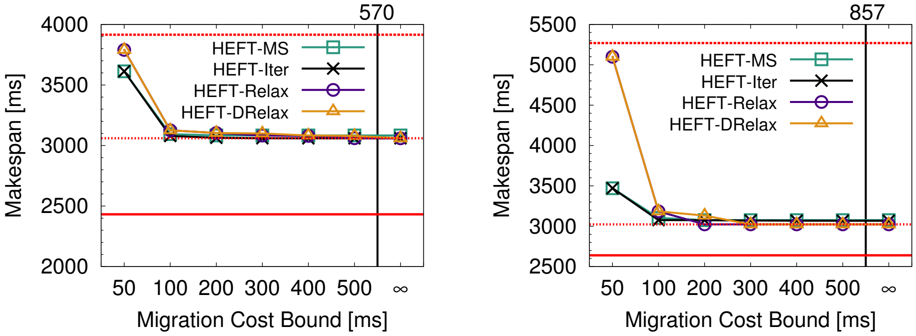
(c) 128 processors

Figure 4.3: Makespan for the simulations with 100 stages and various cluster sizes. Horizontal lines: makespan without adjustment (top), with rerunning HEFT (middle), and the initial makespan (bottom). Vertical lines: migration costs of rerunning HEFT – 1803 ms (a), 2600 ms (b), and 1420 ms (c).

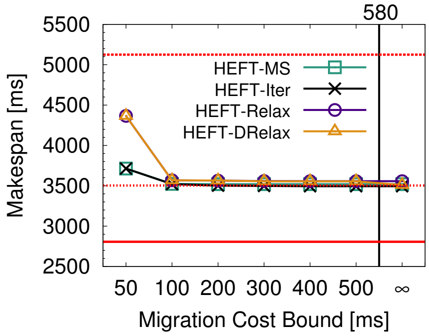
the schedule since adjustment improves the makespan around 25% compared to the case without adjusting the schedule. Although, HEFT-Iter is slightly off the baseline solution (1%) for infinite migration cost bound, it performs the best for other migration cost bounds. Finally, for 128 processors (Figure 4.3 (c)), none of the heuristics converge to the baseline solution; all heuristics are less than 2% off the baseline. For this scenario, rerunning HEFT has an average migration cost of 1420 ms while HEFT-Iter and HEFT-MS are less than 2% off the baseline solution with 130 ms and 60 ms of migration, respectively. We conclude that incremental heuristics significantly improve the makespan compared to the case without adjusting the schedule while reducing the resulting churn noticeably.

Perturb a random stage: workload parameters

Figure 4.4 shows the results of the simulations with different communication to computation ratios (CCR). Communication to computation ratio is the ratio of the average communication cost of a graph to its average computation cost. A low CCR for a graph



(a) $CCR = 0.4$ (b) $CCR = 1.6$

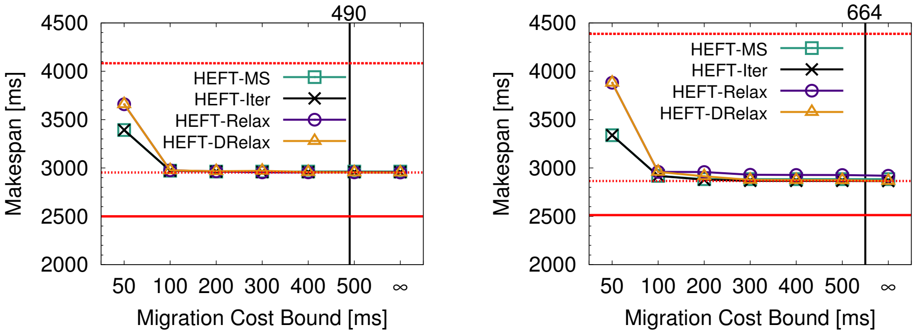


(c) $CCR = 2.4$

Figure 4.4: Makespan for the simulations with 50 stages, 32 processors, and various communication to computation ratios (CCRs). Horizontal lines: makespan without adjustment (top), with rerunning HEFT (middle), and the initial makespan (bottom). Vertical lines: migration costs of rerunning HEFT – 570 ms (a), 857 ms (b), and 580 ms (c).

indicates that the application is compute intensive. To perform simulations with different CCR values, we artificially scaled the communication requirements of the stages for which we use the statistical model that we describe in Section 4.6. Heuristics perform similar to other scenarios across difference CCR values; adjusting the schedule with the incremental placement heuristics improves the makespan by 21%, 42% and 31% for increasing CCR values of 0.4, 1.6, and 2.4, respectively. In addition, for all CCR values, heuristics are less than 2% off the baseline when they do not converge to the baseline solution. Moreover, HEFT-Iter is able to find schedules that are less than 2% off the baseline solution with significant improvements in churn, as much as 70%.

Finally, Figure 4.5 shows the results of the simulations when using different horizontal and vertical split probabilities for generating the random graphs (see Section 4.6.1). A higher vertical split probability means less degree of parallelism and a smaller branching factor for a graph. For higher h values, since the degree of parallelism increases the



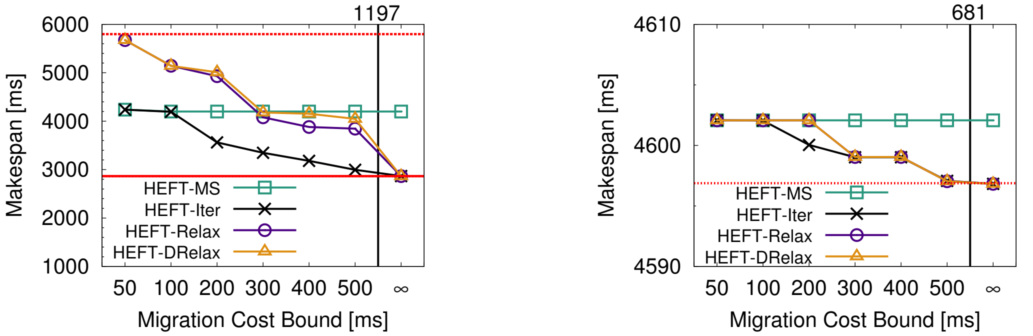
(a) $v = 0.1, h = 0.7$ (b) $v = 0.7, h = 0.1$

Figure 4.5: Makespan for the simulations with 50 stages, 32 processors, and various horizontal and vertical split probabilities. Horizontal lines: makespan without adjustment (top), with rerunning HEFT (middle), and the initial makespan (bottom). Vertical lines: migration costs of rerunning HEFT – 490 ms (a) and 664 ms (b).

resulting makespan decreases. The initial makespan values in both scenarios are close but not identical; the initial makespan for $v = 0.1$ and $h = 0.7$ is 2499 ms, and for $v = 0.7$ and $h = 0.1$ the initial makespan is 2513 ms. For $v = 0.7$ and $h = 0.1$, adjusting the schedule with the incremental placement heuristics improves the makespan by 34% and the improvement is 27% for $v = 0.1$ and $h = 0.7$. For higher h values, heuristics perform similarly, and for lower h values HEFT-Iter and HEFT-DRelax perform slightly better, and converge to the baseline solution eventually while HEFT-MS and HEFT-Relax are slightly off the baseline solution (1%). We conclude that although different graph structures have an impact on the convergence speed and the degree of parallelism of the graph, the behavior of the heuristics are similar to the other scenarios that we investigate.

Perturb a random processor

Figure 4.6(a) shows the makespan for the simulations with graphs of size 50 stages and a cluster of 32 processors where a random processor is perturbed. The top horizontal line shows the makespan without adapting the schedule. Since the initial makespan and the makespan calculated with HEFT are very close, they are overlapping and shown as a single horizontal line at the bottom. For this experiment, 11 stages were affected on average after the perturbation which in turn increases the makespan of the application significantly. In addition, affecting the execution of many stages increases the chances of the heuristics to perform better adaptation. Therefore, adapting the schedule after perturbation improves the makespan noticeably, by roughly 50%, over the case without adaptation. We observe that for small migration costs HEFT-Relax and HEFT-DRelax solutions are close to the case without adaptation. However, these two heuristics are able to improve their solutions



(a) Perturb a random processor

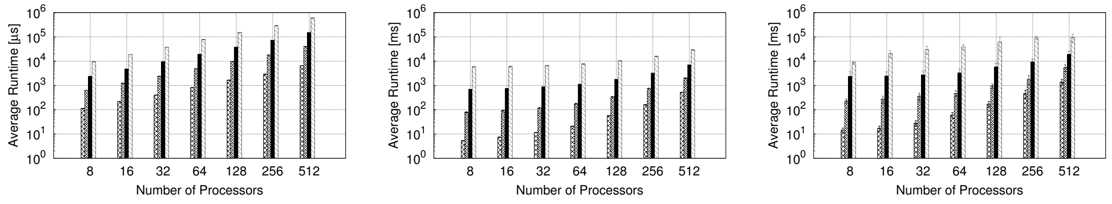
(b) Add a data parallel stage instance to the graph

Figure 4.6: Makespan for the simulations with 50 stages and 32 processors where a random processor is perturbed (a) and a data parallel instance of a stage is added to the graph (b). For the graph on the left, horizontal lines: makespan without adjustment (top), and the makespan with rerunning HEFT and the initial makespan (bottom). Rerunning HEFT has a migration cost of 1197 ms (vertical line). For the graph on the right, the horizontal line shows the makespan obtained with rerunning the HEFT algorithm which has a migration cost of 681 ms (vertical line). The initial makespan (4755 ms) is not shown for better visibility.

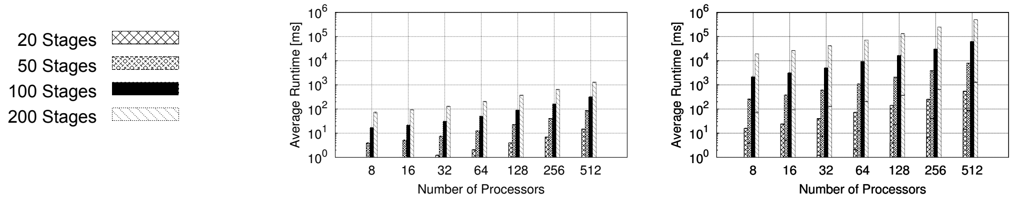
with increasing migration costs. Eventually, all heuristics but HEFT-MS converge to the baseline solution while HEFT-MS is 46% off the baseline since HEFT-MS is not able to adapt the schedule even with increasing migration cost bounds. Other heuristics are able to achieve the baseline performance with smaller migration costs compared to the case with rerunning the HEFT algorithm. In particular, for this experiment rerunning HEFT has a migration cost of 1197 ms, and HEFT-Iter achieves the same performance with a migration cost of 689 ms reducing the churn in the system by 42%.

Add a data parallel stage instance to the graph

Figure 4.6(b) shows the makespan for the simulations with graphs of size 50 stages and a cluster of 32 processors where a new data parallel instance of a stage is added to the graph during execution. For this scenario there is not a case without adjustment since the new data parallel instance must be placed and scheduled. In addition, we do not show the initial makespan for better visibility since the initial makespan is 4755 ms. We show the makespan calculated by rerunning HEFT as a horizontal line in the graph. We observe that the makespan that we obtain after rerunning the HEFT algorithm is less than the initial makespan, as expected, since when a new data parallel instance of a stage is added, the same amount of data is processed by more stage instances which reduces the latency. For



(a) HEFT (b) HEFT-MS (c) HEFT-Iter



(d) HEFT-Relax (e) HEFT-DRelax

Figure 4.7: Average runtime of the HEFT algorithm and the incremental heuristics with graphs and clusters of different sizes. The vertical axis has a logarithmic scale.

this scenario, rerunning HEFT has a migration cost of 681 ms and all heuristics except HEFT-MS converge to the baseline solution; HEFT-Iter with a migration cost of 185 ms and HEFT-DRelax and HEFT-Relax with a migration cost of 358 ms reducing the resulting churn in the system. When the migration cost bound is infinite, HEFT-MS is roughly 1% off the baseline solution with a migration cost of only 42 ms. In addition, for this scenario HEFT-MS is not able to improve the schedule as the migration cost bound increases which shows that for some scenarios HEFT-MS is not as good as other heuristics at adjusting the schedule while it is good at minimizing the churn in the system.

4.7.2 Algorithm Scalability

Before evaluating the runtime of the heuristics, we first profiled and manually optimized our code, then we used the `-O3` flag of the `g++` compiler for further compiler optimizations. During this analysis we set the migration cost bound to infinity to characterize the worst case performance of the heuristics.

Figure 4.7(a) shows the runtime of the HEFT algorithm, which is in microseconds, and Figure 4.7(b-e) show the runtimes of the incremental heuristics, which are in milliseconds. Note that even though HEFT may be much faster than the incremental placement heuristics, it may result in very much churn, that is, it may greatly exceed the migration cost bound.

The runtimes of all heuristics increase with the increasing size of the graphs and the clusters, as expected. Since the graphs in our workload are sparse, we expect the complexity of HEFT to be $O(ep)$ where e is the number of edges and p is the number of processors. For graphs of the same size and for clusters of different sizes, the runtime of the HEFT algorithm increases roughly linearly with the number of processors (Figure 4.7(a)). For clusters of the same size and for graphs with different sizes, the runtime of the HEFT algorithm increases roughly proportional to the number of edges as the graph size increases. Even for graphs having 200 stages and a cluster of 512 processors, HEFT has a runtime of less than 600 ms, which is evidence of how well list scheduling heuristics scale.

Among the incremental heuristics, HEFT-Relax is the least computationally complex. For 20 stages, the runtime is not visible until 32 processors since it is in the order of microseconds. Even for graphs of size 200 and a cluster with 256 processors, the runtime of HEFT-Relax is below 650 ms, making this heuristic an ideal candidate for large-scale settings. Finally, for graphs of size 200 and a cluster having 512 processors, the HEFT-Relax runtime exceeds 1 s only slightly.

Compared to the other heuristics, HEFT-MS is less costly than HEFT-Iter and HEFT-DRelax, and more costly than HEFT-Relax for both small and large graphs and clusters (Figure 4.7(b)). For graphs having fewer than 100 stages, the HEFT-MS runtime is in the order of hundreds of milliseconds (less than 750 ms) except for the case when the cluster has 512 processors. For larger graphs and clusters having more than 64 processors, the runtime of HEFT-MS is in the order of seconds.

For graphs having fewer than 100 stages and clusters having fewer than 256 processors, the HEFT-Iter runtime is in the order of hundreds of milliseconds (less than 950 ms), however, increasing the scale further increases the runtime, causing it to be in the order of seconds (Figure 4.7(c)).

For graphs having fewer than 100 stages, HEFT-DRelax and HEFT-Iter have similar performance (Figure 4.7(e)). However, for larger graphs and especially for large clusters, with more than 128 processors, HEFT-DRelax is more costly than HEFT-Iter, and the difference in cost increases as the cluster size increases. Because until the migration cost bound is exceeded HEFT-DRelax relaxes a pair of stages and lets HEFT run through all processors to *place and schedule* these stages. However, HEFT-Iter when searching the best swap operations, which dominates the runtime for a large number of stages, calls HEFT with stages already placed and HEFT only *schedules* those stages. For graphs having more than 100 stages, both HEFT-Iter and HEFT-DRelax may have significant costs in the order of tens of seconds.

To conclude, HEFT-Relax has the best performance for both small- and large-scale settings. For small-scale settings, all heuristics perform well (with runtimes in the order of at most hundreds of milliseconds). However, for large-scale setting, that is, graphs having more than 50 stages and clusters having more than 64 processors, all heuristics may have runtimes in the order of seconds. It is important to note that these runtimes are

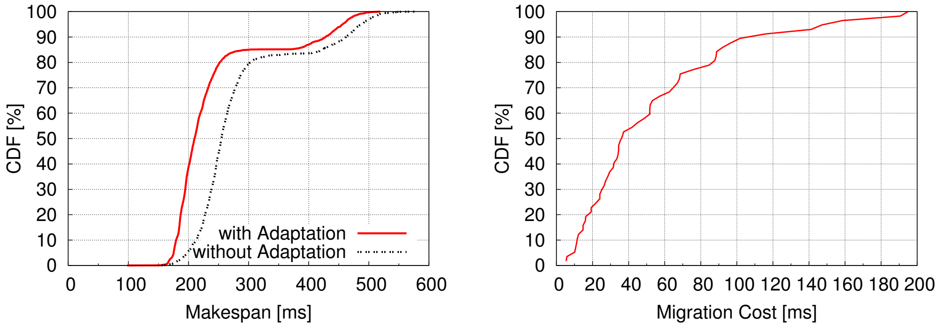


Figure 4.8: Cumulative distribution function (CDF) of the makespan (left) and migration cost (right) for the real system experiments with the pose detection application. The migration cost bound is 200 ms.

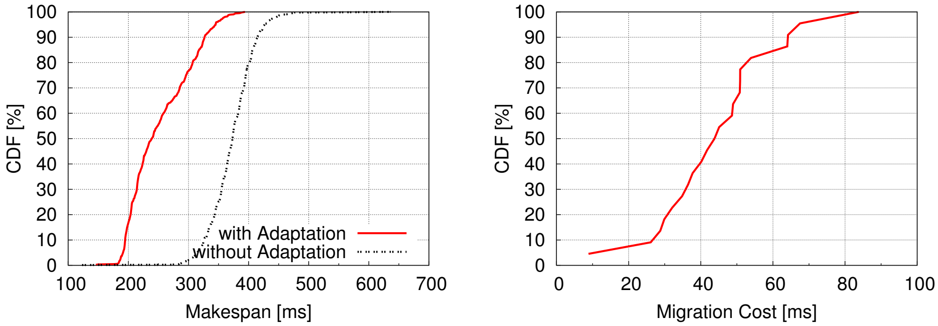


Figure 4.9: Cumulative distribution function (CDF) of the makespan (left) and migration cost (right) for the real system experiments with the TV control application. The migration cost bound is 200 ms.

for the worst case when the migration cost bound is infinite.

4.8 Results in a Real System

In this section we evaluate incremental placement using the pose detection and the TV control applications presented in Section 4.1. We run two experiments for each application. In the first, the placement is adapted as described in Section 4.5, and in the second, the placement is static. Our setup measures and records the (per-frame) latency (or makespan) as the time elapsed between the arrival of a new video frame and the completion of the final processing stage for that frame on a frame-by-frame basis. The applications run until all input frames have been processed, and as a result of our experiments, we report the distributions of the makespan with and without incremental placement, and the distribution of the migration cost.

Figure 4.8 shows the results of the experiments with the pose detection application with a migration cost bound of 200 ms. The application was configured with 53 stages, each with a demand of 1. For this experiment, as input to the application we use a HD (1280x720 pixels) video sequence comprising 3600 frames. The video starts with an empty scene which is slowly populated with many objects, and then the objects are removed. Therefore, both the number of SIFT features and model matching operations increase over time putting more load on the system. This input data creates a scenario where the stages, which have runtimes strictly depending on the structure of the input data, get perturbed as the complexity of the input changes. Figure 4.8(left) shows that the system without adaptation has a median (maximum) makespan of 255 (578) ms, while with adaptation the median (maximum) latency is 211 (519) ms, which is an 18% (10%) improvement, thus noticeably improving the system's responsiveness. For this experiment, the median migration cost is 37 ms, which is 19% of the migration cost bound.

Figure 4.9 shows the results of the experiments with the TV control application with a migration cost bound of 200 ms. The application was configured with 32 stages, each with a demand of 1. For this experiment, we evaluate the same scenario as the previous experiment with an input video where a user performs specific gestures. However, compared to the previous experiment the input video has lower variability. Even with lower input variability, the median (maximum) latency with adaptation is 239 (374) ms, while without adaptation the median (maximum) latency is 374 (636) ms which is a 36% (38%) improvement, thus improving the system responsiveness significantly. For this experiment, the median migration cost is 43 ms, which is 22% of the migration cost bound. We conclude that adaptation is able to improve the makespan significantly while causing little churn in the system.

Using our incremental placement heuristics with two real perception applications we have demonstrated significant improvements in median latency by up to 36%, and the application latencies are relatively close to the latency upper bound of 200 ms stated in the Introduction. However, neither application achieves the desired 50–100 ms latency specified. This may be remedied by using techniques such as runtime adaptation of stage tunable parameters, increasing the degree of parallelism, using more powerful resources, and scaling down the input frames, but these techniques are complementary to and outside the scope of this paper.

4.9 Related Work

The problem of mapping task graphs to multiprocessors has been studied extensively (see [128] for a survey). The problem is NP-complete even in the case of two processors and non-uniform execution times [84]. As the related efforts, particularly in the real-time systems and multimedia processing communities, are too numerous to list, we consider here only those that make assumptions most consistent with our application model and execution environment: task precedence graphs with arbitrary execution times and communication costs, no task duplication, and a bounded number of fully connected, heterogeneous processors. Of these assumptions, heterogeneity is the most restrictive.

Static task scheduling algorithms that satisfy these assumptions fall into three categories: optimal approaches [182], guided random search based algorithms such as genetic algorithms and simulated annealing [127, 186, 189, 210], and heuristics. In our system, task placement is run repeatedly over a dynamic task graph, so efficient run times are essential. For this reason, heuristics are the more practical approach in our environment.

Clustering heuristics [51] assign tasks to an unbounded set of clusters. Tasks belonging to the same cluster execute on the same processor. Clusters are merged at each iteration, often by decreasing amount of inter-task communication. A final mapping step assigns clusters to processors, and then orders the tasks on each processor. A practical issue with clustering algorithms is ensuring that the resulting mapping satisfies machine resource constraints.

List scheduling heuristics [25, 73, 112, 178, 188, 202] maintain a list of tasks ordered by a ranking function, or priority. Tasks are mapped to available processors in this order according to a cost function, such as the earliest start time of the task. As discussed in Section 4.2, HEFT [202] produces competitive schedules with low computational complexity.

Because task execution times and data amounts may not be known a priori, and may change over time, recent approaches adjust task assignments dynamically based on empirical measurements. Kwok et al. [129] use a semi-static method that creates a set of assignments offline using a genetic algorithm, and then selects between the pre-computed assignments online based on observed performance. In task rescheduling [138, 179], an initial schedule is generated for the task graph; as tasks complete, their runtime information is used to improve the schedule for the remaining tasks. This scheme is intended for applications in which the tasks execute once, unlike in our environment where the tasks execute repeatedly over a stream of data.

Dynamic adjustment of task or workload placement with cost constraints has been studied in related problems. Aggarwal et al. [7] provide a 1.5-approximation algorithm for load rebalancing with a cost constraint for multiprocessor scheduling without precedence constraints. Chen et al. [44] propose an an algorithm that minimizes migration cost for independent tasks in a computing grid subject to load constraints. Their algorithm is a local search with special strategies for finding low cost solutions. Tang et al. [197] propose an algorithm that dynamically determines the number of instances of web applications to run on a set of machines. Unlike multiprocessor scheduling with precedence constraints, the problem is formulated as a knapsack problem, with objectives that include maximizing the demand satisfied and minimizing changes to the running system.

4.10 Summary

To achieve low latency for interactive perception applications, clusters of machines can be used to exploit the inherent parallelism in these applications. In this chapter we have addressed the problem of placing the stages of these applications on clusters of machines. In particular, we have tackled the incremental placement problem of adjusting an initial placement to minimize the makespan subject to migration cost constraints.

We have proposed four heuristics for incremental placement that cover a broad range of tradeoffs of computational complexity, churn in the placement, and ultimate improvement in makespan. HEFT-MS is good at minimizing the churn, but not as good as the others at adjusting the schedule. Although HEFT-Iter and HEFT-DRelax are computationally complex, they perform well at improving the schedule; with increasing migration cost bounds they can significantly improve the makespan. HEFT-Relax performs in between; it is computationally less complex, performs better than HEFT-MS at adjusting the schedule, but produces more churn in the system.

Through simulations and real system experiments we have shown that it is worth adjusting the schedule using our incremental placement heuristics. A broad range of simulations show up to 50% improvement in makespan, and experiments with two applications on a real system demonstrate 18% (10%) and 36% (38%) improvements in median (maximum) makespan, respectively. In addition, we have shown that our incremental heuristics can approach the improvements achieved by completely rerunning a static placement algorithm, but with lower migration costs and churn in the schedule.

Chapter 5

Performance evaluation of public clouds[*]

Scientific computing requires an ever-increasing number of resources to deliver results for ever-growing problem sizes in a reasonable time frame. In the last decade, while the largest research projects were able to afford (access to) expensive supercomputers, many projects were forced to opt for cheaper resources such as commodity clusters and grids. Cloud computing proposes an alternative in which resources are no longer hosted by the researchers' computational facilities, but are leased from big data centers only when needed. Despite the existence of several cloud computing offerings by vendors such as Amazon [12] and GoGrid [87], the potential of clouds for scientific computing remains largely unexplored. To address this issue, in this chapter we present a performance analysis of cloud computing services for scientific computing.

The cloud computing paradigm holds great promise for the performance-hungry scientific computing community: Clouds can be a cheap alternative to supercomputers and specialized clusters, a much more reliable platform than grids, and a much more scalable platform than the largest of commodity clusters. Clouds also promise to "scale by credit card," that is, to scale up instantly and temporarily within the limitations imposed only by the available financial resources, as opposed to the physical limitations of adding nodes to clusters or even supercomputers and to the administrative burden of overprovisioning resources. However, clouds also raise important challenges in many aspects of scientific computing, including performance, which is the focus of this chapter.

There are three main differences between scientific computing workloads and the initial target workload of clouds: in required system size, in performance demand, and in the job execution model. Size-wise, top scientific computing facilities comprise very

---

[*] This chapter is based on previous work published in the *IEEE Transactions on Parallel and Distributed Systems* [105], the *International Conference on Cloud Computing* (CloudComp'09) [162], and the *International Workshop on Cloud Computing* (Cloud'09) [226].

large systems, with the top ten entries in the Top500 Supercomputers List together totaling about one million cores as of 2009, while cloud computing services were designed to replace the small-to-medium size enterprise data centers. Performance-wise, scientific workloads often require High Performance Computing (HPC) or High-Throughput Computing (HTC) capabilities. The job execution model of scientific computing platforms is based on the exclusive, space-shared usage of resources. In contrast, most clouds time-share resources and use virtualization to abstract away from the actual hardware, thus increasing the concurrency of users but potentially lowering the attainable performance.

These three main differences between scientific computing workloads and the target workloads of clouds raise an important research question: *Is the performance of clouds sufficient for scientific computing?*, or, in other words, *Can current clouds execute scientific workloads with similar performance (that is, for traditional performance metrics [79]) and at lower cost?* Though early attempts to characterize clouds and other virtualized services exist [231, 63, 163, 208, 211], this question remains largely unexplored. In this chapter, to answer this research question first we evaluate with well-known micro-benchmarks and application kernels the performance of four commercial cloud computing services that can be used for scientific computing, among which the Amazon Elastic Compute Cloud (EC2), the largest commercial computing cloud in production. Then, we compare the performance of clouds with that of scientific computing alternatives such as grids and parallel production infrastructures. Our comparison uses trace-based simulation and the empirical performance results of our cloud performance evaluation. We also perform a preliminary assessment of the performance consistency of these four public clouds. However, our assessment only considers the performance consistency of repeated benchmark executions over short periods of time and with low-level operations, such as floating point additions or memory read/writes, thus motivating us to explore the performance variability in depth in the next chapter.

The rest of this chapter is organized as follows. In Section 5.1 we give a general introduction to the use of cloud computing services for scientific computing, and select four exemplary clouds for use in our investigation. Then, in Section 5.2 we evaluate empirically the performance of four commercial clouds. In Section 5.3 we compare the performance of clouds and of other scientific computing environments. Finally, we compare our investigation with related work in Section 5.4, and we summarize the chapter in Section 5.5.

5.1 Cloud Computing Services for Scientific Computing

In this section we provide a background to analyzing the performance of cloud computing services for scientific computing. We first describe the main characteristics of the common scientific computing workloads, based on previous work on analyzing and modeling of workload traces taken from PPIs [134] and grids [99, 104]. Then, we introduce the cloud computing services that can be used for scientific computing, and select four commercial clouds whose performance we will evaluate empirically.

5.1.1 Scientific Computing

Job structure and source PPI workloads are dominated by parallel jobs [134], while grid workloads are dominated by small bags-of-tasks (BoTs) [107] and sometimes by small workflows [201, 160] comprising mostly sequential tasks. Source-wise, it is common for PPI grid workloads to be dominated by a small number of users.

Bottleneck resources Overall, scientific computing workloads are highly heterogeneous, and can have either one of CPU, I/O, memory, and network as the bottleneck resource. Thus, in Section 5.2 we investigate the performance of these individual resources.

Job parallelism A large majority of the parallel jobs found in published PPI [4] and grid [104] traces have up to 128 processors [134, 99]. Moreover, the average scientific cluster size was found to be around 32 nodes [118] and to be stable over the past five years [108]. Thus, in Section 5.2 we look at the the performance of executing parallel applications of up to 128 processors.

5.1.2 Four Selected Clouds: Amazon EC2, GoGrid, ElasticHosts, and Mosso

We identify three categories of cloud computing services [229, 15]: Infrastructure-as-a-Service (IaaS), that is, raw infrastructure and associated middleware, Platform-as-a-Service (PaaS), that is, APIs for developing applications on an abstract platform, and Software-as-a-Service (SaaS), that is, support for running software services remotely. Many clouds already exist, but not all provide virtualization, or even computing services. The scientific community has not yet started to adopt PaaS or SaaS solutions, mainly to avoid porting legacy applications and for lack of the needed scientific computing services, respectively. Thus, in this study we are focusing only on IaaS providers. We also focus only on public clouds, that is, clouds that are not restricted within an enterprise; such clouds can be used by our target audience, scientists.

Based on our recent survey of the cloud computing providers [169], we have selected for this work four IaaS clouds. The reason for this selection is threefold. First, not all the clouds on the market are still accepting clients; FlexiScale puts new customers on a waiting list for over two weeks due to system overload. Second, not all the clouds on the market are large enough to accommodate requests for even 16 or 32 co-allocated resources. Third, our selection already covers a wide range of quantitative and qualitative cloud characteristics, as summarized in Tables 5.1 and our cloud survey [169], respec-

Name	Cores (ECUs)	RAM [GB]	Archi. [bit]	Disk [GB]	Cost [$/h]
Amazon EC2					
m1.small	1 (1)	1.7	32	160	0.1
m1.large	2 (4)	7.5	64	850	0.4
m1.xlarge	4 (8)	15.0	64	1,690	0.8
c1.medium	2 (5)	1.7	32	350	0.2
c1.xlarge	8 (20)	7.0	64	1,690	0.8
GoGrid (GG)					
GG.small	1	1.0	32	60	0.19
GG.large	1	1.0	64	60	0.19
GG.xlarge	3	4.0	64	240	0.76
Elastic Hosts (EH)					
EH.small	1	1.0	32	30	£0.042
EH.large	1	4.0	64	30	£0.09
Mosso					
Mosso.small	4	1.0	64	40	0.06
Mosso.large	4	4.0	64	160	0.24

Table 5.1: The resource characteristics for the instance types offered by the four selected clouds.

tively. We describe in the following Amazon EC2; the other three, GoGrid (GG), ElasticHosts (EH), and Mosso, are IaaS clouds with provisioning, billing, and availability and performance guarantees similar to Amazon EC2's.

The **Amazon Elastic Computing Cloud (EC2)** is an IaaS cloud computing service that opens Amazon's large computing infrastructure to its users. The service is elastic in the sense that it enables the user to extend or shrink its infrastructure by launching or terminating new virtual machines (*instances*). The user can use any of the *instance types* currently available on offer, the characteristics and cost of the five instance types available in June 2009 are summarized in Table 5.1. An ECU is the equivalent CPU power of a 1.0-1.2 GHz 2007 Opteron or Xeon processor. The theoretical peak performance can be computed for different instances from the ECU definition: a 1.1 GHz 2007 Opteron can perform 4 flops per cycle at full pipeline, which means at peak performance one ECU equals 4.4 gigaflops per second (GFLOPS).

To create an infrastructure from EC2 resources, the user specifies the instance type and the VM image; the user can specify any VM image previously registered with Amazon, including Amazon's or the user's own. Once the VM image has been transparently deployed on a physical machine (the resource status is *running*), the instance is booted; at the end of the boot process the resource status becomes *installed*. The installed resource can be used as a regular computing node immediately after the booting process has finished, via an `ssh` connection. A maximum of 20 instances can be used concurrently by regular users by default; an application can be made to increase this limit, but the process involves an Amazon representative. Amazon EC2 abides by a Service Level Agreement (SLA) in which the user is compensated if the resources are not available for acquisition at least 99.95% of the time. The security of the Amazon services has been investigated elsewhere [163].

5.2 Cloud Performance Evaluation

In this section we present an empirical performance evaluation of cloud computing services. Toward this end, we run micro-benchmarks and application kernels typical for scientific computing on cloud computing resources, and compare whenever possible the obtained results to the theoretical peak performance and/or the performance of other scientific computing systems.

5.2.1 Method

Our method stems from the traditional system benchmarking. Saavedra and Smith [175] have shown that benchmarking the performance of various system components with a wide variety of micro-benchmarks and application kernels can provide a first order estimate of that system's performance. Similarly, in this section we evaluate various components of the four clouds introduced in Section 5.1.2. However, our method is not a straightforward application of Saavedra and Smith's method. Instead, we add a cloud-specific component, select several benchmarks for a comprehensive platform-independent evaluation, and focus on metrics specific to large-scale systems (such as efficiency and variability).

Cloud-specific evaluation An attractive promise of clouds is that they can always provide resources on demand, without additional waiting time [15]. However, since the load of other large-scale systems varies over time due to submission patterns [134, 99] we want to investigate if large clouds can indeed bypass this problem. To this end, one or more instances of the same instance type are repeatedly acquired and released during a few minutes; the resource acquisition requests follow a Poisson process with arrival rate $\lambda = 1s^{-1}$.

Infrastructure-agnostic evaluation There currently is no single accepted benchmark for scientific computing at large-scale. To address this issue, we use several traditional benchmark suites comprising micro-benchmarks and (scientific) application kernels. We further design two types of test workloads: SI–run one or more single-process jobs on a single instance (possibly with multiple cores), and MI–run a single multi-process job on multiple instances. The SI workloads execute in turn one of the *LMbench* [142], *Bonnie* [37], and *CacheBench* [150] benchmark suites. The MI workloads execute the *HPC Challenge Benchmark (HPCC)* [136] scientific computing benchmark suite. The charac-

Type	Suite/Benchmark	Resource	Unit
SI	LMbench/all [23]	Many	Many
SI	Bonnie/all [125, 20]	Disk	MBps
SI	CacheBench/all [207]	Memory	MBps
MI	HPCC/HPL [136, 177]	CPU	GFLOPS
MI	HPCC/DGEMM [69]	CPU	GFLOPS
MI	HPCC/STREAM [69]	Memory	GBps
MI	HPCC/RandomAccess [9]	Network	MUPS
MI	HPCC/b_{eff}(lat.,bw.) [31]	Comm.	μs, GBps

Table 5.2: The benchmarks used for cloud performance evaluation. B, FLOP, U, and PS stand for bytes, floating point operations, updates, and per second, respectively.

teristics of the used benchmarks and the mapping to the test workloads are summarized in Table 5.2; we refer to the benchmarks' references for more details.

Performance metrics We use the performance metrics defined by the benchmarks presented in Table 5.2. We also define and use the *HPL efficiency* of a virtual cluster based on the instance type *T* as the ratio between the HPL benchmark performance of the real cluster and the peak theoretical performance of a same-sized *T*-cluster, expressed as a percentage. Job execution at large-scale often leads to performance variability. To address this problem, in this chapter we report not only the average performance, but also the variability of the results.

5.2.2 Experimental Setup

We now describe the experimental setup in which we use the performance evaluation method presented earlier.

Performance Analysis Tool We have recently [227] extended the GrenchMark [100] large-scale distributed testing framework with new features which allow it to test cloud computing infrastructures. The framework was already able to generate and submit both real and synthetic workloads to grids, clusters, clouds, and other large-scale distributed environments. For this work, we have added to GrenchMark the ability to execute and analyze the benchmarks described in the previous section.

Environment We perform our measurements on homogeneous virtual environments built from virtual resources belonging to one of the instance types described in Table 5.1; the used VM images are summarized in Table 5.3. The experimental environments comprise from 1 to 128 cores. Except for the use of internal IP addresses, described below, we have used in all our experiments the standard configurations provided by the cloud. Due to our choice of benchmarks, our Single-Job results can be readily compared with the benchmarking results made public for many other scientific computing systems, and in particular by the HPCC effort [3].

VM image	OS, MPI	Archi	Benchmarks
EC2/ami-2bb65342	FC6	32bit	SI
EC2/ami-36ff1a5f	FC6	64bit	SI
EC2/ami-3e836657	FC6, MPI	32bit	MI
EC2/ami-e813f681	FC6, MPI	64bit	MI
GG/server$_1$	RHEL 5.1, MPI	32&64bit	SI&MI
EH/server$_1$	Knoppix 5.3.1	32bit	SI
EH/server$_2$	Ubuntu 8.10	64bit	SI
Mosso/server$_1$	Ubuntu 8.10	32&64bit	SI

Table 5.3: The VM images used in our experiments.

MPI library and network The VM images used for the HPCC benchmarks also have a working pre-configured MPI based on the `mpich2-1.0.5` [219] implementation. For the MI (parallel) experiments, the network selection can be critical for achieving good results. Amazon EC2 and GoGrid, the two clouds for which we have performed MI experiments, use internal IP addresses (IPs), that is, the IPs accessible only within the cloud, to optimize the data transfers between closely-located instances. (This also allows the clouds to better shape the traffic and to reduce the number of Internet-accessible IPs, and in turn to reduce the cloud's operational costs.) EC2 and GoGrid give strong incentives to their customers to use internal IP addresses, in that the network traffic between internal IPs is free, while the traffic to or from the Internet IPs is not. We have used only the internal IP addresses in our experiments with MI workloads.

Optimizations, tuning The benchmarks were compiled using GNU C/C++ 4.1 with the `-O3 -funroll-loops` command-line arguments. We did not use any additional architecture- or instance-dependent optimizations. For the HPL benchmark, the performance results depend on two main factors: the the Basic Linear Algebra Subprogram (BLAS) [68] library, and the problem size. We used in our experiments the GotoBLAS [89]library, which is one of the best portable solutions freely available to scientists. Searching for the problem size that can deliver peak performance is an extensive (and costly) process. Instead, we used a free analytical tool [5] to find for each system the problem sizes that can deliver results close to the peak performance; based on the tool advice we have used values from 13,000 to 110,000 for N, the size (order) of the coefficient matrix A [67, 136].

5.2.3 Results

Resource Acquisition and Release

We study two resource acquisition and release scenarios: for single instances, and for multiple instances allocated at once.

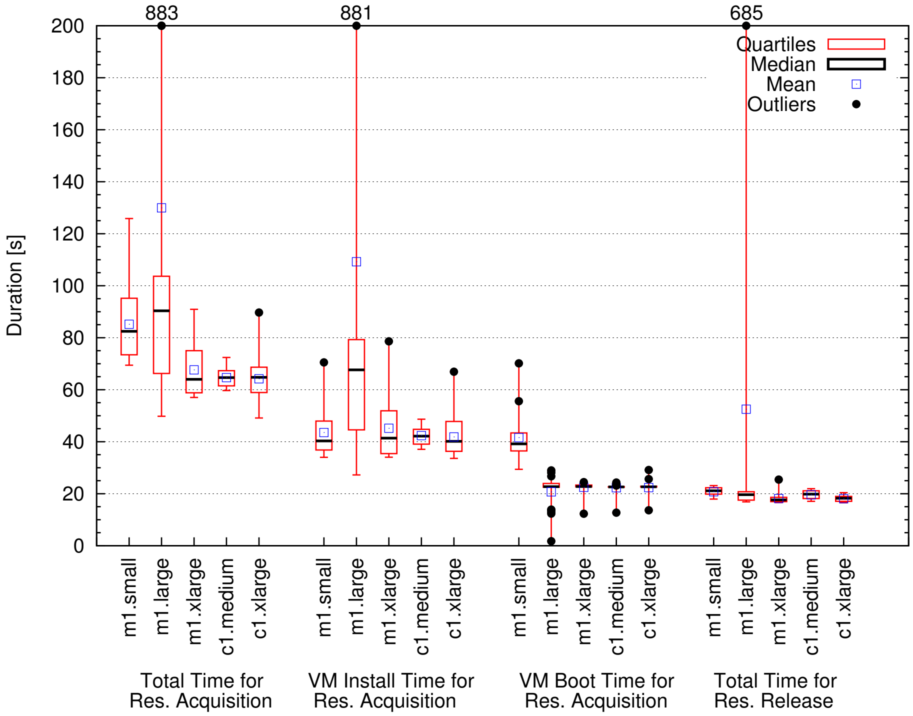


Figure 5.1: Resource acquisition and release overheads for acquiring single EC2 instances. Lower values are better.

Single instances We first repeat 20 times for each instance type a resource acquisition followed by a release as soon as the resource status becomes installed (see Section 5.1.2). Figure 5.1 shows the overheads associated with resource acquisition and release in EC2. The total resource acquisition time (*Total*) is the sum of the *Install* and *Boot* times. The *Release* time is the time taken to release the resource back to EC2; after it is released the resource stops being charged by Amazon. The `c1.*` instances are surprisingly easy to obtain; in contrast, the `m1.*` instances have for the resource acquisition time higher expectation (63-90s compared to around 63s) and variability (much larger boxes). With the exception of the occasional outlier, both the VM *Boot* and *Release* times are stable and represent about a quarter of *Total* each. Table 5.4 presents basic statistics for single resource allocation and release. Overall, **Amazon EC2 has one order of magnitude lower single resource allocation and release durations than GoGrid.** From the EC2 resources, the `m1.small` and `m1.large` instances have higher average allocation duration, and exhibit outliers comparable to those encountered for GoGrid. **The resource acquisition time of GoGrid resources is highly variable**; here, GoGrid behaves similarly to to grids [99] and unlike the promise of clouds.

Multiple instances We investigate next the performance of requesting the acquisition of multiple resources (2,4,8,16, and 20) *at the same time*; a scenario common for creating homogeneous virtual clusters. When resources are requested in bulk, we record acquisi-

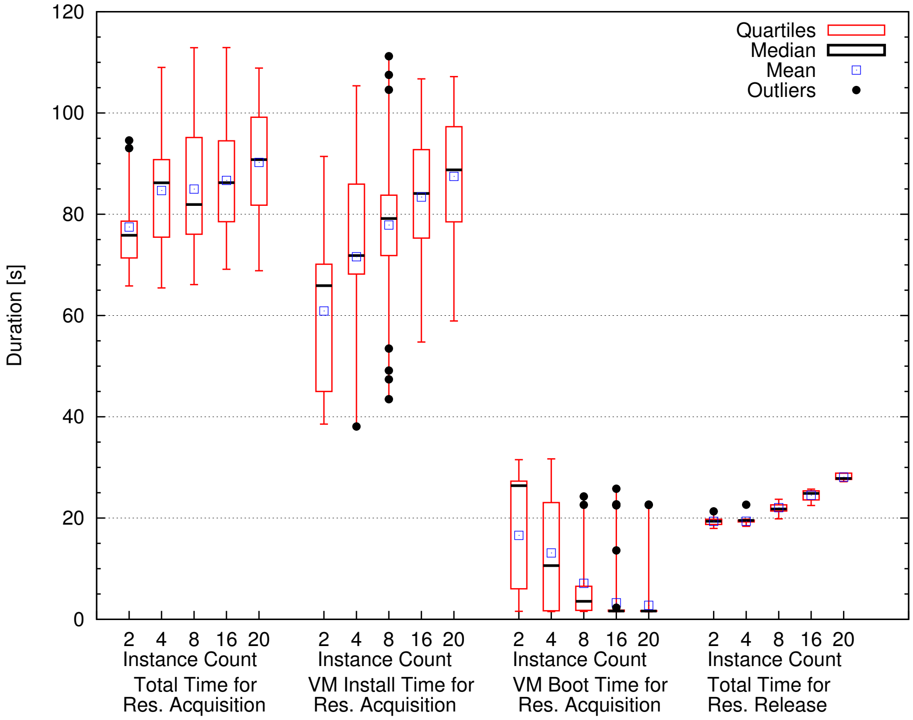


Figure 5.2: Single-instance resource acquisition and release overheads when acquiring multiple `c1.xlarge` instances at the same time. Lower values are better.

tion and release times for each resource in the request, separately. Figure 5.2 shows the basic statistical properties of the times recorded for `c1.xlarge` instances. The expectation and the variance are both higher for multiple instances than for a single instance.

Single-Machine Benchmarks

In this set of experiments we measure the raw performance of the CPU, I/O, and memory hierarchy using the Single-Instance benchmarks listed in Section 5.2.1. We run each benchmark 10 times and report the average results.

Compute performance We assess the computational performance of each instance type using the entire LMbench suite. The performance of int and int64 operations, and of the float and double-precision float operations is depicted in Figure 5.3 left and right, respectively. *The GOPS recorded for the floating point and double-precision float operations is six to eight times lower than the theoretical maximum of ECU (4.4 GOPS).* Also, the double-precision float performance of the `c1.*` instances, arguably the most important for scientific computing, is mixed: excellent addition but poor multiplication capabilities. Thus, as many scientific computing applications use heavily both of these operations, the user is faced with the difficult problem of selecting between two wrong choices. Finally, several double and float operations take longer on `c1.medium` than on

Instance	Res. Allocation			Res. Release		
Type	Min	Avg	Max	Min	Avg	Max
m1.small	69	82	126	18	21	23
m1.large	50	90	883	17	20	686
m1.xlarge	57	64	91	17	18	25
c1.medium	60	65	72	17	20	22
c1.xlarge	49	65	90	17	18	20
GG.large	240	540	900	180	210	240
GG.xlarge	180	1,209	3,600	120	192	300

Table 5.4: Statistics for single resource allocation/release.

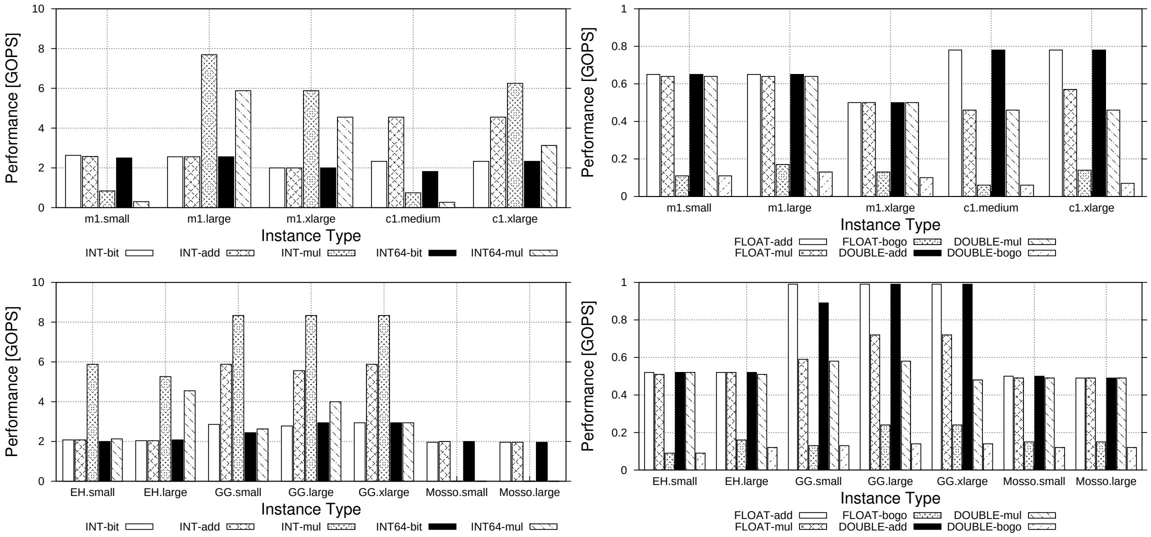


Figure 5.3: LMbench results (*top*) for the EC2 instances, and (*bottom*) for the other instances. Each row depicts the performance of 32- and 64-bit integer operations in giga-operations per second (GOPS) (*left*), and of floating operations with single and double precision (*right*).

`m1.small`. For the other instances, `EH.*` and `Mosso.*` instances have similar performance for both integer and floating point operations. `GG.*` instances have the best float and double-precision performance, and good performance for integer operations, which suggests the existence of better hardware support for these operations on these instances.

I/O performance We assess in two steps the I/O performance of each instance type with the Bonnie benchmarking suite. The first step is to determine the smallest file size that invalidates the memory-based I/O cache, by running the Bonnie suite for thirteen file sizes in the range 1024 Kilo-binary byte (KiB) to 40 GiB. The results of this preliminary step have been described in our previous work [161]; we only summarize them here. For all instance types, a performance drop begins with the 100MiB test file and ends at 2GiB, indicating a capacity of the memory-based disk cache of 4-5GiB (twice 2GiB). Thus, the results obtained for the file sizes above 5GiB correspond to the real I/O performance of the

Instance Type	Seq. Output			Seq. Input		Rand. Input
	Char [MB/s]	Block [MB/s]	ReWr [MB/s]	Char [MB/s]	Block [MB/s]	[Seek/s]
m1.small	22.3	60.2	33.3	25.9	73.5	74.4
m1.large	50.9	64.3	24.4	35.9	63.2	124.3
m1.xlarge	57.0	87.8	33.3	41.2	74.5	387.9
c1.medium	49.1	58.7	32.8	47.4	74.9	72.4
c1.xlarge	64.8	87.8	30.0	45.0	74.5	373.9
GG.small	11.4	10.7	9.2	29.2	40.24	39.8
GG.large	17.0	17.5	16.0	34.1	97.5	29.0
GG.xlarge	80.7	136.9	92.6	79.26	369.15	157.5
EH.large	7.1	7.1	7.1	27.9	35.7	177.9
Mosso.sm	41.0	102.7	43.88	32.1	130.6	122.6
Mosso.lg	40.3	115.1	55.3	41.3	165.5	176.7
’02 Ext3	12.2	38.7	25.7	12.7	173.7	-
’02 RAID5	14.4	14.3	12.2	13.5	73.0	-
’07 RAID5	30.9	40.6	29.0	41.9	112.7	192.9

Table 5.5: The I/O of clouds vs. 2002 [125] and 2007 [20] systems.

system; lower file sizes would be served by the system with a combination of memory and disk operations. We analyze the I/O performance obtained for files sizes above 5GiB in the second step; Table 5.5 summarizes the results. We find that the I/O performance indicated by Amazon EC2 (see Table 5.1) corresponds to the achieved performance for random I/O operations (column ’Rand. Input’ in Table 5.5). The `*.xlarge` instance types have the best I/O performance from all instance types. *For the sequential operations more typical to scientific computing all Amazon EC2 instance types have in general better performance when compared with similar modern commodity systems*, such as the systems described in the last three rows in Table 5.5; EC2 may be using better hardware, which is affordable due to economies of scale [15].

Multi-Machine Benchmarks

In this set of experiments we measure the performance delivered by homogeneous clusters formed with Amazon EC2 and GoGrid instances when running the Single-Job-Multi-Machine benchmarks. For these tests we execute 5 times the HPCC benchmark on homogeneous clusters of 1–16 (1–8) instances on EC2 (GoGrid), and present the average results.

HPL performance The performance achieved for the HPL benchmark on various virtual clusters based on the `m1.small` and `c1.xlarge` instance types is depicted in Figure 5.4. For the `m1.small` resources one node was able to achieve a performance of 1.96 GFLOPS, which is 44.54% from the peak performance advertised by Amazon. Then,

Name	Peak Perf.	GFLOPS	GFLOPS /Unit	GFLOPS /$1
m1.small	4.4	2.0	2.0	19.6
m1.large	17.6	7.1	1.8	17.9
m1.xlarge	35.2	11.4	1.4	14.2
c1.medium	22.0	3.9	0.8	19.6
c1.xlarge	88.0	50.0	2.5	62.5
GG.large	12.0	8.8	8.8	46.4
GG.xlarge	36.0	28.1	7.0	37.0

Table 5.6: HPL performance and cost comparison for various EC2 and GoGrid instance types.

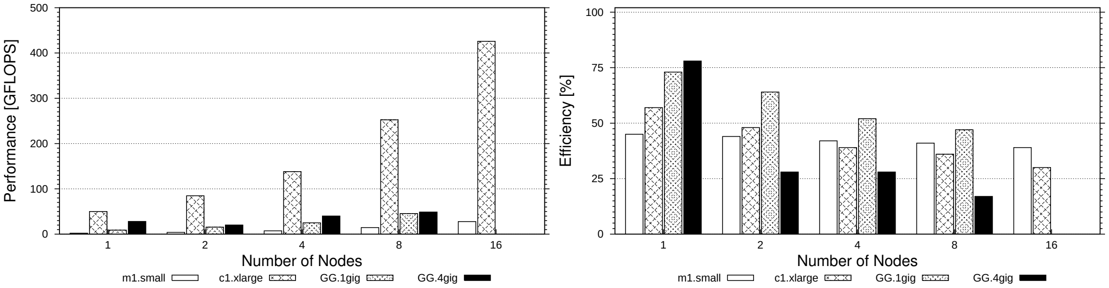


Figure 5.4: The HPL (LINPACK) performance of virtual clusters formed with EC2 `m1.small`, EC2 `c1.xlarge`, GoGrid `large`, and GoGrid `xlarge` instances in terms of throughput (*left*) and efficiency (*right*).

the performance increased to up to 27.8 GFLOPS for 16 nodes, while the efficiency decreased slowly to 39.4%. The results for a single `c1.xlarge` instance are better: the achieved 49.97 GFLOPS represent 56.78% from the advertised peak performance. However, while the performance scales when running up to 16 instances to 425.82 GFLOPS, the efficiency decreases to only 30.24%. The HPL performance loss from one to 16 instances can therefore be expressed as 53.26% which results in rather bad qualification for HPC applications and their need for fast inter-node communication. We have obtained similar results the `GG.large` and `GG.xlarge` instances, as shown in Figure 5.4. For `GG.large` instances, the efficiency decreases quicker than for EC2 instances, down to 47.33% while achieving 45.44 GFLOPS on eight instances. The `GG.xlarge` performed even poorer in our tests. We further investigate the performance of the HPL benchmark for different instance types; Table 5.6 summarizes the results. *The efficiency results* presented in Figure 5.4 and Table 5.6 *place clouds below existing environments for scientific computing, for which the achieved performance is 60-70% of the theoretical peak even for demanding real applications [165, 119, 164].*

HPCC performance To obtain the performance of virtual EC2 and GoGrid clusters we run the HPCC benchmarks on *unit clusters* comprising a single instance, and on *128-core clusters* comprising 16 `c1.xlarge` instances. Table 5.7 summarizes the obtained

Provider, System	Cores or Capacity	Peak Perf. [GFLOPS]	HPL [GFLOPS]	HPL N	DGEMM [GFLOPS]	STREAM [GBps]	RandomAccess [MUPs]	Latency [μs]	Bandwidth [GBps]
EC2, 1 x m1.small	1	4.40	1.96	13,312	2.62	3.49	11.60	-	-
EC2, 1 x m1.large	2	17.60	7.15	28,032	6.83	2.38	54.35	20.48	0.70
EC2, 1 x m1.xlarge	4	35.20	11.38	39,552	8.52	3.47	168.64	17.87	0.92
EC2, 1 x c1.medium	2	22.00	-	13,312	11.85	3.84	46.73	13.92	2.07
EC2, 1 x c1.xlarge	8	88.00	51.58	27,392	44.05	15.65	249.66	14.19	1.49
EC2, 2 x c1.xlarge	16	176.00	84.63	38,656	34.59	15.65	223.54	19.31	1.10
EC2, 4 x c1.xlarge	32	352.00	138.08	54,784	27.74	15.77	280.38	25.38	1.10
EC2, 8 x c1.xlarge	64	704.00	252.34	77,440	3.58	15.89	250.40	35.93	0.97
EC2, 16 x c1.xlarge	128	1,408.00	425.82	109,568	0.23	16.38	207.06	45.20	0.75
EC2, 16 x m1.small	16	70.40	27.80	53,376	4.36	11.95	77.83	68.24	0.10
GoGrid, 1 x GG.large	1	12.00	8.805	10,240	10.01	2.88	17.91	-	-
GoGrid, 4 x GG.large	4	48.00	24.97	20,608	10.34	20.17	278.80	110.11	0.06
GoGrid, 8 x GG.large	8	96.00	45.436	29,184	10.65	20.17	351.68	131.13	0.07
GoGrid, 1 x GG.xlarge	3	36.00	28.144	20,608	10.82	45.71	293.30	16.96	0.97
GoGrid, 4 x GG.xlarge	12	144.00	40.03	41,344	11.31	19.95	307.64	62.20	0.24
GoGrid, 8 x GG.xlarge	24	288.00	48.686	58,496	18.00	20.17	524.33	55.54	1.33
HPCC-227, TopSpin/Cisco	16	102.40	55.23	81,920	4.88	2.95	10.25	6.81	0.66
HPCC-224, TopSpin/Cisco	128	819.20	442.04	231,680	4.88	2.95	10.25	8.25	0.68
HPCC-286, Intel Endeavor	16	179.20	153.25	60,000	10.50	5.18	87.61	1.23	1.96
HPCC-289, Intel Endeavor	128	1,433.60	1,220.61	170,000	10.56	5.17	448.31	2.78	3.47

Table 5.7: The HPCC performance for various platforms. HPCC-*x* is the system with the HPCC ID *x* [3]. The machines HPCC-224 and HPCC-227, and HPCC-286 and HPCC-289 are of brand TopSpin/Cisco and by Intel Endeavor, respectively. Smaller values are better for the Latency column, and worse for the others.

results and, for comparison, results published by HPCC for four modern and similarly-sized HPC clusters [3]. For HPL, only the performance of the `c1.xlarge` is comparable to that of an HPC system. However, for STREAM, and RandomAccess the performance of the EC2 clusters is similar or better than the performance of the HPC clusters. We attribute this mixed behavior to the network characteristics: the EC2 platform has much higher latency, which has an important negative impact on the performance of the HPL benchmark. In particular, this relatively low network performance means that the ratio between the theoretical peak performance and achieved HPL performance increases with the number of instances, making the virtual EC2 clusters poorly scalable. Thus, for scientific computing applications similar to HPL the virtual EC2 clusters can lead to an order of magnitude lower performance for large system sizes (1024 cores and higher). The performance of the GoGrid clusters with the single core instances is as expected, but we observe scalability problems with the 3 core `GG.xlarge` instances. In comparison with previously reported results, the DGEMM performance of `m1.large` (`c1.medium`) instances is similar to that of Altix4700 (ICE) [177], and the memory bandwidth of Cray X1 (2003) is several times faster than that of the fastest cloud resource currently available [69].

Performance Consistency

An important question related to clouds is *Is the performance consistent?* Previous work on virtualization has shown that many virtualization packages deliver the same performance under identical tests for virtual machines running in an isolated environment [52]. However, it is unclear if this holds for virtual machines running in a large-scale cloud (shared) environment. Therefore, we now present a preliminary assessment of the performance consistency.

To get a better picture of the side effects caused by the sharing with other users the same physical resource, we have assessed the performance consistency of different clouds by running the LMBench (computation and OS) and CacheBench (I/O) benchmarks multiple times on the same type of virtual resources.

Figure 5.5 shows the performance consistency for the LMBench benchmark with the float (top) and double (bottom) operations. We observe that although the performance is consistent for the `m1.xlarge` and `Mosso.large` instances, there is noticeable performance variability with the `GG.xlarge` and `EH.small` instances; performance variability depends on the instances used, and probably to the background activity on the virtual machines.

Figure 5.6 summarizes the results for the CacheBench suite for read (top), write (middle), and Rd-Mod-Wr (bottom) operations. Similarly to the LMBench benchmark, the `GG.large` and `EH.small` types have important differences between the min, mean,

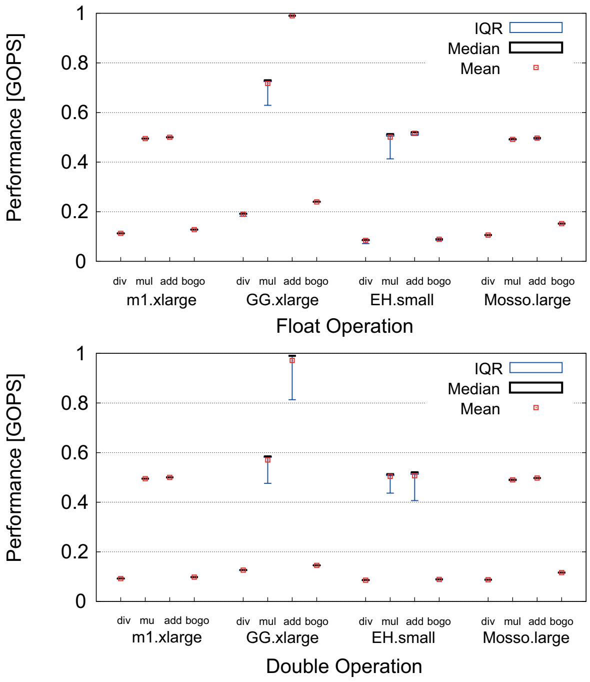


Figure 5.5: Performance consistency of cloud instance types with the LmBench benchmark with various float (top) and double (bottom) operations. IQR denotes the interquartile range.

and max performance even for medium working set sizes, such as $10^{10}B$. The best-performer in terms of computation, `GG.xlarge`, is unstable; this makes cloud vendor selection an even more difficult problem. The different level of performance consistency we have observed across different benchmarks and different instance types motivates us to explore the performance variability of public clouds in depth in the next chapter.

5.3 Clouds versus Other Scientific Computing Infrastructures

In this section we present a comparison between clouds and other scientific computing infrastructures.

5.3.1 Method

We use trace-based simulation to compare clouds with scientific computing infrastructures. To this end, we first extract the performance characteristics from long-term workload traces of scientific computing infrastructures; we call these infrastructures *source environments*. Then, we compare these characteristics with those of a cloud execution.

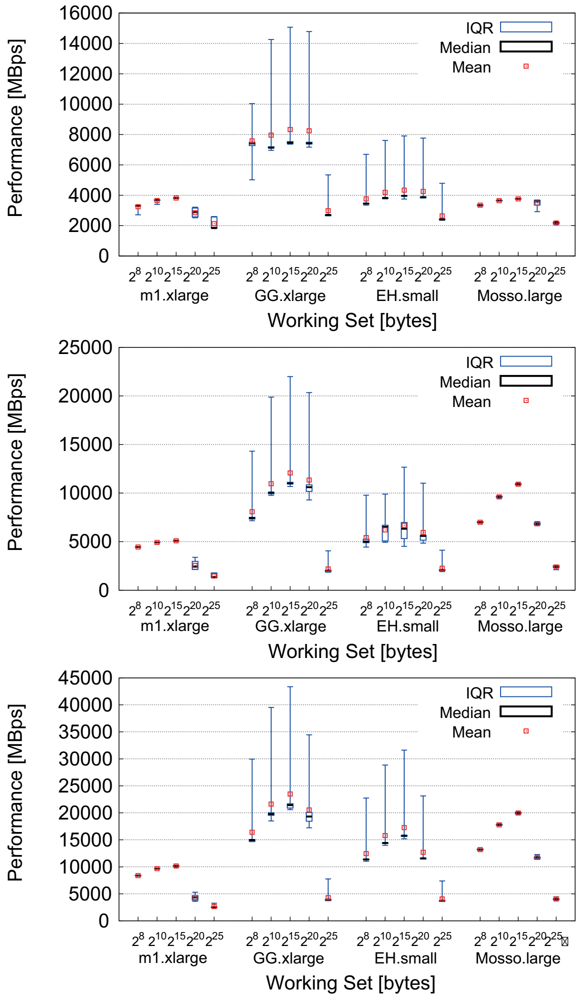


Figure 5.6: Performance consistency of cloud instance types with the CacheBench benchmark with read (top), write (middle), and Rd-Mod-Wr (bottom) operations. IQR denotes the inter-quartile range.

System model We define two performance models of clouds, which differ by the factor that jobs are slowed down. The *cloud with source-like performance* is a theoretical cloud environment that comprises the same resources as the source environment. In this cloud model, the runtimes of jobs executed in the cloud are equal to those recorded in the source environment's workload traces (no slowdown). This model is akin to having a grid being converted into a cloud of identical performance and thus it is useful for assessing the theoretical performance of future and more mature clouds. However, as we have shown in Section 5.2, in real clouds performance is below the theoretical peak, and for parallel jobs the achieved efficiency is lower than that achieved in grids. Thus, we introduce the second model, the *clouds with real performance*, in which the runtimes of jobs executed in the cloud are extended by a factor, which we call the *slowdown factor*, derived from the empirical evaluation presented in Section 5.2. The system equivalence between clouds and source environments is assumed in this model, and ensured in practice by the complete system virtualization [120] employed by all the clouds investigated in this chapter.

Job execution model For job execution we assume exclusive resource use: for each job in the trace, the necessary resources are acquired from the cloud, then released after the job has been executed. We relax this assumption in Section 5.3.3.

System workloads To compare the performance of clouds with other infrastructures, we use the workload traces shown in Table 5.8, where the ID of the trace indicates the system from which it was taken; please see [104, 4] for more details about each trace.

Performance metrics We measure the performance of all environments using the three traditional metrics [79]: *wait time (WT)*, *response time (ReT)*, and *bounded slowdown (BSD)*)–the ratio between the job response time in the real vs. an exclusively-used environment, with a bound that eliminates the bias towards short jobs. The BSD is expressed as $BSD = \max(1, ReT/\max(10, ReT - WT))$, where 10 is the bound that eliminates the bias of jobs with runtime below 10 seconds. We compute for each job the three metrics and report for a complete workload the average values for these metrics, AWT, AReT, and ABSD, respectively.

Cost metrics We report for the two cloud models the total cost of workload execution, defined as the number of instance-hours used to complete all the jobs in the workload. This value can be converted directly into the cost for executing the whole workload for $/CPU-hour and similar pricing models, such as Amazon EC2's.

5.3.2 Experimental Setup

System setup We use the DGSIM simulator [108] to analyze the performance of cloud environments. We have extended DGSIM with the two cloud models, and used it to simulate the execution of real scientific computing workloads on cloud computing infrastructures. To model the slowdown of jobs when using clouds with real performance, we have used different slowdown factors. Specifically, single-processor jobs are slowed-down by a factor of 7, which is the average performance ratio between theoretical and achieved performance analyzed in Section 5.2.3, and parallel jobs are slowed-down by a factor up to 10 depending on the job size, due to the HPL performance degradation with job size described in Section 5.2.3.

Workload setup We use as input workload the ten workload traces presented in Table 5.8. The traces Grid3 and LCG do not include the job waiting time information; only for these two traces we set the waiting time for all jobs to zero, which favors these two grids in comparison with clouds. The wait time of jobs executed in the cloud (also their AWT) is set to the resource acquisition and release time obtained from real measurements

Trace ID, Source (Trace ID in Archive)	Trace			System		
	Time [mo.]	Number of		Size		Load [%]
		Jobs	Users	Sites	CPUs	
Grid Workloads Archive [104], 6 traces						
1. DAS-2 (1)	18	1.1M	333	5	0.4K	15+
2. RAL (6)	12	0.2M	208	1	0.8K	85+
3. GLOW (7)	3	0.2M	18	1	1.6K	60+
4. Grid3 (8)	18	1.3M	19	29	3.5K	-
5. SharcNet (10)	13	1.1M	412	10	6.8K	-
6. LCG (11)	1	0.2M	216	200+	24.4K	-
Parallel Workloads Archive [4], 4 traces						
7. CTC SP2 (6)	11	0.1M	679	1	0.4K	66
8. SDSC SP2 (9)	24	0.1M	437	1	0.1K	83
9. LANLO2K (10)	5	0.1M	337	1	2.0K	64
10. SDSC DS (19)	13	0.1M	460	1	1.7K	63

Table 5.8: The characteristics of the workload traces.

(see Section 5.2.3).

Performance analysis tools We use the Grid Workloads Archive tools [104] to extract the performance metrics from the workload traces of grids and PPIs. We extend these tools to also analyze cloud performance metrics such as cost.

5.3.3 Results

Our experiments follow two main aspects: performance comparison of the workload execution in source environments (grids, PPIs, etc.) and in clouds, and the performance-cost-security trade-off. We present the experimental results for each main aspect, in turn.

Source environments (grids, PPIs, etc.) vs. clouds

We compare the execution in source environments (grids, PPIs, etc.) and in clouds of the ten workload traces described in Table 5.8. Table 5.9 summarizes the results of this comparison, on which we comment below.

An order of magnitude better performance is needed for clouds to be useful for daily scientific computing. The performance of the cloud with real performance model is insufficient to make a strong case for clouds replacing grids and PPIs as a scientific computing infrastructure. The response time of these clouds is higher than that of the source environment by a factor of 4-10. In contrast, the response time of the clouds with source-like performance is much better, leading in general to significant gains (up to 80% faster average job response time) and at worst to 1% higher AWT (for traces of Grid3 and

	Source env. (Grid/PPI)			Cloud (real performance)			Cloud (source performance)		
Trace ID	AWT [s]	AReT [s]	ABSD (10s)	AReT [s]	ABSD (10s)	Total Cost [CPU-h,M]	AReT [s]	ABSD (10s)	Total Cost [CPU-h,M]
DAS-2	432	802	11	2,292	2.39	2	450	2	1.19
RAL	13,214	27,807	68	131,300	1	40	18,837	1	6.39
GLOW	9,162	17,643	55	59,448	1	3	8,561	1	0.60
Grid3	-	7,199	-	50,470	3	19	7,279	3	3.60
SharcNet	31,017	61,682	242	219,212	1	73	31,711	1	11.34
LCG	-	9,011	-	63,158	1	3	9,091	1	0.62
CTC SP2	25,748	37,019	78	75,706	1	2	11,351	1	0.30
SDSC SP2	26,705	33,388	389	46,818	2	1	6,763	2	0.16
LANL O2K	4,658	9,594	61	37,786	2	1	5,016	2	0.26
SDSC DS	32,271	33,807	516	57,065	2	2	6,790	2	0.25

Table 5.9: The results of the comparison between workload execution in source environments (grids, PPIs, etc.) and in clouds. The "-" sign denotes missing data in the original traces. For the two Cloud models AWT=80s (see text). The total cost for the two Cloud models is expressed in millions of CPU-hours.

LCG, which are assumed conservatively to always have zero waiting time[1]). We conclude that if clouds would offer an order of magnitude higher performance than the performance observed in this study, they would form an attractive alternative for scientific computing, not considering costs.

Price-wise, clouds are reasonably cheap for scientific computing, if the usage and funding scenarios allow it (but usually they do not). Looking at costs, and assuming the external operational costs in the cloud to be zero, one million EC2-hours equate to \$100,000. Thus, to execute the total workload of RAL over one year (12 months) would cost \$4,029,000 on Amazon EC2. Similarly, the total workload of DAS-2 over one year and a half (18 months) would cost \$166,000 on Amazon EC2. Both these sums are much lower than the cost of these infrastructures, which includes resource acquisition, operation, and maintenance. To better understand the meaning of these sums, consider the scenario (disadvantageous for the clouds) in which the original systems would have been sized to accommodate strictly the average system load, and the operation and maintenance costs would have been zero. Even in this scenario using Amazon EC2 is cheaper. We attribute this difference to the economy of scale discussed in a recent study [15]: the price of the basic operations in a very large data center can be an order of magnitude lower than in a grid or data center of regular size. However, despite the apparent cost saving it is not clear that the transition to clouds would have been possible for either of these grids. Under the current performance exhibited by clouds, the use of EC2 would have resulted in response times three to four times higher than in the original system, which would have been in conflict with the mission of RAL as a production environment.

---

[1] Although we realize the Grid3 and LCG grids do not have zero waiting time, we follow a conservative approach in which we favor grids against clouds, as the latter are the *new* technology.

Relative Cost	DAS-2	Grid3	LCG	LANL O2K
$\frac{\lvert S2-S1 \rvert}{S1} \times 100$ [%]	30.2	11.5	9.3	9.1

Table 5.10: Relative strategy performance: resource bulk allocation (S2) vs. resource acquisition and release per job (S1). Only performance differences above 5% are shown.

A similar concern can be formulated for DAS-2. Moreover, DAS-2 is specifically targeting research in computer science, and its community would not have been satisfied to use commodity resources instead of a state-of-the-art environment comprising among others high-performance lambda networks; other new resource types, such as GPUs and Cell processors, are currently available in grids but not in clouds. Looking at the funding scenario, it is not clear if finance could have been secured for virtual resources; one of the main outcomes of the long-running EGEE project is the creation of a European Grid infrastructure. Related concerns have been formulated elsewhere [15].

Clouds are now a viable alternative for short deadlines. A low and steady job wait time leads to much lower (bounded) slow-down for any cloud model, when compared to the source environment. The average bounded slowdown (ABSD, see Section 5.3.1) observed in real grids and PPIs is for our traces between 11 and over 500**!**, but below 3.5 and even 1.5 for the cloud models with low and with high utilization. The meaning of the ABSD metric is application-specific, and the actual ABSD value may seem to overemphasize the difference between grids and clouds. However, the presence of high and unpredictable wait times even for short jobs, captured here by the high ABSD values, is one of the major concerns in adopting shared infrastructures such as grids [99, 156]. We conclude that cloud is already a viable alternative for scientific computing projects with tight deadline and few short-running jobs remaining, if the project has the needed funds.

Performance and Security vs. Cost

Currently, clouds lease resources but do not offer a resource management service that can use the leased resources. Thus, the cloud adopter may use any of the resource management middleware from grids and PPIs; for a review of grid middleware we refer to our recent work [102]. We have already introduced the basic concepts of cloud resource management in Section 5.2.2, and explored the potential of a cloud resource management strategy (*strategy S1*) for which resources are acquired and released for each submitted job in Section 5.3. This strategy has good security and resource setup flexibility, but may incur high time and cost overheads, as resources that could otherwise have been reused are released as soon as the job completes. As an alternative, we investigate now the potential of a cloud resource management strategy in which resources are allocated in bulk for all users, and released only when there is no job left to be served (*strategy S2*). To compare these two cloud resource management strategies, we use the experimental setup described in Section 5.3.2; Table 5.10 shows the obtained results. The maximum relative cost difference between the strategies is for these traces around 30% (the DAS-2 trace); in three cases, around 10% of the total cost is to be gained. Given these cost differences, *we raise as a future research problem optimizing the application execution as a cost-performance-security trade-off*.

5.4 Related work

In this section we review related work from three areas: clouds, virtualization, and system performance evaluation.

Clouds and Virtualization There has been a spur of research activity in assessing the performance of virtualized resources, in cloud computing environments [63, 163, 208, 157, 170] and in general [23, 52, 145, 231, 194, 152, 230]. In contrast to this body of previous work, ours is different in scope: we perform extensive measurements using general purpose and high-performance computing benchmarks to compare several clouds, and we compare clouds with other environments based on real long-term scientific computing traces. Our study is also much broader in size: we perform in this chapter an evaluation using over 25 individual benchmarks on over 10 cloud instance types, which is an order of magnitude larger than previous work (though size does not simply add to quality).

Performance studies using general purpose benchmarks have shown that the overhead incurred by virtualization can be below 5% for computation [23, 52] and below 15% for networking [23, 145]. Similarly, the performance loss due to virtualization for parallel I/O and web server I/O has been shown to be below 30% [232] and 10% [47, 148], respectively. In contrast to these, our work shows that virtualized resources obtained from public clouds can have a much lower performance than the theoretical peak.

Recently, much interest for the use of virtualization has been shown by the HPC community, spurred by two seminal studies [231, 98] that find virtualization overhead to be negligible for compute-intensive HPC kernels and applications such as the NAS NPB benchmarks; other studies have investigated virtualization performance for specific HPC application domains [85, 230], or for mixtures of Web and HPC workloads running on virtualized (shared) resources [233]. Our work differs significantly from these previous approaches in target (clouds as black boxes vs. owned and controllable infrastructure) and in size. For clouds, the study of performance and cost of executing a scientific workflow, Montage, in clouds [63] investigates cost-performance trade-offs between clouds and grids, but uses a single application on a single cloud, and the application itself is remote from the mainstream HPC scientific community. Also close to our work is the seminal study of Amazon S3 [163], which also includes a performance evaluation of file transfers between Amazon EC2 and S3. Our work complements this study by analyzing the performance of Amazon EC2, the other major Amazon cloud service; we also test more clouds and use scientific workloads. Several small-scale performance studies of Amazon EC2 have been recently conducted: the study of Amazon EC2 performance using the NPB benchmark suite [208] or selected HPC benchmarks [78], the early comparative study of Eucalyptus and EC2 performance [157], the study of file transfer performance between Amazon EC2 and S3 [28], etc. An early comparative study of the DawningCloud and several operational models [211] extends the comparison method employed for Eucalyptus [157], but uses job emulation instead of job execution. Our performance evaluation results extend and complement these previous findings, and gives more insights into the performance of EC2 and other clouds.

Other (Early) Performance Evaluation Much work has been put into the evaluation of novel supercomputers [165, 69, 119, 207, 177, 9] and non-traditional systems [31, 217, 100, 99, 164] for scientific computing. We share much of the used methodology with previous work; we see this as an advantage in that our results are readily comparable with existing results. The two main differences between this body of previous work and ours are that we focus on a different platform (that is, clouds) and that we target a broader scientific computing community (e.g., also users of grids and small clusters).

5.5 Summary

With the emergence of cloud computing as a paradigm in which scientific computing can done exclusively on resources leased only when needed from big data centers, e-scientists are faced with a new platform option. However, the initial target workloads of clouds does not match the characteristics of scientific computing workloads. Thus, in this chapter we seek to answer the research question *Is the performance of clouds sufficient for scientific computing?* To this end, we have first performed an empirical performance evaluation of four public computing clouds, including Amazon EC2, one of the largest commercial clouds currently in production. Our main finding here is that the compute performance of the tested clouds is low. Last, we have compared the performance and cost of clouds with those of scientific computing alternatives such as grids and parallel production infrastructures. We have found that, while current cloud computing services are insufficient for scientific computing at large, they may still be a good solution for the scientists who need resources instantly and temporarily.

Chapter 6

Performance variability of production cloud services[*]

Cloud computing is emerging as an alternative to traditional computing and software services such as grid computing and online payment. With cloud computing, resources and software are no longer hosted and operated by the user, but instead leased from large-scale data centers and service specialists strictly when needed. An important hurdle to cloud adoption is trusting that the cloud services are dependable, for example that their performance is stable over long time periods. However, providers do not disclose their infrastructure characteristics or how they change, and operate their physical resources in time-sharing mode; this situation may cause significant performance variability. To find out if the performance variability of cloud services is significant, in this chapter we present the first long-term study on the variability of performance as exhibited by ten production cloud services of two popular cloud service providers, Amazon and Google.

Ideally, clouds should provide services such as running a user-given computation with performance equivalent to that of dedicated environments with similar characteristics. However, the performance characteristics of a cloud may vary over time as a result of changes that are not discussed with the users. Moreover, unlike current data centers and grids, clouds time-share their resources, and time-shared platforms have been shown [16] since the 1990s to cause complex performance variability and even performance degradation.

Although it would be beneficial to both researchers and system designers, there currently exists no investigation of performance variability for cloud services. Understanding this variability guides in many ways research and system design. For example, it can help in selecting the service provider, designing and tuning schedulers [106], and detecting and predicting failures [234]. Tens of clouds [95, 137] started to offer services in the past few

---

[*] This chapter is based on previous work published in the *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGRID'11) [109].

years; of these, Amazon Web Services (AWS) and Google App Engine (GAE) are two popular production clouds [15]. A number of studies [63, 163, 208, 157, 163, 161, 15], including our previous work [161], investigate the performance of AWS, but none investigates the performance variability or even system availability for a period of over two months.

In this chapter, our goal is to perform a comprehensive investigation of the long-term variability of performance for production cloud services. Toward this end, we first collect performance traces corresponding to ten production cloud services provided by Amazon Web Services and Google App Engine, currently two of the largest commercial clouds. Then we analyze the collected traces, revealing for each service both summary statistics and the presence or absence of performance time patterns. Finally, we evaluate through trace-based simulations the impact of the variability observed in the studied traces on three large-scale applications that are executed today or may be executed in the cloud in the (near) future: executing scientific computing workloads on cloud resources, selling virtual goods through cloud-based payment services, and updating the virtual world status of social games through cloud-based database services.

The rest of the chapter is structured as follows. In Section 6.1 we present an overview of the production cloud services that we investigate in this chapter. Then, in Section 6.2 we describe the method of our performance variability analysis. In Section 6.3 and Section 6.4 we present the results of our analysis for the AWS and GAE datasets, respectively. Then, in Section 6.5 we assess the impact of the variability of cloud service performance on large-scale applications using trace-based simulations. Finally, we compare our analysis with related work in Section 6.6, and we summarize the chapter in Section 6.7.

6.1 Production Cloud Services

Cloud computing comprises both the offering of infrastructure and software services [15, 95]. A cloud offering infrastructure services such as computing cycles, storage space or queueing services acts as Infrastructure as a Service (IaaS). A cloud offering platform services such as a runtime environment for compiled/interpreted application code operating on virtualized resources acts as Platform as a Service (PaaS). A third category of clouds, Software as a Service (SaaS), incorporate the old idea of providing applications to users, over the Internet.

To accommodate this broad definition of clouds, in our model each cloud provides a set of *services*, and each service a set of *operations*. In our terminology, a *production cloud* is a cloud that operates on the market, that is, it has real customers that use its services. Tens of cloud providers have entered the market in the last five last years, including Amazon Web Services (2006), ENKI (2003), Joyent (2004), Mosso (2006), RightScale (2008), GoGrid (2008), Google App Engine (2008) and recently Microsoft Azure(2010).

From the clouds already in production, Amazon Web Services and Google App Engine are reported to have the largest number of clients [137] which we describe in turn.

6.1.1 Amazon Web Services

Amazon Web Services (AWS) is an IaaS cloud comprising services such as the Elastic Compute Cloud (EC2, performing computing resource provisioning or web hosting operations), Elastic Block Storage and its frontend Simple Storage Service (S3, storage), Simple Queue Service (SQS, message queuing and synchronization), Simple DB (SDB, database), and the Flexible Payments Service (FPS, micro-payments). As operation examples, the EC2 provides three main operations, for resource acquisition, resource release, and resource status query.

Through its services EC2 and S3, AWS can rent infrastructure resources; the EC2 offering comprises more than 10 types of virtual resources (*instance types*) and the S3 offering comprises 2 types of resources. Estimates based on the numerical properties of identifiers given to provided services indicate that Amazon EC2 rents over 40,000 virtual resources per day [172, 173], which is two orders of magnitude more than its competitors GoGrid and RightScale [173], and around the size of the largest scientific grid in production.

6.1.2 Google App Engine

The Google App Engine (GAE) is an PaaS cloud comprising services such as Java and Python Runtime Environments (Run, providing application execution operations), the Datastore (database), Memcache (caching), and URL Fetch (web crawling). Although through the Run service users consume computing and storage resources from the underlying GAE infrastructure, GAE does not provide root access to these resources, like the AWS.

6.2 Method

To characterize the long-term performance variability of cloud services we first build meaningful datasets from performance traces taken from production clouds, and then we analyze these datasets and characterize the performance variability.

Our method is built around the notion of *performance indicators*. We call a performance indicator the stochastic variable that describes the performance delivered by one operation or by a typical sequence of operations over time. For example, the performance indicators for Amazon include the response time of the resource acquisition operation of the EC2 service.

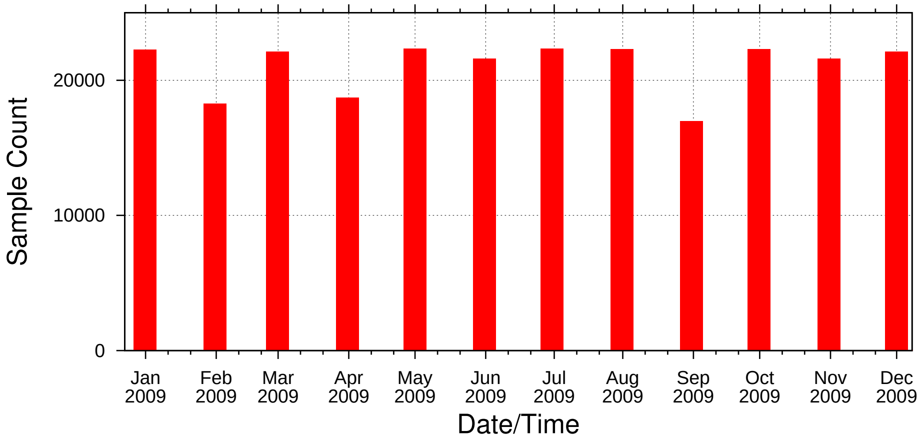


Figure 6.1: Number of monthly data samples.

6.2.1 Performance Traces of Cloud Services

Data Source To characterize AWS and GAE we first acquire data from the performance database created by Hyperic's CloudStatus team [2]. CloudStatus provides real-time values and weekly averages of about thirty performance indicators for AWS and GAE. In particular, it provides performance indicators for five main services provided by AWS (EC2, S3, SDB, SQS, and FPS) and for four main services provided by GAE (Run, Datastore, Memcache, and URL Fetch). CloudStatus obtains values for the various performance indicators by running performance probes periodically, with a sampling rate of under 2 minutes. The CloudStatus probes can be reimplemented easily; we have repeated some of the CloudStatus experiments in our previous work [161, 105], with similar results. We conclude that using CloudStatus data reduces the cost of our study, but does not reduce the applicability of the results.

Data Sanitation We have acquired data from CloudStatus through a sequence of web crawls (samples). The availability and robustness of our crawling setup resulted in 253,174 useful samples, or 96.3% of the maximum number of samples possible for the year. Figure 6.1 shows the number of samples taken every month; during February, April, and September 2009 our crawling infrastructure did not manage to obtain useful samples repeatedly (indicated by the reduced height of the ”Sample Count” bars). Mostly during these month we have lost 9,626 samples due to missing or invalid JSON data; however, we have obtained 76–96% of the maximum number of samples during these three months.

6.2.2 Method of Analysis

For each of the traces we extract the performance indicators, to which we apply independently an analysis method with three steps: find out if variability is present at all, find out the main characteristics of the variability, and analyze in detail the variability time patterns. We explain each step in the following, in turn.

To find out if variability is present at all we select one month of data from our traces and plot the values of the performance indicator where a wide range of values may indicate variability. The month selection should ensure that the selected month does not correspond to a single calendar month (to catch some human-scheduled system transitions), is placed towards the end of the year 2009 (to be more relevant) but does not overlap with December 2009 (to avoid catching Christmas effects).

To find out the characteristics of the variability we compute six basic statistics, the five quartiles (Q_0–Q_4) including the median (Q_2), the mean, and the standard deviation. We also compute one derivative statistic, the Inter-Quartile Range (IQR, defined as $Q_3 - Q_1$). We thus characterize for each studied parameter its location (mean and median), and its variability or scale (the standard deviation, the IQR, and the range). Either a relative difference between the mean and the median of over 10 percent, or a coefficient of variation above 1.10 indicate high variability and possibly a non-normal distribution of values which impacts negatively the ability to enforce soft performance guarantees. Similarly, a ratio between the IQR and the median above 0.5 indicates that the bulk of the performance observations have high variability, and a ratio between range and the IQR above 4 indicates that the performance outliers are severe.

Finally, to analyze the variability over time we investigate for each performance indicator the presence of yearly (month-of-year and week-of-year), monthly (day-of-month), weekly (day-of-week and workday/weekend), and daily patterns (hour-of-day). To this end, we first split for each time pattern investigated the complete dataset into subsets, one for each category corresponding to the time pattern. For example, to investigate the monthly time pattern we split the complete dataset into twelve subsets comprising the performance value samples observed during a specific month. Then, we compute for each subset the basic and derivative statistics performed over the complete dataset in the second step, and plot them for visual inspection. Last, we analyze the results and the plots, record the absence/presence of each investigated time pattern, and attempt to detect new time patterns.

6.2.3 Is Variability Present?

An important assumption of this chapter is that the performance variability of production cloud services indeed exists. We follow in this section the first step of our analysis method and verify this assumption.

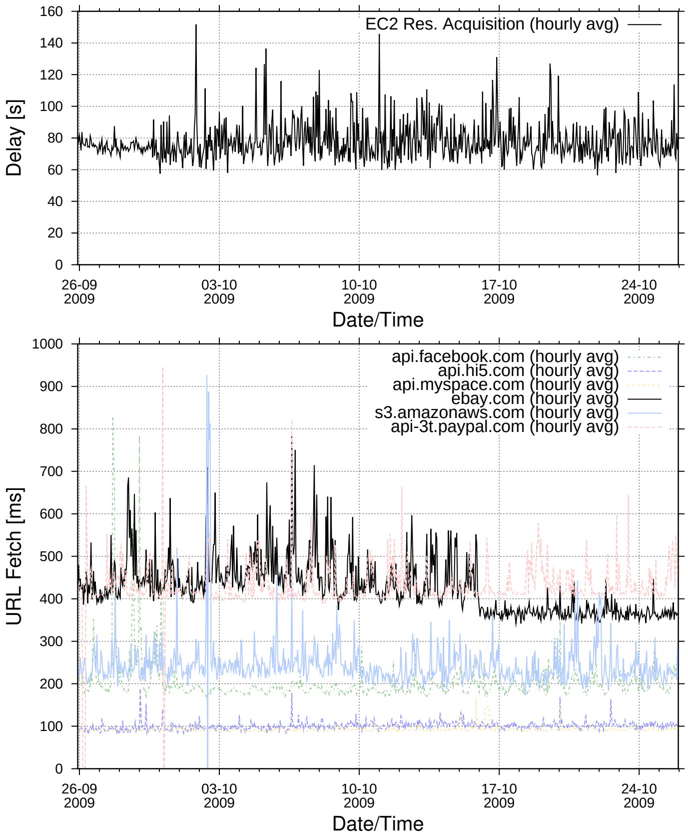


Figure 6.2: Performance variability for two selected cloud services during the period Sep 26, 2009 to Oct 26, 2009: (top) for Amazon EC2, and (bottom) for Google URL Fetch.

Towards this end, we present the results for the selection of data from Sep 26 to Oct 26, 2009. For this month, we present here only the results corresponding to one sample service from each of the Amazon and Google clouds. Figure 6.2 shows the performance variability exhibited by the Amazon EC2 service (top of the figure, one performance indicator) and by the Google URL Fetch service (bottom of the figure, six performance indicators) during the selected month. For EC2, the range of values indicates moderate-to-high performance variability. For URL Fetch, the wide ranges of the six indicators indicate high variability for all URL Fetch operations, regardless of the target URL. In addition, the URL Fetch service targeting eBay web pages suffers from a visible decrease of performance around Oct 17, 2009. We have also analyzed the results for the selected month for all the other cloud services we investigate in this chapter, and have experimented with multiple one-month selections that follow the rules stated by our analysis method; in all cases we have obtained similar results (for brevity reasons not shown). To conclude, the effects observed in this section give strong evidence of the presence of performance variability in cloud services, and motivate an in-depth analysis of the performance variability of both Amazon and Google cloud services.

Service	Min	Q1	Median	Q3	Max	Mean	SD
EC2 [s]							
Deployment Latency	57.00	73.59	75.70	78.50	122.10	76.62	5.17
S3 [MBps]							
GET EU HIGH	0.45	0.65	0.68	0.70	0.78	0.68	0.30
GET US HIGH	8.60	15.50	17.10	18.50	25.90	16.93	2.39
PUT EU HIGH	1.00	1.30	1.40	1.40	1.50	1.38	0.10
PUT US HIGH	4.09	8.10	8.40	8.60	9.10	8.26	0.55
SDB [ms]							
Query Response Time	28.14	31.76	32.81	33.77	85.40	32.94	2.39
Update Latency	297.54	342.52	361.97	376.95	538.37	359.81	26.71
SQS [s]							
Lag Time	1.35	1.47	1.50	1.79	6.62	1.81	0.82
FPS [ms]							
Latency	0.00	48.97	53.88	76.06	386.43	63.04	23.22

Table 6.1: Summary statistics for Amazon Web Services's cloud services.

6.3 The Analysis of the AWS Dataset

In this section, we present the analysis of the AWS dataset. Each service comprises several operations, and for each operation, we investigate the performance indicators to understand the performance variability delivered by these operations.

6.3.1 Summary Statistics

In this section we follow the second step of our analysis method and analyze the summary statistics for AWS; Table 6.1 summarizes the results. Although the EC2 deployment latency has low IQR, it has a high range. We observe higher range and IQR for the performance of S3 measured from small EC2 instances (see Section 6.3.3) compared to performance measured from large and extra large EC2 instances. Similar to EC2, SDB also has low IQR but a high range especially for the update operations. Finally, FPS latency is highly variable which has implications for the applications using this service for payment operations as we present in Section 6.5.3.

6.3.2 Amazon Elastic Compute Cloud (EC2)

CloudStatus.com reports the following performance indicator for the EC2 service:

1. **Deployment Latency** - The time it takes to start an m1.small instance, from the time startup is initiated to the time that the instance is available.

Figure 6.3 shows weekly statistical properties of the EC2 Resource Acquisition operation. We observe higher IQR and range for deployment latency from week 41 till the

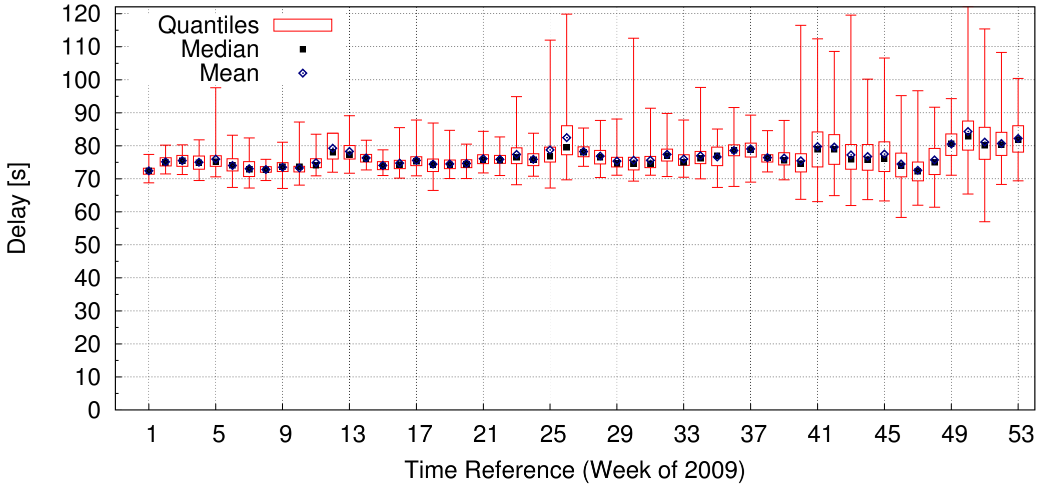


Figure 6.3: Amazon EC2: The weekly statistical properties of the resource acquisition operation. The box and whiskers show min-Q_1-Q_3-max.

end of the year compared to the remainder of the year probably due to increasing user base of EC2. Steady performance for the deployment latency is especially important for applications which uses the EC2 for auto-scaling.

6.3.3 Amazon Simple Storage Service (S3)

CloudStatus.com reports the throughput of S3 where the throughput is measured by issuing S3 requests from US-based EC2 instances to S3 buckets in the US and Europe. ”High I/O” metrics reflect throughput for operations on Large and Extra Large EC2 instances.

The following performance indicators are reported:

1. **Get Throughput (bytes/second)** - Estimated rate at which an object in a bucket is read (GET).

2. **Put Throughput Per Second (bytes/second)** - Estimated rate at which an object in a bucket is written (PUT).

Figure 6.4 (top) depicts the hourly statistical properties of the S3 service GET EU HI operation. The range has a pronounced daily pattern, with evening and night hours (from 7PM to 2AM the next day) exhibiting much lower minimal transfer rates, and the work day hours (from 8AM to 3PM) exhibiting much higher minimal transfer rates.

Figure 6.4 (middle) shows the monthly statistical properties of the S3 service GET EU HI operation. The operation’s performance changes its pattern in August 2009: the last five months of the year exhibit much lower IQR and range, and have significantly better performance – the median throughput increases from 660 KBps to 710 KBps.

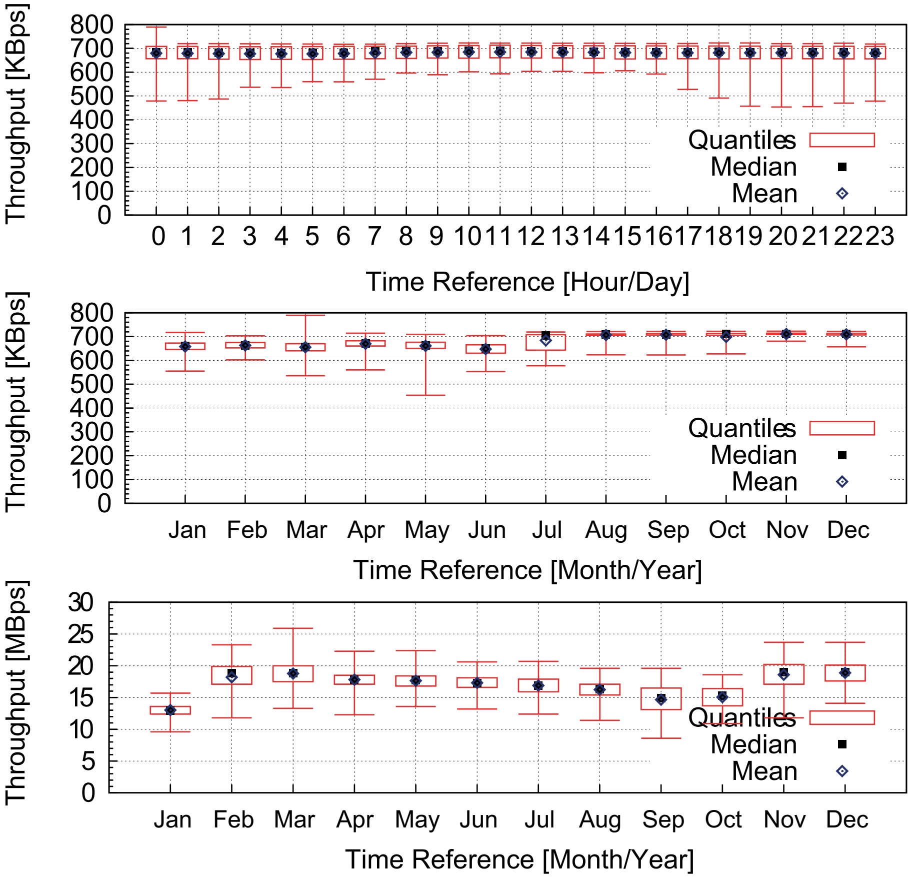


Figure 6.4: Amazon S3: The hourly statistical properties of GET EU HI operations (top), and the monthly statistical properties of the GET EU HI operations (middle) and of GET US HI operations (bottom).

Figure 6.4 (bottom) shows the monthly statistical properties of the S3 service GET US HI operation. The operation exhibits pronounced yearly patterns, with the months January, September, and October 2009 having the lowest mean (and median) performance. Figure 6.4 (bottom) also shows that there exists a wide range of median monthly performance values, from 13 to 19 MBps over the year.

6.3.4 Amazon Simple DB (SDB)

CloudStatus.com reports the following performance indicators for the SDB service:

1. **Query Response Time (ms)** - The time it takes to execute a GetAttributes operation that returns 100 attributes.
2. **Update Latency (ms)** - The time it takes for the updates resulting from a PutAttributes operation to be available to a subsequent GetAttributes operation.

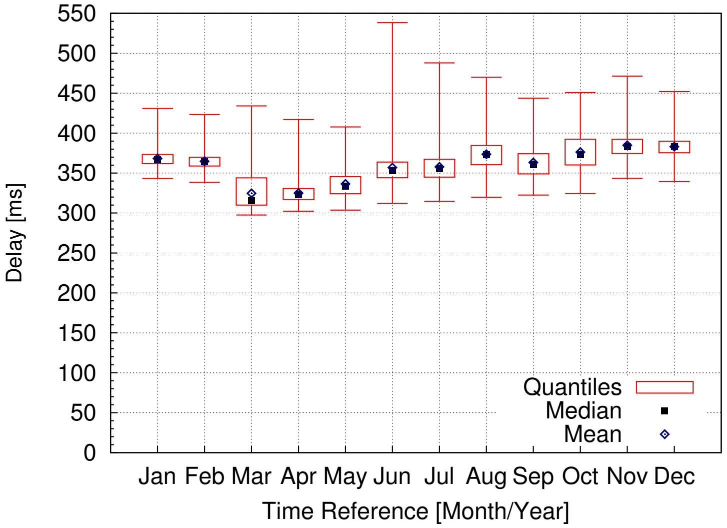


Figure 6.5: Amazon SDB: The monthly statistical properties of the update operation.

Figure 6.5 shows the monthly statistical properties of the SDB Update operation. The monthly median performance has a wide range, from 315 to 383 ms. There is a sudden jump in range in June 2009; the range decreases steadily from June to December to the nominal values observed in the first part of the year. This is significant for applications such as online gaming, in which values above the 99% performance percentile are important, as unhappy users may trigger massive customer departure through their social links (friends and friends-of-friends).

6.3.5 Amazon Simple Queue Service (SQS)

CloudStatus.com reports the following performance indicators for the SQS service:

1. **Average Lag Time (s)** - The time it takes for a posted message to become available to be read. Lag time is monitored for multiple queues that serve requests from inside the cloud. The average is taken over the lag times measured for each monitored queue.

Figure 6.6 depicts the weekly statistical properties of the SQS service. The service exhibits long periods of stability (low IQR and range, similar median performance week after week), for example weeks 5–9 and 26–53, but also periods of high performance variability, especially in weeks 2–4, 13–16, and 20–23. The periods with high performance variability are not always preceded by weeks of moderate variability. The duration of a period with high performance variability can be as short as a single week, for example during week 18.

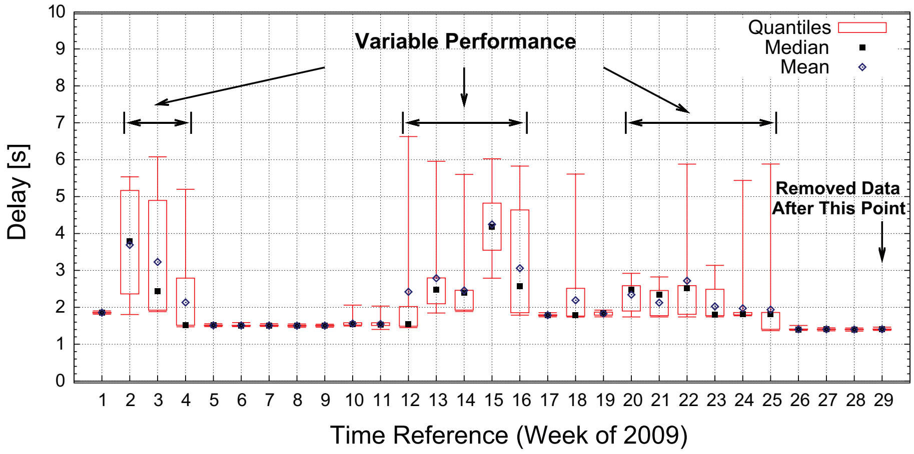


Figure 6.6: Amazon SQS: The weekly statistical properties. The statistics for the weeks 30–53 (not shown) are very similar to those for weeks 26–29.

6.3.6 Amazon Flexible Payment Service (FPS)

CloudStatus.com reports the following performance indicators for the FPS service:

1. **Response Time (s)** - The time it takes to execute a payment transaction. The response time does not include the round trip time to the FPS service nor the time taken to setup pay tokens. Since Amazon reports the response time to the nearest second, payments that complete in less than a second will be recorded as zero.

Figure 6.7 depicts the monthly statistical properties of the FPS service. There is a sudden jump in the monthly median performance in September 2009, from about 50 to about 80 ms; whereas the median is relatively constant before and after the jump. We also observe high variability in the maximum performance values of the FPS service across months.

6.3.7 Summary of the AWS Dataset

The performance results indicate that all Amazon services we analyzed in this section exhibit one or more time patterns and/or periods of time where the service shows special behavior, as summarized in Table 6.2. EC2 exhibits periods of special behavior for the resource acquisition operation (Section 6.3.2). Both storage services of Amazon, SDB and S3, present daily, yearly, and monthly patterns for different operations (Section 6.3.4 and Section 6.3.3). Finally, SQS and FPS show special behavior for specific time periods (Section 6.3.5 and Section 6.3.6).

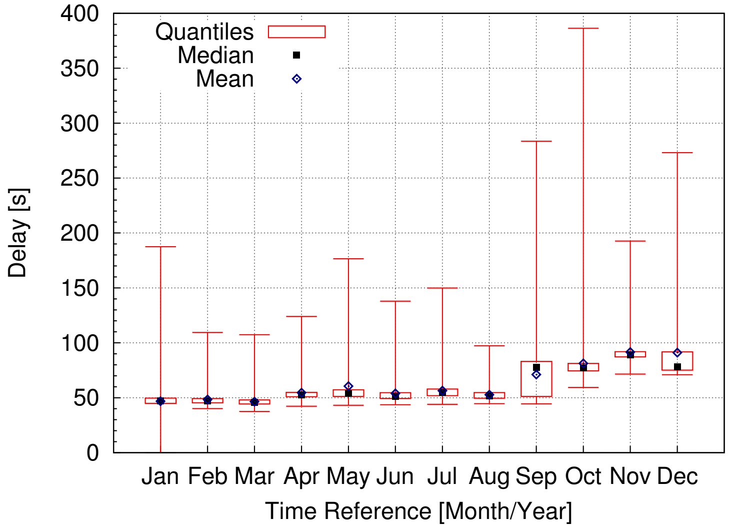


Figure 6.7: Amazon FPS: The monthly statistical properties.

Perf. Indicator	Yearly (Month)	Monthly (Day)	Weekly (Day)	Daily (Hour)	Special Period
Amazon AWS					
EC2					Y
S3	Y		Y	Y	Y
SDB	Y			Y	
SQS					Y
FPS					Y

Table 6.2: Presence of time patterns or special periods for the AWS services. A cell value of Y indicates the presence of a pattern or a special period.

6.4 The Analysis of the Google App Engine Dataset

In this section, we present the analysis of the Google App Engine dataset. Each service comprises several operations, and for each operation, we investigate the performance indicators in detail to understand the performance variability delivered by these operations.

6.4.1 Summary Statistics

In this section we follow the second step of our analysis method and analyze the summary statistics for GAE; Table 6.3 summarizes the results. The GAE Python runtime and Datastore have high range and IQRs leading to highly variable performance. However, we observe relatively stable performance for the Memcache service.

Service	Min	Q1	Median	Q3	Max	Mean	SD
Python Runtime [ms]	1.00	284.14	302.31	340.37	999.65	314.95	76.39
Datastore [ms]							
Create	1040	1280	1420	1710	5590	1600	600
Delete	1.00	344.40	384.22	460.73	999.86	413.24	102.90
Read	1.00	248.55	305.68	383.76	999.27	336.82	118.20
Memcache [ms]							
Get	45.97	50.49	58.73	65.74	251.13	60.03	11.44
Put	33.21	44.21	50.86	60.44	141.25	54.84	13.54
Response	3.04	4.69	5.46	7.04	38.71	6.64	3.39
URL Fetch [ms]							
s3.amazonaws.com	1.01	198.60	226.13	245.83	983.31	214.21	64.10
ebay.com	1.00	388.00	426.74	460.03	999.83	412.57	108.31
api.facebook.com	1.00	172.95	189.39	208.23	998.22	195.76	44.40
api.hi5.com	71.31	95.81	102.58	113.40	478.75	107.03	25.12
api.myspace.com	67.33	90.85	93.36	103.85	515.88	97.90	14.19
paypal.com	1.00	406.57	415.97	431.69	998.39	421.76	35.00

Table 6.3: Summary statistics for Google App Engine's cloud services.

6.4.2 The Google Run Service

CloudStatus.com reports the following performance indicator for the Run service:

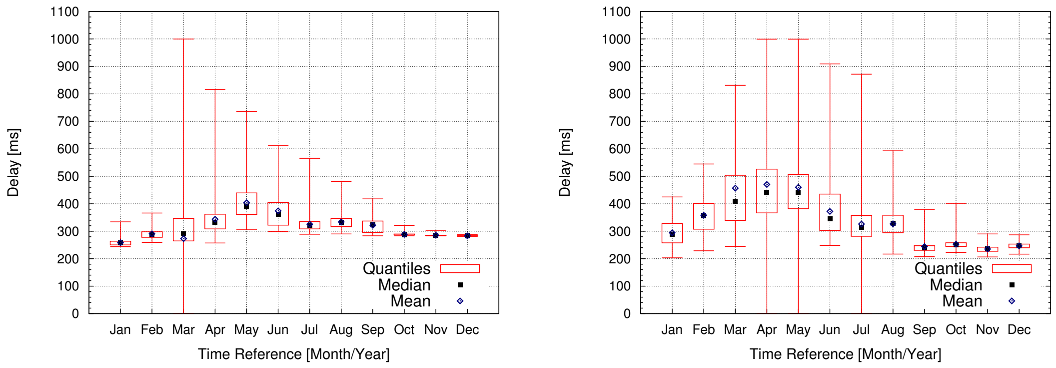
1. **Fibonacci (ms)** - The time it takes to calculate the 27th Fibonacci number in the Python Runtime Environment.

Figure 6.8 (a) depicts the monthly statistical properties of the GAE Python Runtime. The last three months of the year exhibit stable performance, with very low IQR and narrow range, and with steady month-to-month median. Similar to the Amazon SDB service (see Section 6.3.4), the monthly median performance has a wide range, from 257 to 388 ms. Independently of the evolution of the median, there is a sudden jump in range in March 2009; the maximum response time (lowest performance) decreases steadily up to October, from which point the performance becomes steady.

6.4.3 The Google Datastore Service

To measure create/delete/read times CloudStatus uses a simple set of data which we refer to the combination of all these entities as a 'User Group'. CloudStatus.com reports the following performance indicators for the Datastore service:

1. **Create Time (s)** - The time it takes for a transaction that creates a User Group.
2. **Read Time (ms)** - The time it takes to find and read a User Group. Users are randomly selected, and the user key is used to look up the user and profile picture records. Posts are found via a GQL (Google Query Language) ancestor query.



(a) Google Run

(b) Google Datastore

Figure 6.8: The monthly statistical properties of running an application in the Python Runtime Environment with the Google Run service (a) and the read operation for the Google Datastore service (b).

3. **Delete Time (ms)** - The time it takes for a transaction that deletes a User Group.

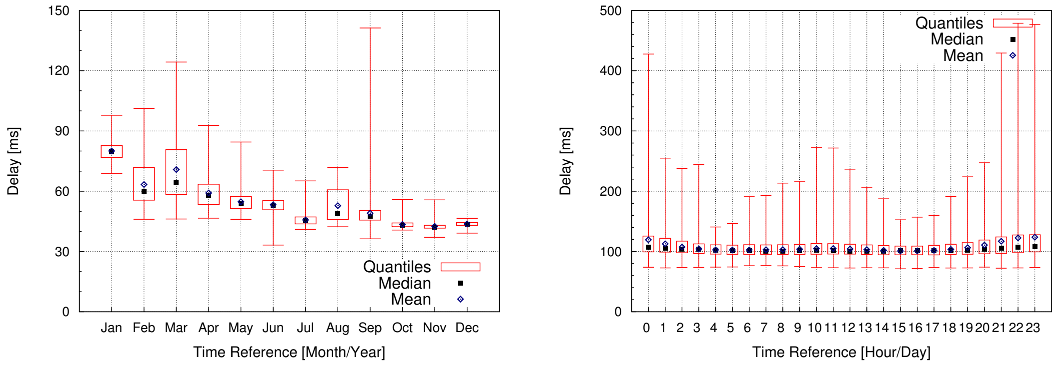
Figure 6.8 (b) depicts the monthly statistical properties of the GAE Datastore service read performance. The last four months of the year exhibit stable performance, with very low IQR and relatively narrow range, and with steady month-to-month median. In addition we observe yearly patterns for the months January through August. Similar to Amazon S3 GET operations, the Datastore service exhibits a high IQR with yearly patterns (Section 6.3.3), and in contrast to S3, the Datastore service read operations exhibit a higher range. Overall, the Update operation exhibits a wide yearly range of monthly median values, from 315 to 383 ms.

6.4.4 The Google Memcache Service

CloudStatus.com reports the following performance indicators for the Memcache service:

1. **Get Time (ms)** - The time it takes to get 1 MB of data from memcache.
2. **Put Time (ms)** - The time it takes to put 1 MB of data in memcache.
3. **Response Time (ms)** - The round-trip time to request and receive 1 byte of data from cache. This is analogous to Get Time, but for a smaller chunk of data.

Figure 6.9 (a) depicts the monthly statistical properties of the Memcache service PUT operation performance. The last three months of the year exhibit stable performance, with very low IQR and relatively narrow range, and with steady month-to-month median. The same trend can be observed for the Memcache GET operation. Uniquely for the



(a) Google Memcache

(b) Google URL Fetch

Figure 6.9: The monthly statistical properties of the PUT operation for the Google Memcache service (a) and the hourly statistical properties for the Google URL Fetch service (b) where the target web site is the Hi5 social network.

Perf. Indicator	Yearly (Month)	Monthly (Day)	Weekly (Day)	Daily (Hour)	Special Period
Google App Engine					
Run				Y	Y
Datastore	Y				Y
Memcache					
URL Fetch			Y	Y	Y

Table 6.4: Presence of time patterns or special periods for the GAE services. A cell value of Y indicates the presence of a pattern or a special period.

Memcache PUT operation, the median performance per month has an increasing trend over the first ten months of the year, with the response time decreasing from 79 to 43 ms.

6.4.5 The Google URL Fetch Service

CloudStatus.com reports the response time (ms) which is obtained by issuing web service requests to several web sites: api.facebook.com, api.hi5.com, api.myspace.com, ebay.com, s3.amazonaws.com, and paypal.com.

Figure 6.9 (b) depicts the hourly statistical properties of the URL Fetch service when the target web site is the Hi5 social network. The ranges of values for the service response times vary greatly over the day, with several peaks. We have observed a similar pattern for other target web sites for which a URL Fetch request is issued.

Section	Application	Used Service
Section 6.5.2	Job execution	GAE Run
Section 6.5.3	Selling virtual goods	AWS FPS
Section 6.5.4	Game status management	AWS SDB GAE Datastore

Table 6.5: Large-scale applications used to analyze the impact of variability.

6.4.6 Summary of the Google App Engine Dataset

The performance results indicate that all GAE services we analyzed in this section exhibit one or more time patterns and/or periods of time where the service provides special behavior, as summarized in Table 6.4. The Python Runtime exhibits periods of special behavior and daily patterns (Section 6.4.2). The Datastore service presents yearly patterns and periods of time with special behavior (Section 6.4.3). The Memcache service performance has also monthly patterns and time patterns of special behavior for various operations (Section 6.4.4). Finally, the URL Fetch service presents weekly and daily patterns, and also shows special behavior for specific time periods for different target websites (Section 6.4.5).

6.5 The Impact of Variability on Large-Scale Applications

In this section we assess the impact of the variability of cloud service performance on large-scale applications using trace-based simulations. Since there currently exists no accepted traces or models of cloud workloads, we propose scenarios in which three realistic applications would use specific cloud services. Table 6.5 summarizes these applications and the main cloud service that they use.

6.5.1 Experimental Setup

Input Data For each application, we use the real system traces described in the section corresponding to the application (column ”Section” in Table 6.5), and the monthly performance variability of the main service leveraged by the ”cloudified” application (column ”Used Service” in Table 6.5).

Simulator We design for each application a simulator that considers from the trace each unit of information, that is, a job record for the Job Execution scenario and the number of daily unique users for the other two scenarios, and assesses the performance for a cloud with stable performance vs variable performance. For each application we select one performance indicator, corresponding to the main cloud service that the ”cloudified” application would use. In our simulations, the variability of this performance indicator, which, given as input to the simulator, is the monthly performance variability analyzed earlier in this chapter. We define the *reference performance* P_{ref} as the average of the twelve monthly medians, and attribute this performance to the cloud with stable performance. To ensure that results are representative, we run each simulation 100 times and report the average results.

Metrics We report the following metrics:

- For the Job Execution scenario, which simulates the execution of compute-intensive jobs from grid and parallel production environments (PPEs), we first report two traditional metrics for the grid and PPE communities: the average response time (**ART**), the average bounded slowdown (**ABSD**) with a threshold of 10 seconds [79]; the ABSD threshold of 10 eliminates the bias of the average toward jobs with runtime below 10 seconds. We also report one cloud-specific metric, **Cost**, which is the total cost for running the complete workload, expressed in millions of consumed CPU-hours.

- For the other two scenarios, which do not have traditional metrics, we devise a performance metric that aggregates two components, the relative performance and the relative number of users. We design our metric so that the lower values for the relative performance are better. We define the **Aggregate Performance Penalty** as $APR(t) = \frac{P(t)}{P_{ref}} \times \frac{U(t)}{U_{max}}$, where $P(t)$ is the performance at time t, P_{ref} is the reference performance, $U(t)$ is the number of users at time t, and U_{max} is the maximum number of users over the course of the trace; $P(t)$ is a random value sampled from the distribution corresponding to the current month at time t. The relative number of users component is introduced because application providers are interested in bad performance only to the extent it affects their users; when there are few users of the application, this component ensures that the $APR(t)$ metric remains low for small performance degradation. Thus, the APR metric does not represent well applications for which good and stable performance is important at all times. However, for such applications the impact of variability can be computed straightforwardly from the monthly statistics of the cloud service; this is akin to excluding the user component from the APR metric.

6.5.2 Grid and PPE Job Execution

Scenario In this scenario we analyze the execution of compute-intensive jobs typical for grids and PPEs on cloud resources.

Trace ID, Source (Trace ID in Archive)	Trace			System		
	Number of			Size		Load [%]
	Mo.	Jobs	Users	Sites	CPUs	
Grid Workloads Archive [104], 3 traces						
1. RAL (6)	12	0.2M	208	1	0.8K	85+
2. Grid3 (8)	18	1.3M	19	29	3.5K	-
3. SharcNet (10)	13	1.1M	412	10	6.8K	-
Parallel Workloads Archive [4], 2 traces						
4. CTC SP2 (6)	11	0.1M	679	1	430	66
5. SDSC SP2 (9)	24	0.1M	437	1	128	83

Table 6.6: Job Execution (GAE Run Service): The characteristics of the input workload traces.

Trace ID	Cloud with					
	Stable Performance			Variable Performance		
	ART [s]	ABSD (10s)	Cost	ART [s]	ABSD (10s)	Cost
RAL	18,837	1.89	6.39	18,877	1.90	6.40
Grid3	7,279	4.02	3.60	7,408	4.02	3,64
SharcNet	31,572	2.04	11.29	32,029	2.06	11.42
CTC SP2	11,355	1.45	0.29	11,390	1,47	0.30
SDSC SP2	7,473	1.75	0.15	7,537	1.75	0.15

Table 6.7: Job Execution (GAE Run Service): Head-to-head performance of workload execution in clouds delivering steady and variable performance. The ”Cost” column presents the total cost of the workload execution, expressed in millions of CPU-hours.

Input Traces We use five long-term traces from real grids and PPEs as workloads; Table 6.6 summarizes their characteristics, with the ID of each trace indicating the system from which the trace was taken; see [104, 4] for more details about each trace.

Variability We assume that the execution performance for the cloud with steady performance is equivalent to the performance of the grid from which the trace was obtained. We also assume that the GAE Run service can run the input workload, and exhibits the monthly variability evaluated in Section 6.4.2. Thus, we assume that the cloud with variable performance introduces for each job a random slowdown factor derived from the real performance distribution of the service for the month in which the job was submitted.

Results Table 6.7 summarizes the results for the job execution scenario. The performance metrics ART, ABSD, and Cost differ by less than 2% between the cloud with stable performance and the cloud with variable performance. Thus, the main finding is that the impact of service variability is low for this scenario.

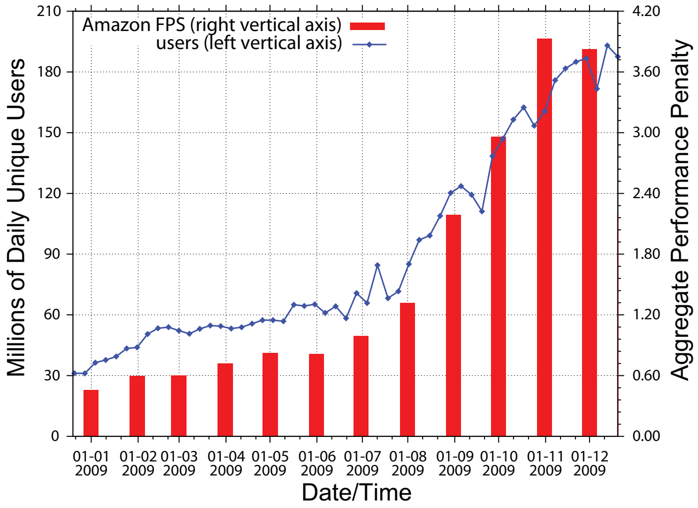


Figure 6.10: Selling Virtual Goods in Social Networks (Amazon FPS): Aggregate Performance Penalty when using Amazon FPS as the micro-payment backend. (Data source for the number of FaceBook users: `http://www.developeranalytics.com/`)

6.5.3 Selling Virtual Goods in Social Networks

Scenario In this scenario we look at selling virtual goods by a company operating a social network such as FaceBook, or by a third party associated with such a company. For example, FaceBook facilitates selling virtual goods through its own API, which in turn could make use of Amazon's FPS service for micro-payments.

Input Traces We assume that the number of payment operations depends linearly with the number of daily unique users, and use as input traces the number of daily unique users present on FaceBook (Figure 6.10).

Variability We assume that the cloud with variable performance exhibits the monthly variability of Amazon FPS, as evaluated in Section 6.3.6.

Results The main result is that our APR metric can be used to trigger and motivate the decision of switching cloud providers. Figure 6.10 shows the APR when using Amazon's FPS as the micro-payment backend of the virtual goods vendor. The significant performance decrease of the FPS service during the last four months of the year, combined with the significant increase in the number of daily users, is well captured by the APR metric–it leads to APR values well above 1.0, to a maximum of 3.9 in November 2009. If the clients respond to high payment latency similarly to other consumers of Internet newmedia [43, 53], that is, they become unsatisfied and quit, our APR metric is a clear indicator for the virtual goods vendor that the cloud provider should be changed.

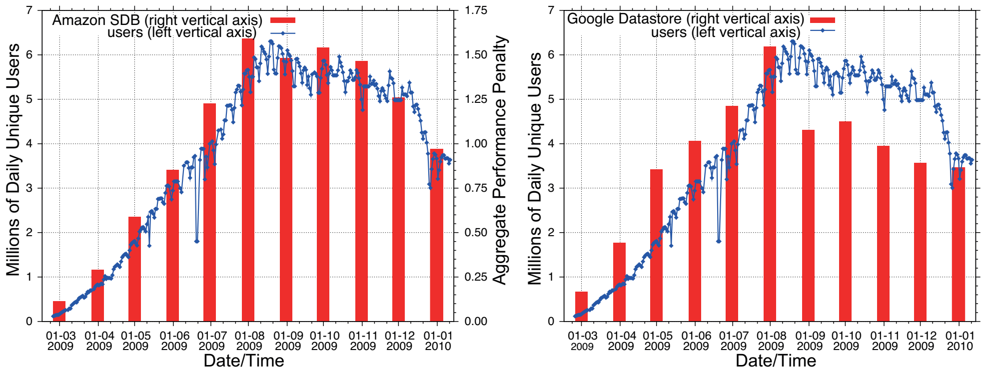


Figure 6.11: Game Status Maintenance for Social Games (Amazon SDB and Google App Engine Datastore): Aggregate Performance Penalty when using Amazon SDB as the database backend (left) and when using Google App Engine Datastore as the database backend (right). (Data source for the number of Farm Town users: `http://www.developeranalytics.com/`)

6.5.4 Game Status Maintenance for Social Games

Scenario In this scenario we investigate the maintenance of game status for a large-scale social game such as Farm Town or Mafia Wars which currently have millions of unique users daily. In comparison with traditional massively multiplayer online games such as World of Warcraft and Runescape, which also gather millions of unique players daily, social games have very little player-to-player interaction (except for messaging, performed externally to the game, for example through FaceBook channels). Hence, maintaining the game status for social gaming is based on simpler database operations, without the burden of cross-updating information for concurrent players, as we have observed for Runescape in our previous work [151]. Thus, this scenario allows us to compare a pair of cloud database services, Amazon's SDB and Google's Datastore.

Input Traces Similarly to the previous scenario, we assume that the number of operations, database accesses in this scenario, depends linearly on the number of daily unique users. We use as input trace the number of daily unique users for the Farm Town social game (Figure 6.11).

Variability We assume, in turn, that the cloud with variable performance exhibits the monthly variability of Amazon SDB (Section 6.3.4) and of Google Datastore (Section 6.4.3). The input traces span the period March 2009 to January 2010; thus, we do not have a direct match between the variability data, which corresponds to only to months in 2009, and the month January 2010 in the input traces. Since the Datastore operations exhibit yearly patterns (Section 6.4.6), we use in simulation the variability data of January 2009 as the variability data for January 2010.

Results The main finding is that there is a big discrepancy between the two cloud services, which would allow the application operator to select the most suitable provider. Figures 6.11 depicts the APR for the application using the Amazon SDB Update opeation (top) and for the application using the Google Datastore Read operation (bottom). During September 2009–January 2010, the bars depicting the APR of Datastore are well below the curve representing the number of users. This corresponds to the performance improvements (lower median) of the Datastore Read performance indicator in the last part of 2009 (see also Figure 6.8 (b)). In contrast, the APR values for SDB Update go above the users curve. These visual clues indicate that, for this application, Datastore is superior to SDB over a long period of time. An inspection of the APR values confirms the visual clues: the APR for the last five depicted months is around 1.00 (no performance penalty) for Datastore and around 1.4 (40% more) for SDB. The application operator has solid grounds for using the Datastore services for the application studied in this scenario.

6.6 Related work

Much effort has been put recently in assessing the performance of virtualized resources, in cloud computing environments [63, 163, 208, 157, 161, 72, 226, 64] and in general [23, 52, 145, 141]. In contrast to this body of previous work, ours is different in scope: we do not focus on the (average) performance values, but on their variability and evolution over time. In particular, our work is the first to characterize the long-term performance variability of production cloud services.

Close to our work is the seminal study of Amazon S3 [163], which also includes a 40 days evaluation of the service availability. Our work complements this study by analyzing the performance of eight other AWS and GAE services over a year; we also focus on different applications. Several small-scale performance studies of Amazon EC2 have been recently conducted: the study of Amazon EC2 performance using the NPB benchmark suite [208], the early comparative study of Eucalyptus and EC2 performance [157], the study of performance and cost of executing a scientific workflow in clouds [63], the study of file transfer performance between Amazon EC2 and S3, etc. Our results complement these studies and give more insight into the (variability of) performance of EC2 and other cloud services.

Recent studies using general purpose benchmarks have shown that virtualization overhead can be below 5% for computation [23] and below 15% for networking [23, 145]. Similarly, the performance loss due to virtualization for parallel I/O and web server I/O has been shown to be below 30% [232] and 10% [47], respectively. Our previous work [161, 105] has shown that virtualized resources in public clouds can have a much lower performance than the theoretical peak, especially for computation and network-intensive applications. In contrast to these studies, we investigate in this chapter the performance variability, and find several examples of performance indicators whose monthly median's variation is above 50% over the course of the studied year. Thus, our current study complements well the findings of our previous work, that is, the performance results obtained for small virtualized platforms are optimistic estimations of the performance observed in clouds.

6.7 Summary

Production cloud services may incur high performance variability, due to the combined and non-trivial effects of system size, workload variability, virtualization overheads, and resource time-sharing. In this chapter we have set to identify the presence and extent of this variability, and to understand its impact on large-scale cloud applications. Our study is based on the year-long traces that we have collected from CloudStatus and which comprise performance data for Amazon Web Services and Google App Engine services. The two main achievements of our study are described in the following.

First, we have analyzed the time-dependent characteristics exhibited by the traces, and found that the performance of the investigated services exhibits on the one hand yearly and daily patterns, and on the other hand periods of stable performance. We have also found that many services exhibit high variation in the monthly median values, which indicates large performance changes over time.

Second, we have found that the impact of the performance variability varies greatly across application types. For example, we found that the service of running applications on GAE, which exhibits high performance variability and a three-months period of low variability and improved performance, has a negligible impact for running grid and parallel production workloads. In contrast, we have found that and explained the reasons for which the GAE database service, having exhibited a similar period of better performance as the GAE running service, outperforms the AWS database service for a social gaming application.

Chapter 7

Space-correlated failures in large-scale distributed systems[*]

Millions of people rely daily on the availability of distributed systems such as peer-to-peer file-sharing networks, grids, and the Internet. Since the scale and complexity of contemporary distributed systems make the occurrence of failures the rule rather than the exception, many fault tolerant resource management techniques have been designed recently [92, 30, 176]. The deployment of these techniques and the design of new ones depend on understanding the characteristics of failures in real systems. While many failure models have been proposed for various computer systems [198, 176, 183, 99], few of these models consider the occurrence of failure bursts. In this chapter we present a new model that focuses on failure bursts, and validate it with real failure traces collected from a diverse set of distributed systems.

The foundational work on the failures of computer systems [41, 114, 198, 91] has already revealed that computer system failures occur often in bursts, that is, the occurrence of a failure of a system component can trigger within a short period a sequence of failures in other components of the system. It turned out that the fraction of bursty system failures is high in distributed systems; for example, in the VAXcluster 58% of all errors and occurred in bursts and involved multiple machines [198], and in both the VAXcluster and in Grid'5000 about 30% of all failures involve multiple machines [198, 103].

A bursty arrival breaks an important assumption made by numerous fault tolerant algorithms [92, 234, 147], that of independent and identical distribution of failures among the components of the system. However, few studies [198, 34, 103] investigate the bursty arrival of failures for distributed systems. Even for these studies, the findings are based on data corresponding to a single system–until the recent creation of online repositories such as the Failure Trace Archive [123] and the Computer Failure Data Repository [183],

---

[*] This chapter is based on previous work published in the *International Euro-Par Conference on Parallel Processing* (EuroPar'10) [83].

failure data for distributed systems were largely inaccessible to the researchers in this area.

The occurrence of failure bursts often makes the availability behavior of different system components to be correlated; thus, they are often referred to as component or *space-correlated failures*. The importance of space-correlated failures has been repeatedly noted: the availability of a distributed system may be overestimated by an order of magnitude when as few as 10% of the failures are correlated [198], and a halving of the work loss may be achieved when taking into account space-correlated failures [234].

In this chapter we address both scarcity problems, of the lack of traces, and of the lack of a model for space-correlated failures. With this study we make publicly and freely available through the Failure Trace Archive six new traces in standard format. We further propose a novel model for space-correlated failures based on moving windows. Then, we propose a fully automated method for identifying space-correlated failures. We validate our model using real failure traces taken from fifteen diverse distributed systems, and present for them the extracted model parameters.

The rest of the chapter is organized as follows. Section 7.1 introduces the terminology and the failure traces used in this chapter. Section 7.2 presents our model for space-correlated failures. Section 7.3 shows that space-correlated failures are indeed present and significant in the failure traces of distributed systems, which is an important assumption of this chapter. Section 7.4 presents the results of fitting common distributions to the empirical distributions extracted from the failure traces. Section 7.5 reviews the related work on space-correlated failures in distributed systems, and finally, Section 7.6 summarizes the chapter.

7.1 Background

In this section we present the terminology and the datasets used in this chapter.

7.1.1 Terminology

We follow throughout this chapter the basic concepts and definitions associated with system dependability as summarized by Avizienis et al. [18]. The basic threats to reliability are failures, errors, and faults occurring in the system. A *failure* (*unavailability event*) is an event in which the system fails to operate according to its specifications. A failure is observed as a deviation from the correct state of the system. An *error* is part of the system state that may lead to a failure. An *availability event* is the end of the recovery of the system from failure. As in our previous work [123], we define an *unavailability interval* (*downtime*) as a continuous period of a service outage due to a failure. Conversely, we define an *availability interval* as a contiguous period of service availability.

System	Type	# of Nodes	Period	Year	# of Events
GRID'5000	Grid	1,288	1.5 years	2005-2006	588,463
WEBSITES	Web servers	129	8 months	2001-2002	95,557
LDNS	DNS servers	62,201	2 weeks	2004	384,991
LRI	Desktop Grid	237	10 days	2005	1,792
DEUG	Desktop Grid	573	9 days	2005	33,060
SDSC	Desktop Grid	207	12 days	2003	6,882
UCB	Desktop Grid	80	11 days	1994	21,505
LANL	SMP, HPC Clusters	4,750	9 years	1996-2005	43,325
MICROSOFT	Desktop	51,663	35 days	1999	1,019,765
PLANETLAB	P2P	200-400	1.5 year	2004-2005	49,164
OVERNET	P2P	3,000	2 weeks	2003	68,892
NOTRE-DAME [1]	Desktop Grid	700	6 months	2007	300,241
NOTRE-DAME [2]	Desktop Grid	700	6 months	2007	268,202
SKYPE	P2P	4,000	1 month	2005	56,353
SETI	Desktop Grid	226,208	1.5 years	2007-2009	202,546,160

[1] The host availability version of the NOTRE-DAME trace.
[2] The CPU availability version of the NOTRE-DAME trace.

Table 7.1: Summary of fifteen datasets in the Failure Trace Archive.

7.1.2 The Datasets

The datasets used in this chapter are part of the Failure Trace Archive (FTA) [123]. The FTA is an online public repository of availability traces taken from diverse parallel and distributed systems.

The FTA makes available online failure traces in a common, unified format. The format records the occurrence time and duration of resource failures as an alternating time series of availability and unavailability intervals. Each availability or unavailability event in a trace records the start and the end of the event, and the resource that was affected by the event. Depending on the trace, the resource affected by the event can be either a node of a distributed system such as a node in a grid, or a component of a node in a system such as CPU or memory.

Prior to the work leading to this study, the FTA made available in its standard format nine failure traces; as a result of our work, the FTA now makes available fifteen failure traces. Table 7.1 summarizes the characteristics of these fifteen traces, which we use throughout this chapter. The traces originate from systems of different types (multi-cluster grids, desktop grids, peer-to-peer systems, DNS and Web servers) and sizes (from hundreds to tens of thousands of resources), which makes these traces ideal for a study among different systems. Furthermore, the traces cover statistically relevant periods of time, and many of the traces cover several months of system operation. A more detailed description of each trace is available on the FTA web site (`http://fta.inria.fr`).

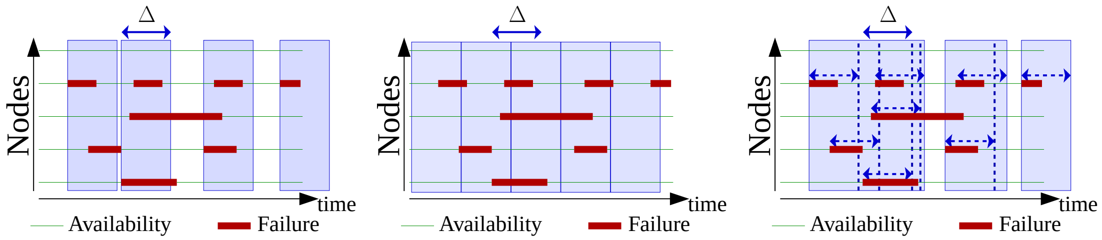


Figure 7.1: Generative processes for space-correlated failures: (left) moving windows; (middle) time partitioning; (right) extending windows.

7.2 Model Overview

In this section we propose a novel model for failures occurring in distributed systems. We first introduce our notion of space-correlated failures, and then build a model around it.

7.2.1 Space-Correlated Failures

We call *space-correlated failures* a group of failures that occur within a short time interval; the seminal work of Siewiorek [41, 131], Iyer [114, 198], and Gray [90, 91] has shown that for tightly coupled systems space-correlated failures are likely to occur. Our investigation of space-correlated failures is hampered by the lack of information present in failure traces—none of the computer system failure traces we know records failures with sufficient detail to reconstruct groups of failures. We adopt instead an approach that groups failures based on their start and finish timestamps. We identify three such approaches, moving windows, time partitioning, and extending windows, which we describe in turn.

Let $TS(\cdot)$ be the function that returns the time stamp of an event, either failure or repair. Let O be the sequence of failure events ordered according to increasing event time stamp, that is, $O = [E_i | TS(E_{i-1}) \leq TS(E_i), \forall i \geq 1]$.

Moving Windows We consider the following iterative process that, starting from O, generates the space-correlated failures with time parameter Δ. At each step in the process we select as the *group generator* F the first event from O unselected yet, and generate the space-correlated failure by further selecting from O all events E occurring within Δ time units from $TS(F)$, that is, $TS(E) \leq TS(F) + \Delta$. The process we employ ends when all the events in O have been selected. The maximum number of generated space-correlated failures is $|O|$, the number of events in O. The process uses a time window of size Δ, where the window ”moves” to the next unselected event in O at each step. Thus, we call this process the generation of space-correlated failures through *moving windows*. Figure 7.1 (left) depicts the use of the moving windows for various values of Δ.

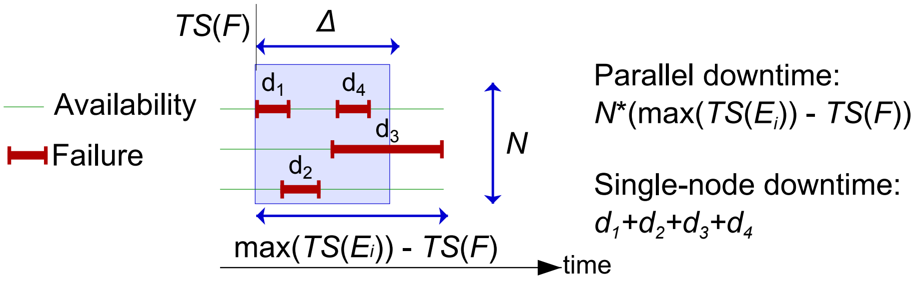


Figure 7.2: Parallel and single-node job downtime for a sample space-correlated failure.

Time Partitioning This approach partitions time in windows of fixed size Δ, starting either from a hypothetical time 0 or from the first event in O. We call this process generation of space-correlated failures through *time partitioning*.

Extending Windows A group of failures in this approach is a maximal subsequence of events such that each two consecutive events are at most a time Δ apart, i.e., for each consecutive events E and F in O, $TS(F) \leq TS(E) + \Delta$. Thus, Δ is the size of the window that extends the horizon for each new event added to the group; thus, we call this second process generation of space-correlated failures through *extending windows*. We have already used this process to model the failures occurring in Grid'5000 [103].

The three generation processes, moving windows, time partitioning, and extending windows, can generate very different space-correlated failures from the same input set of events O (see Figure 7.1). The following two considerations motivate our selection of a single generation process from these three. First, time partitioning may introduce artificial time boundaries between failure events belonging to consecutive space-correlated failures, because each space-correlated failure starts at a multiple of Δ. Thus, the groups identified through time partitioning do not relate well to groups naturally occurring in the system, and may confuse the fault-tolerant mechanisms and algorithms based on them; the moving and extending windows do not suffer from this problem. Second, the extending windows process may generate infinitely-long space-correlated failures: as the extending window is considered between consecutive failures, a failure can occur long after its group generator (its first occurring failure). Thus, the groups generated through extending windows may reduce the efficiency of fault tolerance mechanisms that react to instantaneous bursts of failures. Thus, we select and use in the remainder of this chapter the generative processes for space-correlated failures through moving windows.

7.2.2 Model Components

We now build our model around the notion of space-correlated failures (groups) introduced in the previous section. The model comprises three components: the group inter-arrival time, the group size, and the group downtime. We describe each of these three components in turn.

Inter-Arrival Time This component characterizes the process governing the arrival of new space-correlated failures (including groups of size 1).

Size This component characterizes the number of failures present in each space-correlated failure.

Downtime This component characterizes the downtime caused by each space-correlated failure. When failures are considered independently instead of in groups, the downtime is simply the duration of the unavailability corresponding to each failure event. A group of failure may, however, affect users in ways that depend on the user application. We consider in this chapter two types of user applications: parallel jobs and single-node jobs. We define the *parallel job downtime* (D_{Max}) of a failure group as the product of the number of individual nodes affected by the failure events within the group, and the time elapsed between the earliest failure event and the latest availability event corresponding to a failure within the group. We further define the *single-node job downtime* (D_{Σ}) as the sum of the downtimes of each individual failure within the failure group. Figure 7.2 depicts these two downtime notions. The parallel job downtime gives an upper bound to the downtime caused by space-correlated failures for parallel jobs that would run on any of the nodes affected by failures. Similarly, the single-node job downtime characterizes the impact of a failure group on workloads dominated by single-node jobs, which is the case for many grid workloads [99].

7.2.3 Method for Modeling

Our method for modeling is based on analyzing in two steps failure traces taken from real distributed systems; we describe each step, in turn, in the following.

The first step is to analyze for each trace the presence of space-correlated failures comprising two or more failure events, for values of Δ between 1 second and 10 minutes. Tolerating such groups of failures is important for interactive and deadline-oriented system users.

The second step follows the traditional modeling steps for failures in computer systems [114, 183]. We first characterize the properties of the empirical distributions using basic statistics such as the mean, the standard deviation, the min and the max, etc. This allows us to get a first glimpse of the type of probability distribution that could characterize the real data. We then try to find a good fit, that is, a well-known probability distribution and the parameters that lead to the best fit between that distribution and the empirical data. When selecting the probability distributions, we look at the degrees of freedom (number of parameters) of that distribution; while a distribution with more degrees of freedom may provide a better fit for the data, such a distribution can make the understanding of the model more difficult, can increase the difficulty of mathematical analysis based on the model, and may also lead to overfitting to the empirical datasets. Thus, we select five probability distributions to fit to the empirical data: exponential, Weibull, Pareto, lognormal, and gamma. The fitting of the probability distributions to the empirical datasets uses the Maximum Likelihood Estimation (MLE) method [10], which delivers good accuracy for the large data samples specific to failure traces.

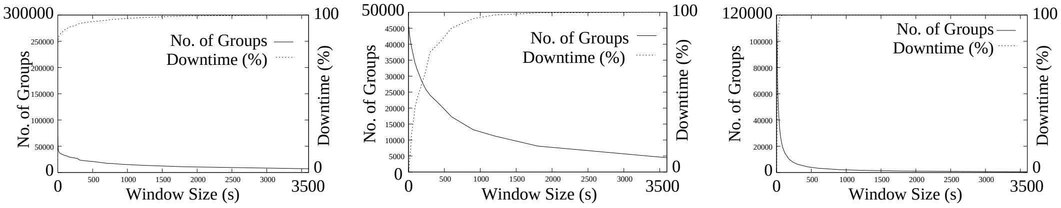
After finding the best fits for each candidate distribution, goodness-of-fit tests are used to assess the quality of the fitting for each distribution, and to establish the best fit. We use for this purpose both the Kolmogorov-Smirnov (KS) and the Anderson-Darling (AD) tests, which essentially assess how close the cumulative distribution function (CDF) of the probability distribution is to the CDF of the empirical data. For each candidate distribution with the parameters found during the fitting process, we formulate the hypothesis that the empirical data are derived from it (the null-hypothesis of the goodness-of-fit test). Neither of the KS and AD tests can confirm the null-hypothesis, but both are useful in understanding the goodness-of-fit. For example, the KS-test provides a test statistic, D, which characterizes the maximal distance between the CDF of the empirical distribution of the input data and that of the fitted distribution.

7.3 Failure Group Window Size

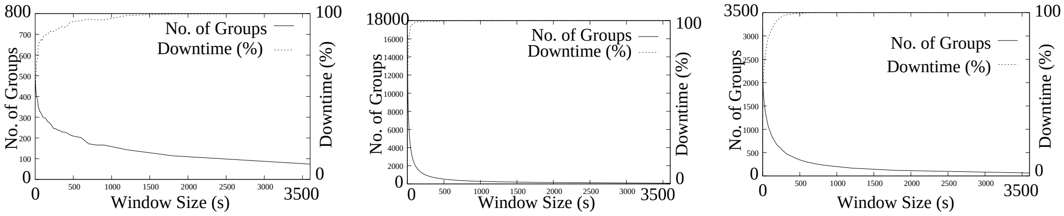
An important assumption in this chapter is that space-correlated failures are present and significant in the failure traces of distributed systems. In this section we show that this is indeed the case. Section 7.2.1 the characteristics of the space-correlated failures are dependent on the window size Δ; we investigate this dependency in this section.

The importance of a failure model derives from the fraction of downtime caused by the failures whose characteristics it explains, from the total downtime of the system. For the model we have introduced in Section 7.2 we are interested in space-correlated failures of at least two failures. As explained in Section 7.2.1, the characteristics of the space-correlated failures depend on the window size Δ. Large values for Δ lead to more groups of at least two failures, but reduce the usefulness of the model for predictive fault tolerance. Conversely, small values for Δ lead to few groups of at least two failures, and effectively convert our model into the model for individual failures we have investigated elsewhere [123].

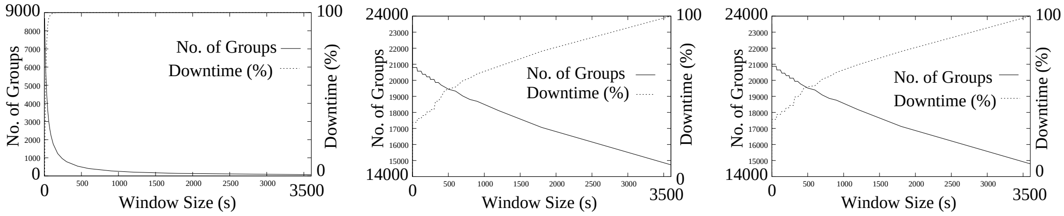
We assess the effect of Δ on the number of and downtime caused by space-correlated failures by varying Δ from one second to one hour; the most interesting values for Δ are below a few minutes, useful for proactive fault tolerance techniques. Figure 7.3 shows the results for each of the fifteen datasets (see Section 7.1.2). We distinguish in the figure the



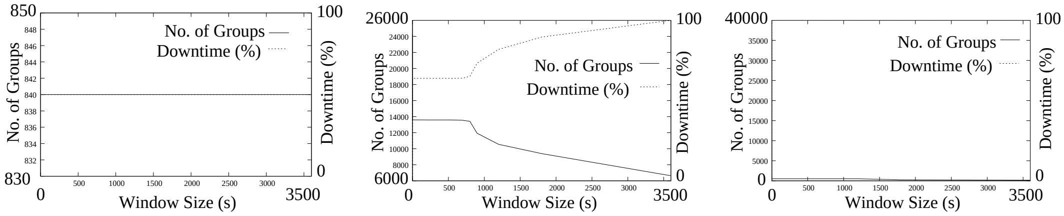
(a) Grid’5000 (b) Websites (c) LDNS



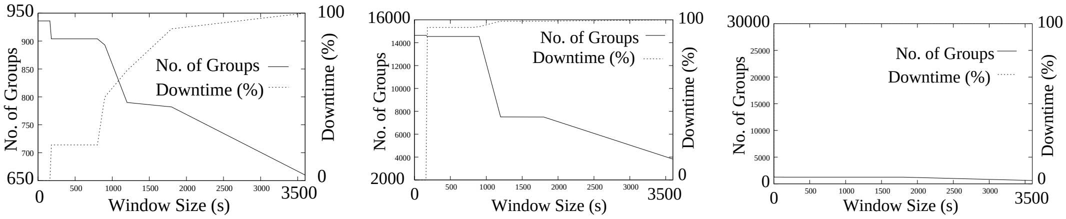
(d) LRI (e) Deug (f) SDSC



(g) UCB (h) LANL (i) LANL (intersection)



(j) Microsoft (k) PlanetLab (l) Overnet



(m) Notre-Dame (n) Notre-Dame (CPU) (o) Skype

Figure 7.3: Number of groups and cumulated downtime for groups of at least two failures.

first seven systems, Grid’5000, Websites, LDNS, LRI, Deug, SDSC, and UCB, for which a significant fraction of the total system downtime is caused by space-correlated

Platform	GRID'5000	WEBSITES	LDNS	LRI	DEUG	SDSC	UCB
Window Size [s]	250	100	150	100	150	120	80

Table 7.2: Selected failure group window size for each system.

failures of size at least 2, when Δ is equal to a few minutes. For similar values of Δ, the space-correlated failures do not cause most of the system downtime for the remaining systems. We do not include in the distinguished systems MICROSOFT, OVERNET, NOTRE-DAME, and SKYPE, since the dependence of the depicted curves on Δ looks more like an artifact of the data, due to the regular probing of nodes.

The seven distinguished traces have similar dependency on Δ: as Δ increases slowly, the number of groups quickly decreases and the cumulative downtime quickly increases. Then, both slowly stabilize; this point, which occurs for values of Δ of a few minutes, is a good trade-off between small window size and large capture of failures into groups. We extract for each of the seven selected traces the best observed trade-off, and round it to the next multiple of 10 seconds; Table 7.2 summarizes our findings.

7.4 Analysis Results

In the previous section we have selected seven systems for which space-correlated failures are responsible for most of the system downtime. In this section, we present the results of fitting common distributions to the empirical distributions extracted from the failure traces of these seven traces selected. The space-correlated failures are generated using the moving windows method introduced in Section 7.2, and the values of Δ selected in Section 7.3.

The Failure Trace Archive already offers a toolbox (see [123] for details) for fitting common distributions to empirical data. We have adapted the tools already present in this toolbox for our model by extending the set of common distributions with the Pareto distribution, by adding a data preprocessing step that extracts groups of failures for a specific value of Δ, and by improving the output of the tools with automated graphing and tabulation support. These additions are now publicly available as part of the FTA toolbox code repository.

7.4.1 Detailed Results

We have fitted to the empirical distributions five common distributions, exponential, Weibull, Pareto, lognormal, and gamma. We now present the results obtained for each model component, in turn.

Failure Group Inter-Arrival Time To understand the failure group inter-arrival time, we consider for each failure group identified in the trace (including groups of size 1), the

	Grid’5000	Websites	LDNS	LRI	Deug	SDSC	UCB
Exp	0.53	0.15	0.18	0.86	1.22	0.47	0.51
Weibull	0.44 0.79	0.16 1.21	0.12 0.74	0.46 0.63	0.23 0.47	0.13 0.57	0.07 0.48
Pareto	0.42 0.29	0.01 0.15	0.36 0.08	0.62 0.25	0.84 0.09	0.40 0.07	0.51 0.03
LogN	**-1.39 1.03**	**-2.17 0.76**	**-2.57 0.81**	**-1.46 1.28**	**-2.28 1.35**	**-2.63 0.86**	**-3.41 0.98**
Gamma	0.79 0.67	1.83 0.08	0.71 0.25	0.48 1.79	0.28 4.33	0.36 1.31	0.26 2.00

Table 7.3: **Failure Group Inter-Arrival Time:** Best found parameters when fitting distributions to empirical data. Values in bold denote the best fit.

	Grid’5000	Websites	LDNS	LRI	Deug	SDSC	UCB
Exp	17.09	2.55	13.44	5.74	10.96	5.19	4.47
Weibull	12.82 0.71	2.87 1.60	15.12 2.29	5.76 1.01	12.12 1.39	4.94 0.93	5.05 2.52
Pareto	0.68 6.75	-0.06 2.68	-0.18 15.09	0.22 4.43	-0.03 11.26	0.22 3.70	-0.41 5.76
LogN	**1.88 1.25**	0.84 0.35	**2.52 0.41**	**1.32 0.77**	**2.15 0.70**	**1.19 0.70**	1.41 0.42
Gamma	0.64 26.78	**5.33 0.48**	6.23 2.16	1.30 4.40	2.22 4.94	1.23 4.24	**6.03 0.74**

Table 7.4: **Failure Group Size:** Best found parameters when fitting distributions to empirical data. Values in bold denote the best fit.

group generator (see Section 7.2.1). We then generate the empirical distribution from the time series corresponding to the inter-arrival time between consecutive group generators. Table 7.3 summarizes for each platform the parameters of the best fit obtained for each of the five common distribution we use in this chapter. These results reveal that the failure group inter-arrival time is not well characterized by a heavy-tail distribution as the p-values for the Pareto are low. Moreover, we identify two categories of platforms. The first category, represented by Grid’5000, Websites, and LRI, is well-fitted by Log-Normal distributions. The second category, represented by LDNS, Deug, SDSC, and UCB, is not well-fitted by any of the common distributions we tried; for these, the best-fits are either the lognormal or the gamma distributions.

Failure Group Size To understand the failure group size, we generate the empirical distribution of the sizes of each group identified in the trace (including groups of size 1). Table 7.4 summarizes for each platform the parameters of the best fit obtained for each of the five common distribution we use in this chapter. Similarly to our findings for the failure group inter-arrival time, the results for the failure group size reveal heavy-tail distributions are not good fits. We find that the lognormal and gamma distributions are good fits for the empirical distributions.

Failure Group Duration The two last components of our model are the parallel- and single-node downtime of the space-correlated failures. To understand these two components, we generate for each the empirical data distribution using the durations of each group identified in the trace (including groups of size 1). The results of the fitting of the parallel downtime component are presented in Table 7.5, and the results of the fitting of the single-node downtime component are given in Table 7.6. Similarly to our previous

	GRID'5000	WEBSITES	LDNS	LRI	DEUG	SDSC	UCB
EXP	3.33e6	21225.18	2.48e6	2.46e5	1.18e5	67183.25	4071.25
WEIBULL	75972.13 0.28	10658.82 0.63	2.430e6 0.96	1.051e5 0.48	61989.86 0.54	35581.34 0.63	4131.60 1.03
PARETO	3.10 2686.08	0.73 5493.50	0.16 2.071e6	**1.71 24187.13**	1.53 15901.44	0.54 20627.60	0.09 3711.35
LOGN	**9.51 3.21**	**8.57 1.36**	**14.16 1.15**	10.41 2.45	**10.03 2.02**	**9.80 1.30**	**7.82 1.03**
GAMMA	0.14 2.362e6	0.46 46006.96	1.01 2.452e6	0.34 7.317e5	0.40 2.950e5	0.49 1.384e5	1.16 3509.88

Table 7.5: **Failure Group Duration, $D_{\max}$:** Best found parameters when fitting distributions to empirical data. Values in bold denote the best fit.

	GRID'5000	WEBSITES	LDNS	LRI	DEUG	SDSC	UCB
EXP	4.40e5	10363.55	4.17e5	1.63e5	29979.27	30139.69	1500.92
WEIBULL	30951.59 0.33	6605.36 0.70	4.576e5 1.37	**80091.30 0.50**	13239.84 0.57	19008.04 0.69	1646.49 1.35
PARETO	2.54 2215.71	0.47 4258.00	-0.11 4.576e5	1.61 20672.26	0.91 5832.36	0.41 12570.49	-0.10 1645.39
LOGN	**8.89 2.71**	**8.20 1.13**	12.64 0.84	10.16 2.40	**8.67 1.62**	**9.25 1.16**	**7.01 0.81**
GAMMA	0.18 2.418e6	0.59 17462.56	**1.82 2.292e5**	0.36 4.484e5	0.40 74867.95	0.59 51497.86	1.82 825.92

Table 7.6: **Failure Group Duration, D_{Σ}:** Best found parameters when fitting distributions to empirical data. Values in bold denote the best fit.

findings in this section, we find that heavy-tail distributions such as Pareto do not fit well the empirical distributions. In contrast, the lognormal distribution is by far the best fit, with only two systems (LDNS and LRI) being better represented by the other distributions (the Gamma and Weibull distributions, respectively).

7.4.2 Results Summary

For all the components of our model and for all platforms, the most well-suited distribution is presented in Table 7.7. The main result is that Log-Normal distributions provide good results for almost all parts of our model. Thus, we can model most of node-level failures in the whole platform by groups of failures, each group being characterized by its size, its parallel downtime and its single-node downtime.

7.5 Related work

From the large body of work already dedicated to modeling the availability of parallel and distributed computer systems–see [198, 176, 183, 99] and the references within–, relatively little attention has been given to space-correlated errors and failures [198, 34, 103], despite their reported importance [92, 176].

The main differences between this work and the previous work on space-correlated errors and failures is summarized in Table 7.8. Our study is the first to investigate the problem in the broad context of distributed systems through the use of a large number of traces. Besides a broader scope, our study is the first to use a generation process based on a moving window, and to propose a method for the selection of the moving window size.

	Group size	Group IAT	$D_{\max}$	D_{Σ}
Grid'5000	LogN (1.88,1.25)	LogN (-1.39,1.03)	LogN (9.51,3.21)	LogN (8.89,2.71)
Websites	Gamma (0.84,0.35)	LogN (-2.17,0.76)	LogN (8.57,1.36)	LogN (8.20,1.13)
LDNS	LogN (2.52,0.41)	LogN (-2.57,0.81)	LogN (14.16,1.15)	Gamma (1.82,2.292e5)
LRI	LogN (1.32,0.77)	LogN (-1.46,1.28)	Weibull (1.051e5,0.48)	Weibull (80091.30,0.50)
Deug	LogN (2.15,0.70)	LogN (-2.28,1.35)	LogN (10.03,2.02)	LogN (8.67,1.62)
SDSC	LogN (1.10,0.70)	LogN (-2.63,0.86)	LogN (9.80,1.30)	LogN (9.25,1.16)
UCB	Gamma (6.03,0.74)	LogN (-3.41,0.98)	LogN (7.82,1.03)	LogN (7.01,0.81)

Table 7.7: Best fitting distribution for all model components, for all systems.

Study	*System Type* System Name (Number of Systems/Total Size [nodes])	Data Source (Length)	Errors/ Failures	Gen. Process	Setup Type (Δ [min])
[198]	*SC* VAXcluster (1 sys./7)	Sys.logs (10 mo.)	Errors	time partitioning	manual (5 min.)
[34]	*NoW* Microsoft (1 sys./>50,000)	Msmts. (5 weeks)	Failures	instantaneous	manual (0 min.)
[103]	*Grid* Grid'5000 (15 cl./>2,500)	Sys.logs (1.5 years)	Failures	extending window	auto (0.5–60)
This study	*Various* Various (15 sys./>500,000)	Various (>6 mo. avg.)	Failures	moving window	auto (0.02–60)

Note: SC, NoW, Sys, Cl, Msmts, and Mo are acronyms for supercomputer, network of workstations, system, cluster, measurements, and months, respectively.

Table 7.8: Research on space-correlated availability in distributed systems.

7.6 Summary

In this chapter we have developed a model for space-correlated failures, that is, for failures that occur within a short time frame across distinct components of the system. For such groups of failures, our model considers three aspects, the group arrival process, the group size, and the downtime caused by the group of failures. We have found that the best models for these three aspects are mainly based on the lognormal distribution.

We have validated this model using failure traces taken from diverse distributed systems. Since the input data available in these traces, and, to our knowledge, in any failure traces available to scientists, do not contain information about the space correlation of failures, we have developed a method based on moving windows for generating space-correlated failure groups from empirical data. Moreover, we have designed an automated way to determine the window size, which is the unique parameter of our method, and we have demonstrated its use on the same traces.

We have found that for seven out of the fifteen traces investigated in this chapter, a majority of the system downtime is caused by space-correlated failures. Thus, these seven traces are better represented by our model than by traditional models, which assume that the failures of the individual components of the system are independent and identically distributed. Finally, with this work we have contributed six new failure traces in standard format to the Failure Trace Archive, which we hope can encourage other researchers to use the archive and also to contribute to it with failure traces.

Chapter 8

Time-correlated failures in large-scale distributed systems[*]

Large-scale distributed systems have reached an unprecedented scale and complexity in recent years. At this scale failures inevitably occur—networks fail, disks crash, packets get lost, bits get flipped, software misbehaves, or systems just crash due to misconfiguration and other human errors. Deadline-driven or mission-critical services are part of the typical workload for these infrastructures, which thus need to be available and reliable despite the presence of failures. Researchers and system designers have already built numerous fault-tolerance mechanisms that have been proven to work under various assumptions about the occurrence and duration of failures. However, most previous work focuses on failure models that assume the failures to be non-correlated, but this may not be realistic for the failures occurring in large-scale distributed systems. For example, such systems may exhibit peak failure periods, during which the failure rate increases, affecting in turn the performance of fault-tolerance solutions. Moreover, we have already shown in Chapter 7 that failures occur as burst, and we have presented a model for burst of failures in large-scale distributed systems. In this chapter, we investigate the time-varying behavior of failures using nineteen traces obtained from several large-scale distributed systems including grids, P2P systems, DNS servers, web servers, and desktop grids.

Recent studies report that in production systems, failure rates can be of over one thousand failures per year, and depending on the root cause of the corresponding problems, the mean time to repair can range from hours to days [183]. The increasing scale of the deployed distributed systems causes the failure rates to increase, which in turn can have a significant impact on the performance and cost, such as degraded response times [234] and increased Total Cost of Operation (TCO) due to increased administration costs and human resource needs [27]. This problem also motivates the need for further research in

---

[*] This chapter is based on previous work published in the *IEEE/ACM International Conference on Grid Computing* (GRID’10) [225].

failure characterization and modeling. Previous studies [199, 158, 155, 176, 220, 183] focused on characterizing failures in several different distributed systems. However, most of these studies assume that failures occur independently or disregard the time correlation of failures, despite the practical importance of these correlations [147, 206, 180]. First of all, understanding if failures are time correlated has important implications for proactive fault-tolerance solutions. Second, understanding the time-varying behavior of failures and peaks observed in failure patterns is required for evaluating design decisions. For example, redundant submissions may all fail during a failure peak period, regardless of the quality of the resubmission strategy. Third, understanding the temporal correlations and exploiting them for smart checkpointing and scheduling decisions provides new opportunities for enhancing conventional fault-tolerance mechanisms [234, 111]. For example, a simple scheduling policy could be to stop scheduling large parallel jobs during failure peaks. Finally, it is possible to devise adaptive fault-tolerance mechanisms that adjust the policies based on the information related to peaks. For example, an adaptive fault-tolerance mechanism can migrate the computation at the beginning of a predicted peak.

In this chapter, to understand the time-varying behavior of failures in large-scale distributed systems, we perform a detailed investigation using data sets from diverse large-scale distributed systems including more than $100K$ hosts and $1.2M$ failure events spanning over 15 years of system operation in total. With this chapter we make the following contributions. First, we make four new failure traces publicly available through the Failure Trace Archive. Secondly, we present a detailed evaluation of the time correlation of failure events observed in traces taken from nineteen (production) distributed systems. Finally, we propose a model for peaks observed in the failure rate process.

The remaining part of this chapter is organized as follows. Section 8.1 introduces the failure traces and the modeling methodology we use in this chapter. Section 8.2 presents our analysis of autocorrelations in the failure events observed in these failure traces. Then, Section 8.3 presents our model for the peaks observed in the failure rate process. Section 8.4 reviews the related work on time-correlated failures in large-scale distributed systems, and finally, Section 8.5 summarizes the chapter.

8.1 Method

8.1.1 Failure Datasets

In this chapter we use and contribute to the data sets in the Failure Trace Archive (FTA) [123]. A general overview of the FTA has already been presented in Section 7.1.2.

With the prior work, the FTA made fifteen failure traces available in its standard format; as a result of our work, the FTA now makes available nineteen failure traces. Ta-

System	Type	Nodes	Period	Year	Events
Grid'5000	Grid	1,288	1.5 years	2005-2006	588,463
COND-CAE[1]	Grid	686	35 days	2006	7,899
COND-CS[1]	Grid	725	35 days	2006	4,543
COND-GLOW[1]	Grid	715	33 days	2006	1,001
TeraGrid	Grid	1001	8 months	2006-2007	1,999
LRI	Desktop Grid	237	10 days	2005	1,792
Deug	Desktop Grid	573	9 days	2005	33,060
Notre-Dame [2]	Desktop Grid	700	6 months	2007	300,241
Notre-Dame [3]	Desktop Grid	700	6 months	2007	268,202
Microsoft	Desktop Grid	51,663	35 days	1999	1,019,765
UCB	Desktop Grid	80	11 days	1994	21,505
PlanetLab	P2P	200-400	1.5 year	2004-2005	49,164
Overnet	P2P	3,000	2 weeks	2003	68,892
Skype	P2P	4,000	1 month	2005	56,353
Websites	Web servers	129	8 months	2001-2002	95,557
LDNS	DNS servers	62,201	2 weeks	2004	384,991
SDSC	HPC Cluster	207	12 days	2003	6,882
LANL	HPC Cluster	4,750	9 years	1996-2005	43,325
PNNL	HPC Cluster	1,005	4 years	2003-2007	4,650

[1] COND-* data sets denote the Condor data sets.

[2] The host availability version of the Notre-Dame trace.

[3] The CPU availability version of the Notre-Dame trace.

Table 8.1: Summary of nineteen data sets in the Failure Trace Archive.

ble 8.1 summarizes the characteristics of these nineteen traces, which we use throughout this chapter. The traces originate from systems of different types (multi-cluster grids, desktop grids, peer-to-peer systems, DNS and web servers) and sizes (from hundreds to tens of thousands of resources), which makes these traces ideal for a study among different distributed systems. Furthermore, many of the traces cover several months of system operation.

8.1.2 Analysis

In our analysis, we use the autocorrelation function (ACF) to measure the degree of correlation of the failure time series data with itself at different time lags. The ACF takes on values between -1 (high negative correlation) and 1 (high positive correlation). In addition, the ACF reveals when the failures are random or periodic. For random data the correlation coefficients will be close to zero, and a periodic component in the ACF reveals that the failure data is periodic or at least it has a periodic component.

8.1.3 Modeling

In the modeling phase, we statistically model the peaks observed in the failure rate process, i.e., the number of failure events per time unit. Towards this end we use the Maximum Likelihood Estimation (MLE) method [10] for fitting the probability distributions to the empirical data as it delivers good accuracy for the large data samples specific to failure traces. After we determine the best fits for each candidate distribution for all data sets, we perform the goodness-of-fit tests to assess the quality of the fitting for each distribution, and to establish the best fit. As the goodness-of-fit tests, we use both the Kolmogorov-Smirnov (KS) and the Anderson-Darling (AD) tests, which essentially assess how close the cumulative distribution function (CDF) of the probability distribution is to the CDF of the empirical data. For each candidate distribution with the parameters found during the fitting process, we formulate the hypothesis that the empirical data are derived from it (the null-hypothesis of the goodness-of-fit test). Neither the KS or the AD tests can confirm the null-hypothesis, but both are useful in understanding the goodness-of-fit. For example, the KS-test provides a test statistic, D, which characterizes the maximal distance between the CDF of the empirical distribution of the input data and that of the fitted distribution; distributions with a lower D value across different failure traces are better. Similarly, the tests return p-values which are used to either reject the null-hypothesis if the p-value is smaller than or equal to the significance level, or confirm that the observation is consistent with the null-hypothesis if the p-value is greater than the significance level. Consistent with the standard method for computing p-values [155, 123], we average 1,000 p-values, each of which is computed by selecting 30 samples randomly from the data set, to calculate the final p-value for the goodness-of-fit tests.

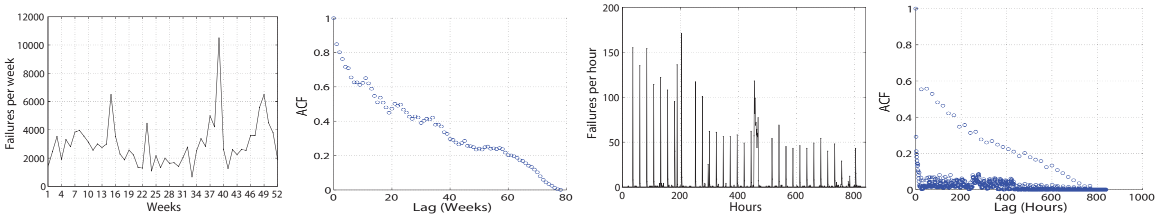
8.2 Analysis of Autocorrelation

In this section we present the autocorrelations in failures using traces obtained from grids, desktop grids, P2P systems, web servers, DNS servers and HPC clusters, respectively. We consider the failure rate process, that is the number of failure events per time unit.

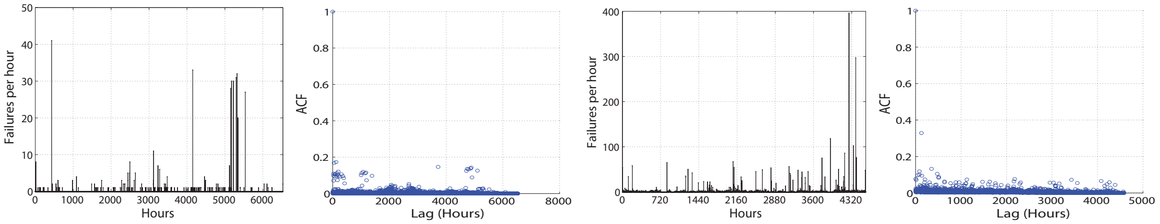
8.2.1 Failure Autocorrelations in the Traces

Our aim is to investigate whether the occurrence of failures is repetitive in our data sets. Towards this end, we compute the autocorrelation of the failure rate for different time lags including hours, weeks, and months. Figure 8.1 shows for several platforms the failure rate at different time granularities, and the corresponding autocorrelation functions.

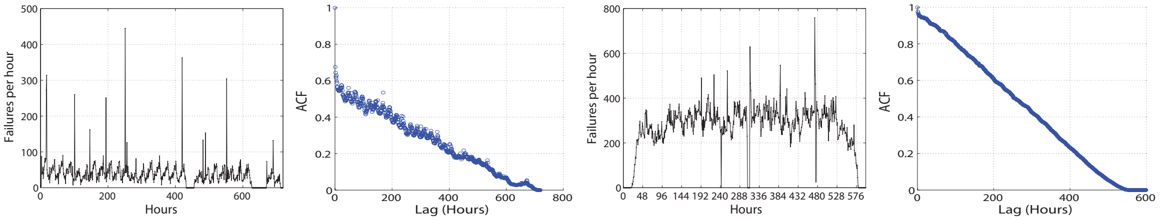
Many of the systems investigated in this chapter exhibit strong autocorrelation for hourly and weekly lags. Figures 8.1(a), 8.1(b), 8.1(e), 8.1(f), and 8.1(g) show the failure rates and autocorrelation functions for the GRID'5000, CONDOR (CAE), SKYPE, LDNS and LANL systems, respectively. The GRID'5000 data set is a one and a half year long trace collected from an academic research grid. Since this system is mostly used for experimental purposes, and is large-scale (∼3K processors) the failure rate is



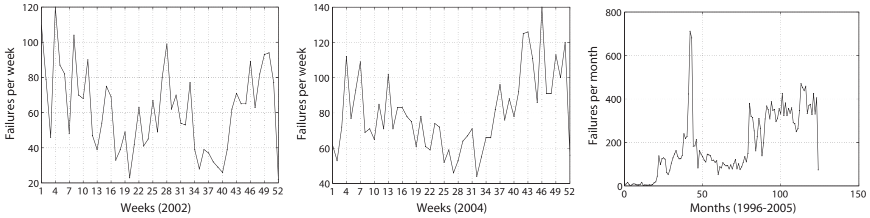
(a) Grid’5000 (b) Condor (CAE)



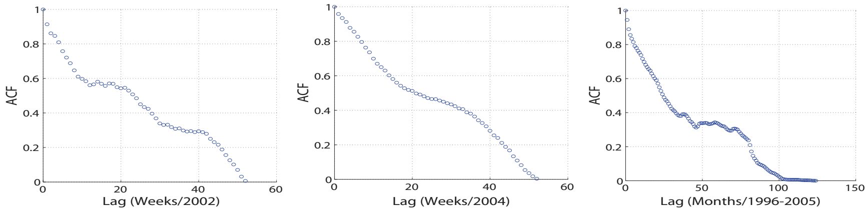
(c) TeraGrid (d) Notre-Dame



(e) Skype (f) LDNS



(g) LANL (For 2002, 2004 and the whole trace)



(h) LANL (For 2002, 2004 and the whole trace)

Figure 8.1: Failure rates at different time granularities for several platforms and the corresponding autocorrelation functions.

quite high. In addition, since most of the jobs are submitted through the OAR resource manager, hence without direct user interaction, the daily pattern is not clearly observable.

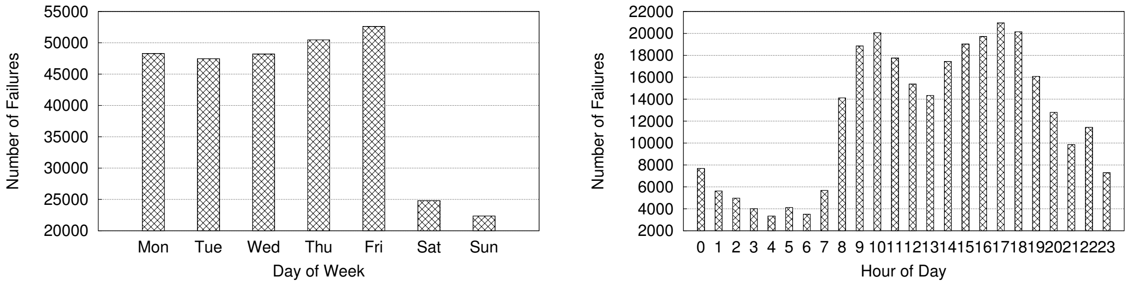
However, during the summer the failure rate decreases, which indicates a correlation between the system load and the failure rate. Finally, as the system size increases over the years, the failure rate does not increase significantly, which indicates system stability. The CONDOR (CAE) data set is a one month long trace collected from a desktop grid using the Condor cycle-stealing scheduler. As expected from a desktop grid, this trace exhibits daily peaks in the failure rate, and hence in the autocorrelation function. In contrast to other desktop grids, the failure rate is lower. The SKYPE data set is a one month long trace collected from a P2P system used by $4,000$ clients. Clients may join or leave the system, and clients that are not online are considered as unavailable in this trace. Similar to desktop grids, there is high autocorrelation at small time lags, and the daily and weekly peaks are more pronounced. The LDNS data set is a two week long trace collected from DNS servers. Unlike P2P systems and desktop grids, DNS servers do not exhibit strong autocorrelation for short time lags with periodic behavior. In addition, as the workload intensity increases during the peak hours of the day, we observe that the failure rate also increases. Finally, the LANL data set is a ten year long trace collected from production HPC clusters. The weekly failure rate is quite low compared to GRID'5000. We do not observe a clear yearly pattern; the failure rate increases during summer 2002 while the failure rate decreases during summer 2004. Since around $3,000$ nodes were added to the LANL system between 2002 and 2003, the failure rate also increases correspondingly.

Last, a few systems exhibit weak autocorrelation in failure occurrence. Figure 8.1(c) and 8.1(d) show the failure rate and the corresponding autocorrelation function for the TERAGRID and NOTRE-DAME systems. The TERAGRID data set is an eight month long trace collected from an HPC cluster that is part of a grid. We observe weak autocorrelation at all time lags, which implies that the failure rates observed over time are independent. In addition, there are no clear hourly or daily patterns, which gives evidence of an erratic occurrence of failures in this system. The NOTRE-DAME data set is a six month long trace collected from a desktop grid. The failure events in this data set consist of the availability/unavailability events of the hosts in this system. Similar to other desktop grids, we observe clear daily and weekly patterns. However, the autocorrelation is low compared to other desktop grids.

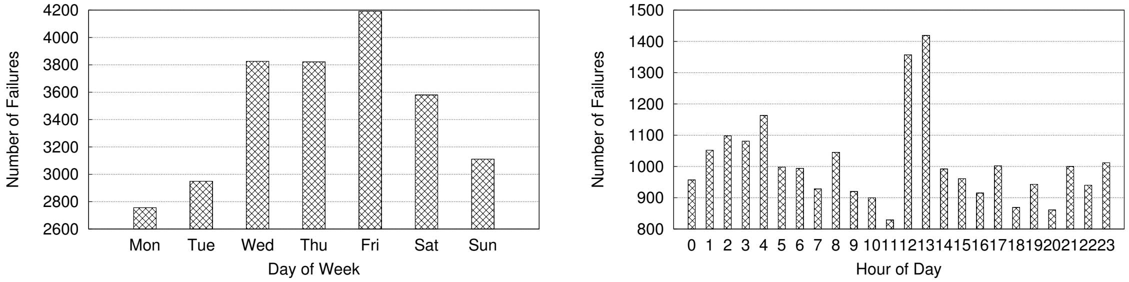
8.2.2 Discussion

As we have shown in the previous section, many systems exhibit strong correlation from small to moderate time lags, which indicates a high degree of predictability. In contrast, a small number of systems (NOTRE-DAME, PNNL, and TERAGRID) exhibit weak autocorrelation; only for these systems, the failure rates observed over time are independent.

We have found that similar systems have similar time-varying behavior, e.g., desktop grids and P2P systems have daily and weekly periodic failure rates, and these systems



(a) GRID'5000



(b) PLANETLAB

Figure 8.2: Daily and hourly failure rates for GRID'5000 and PLANETLAB platforms.

exhibit strong temporal correlation at hourly time lags. Some systems (NOTRE-DAME and CONDOR (CAE)) have direct user interaction, which produces clear daily and weekly patterns in both system load and occurrence of failures—the failure rate increases during work hours and days, and decreases during free days and holidays (the summer).

Finally, not all systems exhibit a correlation between work hours and days, and the failure rate. In the examples depicted in Figure 8.2, while GRID'5000 exhibits this correlation, PLANETLAB exhibit irregular/erratic hourly and daily failure behavior.

Our results are consistent with previous studies [176, 42, 115, 184] as in many traces we observe strong autocorrelation at small time lags, and that we observe correlation between the intensity of the workload and failure rates.

8.3 Modeling the Peaks of Failures

In this section we present a model for the peaks observed in the failure rate process in diverse large-scale distributed systems.

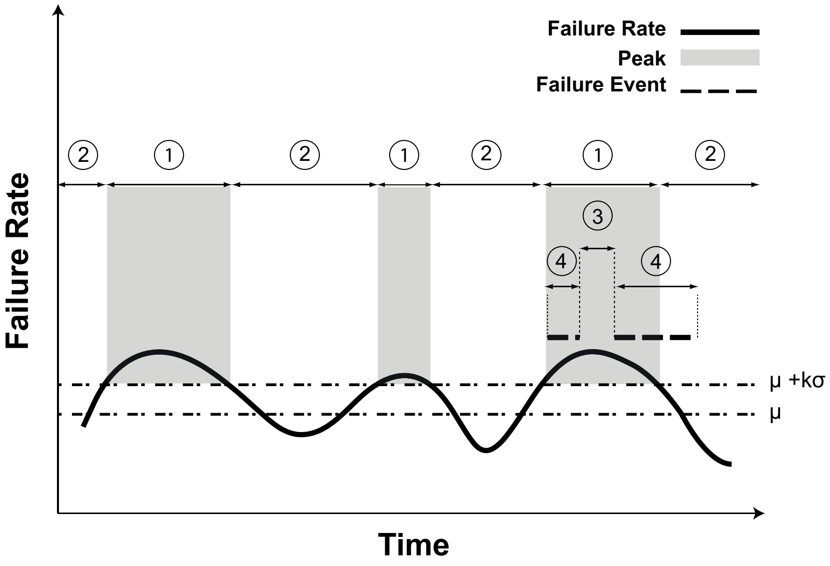


Figure 8.3: Parameters of the peak periods model. The numbers in the figure match the (numbered) model parameters in the text.

8.3.1 Peak Periods Model

Our model of peak failure periods comprises four parameters as shown in Figure 8.3: the peak duration, the time between peaks (inter-peak time), the inter-arrival time of failures during peaks, and the duration of failures during peaks:

1. **Peak Duration**: The duration of peaks observed in a data set.
2. **Time Between Peaks (inter-peak time)**: The time from the end of a previous peak to the start of the next peak.
3. **Inter-arrival Time of Failures During Peaks**: The inter-arrival time of failure events that occur during peaks.
4. **Failure Duration During Peaks**: The duration of failure events that start during peaks. These failure events may last longer than a peak.

Our modeling process is based on analyzing the failure traces taken from real distributed systems in two steps which we describe in turn.

The first step is to identify for each trace the peaks of hourly failure rates. Since there is no rigorous mathematical definition of peaks in time series, to identify the peaks we define a threshold value as $\mu + k\sigma$, where μ is the average and σ is the standard deviation of the failure rate, and k is a positive integer; a period with a failure rate above the threshold is a *peak period*. We adopt this threshold to achieve a good balance between

System	Avg. Peak Duration [s]	Avg. Failure IAT During Peaks [s]	Avg. Time Between Peaks [s]	Avg. Failure Duration During Peaks [s]
Grid'5000	5,047	13	55,101	20,984
Condor (CAE)	5,287	23	87,561	4,397
Condor (CS)	3,927	4	241,920	20,740
Condor (GLOW)	4,200	14	329,040	75,672
TeraGrid	3,680	35	526,500	368,903
LRI	4,080	78	58,371	31,931
Deug	14,914	10	103,800	1,091
Notre-Dame	3,942	21	257,922	280,593
Notre-Dame (CPU)	7,520	33	47,075	22,091
Microsoft	7,200	0	75,315	90,116
UCB	21,272	23	77,040	332
PlanetLab	4,810	264	47,124	166,913
Overnet	3,600	1	14,400	382,225
Skype	4,254	11	112,971	26,402
Websites	5,211	103	104,400	3476
LDNS	4,841	8	42,042	30,212
SDSC	4,984	26	84,900	6,114
LANL	4,122	653	94,968	21,193

Table 8.2: Average values for the model parameters.

capturing in the model extreme system behavior, and characterizing with our model an important part of the system failures (either number of failures or downtime caused to the system). A threshold excluding all but a few periods, for example defining peak periods as distributional outliers, may capture too few periods and explain only a small fraction of the system failures. A more inclusive threshold would lead to the inclusion of more failures, but the data may come from periods with very different characteristics, which is contrary to the goal of building a model for peak failure periods.

In the second step we extract the model parameters from the data sets using the peaks that we identified in the first step. Then we try to find a good fit, that is, a well-known probability distribution and the parameters that lead to the best fit between that distribution and the empirical data. When selecting the probability distributions, we consider the degrees of freedom (number of parameters) of that distribution. Although a distribution with more degrees of freedom may provide a better fit for the data, such a distribution can result in a complex model, and hence it may be difficult to analyze the model mathematically. In this study we use five probability distributions to fit to the empirical data: exponential, Weibull, Pareto, lognormal, and gamma. For the modeling process, we follow the methodology described in Section 8.1.3.

8.3.2 Results

After applying the modeling methodology presented in the previous section and Section 8.1.3, we now present the peak model that we derived from diverse large scale distributed systems.

Table 8.2 shows the average values for all the model parameters for all platforms. The average peak duration varies across different systems, and even for the same type of systems. For example, UCB, MICROSOFT and DEUG are all desktop grids, but the average peak duration widely varies among these platforms. In contrast, for the SDSC, LANL, and PNNL platforms, which are HPC clusters, the average peak duration values are relatively close. The DEUG and UCB platforms have small number of long peak durations resulting in higher average peak durations compared to the other platforms. Finally, as there are two peaks of zero length (single data point) in the OVERNET system, the average peak duration is zero.

The average inter-arrival time during peaks is rather low, as expected, as the failure rates are higher during peaks compared to off-peak periods. For the MICROSOFT platform, as all failures arrive as burst during peaks, average inter-arrival time during peaks is zero.

Similar to the average peak duration parameter, the average time between peaks parameter is also highly variable across different systems. For some systems like TERAGRID, this parameter is in the order of days, and for some systems like OVERNET it is in the order of hours.

Similarly, the duration of failures during peaks highly varies even across similar platforms. For example, the difference between the average failure duration during peaks between the UCB and the MICROSOFT platforms, which are both desktop grids, is huge because the machines in the UCB platform leave the system less often than the machines in the MICROSOFT platform. In addition, in some platforms like OVERNET and TERAGRID, the average failure durations during peaks is in the order of days showing the impact of space-correlated failures, that is multiple nodes failing nearly simultaneously.

Using the AD and KS tests we next determine the best fitting distributions for each model parameter and each system. Since we determine the hourly failure rates using fixed time windows of one hour, the peak duration and the inter-peak time are multiples of one hour. In addition, as the peak duration parameter is mostly in the range $[1h\text{-}5h]$, and for several systems this parameter is mostly $1h$ causing the empirical distribution to have a peak at $1h$, none of the distributions provide a good fit for the peak duration parameter. Therefore, for the peak duration model parameter, we only present an empirical histogram in Table 8.3. We find that the peak duration for almost all platforms are less than $3h$.

Table 8.4 shows the best fitting distributions for the model parameters for all data sets investigated in this study. To generate synthetic yet realistic traces without using a single system as a reference, we create the *average system model* which has the average characteristics of all systems we investigate. We create the average system model as follows. First, we determine the *candidate distributions* for a model parameter with the distributions having the smallest D values for each system. Then, for each model parameter,

Platform / Peak Duration	$1h$	$2h$	$3h$	$4h$	$5h$	$6h$	$\geq 7h$
Grid’5000	**80.56** %	**13.53** %	3.38 %	1.33 %	0.24 %	0.12 %	0.84%
Condor (CAE)	**93.75** %	3.13 %	–	–	–	–	3.12%
Condor (CS)	**90.91** %	9.09 %	–	–	–	–	–
Condor (GLOW)	**83.33** %	**16.67** %	–	–	–	–	–
TeraGrid	**97.78** %	2.22 %	–	–	–	–	–
LRI	**86.67** %	**13.33** %	–	–	–	–	–
Deug	**28.57** %	–	**28.57** %	–	**14.29** %	–	**28.57** %
Notre-Dame	**90.48** %	9.52 %	–	–	–	–	–
Notre-Dame (CPU)	**56.83** %	**17.78** %	9.84 %	3.49 %	5.40 %	3.49 %	3.17
Microsoft	**35.90** %	**33.33** %	**25.64** %	5.13 %	–	–	–
UCB	9.09 %	9.09 %	–	–	–	9.09 %	**72.73** %
PlanetLab	**80.17** %	**13.36** %	3.71 %	1.27 %	0.53 %	0.53 %	0.43
Overnet	**100.00** %	–	–	–	–	–	–
Skype	**90.91** %	4.55 %	–	4.55 %	–	–	–
Websites	**76.74** %	**13.95** %	5.23 %	2.33 %	0.58 %	–	1.16
LDNS	**75.86** %	**13.79** %	**10.34** %	–	–	–	–
SDSC	**69.23** %	**23.08** %	7.69 %	–	–	–	–
LANL	**88.35** %	9.24 %	2.06 %	0.25 %	0.06 %	0.03 %	–
PNNL	**85.99** %	**10.35** %	1.75 %	0.96 %	0.64 %	0.16 %	0.16 %
Avg	**74.79** %	**11.3** %	5.16 %	1.01 %	1.14 %	0.7 %	5.85 %

Table 8.3: Empirical distribution for the peak duration parameter. h denotes hours. Values above 10% are depicted as bold.

we determine the best fitting distribution among the candidate distributions that has the lowest average D value over all data sets. After we determine the best fitting distribution for the average system model, each data set is fit independently to this distribution to find the set of best fit parameters. The parameters of the average system model shown in the ”Avg.” row represent the average of this set of parameters.

For the IAT during peak durations, several platforms do not have a best fitting distribution since for these platforms most of the failures during peaks occur as bursts hence having inter-arrival times of zero. Similarly, for the time between peaks parameter, some platforms (like all CONDOR platforms, Deug, Overnet and UCB platforms) do not have best fitting distributions since these platforms have inadequate number of samples to generate a meaningful model. For the failure duration between peaks parameter, some platforms do not have a best fitting distribution due to the nature of the data. For example, for all CONDOR platforms the failure duration is a multiple of a monitoring interval creating peaks in the empirical distribution at that monitoring interval. As a result, none of the distributions we investigate provide a good fit.

In our model we find that the model parameters do not follow a heavy-tailed distribution since the p-values for the Pareto distribution are very low. For the IAT during peaks parameter, Weibull distribution provides a good fit for most of the platforms. For the time between peaks parameter, we find that the platforms can either be modeled by the lognormal distribution or the Weibull distribution. Similar to our previous model [123], which is

System	IAT During Peaks	Time Between Peaks	Failure Duration During Peaks
Grid’5000	– (see text)	LN(10.30,1.09)	– (see text)
Condor (CAE)	– (see text)	– (see text)	– (see text)
Condor (CS)	– (see text)	– (see text)	– (see text)
Condor (GLOW)	– (see text)	– (see text)	– (see text)
TeraGrid	– (see text)	LN(12.40,1.42)	LN(10.27,1.90)
LRI	LN(3.49,1.86)	LN(10.51,0.98)	– (see text)
Deug	W(9.83,0.95)	– (see text)	LN(5.46,1.29)
Notre-Dame	– (see text)	W(247065.52,0.92)	LN(9.06,2.73)
Notre-Dame (CPU)	– (see text)	W(44139.20,0.89)	LN(7.19,1.35)
Microsoft	– (see text)	G(1.50,50065.81)	W(55594.48,0.61)
UCB	E(23.77)	– (see text)	LN(5.25,0.99)
PlanetLab	– (see text)	LN(10.13,1.03)	LN(8.47,2.50)
Overnet	– (see text)	– (see text)	– (see text)
Skype	– (see text)	W(123440.05,1.37)	– (see text)
Websites	W(66.61,0.60)	LN(10.77,1.25)	– (see text)
LDNS	W(8.97,0.98)	LN(10.38,0.79)	LN(9.09,1.63)
SDSC	W(16.27,0.46)	E(84900)	LN(7.59,1.68)
LANL	G(1.35,797.42)	LN(10.63,1.16)	LN(8.26,1.53)
PNNL	– (see text)	E(160237.32)	– (see text)
Avg	W(193.91,0.83)	LN(10.89,1.08)	LN(8.09,1.59)

Table 8.4: **Peak model:** The parameter values for the best fitting distributions for all studied systems. E,W,LN, and G stand for exponential, Weibull, lognormal, and gamma distributions, respectively.

derived from both peak and off-peak periods, for the failure duration during peaks parameter, we find that the lognormal distribution provides a good fit for most of the platforms. To conclude, for all the model parameters, we find that either the lognormal or the Weibull distributions provide a good fit for the average system model.

Similar to the average system models built for other systems [135], we cannot claim that our average system model represents the failure behavior of an actual system. However, the main strength of the average system model is that it represents a common basis for the traces from which it has been extracted. To generate failure traces for a specific system, individual best fitting distributions and their parameters shown in Table 8.4 may be used instead of the average system.

Next, we compute the average failure duration/inter-arrival time over each data set and only during peaks (Table 8.5). We compare only the data sets used both in this study and our previous study [123], where we modelled each data set individually without isolating peaks. We observe that the average failure duration per data set can be twice as long as the average duration during peaks. In addition, the average failure inter-arrival time per data set is on average nine times the average failure inter-arrival time during peaks. This implies that the distribution per data set is significantly different from the distribution for peaks, and that fault detection mechanisms must be significantly faster during peaks. Likewise, fault-tolerance mechanisms during peaks must have considerably

System	Avg. Failure Duration [h] (Entire)	Avg. Failure Duration [h] (Peaks)	Avg. Failure IAT [s] (Entire)	Avg. Failure IAT [s] (Peaks)
GRID'5000	7.41	5.83	160	13
NOTRE-DAME (CPU)	4.25	6.14	119	33
MICROSOFT	16.49	25.03	6	0
PLANETLAB	49.61	46.36	1,880	264
OVERNET	11.98	106.17	17	1
SKYPE	14.30	7.33	91	11
WEBSITES	1.17	0.97	380	103
LDNS	8.61	8.39	12	8
LANL	5.88	5.89	13,874	653

Table 8.5: The average duration and average IAT of failures for the entire traces and for the peaks.

lower overhead than during non-peak periods.

Finally, for various k values we explore the fraction of downtime caused by failures that originate during peaks and the fraction of the number of failures that originate during peaks (Table 8.6). Noticeably, we find that on average over 50% and up to 95% of the downtime of the systems we investigate are caused by the failures that originate during peaks. This result suggests that failure peaks deserve special attention when designing fault-tolerant distributed systems.

8.4 Related Work

Much work has been dedicated to characterizing and modeling system failures [199, 158, 155, 176, 220, 183]. While the correlation among failure events has received attention since the early 1990s [199], previous studies focus mostly on *space-correlated* failures, that is, on multiple nodes failing nearly simultaneously. Although the *time correlation* of failure events deserve a detailed investigation due to its practical importance [147, 206, 180], relatively little attention has been given to characterize the time correlation of failures in distributed systems. Our work is the first to investigate the time correlation between failure events across a broad spectrum of large-scale distributed systems. In addition, we also propose a model for peaks observed in the failure rate process derived from several distributed systems.

Previous failure studies [199, 158, 155, 176, 220] used few data sets or even data from a single system; their data also span relatively short periods of time. In contrast, we perform a detailed investigation using data sets from diverse large-scale distributed systems including more than $100K$ hosts and $1.2M$ failure events spanning over 15 years of system operation.

Closest to our work, Schroeder and Gibson [183] present an analysis using a large

System	$k = 0.5$		$k = 0.9$		$k = 1.0$		$k = 1.1$		$k = 1.25$		$k = 1.5$		$k = 2.0$	
	Time %	# Failures %	Time %	# Failures %	Time %	# Failures %	Time %	# Failures %	Time %	# Failures %	Time %	# Failures %	Time %	# Failures %
GRID'5000	61.93	74.43	52.11	64.58	49.19	62.56	47.27	60.84	35.93	57.93	33.25	53.60	27.63	44.99
CONDOR (CAE)	63.03	90.91	62.93	90.52	62.93	90.52	62.73	89.88	62.73	89.88	62.57	89.13	62.10	87.08
CONDOR (CS)	80.09	89.56	80.04	88.63	80.04	88.63	80.04	88.63	80.04	88.63	80.04	88.63	80.04	88.63
CONDOR (GLOW)	39.53	70.51	39.37	69.70	39.37	69.70	39.37	69.70	39.37	69.70	39.37	69.70	37.28	67.47
TERAGRID	100	77.10	66.90	77.10	66.90	77.10	66.90	77.10	66.90	77.10	66.90	77.10	62.98	70.61
LRI	87.42	71.87	84.73	62.26	84.73	62.26	77.92	59.47	75.66	57.94	75.66	57.94	73.91	55.71
DEUG	47.31	83.61	26.07	66.46	25.07	63.03	25.07	63.03	22.28	57.76	20.94	53.31	16.83	41.11
NOTRE-DAME	62.62	73.06	58.53	69.72	53.40	69.09	45.19	67.70	45.12	67.19	43.69	64.47	41.79	62.61
NOTRE-DAME (CPU)	73.77	56.92	63.85	43.56	57.92	40.15	56.19	37.93	47.61	33.32	41.88	26.21	28.76	14.99
MICROSOFT	52.16	40.26	37.44	25.10	35.54	23.41	32.20	20.08	28.78	16.81	23.76	12.80	15.35	6.73
UCB	100	100	97.70	97.78	95.31	96.10	95.31	96.10	93.19	94.18	86.75	87.62	54.48	57.88
PLANETLAB	50.55	54.70	38.07	41.81	38.07	41.81	30.02	32.34	30.02	32.34	26.69	24.86	24.02	20.27
OVERNET	68.69	12.90	65.97	7.44	65.97	7.44	65.97	7.44	65.97	7.44	65.97	7.44	65.97	7.44
SKYPE	32.87	36.01	12.06	18.93	7.65	14.93	6.14	13.81	4.32	12.05	3.29	10.74	3.29	10.74
WEBSITES	24.41	31.45	15.33	18.87	13.60	16.59	12.85	14.97	12.33	13.85	9.27	11.92	8.53	10.06
LDNS	38.21	38.80	13.74	13.85	10.14	10.41	7.80	8.28	5.07	5.61	3.25	3.57	2.06	2.45
SDSC	87.57	86.13	67.86	65.25	67.46	64.02	67.46	64.02	65.24	61.01	64.91	59.23	52.20	48.78
LANL	100	44.68	44.69	44.68	44.69	44.68	44.69	44.68	44.69	44.68	44.69	44.68	22.96	21.41
Avg.	65.01	62.93	51.52	53.67	49.88	52.35	47.95	50.88	45.84	49.30	44.04	46.83	37.78	39.94

Table 8.6: Fraction of downtime and fraction of the number of failures due to the failures that originate during peaks.

set of failure data obtained from a high performance computing site. However, this study lacks a time correlation analysis and focuses on well-known failure characteristics like MTTF and MTTR. Sahoo et al. [176] analyze one year long failure data obtained from a single cluster. Similar to the results of our analysis, they report that there is strong correlation with significant periodic behavior. Bhagwan et al. [29] present a characterization of the availability of the OVERNET P2P system. Their study and the study of Chu et al. [50] show that the availability of P2P systems has diurnal patterns. However, neither of these studies characterize the time correlations of failure events.

Traditional failure analysis studies [42, 115] report strong correlation between the intensity of the workload and failure rates. Our analysis brings further evidence supporting the existence of this correlation–we observe more failures during peak hours of the day and during work days in most of the (interactive) traces.

8.5 Summary

Traditional failure models in distributed systems were derived from small scale systems and often under the assumption of independence between failures. However, recent studies have shown evidence that there exist time patterns and other time-varying behavior in the occurrence of failures. Thus, in this chapter we have investigated the time-varying behavior of failures in large-scale distributed systems, and we have proposed a model for time-correlated failures in such systems.

First, we have assessed the presence of time-correlated failures, using traces from nineteen (production) systems, including grids, P2P systems, DNS servers, web servers, and desktop grids. We have found for most of the studied systems that, the failure rates are highly variable, and that the failures exhibit strong periodic behavior and time correlation.

Second, to characterize the periodic behavior of failures and the peaks in failures, we have proposed a peak model with four parameters: the peak duration, the failure inter-arrival time during peaks, the time between peaks, and the failure duration during peaks. We found that the peak failure periods explained by our model are responsible for on average over 50% and up to 95% of the system downtime. We have also found that the Weibull and the lognormal distributions provide good fits for the model parameters. We have provided the best-fitting parameters for these distributions which will be useful to the community when designing and evaluating fault-tolerance mechanisms in large-scale distributed systems.

Last but not least, we have made four new traces available, three Condor traces and one TeraGrid trace, in the Failure Trace Archive, which we hope will encourage others to use the archive and also to contribute to it with new traces.

Chapter 9

Conclusion and Future Work

During the past few decades distributed computing systems have evolved from ARPANET that comprises only a few machines to compute clouds that comprise hundreds of thousands of machines all around the world. The significant advancement in the capabilities of distributed systems make them an important part of our society; in fact, millions of people around the globe depend on distributed infrastructures such as the Internet and the telecommunications networks for various services. As users depend even more on distributed systems, it is inevitable that they also expect more from these infrastructures. It is very important for the users that distributed systems provide consistent performance, that is, the system provides a similar level of performance at all times. It is the focus of this thesis to understand and improve the performance consistency of distributed computing systems.

Towards this end, we have taken an empirical approach, and we have explored diverse distributed systems, such as clusters, multi-cluster grids, and clouds, and diverse workloads, such as Bags-of-tasks (BoTs), interactive perception applications, and scientific workloads. In Chapter 1 of this thesis, we have shown various evidence why this problem is important and non-trivial. In Chapter 2, we have explored overprovisioning as a means to provide consistent performance in multi-cluster grids using realistic simulations. Then in Chapter 3, we have explored the performance of throttling-based overload control techniques in multi-cluster grids with experiments in our DAS-3 research testbed [58]. In Chapter 4, we have proposed four scheduling heuristics for interactive perception applications to minimize their latency subject to migration cost constraints, and we have evaluated these heuristics with two real perception applications on the Open Cirrus testbed [17]. In Chapter 5, we have assessed the performance of four public compute clouds, GoGrid, ElasticHosts, Mosso, and Amazon EC2, which is one of the largest commercial production clouds, using scientific workloads, and we have performed a preliminary analysis of the performance consistency of these clouds. In Chapter 6, we have analyzed the performance variability of ten popular production cloud services provided by Amazon and

Google, and we have assessed the impact of this variability on three large-scale applications. In Chapter 7, we have developed statistical models for failures that occur within a short time frame across distinct components of a system (space-correlated failures) by considering the failure group arrival process, the group size, and the downtime caused by the group of failures using fifteen data sets from the Failure Trace Archive [123]. Finally, in Chapter 8, we have investigated the time-varying behavior of failures in large-scale distributed systems using nineteen data sets from the Failure Trace Archive [123], and we have proposed a model for the peaks in the failure rate.

Overall, with this thesis we have provided evidence that the performance provided by state-of-the-art distributed systems is highly variable, and hence, is far from being consistent. We further show that it is possible to improve the performance consistency of distributed systems using scheduling and resource management techniques that are tailored for particular workloads and systems. Moreover, since failures are one of the primary causes of high performance variability [116, 181, 65, 117, 33, 216, 22, 139], this thesis also provides a fundamental understanding of failures, and provides strong evidence that the fundamental assumption of various previous studies, that is, that failures are independent and identically distributed, is not correct, and hence may lead to suboptimal system designs.

In the rest of this chapter we first present the main conclusions of this thesis (Section 9.1), and then we conclude the chapter with several future research directions (Section 9.2).

9.1 Conclusions

Our work has led to seven major conclusions. The first six conclusions provide insights into the performance of modern distributed systems while the last conclusion provides a fundamental understanding of the behavior of failures in these systems. We present these conclusions in turn.

1. As we have demonstrated with diverse workloads and systems in Chapters 2, 3, and 4, resource management and scheduling is the key to provide consistent performance in distributed computing systems.

2. Overprovisioning is a simple yet effective way to provide consistent performance in multi-cluster grids (Chapter 2).

3. Executing large loosely coupled applications, such as bags-of-tasks, can overload the head-nodes of multi-cluster grids, which results in noticeable degradation in the performance and responsiveness. Throttling the workload rate can help improve

both the raw performance and performance consistency for bursty workloads. Besides, dynamic throttling-based overload control technique can replace the static (hand-tuned) one, which is both slow and costly in multi-cluster grids due to the number of clusters, and difficult to tune due to workload burstiness (Chapter 3).

4. With the incremental placement heuristics that we have designed in Chapter 4 it is possible to provide consistent processing latency to interactive perception applications while at the same time keeping the total cost of migrations, which manifest themselves as latency spikes, within a given bound. We have further shown that using these heuristics it is possible to approach the improvements achieved by rerunning a static placement algorithm, but with less churn in the system.
5. The performance of production cloud infrastructures are currently insufficient for scientific computing at large, although these services are still good alternatives for the users who need resources instantly and temporarily (Chapter 5).
6. The performance of popular production cloud services of Amazon and Google have highly variable performance, which may have noticeable impact on the performance of distributed applications that depend on these services. In particular, the performance of the investigated services exhibits both yearly and daily patterns, and periods of relative stability. Moreover, the impact of this performance variability is significantly different for different types of applications (Chapter 6).
7. Traditional failure models for distributed systems, which assume failure events are independent and identically distributed, are not adequate for large-scale distributed systems, because failure events in these systems are correlated both in time and space. So, system designers need to evaluate their designs under correlated failure events. In addition, a majority of the system downtime in distributed systems is caused by space-correlated failures and peak failure periods; system architects should pay special attention to both space correlations of failures and failure peaks when designing fault-tolerant distributed systems (Chapters 7 and 8).

9.2 Future Research Directions

Although significant research effort has been put into improving the performance consistency of distributed systems both in this thesis and in the literature, as a result of our research we identify six interesting future research directions, which we describe in turn.

1. In Chapter 2 with our simulations with the controller that dynamically overprovisions a multi-cluster grid, we have assumed for simplicity that there is no background load in the system and that the system is homogeneous. A natural extension

to our evaluation would be to explore the impact of background activity and heterogeneity on the performance of our controller. In addition, assessing the performance of the controller in a real deployment will definitely be interesting, and it will pose additional challenges to address.

2. In our performance evaluation of overload control techniques in Chapter 3 we have used workloads that comprise a single BoT. It will definitely be interesting to extend our evaluation to more complex workloads that comprise multiple BoTs of different sizes. Moreover, another interesting future direction is to explore the feasibility of machine learning techniques to control overload in multi-cluster grids as machine learning has shown to be a promising approach for managing the performance of large-scale distributed systems [32].

3. In Chapter 4 we consider the placement problem of interactive perception applications in isolation. However, in practice, we expect an effective system to employ adaptation of the application graph and its degree of parallelism in conjunction with incremental placement to best utilize the cluster resources. Therefore, we raise as an important future work to explore how these two forms of adaptation can interact and be integrated into a runtime system for interactive perception applications.

4. In our performance evaluation of public clouds that we have presented in Chapter 5, we have assessed the network performance with our multi-machine benchmarks using up to 16 Amazon EC2 instances. It would definitely be interesting to perform a more comprehensive evaluation of the network performance at a larger scale. Network performance is particularly important in virtualized systems as it has already been shown that even when the network is lightly utilized, virtualization can cause throughput instability and large latency variations [209]. In addition, Amazon EC2 has recently released Cluster Compute and Cluster GPU Instances that provide high-performance networking capabilities. Exploring the performance of scientific workloads on these instances would be an interesting future work.

5. In our analysis of the performance variability of popular production cloud services in Chapter 6, we have shown the existence of significant performance variability. However, we haven't addressed the question of how to reduce this variability or how to reduce the impact of this variability on large-scale distributed applications, which is a challenging problem on its own. In addition to the challenges we have outlined in the Introduction, namely, the scale of systems, the complexity of systems and workloads, the shared nature of systems, different performance requirements of users, and failures, virtualization and the large scale of cloud infrastructures make this problem even more challenging; modern data centers now comprise up to a million servers [122, 11, 110].

6. In Chapters 7 and 8, we have provided an analysis and modeling of space- and time-correlated failures in large-scale distributed systems. An important contribution would be exploring how these models can be used to reduce the impact of failures on performance variability. For example, our failure models can be used in scheduling and resource management decisions, such as migrating a parallel application when we predict that a failure burst of a particular size is arriving. Similarly, exploring smarter checkpointing algorithms that use these models to predict failure events is also an interesting future work as efficient checkpointing in large-scale distributed systems is an active area of research [36, 154].

Bibliography

[1] Facebook, hadoop, and hive, 2009. http://www.dbms2.com/2009/05/11/facebook-hadoop-and-hive/.

[2] The Cloud Status Team . JSON report crawl, January 2009. http://www.cloudstatus.com/ (consulted in 2009, now defunct).

[3] The HPCC Team . HPCChallenge results, January 2009. [Online]. Available: http://icl.cs.utk.edu/hpcc/hpcc_results.cgi.

[4] The Parallel Workloads Archive Team . The parallel workloads archive logs, January 2009. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[5] Advanced Clustering Tech. . Linpack problem size analyzer, Dec 2008. [Online] Available: http://www.advancedclustering.com/.

[6] A. Adya, W. Bolosky, R. Chaiken, J. Douceur, J. Howell, and J. Lorch. Load management in a large-scale decentralized file system. Technical Report MSR-TR-2004-60, Microsoft Research, 2004.

[7] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. In *Proc. of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 258–265, 2003.

[8] R. Al-Ali, K. Amin, G. von Laszewski, O. Rana, D. Walker, M. Hategan, and N. Zaluzec. Analysis and provision of qos for distributed grid applications. *Journal of Grid Computing*, 2:163–182, 2004.

[9] S. R. Alam, R. F. Barrett, M. Bast, M. R. Fahey, J. A. Kuehn, C. McCurdy, J. Rogers, P. C. Roth, R. Sankaran, J. S. Vetter, P. H. Worley, and W. Yu. Early evaluation of IBM BlueGene/P. In *Proc. of Supercomputing (SC)*, page 23, 2008.

[10] J. Aldrich. R. A. Fisher and the making of maximum likelihood 1912-1922. *Statistical Science*, 12(3):162–176, 1997.

[11] Amazon data center size, 2012. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.

[12] Amazon Inc. . Amazon Elastic Compute Cloud (Amazon EC2), Dec 2008. [Online] Available: `http://aws.amazon.com/ec2/`.

[13] O. Andrew. Data networks are lightly utilized, and will stay that way. *Review of Network Economics*, 2:210–237, 1999.

[14] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models. Technical Report HPL-2002-339, HP, 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-339.html`.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[16] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, A. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 267–278, 1995.

[17] A. I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Ganger, M. A. Kozuch, D. O'Hallaron, M. Kunze, T. T. Kwan, K. Lai, M. Lyons, D. S. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J. Luke, and H. Namgoong. Open cirrus: A global cloud computing testbed. *IEEE Computer*, 43(4):35–43, 2010.

[18] A Avizienis, JC. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[19] Windows azure. `http://www.windowsazure.com/en-us/`.

[20] M. Babcock. XEN benchmarks. Tech. Rep. , Aug 2007. [Online] Available: `mikebabcock.ca/linux/xen/`.

[21] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, T. Kielmann, J. Maassen, R. Nieuwpoort, J. Romein, L. Renambot, T. Rühl, R. Veldema, K. Verstoep, A. Baggio, G. Ballintijn, I. Kuz, G. Pierre, M. Steen, A. Tanenbaum, G. Doornbos, D. Germans, H. Spoelder, E. Baerends, S. Gisbergen, H. Afsermanesh, D. Albada, A. Belloum, D. Dubbeldam, Z. Hendrikse, B. Hertzberger, A. Hoekstra, K. Iskra, D. Kandhai, D. Koelma, F. Linden, B. Overeinder, P. Sloot, P. Spinnato, D. H. J. Epema, A. van Gemund, P. Jonker, A. Radulescu, C. van Reeuwijk, H. Sips, P. Knijnenburg, M. Lew, F. Sluiter, L. Wolters, H. Blom, C. de Laat, and A. Steen. The distributed asci supercomputer project. *SIGOPS Oper. Syst. Rev.*, 34(4):76–96, October 2000.

[22] R. Baldoni, G. Lodi, G. Mariotta, L. Montanari, and M. Rizzuto. Online black-box failure prediction for mission critical distributed systems. Technical report, MIDLAB 3/2012, 2012.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[24] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40:33–37, 2007.

[25] O. Beaumont, V. Boudet, and Y. Robert. The iso-level scheduling heuristic for heterogeneous processors. In *Proc. of the Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 335–350, 2002.

[26] J. Becla and D. L. Wang. Lessons learned from managing a petabyte. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[27] The uc berkeley/stanford recovery-oriented computing (roc) project. `http://roc.cs.berkeley.edu/`.

[28] M. E. Bgin, B. Jones, J. Casey, E. Laure, F. Grey, C. Loomis, and R. Kubli. Comparative study: Grids and clouds, evolution or revolution? Egee-ii report, CERN, June 2008. [Online] Available: `https://edms.cern.ch/file/925013/3/EGEE-Grid-Cloud.pdf`.

[29] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 256–267, 2003.

[30] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total recall: System support for automated availability management. In *Proc. of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, pages 337–350, 2004.

[31] R. Biswas, M. J. Djomehri, R. Hood, H. Jin, C. C. Kiris, and S. Saini. An application-based performance characterization of the Columbia Supercluster. In *Proc. of Supercomputing (SC)*, page 26, 2005.

[32] P. Bodik. *Automating Datacenter Operations Using Machine Learning*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2010.

[33] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proc. of the 5th European conference on Computer systems (EuroSys)*, pages 111–124, 2010.

[34] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 34–43, 2000.

[35] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E. Talbi, and

I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[36] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 33:1–33:11, 2011.

[37] T. Bray. Bonnie, 1996. [Online] Available: `http://www.textuality.com/bonnie/`, Dec 2008.

[38] G. Brumfiel. High-energy physics: Down the petabyte highway. *Nature*, 469(7330):282–283, January 2011.

[39] C. Castillo, G. N. Rouskas, and K. Harfoush. Efficient resource management using advance reservations for heterogeneous grids. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2008.

[40] C. Castillo, G. N. Rouskas, and K. Harfoush. Resource co-allocation for large-scale distributed environments. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 131–140, 2009.

[41] X. Castillo, S. R. McConnel, and D. P. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Transactions on Computers*, 31(7):658–671, 1982.

[42] X. Castillo and D. P. Siewiorek. Workload, performance, and reliability of digital computing systems. In *Proc. of the International Symposium on Fault-Tolerant Computing*, pages 84–89, 1981.

[43] K. T Chen, P. Huang, and C. L. Lei. Effect of network quality on player departure behavior in online games. *IEEE Transactions on Parallel and Distributed Systems*, 20(5):593–606, 2009.

[44] L. Chen, C. Wang, and F. C. M. Lau. Process reassignment with reduced migration cost in grid load rebalancing. In *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–13, 2008.

[45] M. Chen, L. Mummert, P. Pillai, A. Hauptmann, and R. Sukthankar. Controlling your tv with gestures. In *Proc. of the ACM International Conference on Multimedia Information Retrieval (ICMR)*, pages 405–408, 2010.

[46] M. Chen, L. Mummert, P. Pillai, A. Hauptmann, and R. Sukthankar. Exploiting multi-level parallelism for low-latency activity recognition in streaming video. In *Proc. of the ACM Multimedia Systems Conference (MMSys)*, 2010.

[47] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 387–390, 2005.

[48] L. Cherkasova and P. Phaal. Session based admission control: A mechanism for improving the performance of an overloaded web server. Technical Report HPL-98-119, HP, 1998.

[49] D. M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1–14, 1989.

[50] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, 2002.

[51] B. Cirou and E. Jeannot. Triplet: a clustering scheduling algorithm for heterogeneous systems. In *Proc. of the IEEE International Conference on Parallel Processing Workshops (ICPP)*, pages 231–236, 2001.

[52] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 135–144, 2004.

[53] M. Claypool and K. T. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[54] A. Collet, D. Berenson, S. Srinivasa, and D. Ferguson. Object recognition and full pose registration from a single image for robotic manipulation. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3534–3541, 2009.

[55] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in Condor. In *Workflows for e-Science*, pages 357–375. Springer, 2007.

[56] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proc. of the 2nd conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 22–22, 1999.

[57] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable earthquake simulation on petascale supercomputers. In *Proc. of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–20, 2010.

[58] The distributed asci supercomputer 3. `http://www.cs.vu.nl/das3/`.

[59] The distributed asci supercomputer 4. `http://www.cs.vu.nl/das4/`.

[60] M. D. de Assuncao, A. di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 141–150, 2009.

[61] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of the 6th conference on Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, 2004.

[62] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[63] E. Deelman, G. Singh, M. Livny, J. B. Berriman, and J. Good. The cost of doing science on the cloud: the Montage example. In *Proc. of Supercomputing (SC'09)*, page 50, 2008.

[64] J. Dejun, G. Pierre, and CH. Chi. EC2 performance analysis for resource provisioning of service-oriented applications. In *Proc. of the International Conference on Service-Oriented Computing (ICSOC/ServiceWave)*, pages 197–207, 2009.

[65] F. Dinu and T. S. E. Ng. Understanding the effects and implications of compute node related failures in hadoop. In *Proc. of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 187–198, 2012.

[66] M. Dobber, R. van der Mei, and G. Koole. A prediction method for job runtimes on shared processors: Survey, statistical analysis and new avenues. *Performance Evaluation*, 64:755–781, 2007.

[67] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[68] J. Dongarra et al. . Basic linear algebra subprograms technical forum standard. *High Performance Applications and Supercomputing*, 16(1):1–111, 2002.

[69] T. H. Dunigan, M. R. Fahey, J. B. White III, and P. H. Worley. Early evaluation of the Cray X1. In *Proc. of Supercomputing (SC)*, page 18, 2003.

[70] G. B. Dyson. Darwin among the machines: The evolution of global intelligence, 1998. Perseus Books Group.

[71] European grid infrastructure, 2012. `http://www.egi.eu/`.

[72] Y. El-Khamra, H. Kim, S. Jha, and M. Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 383–387, 2010.

[73] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, June 1990.

[74] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proc. of the 13th international Conference on World Wide Web (WWW)*, pages 276–286, 2004.

[75] The history of email, 2012. `http://en.wikipedia.org/wiki/Distributed_computing`.

[76] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.

[77] Callaghan et al. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428–446, 2010.

[78] C. Evangelinos and C. N. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. In *Proc. of the Workshop on Cloud Computing and Its Applications (CCA)*, pages 1–6, 2008.

[79] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1997.

[80] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Computational Grids, pages 15–52. Morgan-Kaufmann, July 1998.

[81] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop (GCE)*, pages 1–10, 2008.

[82] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. *SIGOPS Operating Systems Review*, 31(5):78–91, 1997.

[83] M. Gallet, M. N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. H. J. Epema. A model for space-correlated failures in large-scale distributed systems. In *Proc. of the 16th International Euro-Par Conference on Parallel Processing (EuroPar)*, pages 88–100, 2010.

[84] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[85] L. Gilbert, J. Tseng, R. Newman, S. Iqbal, R. Pepper, O. Celebioglu, J. Hsieh, and M. Cobban. Performance implications of virtualization and hyper-threading on high energy physics applications in a grid environment. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[86] The metrics project, globus metrics. Technical Report v 1. 4, Globus, 2007. `http://incubator.globus.org/metrics/reports/2007-02.pdf`.

[87] GoGrid. GoGrid cloud-server hosting, Dec 2008. [Online] Available: `http://www.gogrid.com`.

[88] Google Inc. . Google App Engine, Run your web applications on Google's infrastructure., Dec 2008. [Online] Available: `http://code.google.com/appengine`.

[89] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, 2008.

[90] J. Gray. Why do computers stop and what can be done about it? In *Proc. of the Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[91] J. Gray. A census of tandem system availability between 1985 and 1990. In *IEEE Transactions on Reliability*, volume 39, pages 409–418, October 1990.

[92] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 217–227, 2002.

[93] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[94] T. Hey and G. Fox. Special issue: Grids and web services for e-science: Editorials. *Concurrency and Computation: Practice and Experience*, 17(2-4):317–322, February 2005.

[95] D. Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical Report GIT-CERCS-09-13, Georgia Institute of Technology, Dec 2008.

[96] V. Hilt and I. Widjaja. Controlling overload in networks of sip servers. In *Proc. of IEEE International Conference on Network Protocols (ICNP)*, pages 83–93, 2008.

[97] A. Hosoi, T. Washio, J. Okada, Y. Kadooka, K. Nakajima, and T. Hisada. A multi-scale heart simulation on massively parallel computers. In *Proc. of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2010.

[98] W. Huang, J. Liu, B. Abali, and D. K. Panda. A case for high performance computing with virtual machines. In *Proc. of the 21st International Conference on Supercomputing (ICS)*, pages 125–134, 2006.

[99] A. Iosup, C. Dumitrescu, D. H. J. Epema, H. Li, and L. Wolters. How are real grids used? The analysis of four grid traces and its implications. In *Proc. of the 7th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 262–269, 2006.

[100] A. Iosup and D. H. J. Epema. GrenchMark: A framework for analyzing, testing, and comparing grids. In *Proc. of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 313–320, 2006.

[101] A. Iosup and D. H. J. Epema. Grid computing workloads: Bags of tasks, workflows, pilots, and others. *IEEE Internet Computing*, 15:19–26, 2011.

[102] A. Iosup, D. H. J. Epema, T. Tannenbaum, M. Farrellee, and M. Livny. Inter-operating grids through delegated matchmaking. In *Proc. of Supercomputing (SC'07)*, page 13, 2007.

[103] A. Iosup, M. Jan, O. O. Sonmez, and D. H. J. Epema. On the dynamic resource availability in grids. In *Proc. of the 8th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 26–33, 2007.

[104] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The Grid Workloads Archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[105] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2010.

[106] A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proc. of the 7th International Symposium on High Performance Distributed Computing (HPDC)*, pages 97–108, 2008.

[107] A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 97–108, 2008.

[108] A. Iosup, O. O. Sonmez, and D. H. J. Epema. DGSim: Comparing grid resource management architectures through trace-based simulation. In *Proc. of the European Conference on Parallel Processing (Euro-Par)*, volume 5168 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 2008.

[109] A. Iosup, M. N. Yigitbasi, and D. H. J. Epema. On the performance variability of production cloud services. In *Proc. of the 11th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 104–113, 2011.

[110] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, April 2007.

[111] T. Z. Islam, S. Bagchi, and R. Eigenmann. Falcon: a system for reliable checkpoint recovery in shared grid environments. In *Proc. of Supercomputing (SC)*, pages 1–12, 2009.

[112] M. A. Iverson, F. Özgüner, and G. J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *Proc. of the IEEE Heterogeneous Computing Workshop (HCW)*, pages 93–100, 1995.

[113] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for web servers. In *In Workshop on Performance and QoS of Next Generation Networks*, pages 225–244, 2000.

[114] R. K. Iyer, S. E. Butner, and E. J. McCluskey. A statistical failure/load relationship: Results of a multicomputer study. *IEEE Transactions on Computers*, 31(7):697–706, 1982.

[115] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Transactions on Computer Systems*, 4(3):214–237, 1986.

[116] S. P. Kavulya, K. Joshi, M. Hiltunen, S. Daniels, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[117] S. P. Kavulya, K. Joshi, Matti M. Hiltunen, S. Daniels, R. Gandhi, and P. Narasimhan. Practical experiences with chronics discovery in large telecommunications systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML)*, pages 7:1–7:8, 2011.

[118] YS. Kee, H. Casanova, and A. A. Chien. Realistic modeling and svnthesis of resources for computational grids. In *Proc. of Supercomputing (SC)*, page 54, 2004.

[119] D. J. Kerbyson, A. Hoisie, and H. J. Wasserman. A performance comparison between the Earth Simulator and other terascale systems on a characteristic ASCI workload. *Concurrency and Computation: Practice and Experience*, 17(10):1219–1238, 2005.

[120] T. Killalea. Meet the virts. *Queue*, 6(1):14–18, 2008.

[121] S. Kleban and S. Clearwater. Quelling queue storms. In *Proc. of the 12th International Symposium on High Performance Distributed Computing (HPDC)*, page 162, 2003.

[122] Data Center Knowledge. Google uses about 900,000 servers, 2011. `http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/`.

[123] D. Kondo, B. Javadi, A. Iosup, and D. H. J. Epema. The Failure Trace Archive: Enabling comparative analysis of failures in diverse distributed systems. In *Proc. of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 1–10, 2010.

[124] S. Kounev, R. Nou, and J. Torres. Autonomic qos-aware resource management in grid computing using online performance models. In *Proc. of the International Conference on Performance Evaluation Methodologies and Tools*, pages 1–10, 2007.

[125] A. Kowalski. Bonnie - file system benchmarks. Tech. Rep. , Jefferson Lab, Oct 2002. [Online] Available: `http://cc.jlab.org/docs/scicomp/benchmark/bonnie.html`.

[126] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxcluster: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[127] Y. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47:58–77, 1997.

[128] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[129] Y. Kwok, A. A. Maciejewski, H. J. Siegel, I. Ahmad, and A. Ghafoor. A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 66:77–98, January 2006.

[130] A. Leff, J. T. Rayfield, and D. M. Dias. Service-level agreements and commercial grids. *IEEE Internet Computing*, 7:44–50, 2003.

[131] T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. In *IEEE Transactions on Reliability*, volume 39, pages 419–432, October 1990.

[132] G. Linden. Make data useful, 2006. `http://home.blarg.net/~glinden/StanfordDataMining.2006-11-29.ppt`.

[133] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[134] U. Lublin and D. G. Feitelson. Workload on parallel supercomputers: modeling characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[135] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[136] P. Luszczek, D. H. Bailey, J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. S12 - The HPC Challenge (HPCC) benchmark suite. . In *Proc. of Supercomputing (SC)*, page 213, 2006.

[137] J. Maguire, J. Vance, and C. Harvey. 85 cloud computing vendors shaping the emerging cloud, Aug 2009. ITManagement Tech. Rep. `itmanagement.earthweb.com/features/article.php/12297$_$3835941$_$2/85-Cloud-Computing-Vendors-Shaping-the-Emerging-Cloud.htm`.

[138] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Proc. of the IEEE Heterogeneous Computing Workshop (HCW)*, pages 57–69, 1998.

[139] O. Malik. Parts of amazon web services suffer an outage, 2012. `http://gigaom.com/cloud/did-amazons-web-services-go-down`.

[140] P. Matikainen, P. Pillai, L. Mummert, R. Sukthankar, and M. Hebert. Prop-free pointing detection in dynamic cluttered environments. In *Proc. of the IEEE Conference on Automatic Face and Gesture Recognition*, 2011.

[141] J. Matthews, T. Garfinkel, C. Hoff, and J. Wheeler. Virtual machine contracts for datacenter and cloud computing environments. In *Proc. of the Workshop on Automated Control for Datacenters and Clouds (ACDC)*, pages 25–30, 2009.

[142] L. McVoy and C. Staelin. LMbench - tools for performance analysis. [Online] Available: `http://www.bitmover.com/lmbench/`, Dec 2008.

[143] M. Mehech. The impact of failures on large distributed storage systems. August 2007.

[144] D. Menascé and E. Casalicchio. A framework for resource allocation in grid computing. In *Proc. of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 259–267, 2004.

[145] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 13–23, 2005.

[146] N. Mi, G. Casale, A. Riska, Q. Zhang, and E. Smirni. Autocorrelation-driven load control in distributed systems. In *Proc. of IEEE International Symposium on the Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[147] J. W. Mickens and B. D. Noble. Exploiting availability prediction in distributed systems. In *Proc. of the 3rd conference on Networked Systems Design and Implementation (NSDI)*, pages 6–6, 2006.

[148] U. F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. Database systems on virtual machines: How much do you lose? In *Proc. of the 24th IEEE International Conference on Data Engineering Workshop (ICDEW)*, pages 35–41, 2008.

[149] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proc. of the SIGOPS/EuroSys European Conference on Computer Systems*, pages 293–304, 2006.

[150] P. J. Mucci and K. S. London. Low level architectural characterization benchmarks for parallel computers. Technical Report UT-CS-98-394, U. Tennessee, 1998.

[151] V. Nae, A. Iosup, S. Podlipnig, R. Prodan, D. H. J. Epema, and T. Fahringer. Efficient management of data center resources for massively multiplayer online games. In *Proc. of Supercomputing (SC)*, page 10, 2008.

[152] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proc. of the 21st International Conference on Supercomputing (ICS)*, pages 23–32, 2007.

[153] H. B. Newman, M. H. Ellisman, and J. A. Orcutt. Data-intensive e-science frontier research. *Commun. ACM*, 46(11):68–77, November 2003.

[154] B. Nicolae and F. Cappello. Blobcr: efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 34:1–34:12, 2011.

[155] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Proc. of the European Conference on Parallel Processing (Euro-Par)*, pages 432–441, 2005.

[156] D. Nurmi, R. Wolski, and J. Brevik. Varq: virtual advance reservations for queues. In *Proc. of the 17th International Symposium on High Performance Distributed Computing (HPDC)*, pages 75–86, 2008.

[157] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proc. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 124–131, 2009.

[158] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 1–1, 2003.

[159] The open science grid project (osg), July 2007.

[160] S. Ostermann, A. Iosup, R. Prodan, T. Fahringer, and D. H. J. Epema. On the characteristics of grid workflows. In *Proc. of the CoreGRID Workshop on Integrated Research in Grid Computing (CGIW)*, pages 431–442, 2008.

[161] S. Ostermann, A. Iosup, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. An early performance analysis of cloud computing services for scientific computing. In *CloudComp*, volume 34 of *LNICST*, pages 115–31, 2009.

[162] S. Ostermann, A. Iosup, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Proc. of the 1st International Conference on Cloud Computing (CloudComp)*, pages 115–131, 2009.

[163] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon S3 for science grids: a viable solution? In *Proc. of the International Workshop on Data-aware Distributed Computing (DADC)*, pages 55–64, 2008.

[164] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.

[165] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of Supercomputing (SC)*, page 55, 2003.

[166] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 1995.

[167] P. Pillai, L. Mummert, S. Schlosser, R. Sukthankar, and C. Helfrich. Slipstream: Scalable low-latency interactive perception on streaming data. In *Proc. of the ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 43–48, 2009.

[168] A. Pras, R. Van De Meent, and M. Mandjes. Qos in hybrid networks - an operator's perspective. In *Proc. of the 13th International Workshop on Quality of Service*, 2005.

[169] R. Prodan and S. Ostermann. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *Proc. of the IEEE/ACM International Conference on Grid Computing (GRID)*, pages 1–10, 2009.

[170] B. Quétier, V. Néri, and F. Cappello. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 5(1):83–98, 2007.

[171] The rackspace cloud. `http://www.rackspace.com/cloud/`.

[172] RightScale. Amazon usage estimates, Aug 2009. [Online] Available: `blog.rightscale.com/2009/10/05/amazon-usage-estimates`.

[173] G. Rosen. Cloud usage analysis series, Aug 2009. [Online] Available: `www.jackofallclouds.com/category/analysis`.

[174] K. Ryu, J. Hollingsworth, and P. Keleher. Efficient network and i/o throttling for fine-grain cycle stealing. In *Proc. of Supercomputing (SC)*, pages 3–3, 2001.

[175] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems*, 14(4):344–384, 1996.

[176] R. Sahoo, A. Sivasubramaniam, M. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pages 772–, 2004.

[177] S. Saini, D. Talcott, D. C. Jespersen, M. J. Djomehri, H. Jin, and R. Biswas. Scientific application-based performance comparison of SGI Altix 4700, IBM POWER5+, and SGI ICE 8200 supercomputers. In *Proc. of Supercomputing (SC)*, page 7, 2008.

[178] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Proc. of the Heterogeneous Computing Workshop (HCW)*, volume 2, page 111b, 2004.

[179] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12:253–262, December 2004.

[180] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42(3):1–42, 2010.

[181] R. R. Sambasivan and G. R. Ganger. Automated diagnosis without predictability is a recipe for failure. In *Proc. of the 4th USENIX Workshop on Hot Topics in Cloud (HotCloud)*, 2012.

[182] P. Scholz and E. Harbeck. Task assignment for distributed computing. In *Proc. of the Advances in Parallel and Distributed Computing Conference*, pages 270–277, 1997.

[183] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pages 249–258, 2006.

[184] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. of the 5th USENIX conference on File and Storage Technologies (FAST)*, page 1, 2007.

[185] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technologies*, 6(1):20–52, 2006.

[186] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *Proc. of the IEEE Heterogeneous Computing Workshop (HCW)*, pages 98–104, 1996.

[187] M. Siddiqui, A. Villazn, and T. Fahringer. Grid allocation and reservation - grid capacity planning with negotiation-based advance reservation for optimized qos. In *Proc. of Supercomputing (SC)*, page 103, 2006.

[188] G. C. Sih and E. A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proc. of the IEEE Symposium on Parallel and Distributed Processing (IPDPS)*, pages 42–49, 1990.

[189] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proc. of the IEEE Heterogeneous Computing Workshop (HCW)*, pages 86–97, 1996.

[190] D. Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *Proc. of the International Symposium on Workload Characterization*, 2005.

[191] W. Smith, I. Foster, and V. E. Taylor. Scheduling with advanced reservations. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 127, 2000.

[192] O. O. Sonmez, M. N. Yigitbasi, S. Abrishami, A. Iosup, and D. H. J. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 49–60, 2010.

[193] O. O. Sonmez, M. N. Yigitbasi, A. Iosup, and D. H. J. Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proc. of International Symposium on High Performance Distributed Computing (HPDC)*, pages 111–120, 2009.

[194] N. Sotomayor, K. Keahey, and I. Foster. Overhead matters: A model for virtual resource management. In *Proc. of the International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, pages 5–5, 2006.

[195] Spec cpu2006 benchmark. [Online]: http://www. spec. org/cpu2006/.

[196] A. Sridhar and A. Sowmya. Multiple camera, multiple person tracking with pointing gesture recognition in immersive environments. In *Advances in Visual Computing*, volume 5358 of *Lecture Notes in Computer Science*, pages 508–519. Springer Berlin / Heidelberg, 2008.

[197] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *Proc. of the International World Wide Web Conference (WWW)*, pages 331–340, 2007.

[198] D. Tang and R. K. Iyer. Dependability measurement and modeling of a multicomputer system. *IEEE Transactions on Computers*, 42(1):62–75, 1993.

[199] D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modeling of a vaxcluster system. In *Proc. International Symposium on Fault-tolerant computing*, pages 244 –251, 1990.

[200] The nsf teragrid. `http://www.teragrid.org`.

[201] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 152–161, 2003.

[202] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[203] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18:789–803, 2007.

[204] Uk e-science (grid) core programme. `http://www.escience-grid.org.uk/index.htm`.

[205] B. Urgaonkar and P. Shenoy. Cataclysm: policing extreme overloads in internet applications. In *Proc. of the 14th International Conference on World Wide Web (WWW)*, pages 740–749, 2005.

[206] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. In *Proc. of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 62–71, 2001.

[207] J. S. Vetter, S. R. Alam, T. H. Dunigan Jr., M. R. Fahey, P. C. Roth, and P. H. Worley. Early evaluation of the Cray XT3. In *Proc. of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006.

[208] E. Walker. Benchmarking Amazon EC2 for HP Scientific Computing. *Login*, 33(5):18–23, Nov 2008.

[209] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *Proc. of the 29th conference on Information communications (INFOCOM)*, pages 1163–1171, 2010.

[210] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47:8–22, November 1997.

[211] L. Wang, J. Zhan, W. Shi, Y. Liang, and L. Yuan. In cloud, do mtc or htc service providers benefit from the economies of scale? In *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2009.

[212] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 26–28, 2003.

[213] J. Widmer, R. Denda, and M. Mauve. A survey on tcp-friendly congestion control. *IEEE Network*, 15(3):28 –37, May 2001.

[214] Computer cluster, 2012. `http://en.wikipedia.org/wiki/Computer_cluster`.

[215] Distributed computing, 2012. `http://www.nethistory.info/HistoryoftheInternet/email.html`.

[216] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[217] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. A. Yelick. The potential of the Cell processor for scientific computing. In *Proc. of the 3rd Conference on Computing Frontiers (CF)*, pages 9–20, 2006.

[218] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. In *Performance Evaluation Review*, pages 575–611, 2006.

[219] J. Worringen and K. Scholtyssik. MP-MPICH: User documentation & technical notes, Jun 2002.

[220] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked windows nt system field failure data analysis. In *Proc. of the Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 178–185, 1999.

[221] M. N. Yigitbasi, K. Datta, N. Jain, and T. Willke. Energy efficient scheduling of mapreduce workloads on heterogeneous clusters. In *Proc. of the 2nd International Workshop on Green Computing Middleware*, pages 1:1–1:6, 2011.

[222] M. N. Yigitbasi and D. H. J. Epema. Overdimensioning for consistent performance in grids. In *Proc. of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 526–529, 2010.

[223] M. N. Yigitbasi and D. H. J. Epema. Static and dynamic overprovisioning strategies for performance consistency in grids. In *Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 145–152, 2010.

[224] M. N. Yigitbasi and D. H. J. Epema. Performance evaluation of overload control in multi-cluster grids. In *Proc. of the 12th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 173–180, 2011.

[225] M. N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. H. J. Epema. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 65–72, 2010.

[226] M. N. Yigitbasi, A. Iosup, D. H. J. Epema, and S. Ostermann. C-meter: A framework for performance analysis of computing clouds. In *Proc. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 472–477, 2009.

[227] M. N. Yigitbasi, A. Iosup, D. H. J. Epema, and S. Ostermann. C-meter: A framework for performance analysis of computing clouds. In *Proc. of the 15th ASCI Conference*, 2009.

[228] M. N. Yigitbasi, L. Mummert, P. Pillai, and D. H. J. Epema. Incremental placement of interactive perception applications. In *Proc. of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, pages 123–134, 2011.

[229] L. Youseff, M. Butrico, and D. Da Silva. Towards a unified ontology of cloud computing. In *Proc. of the Grid Computing Environments Workshop (GCE)*, pages 1–10, 2008.

[230] L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, pages 141–152, 2008.

[231] L. Youseff, R. Wolski, B. C. Gorda, and C. Krintz. Paravirtualization for HPC systems. In *Proc. of the Workshop on Xen in High-Performance Cluster and Grid Computing*, volume 4331 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 2006.

[232] W. Yu and J. S. Vetter. Xen-based HPC: A parallel I/O perspective. In *Proc. of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 154–161, 2008.

[233] J. Zhan, L. Wang, B. Tu, Y. Li, P. Wang, W. Zhou, and D. Meng. Phoenix cloud: Consolidating different computing loads on shared cluster system for large organization. In *Proc. of the Workshop on Cloud Computing and Its Applications – Posters (CCA)*, pages 7–11, 2008.

[234] Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *Proc. of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 233–252, 2004.

[235] Q. Zhu, M. N. Yigitbasi, and P. Pillai. Running interactive perception applications on opencirrus. In *Proc. of the 6th International Open Cirrus Summit*, pages 17–21, 2011.

Summary

During the past few decades, we have seen several major innovations in the field of distributed computing systems, which have really resulted in significant advances in the capabilities of such systems. Initially, around the late 1970s the increasing complexity of workloads resulted in the invention of clusters that comprise multiple machines connected over a local area network. Later, in the 1990s, grid computing was invented to give users access to a large amount of resources from different administrative domains on-demand, similar to the public utilities, and since then various grids have been deployed all around the world. Recently, cloud computing has been emerging as a new large-scale distributed computing paradigm where service providers rent their infrastructures, services, and platforms to their clients on-demand.

With the increasing adoption of distributed systems in both academia and industry, and with the increasing computational and storage requirements of distributed applications, users inevitably demand more from these systems. Moreover, users also depend on these systems for latency and throughput sensitive applications, such as interactive perception applications and MapReduce applications, which make the performance of these systems even more important. Therefore, for the users it is very important that distributed systems provide consistent performance, that is, the system provides a similar level of performance at all times.

In this thesis we address the problem of understanding and improving the performance consistency of state-of-the-art distributed computing systems. Towards this end, we take an empirical approach and we investigate various resource management, scheduling, and statistical modeling techniques with real system experiments in diverse distributed systems, such as clusters, multi-cluster grids, and clouds, using various types of workloads, such as Bags-of-tasks (BoTs), interactive perception applications, and scientific workloads. In addition, as failures are known to be an important source of significant performance inconsistency, we also provide fundamental insights into the characteristics of failures in distributed systems, which is required to design systems that can mitigate the impact of failures on performance consistency.

In Chapter 1 of this thesis we present the performance consistency problem in distributed computing systems, and we describe why this problem is challenging in such

systems.

In Chapter 2, we assess the benefit of overprovisioning on the performance consistency of BoTs in multi-cluster grids. Overprovisioning can be defined as increasing the capacity, by a factor that we define as the overprovisioning factor, of a system to better handle the fluctuations in the workload, and to provide consistent performance even under unexpected user demands. Through simulations with realistic workload models we explore various overprovisioning strategies with different overprovisioning factors and different scheduling policies. We find that beyond a certain value for the overprovisioning factor there is only slight improvement in performance consistency with significant additional costs. We also find that by dynamically tuning the overprovisioning factor, we can significantly increase the number of BoTs that have a makespan within a user specified range, thus improving the performance consistency.

In Chapter 3, we evaluate the performance of throttling-based overload control techniques in multi-cluster grids motivated by our DAS-3 multi-cluster grid, where running hundreds of tasks concurrently leads to overloads in the cluster head-nodes. We find that throttling results in a decrease (in most cases) or at least in a preservation of the makespan of bursty workloads while significantly improving the tail behavior of the application performance, which leads to better performance consistency and reduces the overload of the head-nodes. Our results also show that our adaptive throttling technique significantly improves the application performance and the system responsiveness, when compared with the hand-tuned multi-cluster system without throttling.

In Chapter 4, we address the problem of incremental placement of interactive perception applications on clusters of machines to provide a responsive user experience. These applications require both low latency and, if possible, no latency spikes at all; frequent migrations of the application components can introduce such spikes, which reduces the quality of the user experience. We design and evaluate four incremental placement heuristics that cover a broad range of trade-offs of computational complexity, churn in the placement, and ultimate improvement in the latency. Through simulations and real system experiments in the Open Cirrus testbed we find that it is worth adjusting the schedule using our heuristics after a perturbation to the system or the workload, and that our heuristics can approach the improvements achieved by completely rerunning a static placement algorithm, but with significantly less churn.

In Chapter 5, using various well-known benchmarks, such as LMbench, Bonnie, CacheBench, and the HPC Challenge Benchmark, we conduct a comprehensive performance study with four public clouds, including Amazon EC2, which is one of the largest production clouds. Notably, we find that the compute performance of the tested clouds is low. Furthermore, we also perform a preliminary assessment of the performance consistency of these clouds, and we find that noticeable performance variability is present for some of the cloud resource types we have explored, which motivates us to explore the performance variability of clouds in depth in the next chapter. Finally, we compare the performance and cost of clouds with those of scientific computing alternatives, such as grids and parallel production infrastructures. Our results show that while current cloud computing services are insufficient for scientific computing at scale, they may still be a good alternative for the scientists who need resources instantly and temporarily.

In Chapter 6, we explore the performance variability of production cloud services using year-long traces that comprise performance data for two popular cloud services: Amazon Web Services and Google App Engine. We find that the performance of the investigated cloud services exhibits on the one hand yearly and daily patterns, and on the other hand periods of stable performance. We also find that many of these services exhibit high variation in the monthly median values, which indicates large performance variability over time. In addition, through trace-based simulations of different large-scale distributed applications we find that the impact of the performance variability varies significantly across different application types.

In Chapter 7, we develop a statistical model for space-correlated failures, that is, for failures that occur within a short time period across different system components using fifteen data sets in the Failure Trace Archive, which is an online public repository of availability traces taken from diverse parallel and distributed systems. In our failure model we consider three aspects of failure events: the group arrival process, the group size, and the downtime caused by the group of failures. We find that the lognormal distribution provides a good fit for these parameters. Notably, we also find that for seven out of the fifteen traces we investigate, space-correlated failures are a major cause of the system downtime. Therefore, these seven traces are better represented by our model than by traditional models, which assume that the failures of the individual components of the system are independent and identically distributed.

In Chapter 8, we investigate the time-varying behavior of failure events in diverse large-scale distributed systems using nineteen data sets in the Failure Trace Archive. We find that for most of the studied systems the failure rates are highly variable, and that failures exhibit strong periodic behavior and time correlations. Moreover, to characterize the peaks in the failure rate we develop a model that considers four parameters: the peak duration, the failure inter-arrival time during peaks, the time between peaks, and the failure duration during peaks. We find that the peak failure periods explained by our model are responsible for a significant portion of the system downtime, suggesting that failure peaks deserve special attention when designing fault-tolerant distributed systems. We believe that our failure models can be used for predictive scheduling and resource management decisions, which can help to mitigate the impact of failures on the performance variability in distributed systems.

Finally, in Chapter 9, we present the conclusions of this thesis and we further present several interesting future research directions. With various workloads and distributed computing systems we show empirically how we can improve the performance consistency of such systems. Moreover, this thesis also provides a fundamental understanding of the characteristics of failures in distributed systems, which is required to design systems that can mitigate the impact of failures on performance consistency. A particularly important extension to our work is to investigate how we can improve the performance consistency of commercial cloud computing infrastructures. We believe that our research presented in this thesis has already taken initial steps in this direction.

Samenvatting

Gedurende de afgelopen decennia zijn er verscheidene grote innovaties geweest op het gebied van gedistribueerde systemen die gezorgd hebben voor een belangrijke vooruitgang in de mogelijkheden van dergelijke systemen. De toenemende complexiteit van de werklasten resulteerde in de late jaren zeventig van de vorige eeuw in het ontstaan van *clusters* van computers die via een locaal netwerk met elkaar verbonden waren. Later, in de jaren negentig, werd *grid computing* ontwikkeld om gebruikers op afroep toegang te geven tot een grote hoeveelheid *resources* gespreid over verschillende administratieve domeinen, net zoals nutsvoorzieningen, en sindsdien zijn er vele *grids* over de hele wereld in gebruik genomen. Recent is *cloud computing* opgekomen als een nieuw paradigma voor gedistribueerde verwerking op grote schaal waarin de aanbieders hun infrastructuur, diensten en platforms aan klanten op afroep verhuren.

Met het toenemende gebruik van gedistribueerde systemen in zowel universiteiten als in de industrie, en met de toenemende vereisten wat betreft rekenkracht en opslagcapaciteit van gedistribueerde applicaties, stellen gebruikers steeds hogere eisen aan deze systemen. Bovendien hebben gebruikers deze systemen ook nodig voor applicaties die een snelle respons of een grote doorstroming vereisen, zoals applicaties die interactieve waarneming doen en *MapReduce* applicaties, hetgeen de prestaties van deze systemen alleen maar nog belangrijker maakt. Daarom is het voor gebruikers erg belangrijk dat gedistribueerde systemen consistente prestaties bieden, dat wil zeggen, dat ze te allen tijde een vergelijkbaar niveau van prestaties bieden.

In dit proefschrift behandelen we het probleem van het begrijpen en verbeteren van de consistentie van de prestaties van de huidige gedistribueerde computersystemen. Daarvoor gebruiken we een empirische benadering en onderzoeken we verscheidene technieken voor *resource management*, *scheduling*, en statistisch modelleren met behulp van experimenten in echte systemen zoals *clusters*, *multi-cluster grids*, en *clouds*. Daarbij gebruiken we verschillende typen werklasten, zoals *Bags-of-Tasks* (BoTs), applicaties voor interactieve waarneming, en wetenschappelijke applicaties. Omdat storingen een belangrijke bron van inconsistentie in prestaties vormen, verschaffen we bovendien fundamentele inzichten in de karakteristieken van storingen in gedistribueerde systemen, hetgeen nodig is om systemen te ontwerpen waarin hun invloed op de consistentie van de prestaties wordt verzacht.

In Hoofdstuk 1 van dit proefschrift formuleren we het probleem van de consistentie van de prestaties van gedistribueerde computersystemen, en leggen we uit waarom dit probleem in deze systemen uitdagend is.

In Hoofdstuk 2 gaan we het nut na van overvoorziening op de consistentie in prestaties van BoTs in *multi-cluster grids*. Overvoorziening kan worden gedefinieerd als het vergroten van de capaciteit, met een factor die we definiëren als de overvoorzieningsfactor, van een systeem om beter de fluctuaties in de werklast aan te kunnen, en om consistente prestaties te bieden zelfs als de vraag van gebruikers onverwacht groot is. Door middel van simulaties met realistische modellen voor de werklasten onderzoeken we verscheidene strategieën voor overvoorziening met verschillende overvoorzieningsfactoren en verschillende *scheduling policies*. Het blijkt dat boven een bepaalde waarde van de overvoorzieningsfactor er slechts een kleine verbetering in de consistentie van de prestaties bereikt kan worden tegen hoge additionele kosten. Tevens blijkt dat we met het dynamisch aanpassen van de overvoorzieningsfactor het aantal BoTs dat een verwerkingstijd binnen door de gebruiker gestelde grenzen heeft, aanzienlijk kunnen verhogen, hetgeen de prestatie-consistentie verbetert.

In Hoofdstuk 3 evalueren we de prestaties van technieken voor het beheersen van de overbelasting door middel van werkdosering in *multi-cluster grids* zoals het Nederlandse DAS-3 systeem, waarin het gelijktijdig draaien van honderden applicaties de *head-nodes* van de *clusters* overbelast. Het blijkt dat werkdosering meestal resulteert in een reductie of tenminste het gelijkblijven van de tijdsduur om pieken in de werklast af te handelen, terwijl het de uitschieters in responstijd van applicaties sterk reduceert, hetgeen leidt tot betere consistentie in de prestaties en reductie in de overbelasting van de *head-nodes*. Onze resultaten laten ook zien dat adaptieve doseringstechnieken de prestaties van applicaties en de responsiviteit van systemen beduidend verbetert in vergelijking met het *multi-cluster* systeem zonder dosering dat met de hand is afgesteld.

In Hoofdstuk 4 behandelen we het probleem van de incrementele plaatsing van applicaties voor interactieve waarneming op *clusters* van machines om de gebruiker een snelle respons te laten ervaren. Deze applicaties vereisen een snelle respons zonder uitschieters; frequente migratie van de componenten van een applicatie kunnen zulke uitschieters veroorzaken, hetgeen de kwaliteit van de ervaring van de gebruiker vermindert. We ontwerpen en beoordelen vier heuristieken voor incrementele plaatsing die een breed spectrum bestrijken van de afwegingen van algoritmische complexiteit, frequentie van verplaatsing van de componenten van applicaties, en de uiteindelijke verbetering in de respons. Door middel van simulaties en experimenten in het Open Cirrus testsysteem blijkt dat het de moeite loont om de plaatsing van componenten aan te passen met onze heuristieken na een verstoring van het systeem of de werklast, en dat onze heuristieken de verbeteringen benaderen die kunnen worden bereikt door het opnieuw uitvoeren van een algoritme voor niet-dynamische plaatsing, maar met beduidend minder migraties.

In Hoofdstuk 5 beschrijven we een uitgebreide studie van de prestaties van vier publieke *clouds*, inclusief Amazon EC2, dat één van de grootste productie-*clouds* is. Deze studie is uitgevoerd met bekende *benchmarks* zoals LMbench, Bonnie, CacheBench, en de HPC Challenge Benchmark. Het blijkt dat de prestaties van de onderzochte *clouds* met betrekking tot hun rekenkracht laag zijn. Bovendien hebben we ook een initiële beoordeling van de consistentie in de prestaties van deze *clouds* gedaan, en die blijkt soms aanzienlijk te zijn. Dit leidde ons er toe om de variabiliteit in de prestaties dieper te onderzoeken in het volgende hoofdstuk. Ten slotte vergelijken we de prestaties en kosten van *clouds* met die van alternatieven voor wetenschappelijk rekenen zoals *grids* en parallelle computers. Onze resultaten laten zien dat terwijl de huidige *cloud*-diensten voor rekenwerk onvoldoende zijn voor grootschalig wetenschappelijk rekenen, ze een goed alternatief zijn voor onderzoekers die snel en tijdelijk toegang moeten hebben tot rekencapaciteit.

In Hoofdstuk 6 onderzoeken we de variabiliteit in de prestaties van productie-*clouds* met behulp van jarenlange *traces* met gegevens over de prestaties van twee populaire *clouds*: Amazon Web Services en Google App Engine. Het blijkt dat de prestaties van de onderzochte *cloud*-diensten jaarlijkse en dagelijkse patronen laten zien, maar dat er ook perioden met stabiele prestaties zijn. Tevens blijkt dat veel van deze diensten een hoge variatie in de maandelijkse mediane waarden laten zien, hetgeen duidt op een grote variabiliteit in prestaties over de tijd. Bovendien laten we door middel van op *traces* gebaseerde simulaties van verscheidene grootschalige gedistribueerde applicaties zien dat de invloed van de variabiliteit in prestaties sterk verschilt per applicatie-type.

In Hoofdstuk 7 ontwikkelen we een statistisch model voor ruimte-gecorreleerde storingen, dat wil zeggen, voor storingen die binnen korte tijd in verschillende systeemcomponenten optreden. Hierbij gebruiken we vijftien dataverzamelingen uit de *Failure Trace Archive*, een openbaar archief met de beschikbaarheids-*traces* van diverse parallelle en gedistribueerde systemen. In ons model voor storingen beschouwen we drie aspecten: het aankomstproces van groepen van storingen, de omvang van dergelijke groepen, en de tijdsduur dat een systeem buiten bedrijf is door een groep van storingen. Het blijkt dat de lognormale verdeling deze parameters goed beschrijft. Tevens blijkt dat in zeven van de vijftien *traces* die we onderzoeken, de ruimte-gecorreleerde storingen één van de hoofdoorzaken zijn van het buiten bedrijf zijn van systemen. Derhalve worden deze zeven *traces* beter door ons model verklaard dan door traditionele modellen, die aannemen dat de storingen in de individuele componenten van systemen onafhankelijk en identiek verdeeld zijn.

In Hoofdstuk 8 onderzoeken we het tijdsafhankelijke gedrag van storingen in diverse grootschalige gedistribueerde systemen met behulp van negentien dataverzamelingen in de *Failure Trace Archive*. Het blijkt dat in de meeste onderzochte systemen de frequentie van storingen erg variabel is, en dat storingen een sterk periodiek gedrag en tijdscorrelaties vertonen. Bovendien ontwikkelen we een model om de pieken in die frequentie te karakteriseren dat vier parameters heeft: de tijdsduur van die pieken, de tussenaankomsttijden van de storingen gedurende de pieken, de tijdsduur tussen pieken, en de duur van de storingen gedurende pieken. Het blijkt dat de perioden met de hoogste frequentie van storingen die door ons model worden verklaard, verantwoordelijk zijn voor een beduidend deel van de uitval van het systeem, hetgeen aangeeft dat de piekperioden in storingen speciale aandacht verdienen bij het ontwerp van foutbestendige gedistribueerde systemen. Ons model voor storingen kan worden gebruikt voor voorspellingen bij het nemen van beslissingen voor *scheduling* en *resource management* om de invloed van storingen op de wisselvalligheid van de prestaties in gedistribueerde systemen te verzachten.

Ten slotte presenteren we in Hoofdstuk 9 de conclusies van dit proefschrift en formuleren we verscheidene interessante onderzoeksvragen. Met verscheidene werklasten en gedistribueerde systemen hebben we empirisch laten zien hoe we de consistentie in prestaties van zulke systemen kunnen verbeteren. Bovendien heeft dit proefschrift ook geleid tot een beter begrip van de karakteristieken van storingen in gedistribueerde systemen, wat nodig is bij het ontwerp van systemen die de invloed van storingen op de consistentie van de prestaties kunnen afzwakken. Een uitbreiding van ons werk die van bijzonder belang is, is om te onderzoeken hoe we de consistentie in prestaties van commerciële *cloud*-infrastructuren kunnen verbeteren. Het onderzoek dat in dit proefschrift wordt gepresenteerd heeft al de eerste stappen in die richting gezet.

About the author

Nezih Yiğitbaşı was born in Istanbul, Turkey, on October 4, 1984. He received a BSc and an MSc degree in computer engineering both from Istanbul Technical University, Turkey, in 2006 and 2008, respectively. He worked as a software engineer in the telecommunications industry between 2006 and 2008, where he has developed his passion for building distributed systems. In September 2008, he joined the Parallel and Distributed Systems Group of Delft University of Technology to pursue his PhD degree. During summer 2010 he was an Intel/Carnegie Mellon University summer research fellow in Pittsburgh, PA, and he was a graduate technical intern at Intel Research Labs, Hillsboro, OR during summer 2011 and spring 2012. According to Google Scholar, as of December 2012 Nezih Yiğitbaşı's work has attracted over 350 citations with an h-index of 8. In his spare time, he enjoys reading, traveling, watching movies, and spending time with his family.

List of refereed publications

Journal papers

- A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, 2010

Conference papers

- M. N. Yigitbasi, K. Datta, N. Jain, and T. Willke. Energy efficient scheduling of mapreduce workloads on heterogeneous clusters. In *Proc. of the 2nd International Workshop on Green Computing Middleware*, pages 1:1–1:6, 2011

- Q. Zhu, M. N. Yigitbasi, and P. Pillai. Running interactive perception applications on opencirrus. In *Proc. of the 6th International Open Cirrus Summit*, pages 17–21, 2011

- M. N. Yigitbasi and D. H. J. Epema. Performance evaluation of overload control in multi-cluster grids. In *Proc. of the 12th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 173–180, 2011

- M. N. Yigitbasi, L. Mummert, P. Pillai, and D. H. J. Epema. Incremental placement of interactive perception applications. In *Proc. of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, pages 123–134, 2011

- A. Iosup, M. N. Yigitbasi, and D. H. J. Epema. On the performance variability of production cloud services. In *Proc. of the 11th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 104–113, 2011

- M. N. Yigitbasi and D. H. J. Epema. Static and dynamic overprovisioning strategies for performance consistency in grids. In *Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 145–152, 2010

- M. N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. H. J. Epema. Analysis and modeling of time-correlated failures in large-scale distributed systems. In *Proc. of the 11th IEEE/ACM International Conference on Grid Computing (GRID)*, pages 65–72, 2010

- M. Gallet, M. N. Yigitbasi, B. Javadi, D. Kondo, A. Iosup, and D. H. J. Epema. A model for space-correlated failures in large-scale distributed systems. In *Proc. of the 16th International Euro-Par Conference on Parallel Processing (EuroPar)*, pages 88–100, 2010

- O. O. Sonmez, M. N. Yigitbasi, S. Abrishami, A. Iosup, and D. H. J. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 49–60, 2010

- M. N. Yigitbasi and D. H. J. Epema. Overdimensioning for consistent performance in grids. In *Proc. of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 526–529, 2010

- S. Ostermann, A. Iosup, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Proc. of the 1st International Conference on Cloud Computing (CloudComp)*, pages 115–131, 2009

- O. O. Sonmez, M. N. Yigitbasi, A. Iosup, and D. H. J. Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proc. of Inter-*

national Symposium on High Performance Distributed Computing (HPDC), pages 111–120, 2009

- M. N. Yigitbasi, A. Iosup, D. H. J. Epema, and S. Ostermann. C-meter: A framework for performance analysis of computing clouds. In *Proc. of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 472–477, 2009