3-30-2012

Scheduling Medical Application Workloads on Virtualized Computing Systems

Javier Delgado
Florida International University, javier.delgado@fiu.edu

DOI: 10.25148/etd.FI12050239

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

SCHEDULING MEDICAL APPLICATION WORKLOADS ON VIRTUALIZED COMPUTING SYSTEMS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

by

Javier Delgado

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Javier Delgado, and entitled Scheduling Medical Application Workloads on Virtualized Computing Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Armando Barreto

Jean Andrian

S. Masoud Sadjadi, Co-Major Professor

Malek Adjouadi, Major Professor

Date of Defense: March 30, 2012

The dissertation of Javier Delgado is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

DEDICATION

To my family

ACKNOWLEDGMENTS

I would like to start by thanking my family for their support, and for teaching me many lessons along the way.

I would also like to thank my advisor, Malek Adjouadi, for supporting me through two graduate degrees, for his patience, and for his guidance. I would also like to thank my co-advisor, S. Masoud Sadjadi, for his guidance and for all the opportunities he afforded me. My thanks also go to my committee members Dr. Armando Barreto and Dr. Jean Andrian for their valuable input.

Several others have helped me throughout my graduate career, and there is not enough space on one page to mention them all. My collaborators at IBM, Liana Fong, Norman Bobroff, Yanbin Liu, and Setharami Seelam, without a doubt helped make me a better researcher. My international collaborators in many places: Gabriel Gazolla, Esteban Clua, Hector Duran, Rosa Badia, were very helpful to me both professionally and in helping me get around during me visits. All my friends and collaborators at CATE also provided major help along the way.

I appreciate greatly the support provided through the National Science Foundation under grants CNS-0959985, HRD-0833093, CNS-1042341, and OISE-0730065. I am also thankful for the clinical support provided by the Ware Foundation and the joint Neuro-Engineering Program between FIU and Miami Children's Hospital.

ABSTRACT OF THE DISSERTATION

SCHEDULING MEDICAL APPLICATION WORKLOADS ON VIRTUALIZED COMPUTING SYSTEMS

by

Javier Delgado

Florida International University, 2012

Miami, Florida

Professor Malek Adjouadi, Major Professor

This dissertation presents and evaluates a methodology for scheduling medical application workloads in virtualized computing environments. Such environments are being widely adopted by providers of "cloud computing" services. In the context of provisioning resources for medical applications, such environments allow users to deploy applications on distributed computing resources while keeping their data secure. Furthermore, higher level services that further abstract the infrastructure-related issues can be built on top of such infrastructures. For example, a medical imaging service can allow medical professionals to process their data in the cloud, easing them from the burden of having to deploy and manage these resources themselves.

In this work, we focus on issues related to scheduling scientific workloads on virtualized environments. We build upon the knowledge base of traditional parallel job scheduling to address the specific case of medical applications while harnessing the benefits afforded by virtualization technology. To this end, we provide the following contributions:

- An in-depth analysis of the execution characteristics of the target applications when run in virtualized environments.
- A performance prediction methodology applicable to the target environment.
- A scheduling algorithm that harnesses application knowledge and virtualization-related benefits to provide strong scheduling performance and quality of service guarantees.

In the process of addressing these pertinent issues for our target user base (i.e. medical professionals and researchers), we provide insight that benefits a large community of scientific application users in industry and academia.

Our execution time prediction and scheduling methodologies are implemented and evaluated on a real system running popular scientific applications. We find that we are able to predict the execution time of a number of these applications with an average error of 15%. Our scheduling methodology, which is tested with medical image processing workloads, is compared to that of two baseline scheduling solutions and we find that it outperforms them in terms of both the number of jobs processed and resource utilization by 20-30%, without violating any deadlines. We conclude that our solution is a viable approach to supporting the computational needs of medical users, even if the *cloud computing* paradigm is not widely adopted in its current form.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

INTRODUCTION

I.1 General Statement Of The Problem Area

I.1.1 Overview

Medical imaging workloads can consist of hundreds of individual images of one or more patients, requiring a large amount of computing resources to process. Professionals and researchers who work with these workloads face a complex problem: they need expensive computing infrastructure to process their data in a reasonable amount of time, but their usage pattern, which consists of long periods of little or no CPU requirements followed by occasional *bursts* of CPU demands, makes it difficult to justify the cost of purchasing and maintaining these resources. This problem can be addressed by sharing resources among many users, although this can create problems when multiple users need to access the resources at the same time.

A contemporary solution to this problem is *cloud computing*, as described in [1-3], in which massive cloud *providers* sell resources using a utility-based model. Users submit *jobs*, consisting of input problem(s) to be processed by a given application, to these providers and only pay for the computing resources required for the given job. Providers typically use some cost model to bill users. The cost model may be monetary or based on some other established means of controlling and/or rationing each user's resource consumption. For example, academic users may access nationally funded computing infrastructure such as the Teragrid [4] by applying for *credits* (e.g. compute units).

Advances in virtualization technology [5-6] allow users to deploy custom application environments in minutes and without the upfront and maintenance costs associated with owning the machines. In this context, users can be end users paying to lease the resources directly or providers of higher level services. The cloud usage model is often divided into three main levels of abstraction: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). In IaaS, users deploy arbitrary applications on machines residing somewhere "in the cloud."

As the lowest level offering, IaaS is most applicable to users who need to create their own customized environment without worrying about the initial costs of purchasing equipment or the ongoing costs of fixing hardware and cooling the systems. For example, the Amazon Elastic Compute Cloud (EC2) allows users to lease their machines in increments of an hour according to a pricing model. PaaS solutions provide a programming environment on which users can deploy higher level applications. For example, *Google App Engine* [7] provides a framework for users to deploy web applications. Finally, SaaS providers offer specific services for a target demographic. Going back to medical imaging, a medical image processing service would allow users to upload data sets and apply different algorithms to them. With virtualization technology, a medical image processing service can be deployed in a relatively small amount of time.

In order for the cloud paradigm to be adopted, certain criteria must be met. From a user's perspective, it is necessary to satisfy certain quality of service (QoS) objectives. For example, medical specialists work on tight schedules, so it is necessary to have data available for them by a given time. On the other hand, providers need to justify their investment, which implies that they need to keep their resources highly utilized while minimizing individual response times. Balancing utilization and response time, while satisfying deadlines, necessitates the use of non-trivial job scheduling techniques. In order to satisfy deadlines, the execution times of submitted jobs must be known either by the provider or the user. Since the provider's resources may be heterogeneous, dynamic, and/or confidential, the burden of predicting performance generally lies on the provider.

Within this context, this dissertation addresses the issue of predicting performance and scheduling jobs such that deadlines are met, which we consider a prerequisite for a *medical imaging as a service* offering. We also explore scenarios in which multiple kinds of scientific applications are provided as a service, while running on the same set of hardware. In the process, we explore related issues such as the impact of virtualization on application execution.

I.1.2 State of the Art

A shift towards the cloud paradigm, in which most of the computationally intensive processing required by a large set of users is performed at specialized data centers, is currently receiving much attention in industry and academia. This paradigm can be more cost effective in terms of hardware and operational costs. According to *Google Trends*, the term "cloud computing" has reached "disruptive" status [8]. This usage model is particularly attractive for users who require demanding, but relatively infrequent access to resources, such as medical professionals and meteorologists.

There are currently several commercial providers of cloud resources. While medical professionals can use their services, the fact that resources are provisioned on a best-effort basis implies that they are not guaranteed that their jobs will complete before a certain deadline. The fact that their workloads must be processed in a certain amount of time can deter them from using current cloud offerings, unless the provider can guarantee that their jobs will finish before a given time.

I.1.3 Parallelization

Parallelization of computationally intensive software is becoming an unavoidable requirement. For decades, the performance of individual CPUs increased at a steady rate; this *scaling up* of compute power enabled *soft scaling* of compute-intensive software (i.e. existing software would become faster over time by virtue of increased CPU clock frequency). The "power wall" reached early on in the 21st century motivated the development of multicore processors capable of running multiple application processes simultaneously, in turn requiring software to *scale out* in order to achieve faster execution time.

Different methods of parallelization are possible, depending on the computational requirements and computational characteristics of the problem addressed. For problems that process a large amount of data, the single-program, multiple-data (SPMD) approach is the most popular. In this approach, multiple workers execute the same program on subsets of the input data. Different approaches can be used to parallelize a given problem using SPMD, each with its own benefits and drawbacks, and the choice to be made depends on the problem's *coupling,* size, and available development time and/or manpower.

Coupling, in this context, refers to how interrelated the processing of subsets of data is. For example, in a medical image, a filtering algorithm in which neighboring pixels affect the output value of a particular pixel would be considered *tightly coupled*. Performing the same algorithm on separate images would be considered *loosely coupled*.

One parallelization approach is to implement a parallel algorithm for a given problem; this has several benefits: (1) it works well with tightly coupled programs; and (2) there is greater potential to speed up the program's execution as long as the algorithm itself scales well on the hardware it runs on. There are also several drawbacks: (1) the code may not be implementable in parallel, hence requiring extensive rewriting; and (2) some programs, particularly tightly coupled ones, incur some overhead due to sharing of data during execution, so resources may be wasted during execution. Recall that scalability is dictated by Moore's Law [9]. The alternative (i.e. loosely coupled) approach consists of sending unrelated subsets of the data to different CPUs. This approach is easier, unless parallelization of the algorithm is trivial. The main challenge with this approach is ensuring that all CPUs remain busy. Even though having tasks finish at different times on a loosely coupled program does not affect its results, the results of many loosely coupled problems are only useful after all the data have been processed. For example, a medical professional submitting a batch of medical images to be processed needs to obtain a statistical summary of all results to make further prognosis.

I.1.4 The Role of Virtualization

Cloud computing relies heavily on resource virtualization technology, due to the following key benefits:

Data Isolation: An issue with allowing multiple users to share physical machines is that their data can be compromised. This problem is particularly undesirable when the workloads being processed contain medical data. Operating system access controls can address this issue, but simply deploying separate virtual machines (VMs) is easier and less error prone. Also, by giving each user a VM they each get full control of what software they can install on the system. In contrast, using regular operating system access control mechanisms precludes users from most administrative powers, such as the ability to install software.

Resource Isolation: Virtualization technology provides a finer grain of control of each VM's CPU and memory allocation. There are multiple cases in which this is useful in the context of job scheduling. One is the ability to provide higher allocation to higher-priority jobs. Another is to allow jobs that arrive when all physical machines are occupied running long-duration jobs to execute immediately, possibly sharing the CPU with other jobs, rather than waiting in a queue. For example, consider a scenario in which a large, long-duration parallel job is using all available physical machines and a user wants to submit a small, short-duration job. If only a single job is allowed per physical machine, the short job has to wait for the large job to finish, resulting in poor response time.

Resource isolation can be harnessed to strategically collocate jobs with complementary resource requirements, resulting in better resource utilization. For example, some parallel applications spend a significant portion of their execution performing I/O, which is computationally inexpensive. By combining multiple parallel jobs with this behavior, CPU utilization can be increased.

Historical workload data has shown that scientific workloads have a *bursty* behavior, i.e. occasional periods in which a relatively large number of jobs arrive [10-11]. Collocating jobs can be beneficial for this as well. As a case in point, gang scheduling [12] has demonstrated improved scheduling performance, although it can lead to resource fragmentation if the total number of tasks is not a multiple of the number of available processors. Similarly, in [13] and [14], the authors simulate a scheduling algorithm that leverages CPU sharing, and average job response times improved by a factor of 8 or more, depending on specific optimizations, compared to only running one job at a time per machine using the popular EASY job scheduling algorithm [15].

On the other hand, providing resource isolation requires additional computational processing that can significantly impact application performance. Scientific applications that perform a lot of I/O are particularly susceptible to this overhead. It is important to know the extent of this overhead to decide whether the computational cost is worth the benefits afforded by virtualization for these particular applications.

Migration: Another key feature enabled by virtualization technology is the ability to easily migrate virtual machines across physical machines. This allows applications to be moved around or across data centers for improved load balancing. This feature can be used to retroactively improve job placement when a job scheduling algorithm leaves the system in a sub-optimal state.

I.2 Research Problem

The central goal of this dissertation is to develop a scheduling methodology that addresses the time-sensitive nature of the jobs that are typically submitted by medical professionals, while balancing resource utilization and job response time. To this end, multiple issues need to be addressed. First, it is necessary to have a real-world understanding of the behavior of the applications that these users use, including how the applications respond to different run time conditions and the effect of running them in virtualized systems. In order to satisfy deadlines, an execution time prediction methodology is required. Finally, a scheduling methodology that uses this prediction model to complement theoretical heuristics for optimized job scheduling must be implemented.

I.3 Significance of the Study

As mentioned earlier, the idea of running applications “in the cloud” has generated a lot of interest for academics as well as end users. The scientific community in particular is experiencing a period of expanding growth in the amount of data to be processed. For example, in [16], the authors cite that “the number of genomes has increased approximately 12 fold over the last 5 years.” The amount of data generated by medical acquisition devices is also growing, in turn requiring more computational power to harness the additional data for improved medical diagnosis. The authors of [16] find that using clouds for certain genomics applications is cost effective. Another benefit the cloud may achieve is the “democratization” of science [17], which refers to the fact that a relatively small percentage of institutions have the means to perform large scale scientific experiments due to the large entry costs associated with purchasing and maintaining supercomputers. Cloud providers promise that “anyone with a credit card” can access these huge data centers to run their experiments.

The findings presented in this dissertation benefit a large community that may not necessarily be interested in cloud computing. For example, the benefits of virtualization that were mentioned provide a practical way of improving job scheduling and resource utilization. In terms of resource sharing, it is common for medical establishments to collaborate with each other and even with universities. Sharing compute resources can help reduce the costs of processing medical workloads, in turn reducing overall operating costs and the cost of healthcare. Our studies on performance analysis and modeling provide important insights about the behavior of popular scientific applications.

I.4 Structure of the Research

We separate the work presented in this dissertation into three interrelated contributions. First is an in-depth analysis of the performance of several applications in virtualized and non-virtualized environments. Second is the development of performance prediction methodology. Third is the development of the job scheduling and resource management methodology.

We start by discussing previous and related work in Chapter II. In Chapter III, we summarize our observations from an in-depth look at the performance characteristics of different scientific applications. These observations are based on actual application executions on a compute cluster configured similar to the machines used in cloud data centers. We focus on medical applications, but since in-depth performance analyses are beneficial to a wide audience, we broaden the scope of our analysis by also studying some popular CPU-intensive fluid-dynamics applications. This chapter describes the computing infrastructure used for this study, how it was configured, and the implications and benefits of virtualization. We then give an overview of the applications used, including their purpose, and some general algorithm- and implementation-related details. This section serves two purposes. First, it assists us in implementing the performance prediction methodology described in the following section. Second, we measure the magnitude of the performance impact of virtualization. Studies in the literature have given mixed results on this and we attempt to fill this gap with more in-depth analysis and new insights.

Based on the observations made in Chapter III, we implement and describe our performance prediction methodology in Chapter IV. The prediction methodology is broken down into prediction of computation time (i.e. how will different problem sizes affect computation time?), prediction of scalability (i.e. how do different system configurations affect execution time?) and prediction of execution prolongation due to resource sharing (i.e. in multitenant virtualized environments when the CPU is shared by multiple VMs). Both models rely on historical execution data, which allows us to use mathematical predictors that provide fast predictions, such that scheduling decisions can be performed in real time. The accuracy of the models is tested with different applications and the results are presented.

The performance prediction methodology can then be applied to the scheduler, which constitutes the scheduling methodology described in Chapter V. The scheduler assigns jobs to machines. In addition to the scheduler, we implement a resource manager to track the utilization of all resources in the cluster and a job monitor to ensure that jobs are getting enough resources to satisfy the scheduling objectives. The scheduling methodology is evaluated empirically by comparing to currently available schedulers.

Chapter VI concludes the dissertation with a summary of the results, some discussion, and future work.

CHAPTER II

LITERATURE REVIEW

In this chapter, we discuss related work. We break this down into four sections, one for each of our contributions presented in chapters III-V and one on work related to the main goal of provisioning medical imaging services in cloud-like environments.

II.1 Performance of Scientific Applications on Virtualized Systems

The use of *virtualized data centers* has proven successful for commercial cloud providers, particularly for provisioning resources to web application providers, as evidenced by the list of users [18] available from Amazon, who is currently the most well known public cloud service provider. Noting that virtualization adds computational overhead, as we will describe in Chapter III, others have studied the performance of virtualized applications [5], [19-24]; some of them specifically studied scientific applications [20-25]. For example, in [20], the authors evaluate the performance impact of *Xen* on different parallel application benchmarks. Among their contributions, they analyze the performance penalty of *Xen* on a few applications and observe a virtualization overhead of up to 20%. On the other hand, the authors of [21] do not find significant overhead on similar applications.

We address discrepancies by adding more in-depth insight compared to existing work, particularly on how communication characteristics and problem size impact application performance when running on virtualized systems instead of “bare metal.” To this end, we provide quantitative data about virtualization overhead for different execution scenarios, including different applications, problem sizes, numbers of nodes, and processes per node. Using the discord in the previous paragraph as a specific example of how our findings are beneficial, the authors of [21] concluded that parallel applications whose tasks communicate data frequently experience negligible virtualization overhead. We noticed that the jobs they studied had computation rates below 50% and our studies demonstrate that virtualization overhead is negated when the computation rate of a job is low since virtualization-related overhead processing can be performed while the physical communication of data is taking place. On the other hand, virtualization is significant with moderate computation rates. The results provided in [25] re-enforce this conclusion, as evidenced by the fact that using 8 nodes results in a large execution time penalty with *Xen*, while using 32 nodes result in a minor penalty.

In [23], they use some popular parallel benchmarks to evaluate the overhead of Xen. They use more low-level profiling details such as the number of cache misses. Their goal was to be able to predict virtualized performance by application, but they find that this is not possible with their approach. We will show that predicting virtualization impact is possible using a model that is aware of system configuration and application input.

II.2 Performance Prediction

II.2.1 General Performance Prediction Studies

Performance prediction has a long history, but new uses have emerged, particularly for HPC applications. Early work was done for the purpose of architecture design [26]. Current uses include resource allocation and capacity planning [27-28]. In [28], the authors implement a performance prediction framework, but it relies on users knowing how long they need the computing resources for. Most of the work on capacity planning targets web application environments, which have different execution characteristics than scientific applications. Queuing networks are often used for modeling these environments. In contrast, for scheduling scientific applications, the problem is predicting how quickly a given computational problem can be completed with a given amount of resources.

The approaches for predicting execution time of scientific applications include methods that use popular analysis and optimization tools such as *Dimemas* and *Vampir* [29], tasks scheduling [30], and statistics [31-32]. Other works focus on system specific approaches [33]. In [34], the authors describe the use of the *Dimemas* tool to perform prediction of the execution behavior of message passing applications. Their results are good, but the tools used require intimate knowledge about the execution domain and special trace files need to be generated for each application. It is possible to dynamically link the trace-file-generation library with most applications, but in some cases a complete recompilation of the application being profiled is necessary.

Like us, the authors of [32] use a prediction model based on linear regression. Their approach relies on user input, while we attempt to provide a prediction without any user input required, although is information could help increase prediction accuracy.

The effort described in [36] allows for cross-platform performance prediction of parallel applications. The prediction is achieved by combining the application's performance in a reference system and the relative performance between the two systems derived from a partial execution on the target platform. The source code of an application is analyzed to identify the major time step loops and the source code is then modified to include the API for the partial execution measurements. A key drawback of this effort in the context of what our prediction methodology tries to achieve is that since run-time profiling is not incorporated, dynamic changes in the system environment unavoidably lead to inaccurate predictions.

The authors of [37] present a similar approach in trying to model the execution time of an application on a particular set of resources for use in meta-scheduling decisions in a grid environment as part of the *Ianos* project. Their model includes several application-specific parameters and characteristics, which can be done with our model, but is not necessarily a prerequisite for good results. A large number of parameters may need to be input by the user in regards to the application and target architecture, which can result in longer times for deployment.

Reference [37] provides an approach similar to ours. The authors predict application scalability on up to 1024 processors. They estimate communication and computation times separately. Our approach models execution as a function of communication time and computation time. Another fundamental difference with their work is that they use PMPI, which works at the source code level, to instrument the communication. Our approach is to profile externally using operating system facilities. Unlike our experiments, theirs keep the parallelism level constant and they do not show their model's cross-platform prediction accuracy, since they test on a single system.

II.2.2 Virtualization-Related Studies

The authors of [38] studied how accurately Xen enforces the CPU utilization constraints and found that it does so effectively when VMs run CPU-bound applications. Our experiments confirmed this, but we noticed that the share of CPU given to different jobs is not distributed as expected when one of them creates virtualization overhead.

There has been work on predicting the execution time of applications run in virtualized environments. Wood [19] provided a model for predicting the virtualization penalty of web workload benchmarks, given historic run time information in a non-virtualized environment. Our work is focused on analyzing virtualization overhead for scientific applications in terms of their communication requirements. A comprehensive virtualization overhead prediction model would complement our work, although we find that for the jobs we experiment with, simply monitoring the execution host and measuring overhead works well due to their iterative steady state execution behavior. We also look into the effects of CPU sharing and its effect on execution time. We devise a model for predicting the execution time of tightly-coupled jobs when they share the CPU with other jobs, a topic that providers need to study in order to maximize CPU utilization while satisfying service level agreement (SLA) requirements.

Resource requirements prediction for capacity planning for multi-tenant environments that handle web workloads has also been addressed in [39-40]. Our work on multi-tenancy is focused on the prediction of execution time for making on-the-fly job scheduling decisions. This is related to parallel job multiprogramming, in which multiple parallel jobs are executed on each node of a compute cluster, which was studied in the past [41-43] (albeit without the fine-grained resource control allowed by VMs). In [41], the authors compared different strategies for load balancing among multiprogrammed clusters and found that gang scheduling provides the best CPU utilization compared to using blocking I/O during non-working cycles. Gang scheduling involves using predefined time quanta rather than voluntary blocking, which can be advantageous since it ensures threads of the same job are active at the same time, but reduces the ability to mask I/O cycles. The VM model of our work is similar to the voluntary blocking model, but we suspect that the smaller cache sizes and larger context switch overhead of processors used for those earlier studies favored the gang scheduling model. Another benefit of the VM model is that it does not require any major modifications, whereas gang scheduling requires mechanisms to ensure all tasks of a job enter the working state at the same time.

Our performance models complement simulation-based studies. For example, Stillwell [13-14] evaluated the use of dynamic fractional scheduling via simulation using real-world traces, while our work provides empirical results and a mathematical model to align such simulations with real-world data.

Having realized the potential benefits of clouds for scientists, others have examined the cost-effectiveness and performance of using commercial clouds for science [44]. One of the issues they have found is inconsistency in both resource acquisition times and application scalability. This could be due to different reasons, e.g. users may have to wait in a queue during peak times or there may be over-commitment of resources (e.g. when time sharing the CPU). This finding confirms the need for implementing mechanisms for deadline satisfaction. Unlike our work, theirs looks at using the cloud from a user's perspective, while we look at it from a provider's perspective.

There have been many efforts on other aspects of enhancing virtualization for scientific computing. In [30] they enhance an MPI implementation for improved intra-node (shared) memory performance. In [22], authors implemented a methodology that achieves near-native performance on Xen using VMM-bypass, but it requires *Infiniband*-based network infrastructure.

II.2.3 Parallel Job Scheduling

Parallel job scheduling, which entails scheduling tasks requiring some amount of computational processing on a fixed set of resources, has been studied for several years. Feitelson published a survey of the state of the art in 2008 [45]. The problem is essentially to minimize response times. Due to the high cost of computing systems, it is necessary to compromise between response time and system utilization. A major problem is that job arrivals are known to be "bursty" [10-11], i.e. there are periods of low utilization followed by periods of high utilization. Overprovisioning data centers to accommodate occasional bursts results in low return on investment, so it is preferable to focus on using job scheduling algorithms and heuristics to ensure satisfactory job response times with the available resources.

Recently, the interest in scheduling on virtual machines has surfaced. For example, the authors of [13-14] explore dynamic fractional scheduling, in which fractions of resources are given to jobs. Although they use simulations, VMs make this kind of scheduling possible, as our results show. However, their work did not address the issue of satisfying deadlines, which was central to our work.

Scheduling of *bags-of-tasks* workloads, which consist of large groups of independent tasks, has seen particular interest. In [46], the authors present solutions for solving this problem in the non-clairvoyant case and give insight into solving it in the clairvoyant case.

Their studies are not applicable to ours since they assume that all jobs arrive at the same time.

In terms of cloud computing, the issue of SLA satisfaction has received a lot of attention. In [47], they describe a monitoring framework for providing quality-of-service (QoS) guarantees, but for multimedia applications whose QoS constraints are different from scientific applications. Relatedly, in [48] they describe a framework for scheduling jobs on VMs. They also discuss the issue of resource monitoring. However, no performance studies are performed.

Gang scheduling [41-42] is closely related to VM-based scheduling in the sense that the processor is over-provisioned. However, the resource allocation controls enabled by virtualization add more flexibility (and complexity) to the scheduling algorithm. In our work, we harness this flexibility in order to satisfy deadlines at the expense of additional scheduling complexity.

Scheduling heuristics such as backfill [15] use execution time predictions to improve scheduling performance. Traditionally, job submitters were required to provide this information. Apart from being a burden on them, with cloud computing they may not even know the compute power they are getting since the details of the infrastructure are abstracted from them. Also, the predicted times have been shown to be inaccurate [49]. Hence, system generated predictions are necessary and are what we use in our work. In [50], the authors demonstrate that system-generated predictions can improve scheduling performance. In [10], the authors also find that, specifically when scheduling jobs in compute grids, system generated predictions can help scheduling performance. Our work also demonstrates the benefits of performance prediction, but unlike their approaches, which rely solely on past execution data for job submitters, we extrapolate for execution times with different types of resources and parallelism levels, in order to accommodate deadlines and to account for the fact that CPU power might be shared by multiple virtual machines.

II.3 Medical Imaging as a Service

In [51], the authors describe the implementation of a service that uses the Aneka [52] framework to processes electrocardiogram (ECG) data in real time. This work differs from ours in that it is focused on using existing cloud frameworks and elasticity methods to provide an ECG processing service. They do not address the issue of satisfying deadlines nor provide analysis on scheduling objectives. Our work is focused on these scheduling issues and our findings can be applied to different medical applications, even though we test it with only one well-known set of such applications. As the authors of [10] found, the resource acquisition times for EC2 are not dependable, which implies the need for deadline satisfaction, which we address.

The work summarized in [53] is also closely related. The authors analyze the performance of public and private cloud infrastructure for processing medical image segmentation workloads. They harness the *CometCloud* [54] framework to manage the executions based on cost and time constraints. Although their implementation optimizes timeliness upon user request, they do not have mechanisms for deadline satisfaction. Rather, it just allocates enough nodes to finish jobs as quickly as possible. They do not compare the scheduling performance of different algorithms either.

We consider it necessary to support hard deadlines, rather than using best-effort scheduling, to satisfy the needs of the medical community. Therefore, we focus on this specific issue and at the same time analyze different scheduling algorithms in order to maximize system utilization while also considering job response times. Both of the works cited in this subsection can harness the scheduling methodology that we implemented to provide a better experience to users of these technologies.

CHAPTER III

PERFORMANCE ANALYSIS

The purpose of this chapter is to provide a detailed analysis of the performance characteristics of different scientific applications when run on virtualized and non-virtualized computing resources. In-depth analysis is provided with emphasis on medical applications. Some interesting findings were found with other applications, which we describe in this chapter for the benefit of a wider audience. This analysis can be used for evaluating the performance impact of the virtualization technology used, for capacity planning, to observe the scalability characteristics of different parallel applications, and/or to assist with job scheduling. In the next chapter, we apply the knowledge gained in this section to a performance prediction methodology, which is later used in our scheduling framework. Due to the important role of virtualization in the cloud paradigm, we provide an in-depth analysis of the performance impact of virtualization on medical applications as well as on some scientific benchmarks that have execution behavior representative of a wide range of parallel applications.

III.1 Infrastructure Used

We use a 16-node compute cluster at the Center for Advanced Technology and Education (CATE). We refer to this cluster as *Mind*. Each node in the cluster contains 2 single core Intel Xeon processors with *hyperthreading* technology rated at 3.6GHz; they are based on the *Netburst* CPU architecture. Each node has at least 2 GB of main memory. The nodes are connected using a 1-gigabit Ethernet interconnection. The operating system is the *CentOS* Linux distribution*,* version 5.3, included in version 5.2 of the *Rocks* cluster distribution [24]. For comparing virtualized and non-virtualized experiments, half of the physical nodes were configured as *compute* nodes and the other half were configured as *vm-container* nodes using the *Rocks Xen* roll, which deploys selected worker nodes with the *Xen* virtualization software, version 3.0.3 [5]. VM images used for the virtualized experiments contain the same *CentOS* distribution, including the same kernel version. VMs are deployed using *OpenNebula* [55].

For these experiments, we allot 2 CPU cores to each VM and execute up to 2 MPI processes per node. Unless otherwise stated, each VM can use the full processing power of each allotted processor. In the *vm-container* nodes, the *dom0* (i.e. hypervisor) VM is allowed to use the virtual processors (i.e. *hyperthreads*). We noticed performance degradation when allowing *Xen* to dynamically change the virtual to physical CPU mappings, so each virtual CPU in a VM is pinned to a specific physical CPU.

All software and guest VM images were installed on a shared file system, which is hosted on the master node of the cluster. A virtual network was created using *OpenNebula* to link the VMs. Xen's standard network bridging configuration was used.

III.2 Benefits and Drawbacks of Virtualization

Virtualization is integral to cloud computing. Many benefits of virtualization were highlighted in section I.1.4. In regards to the resource utilization benefits of virtualization, although scientific applications are computationally intensive, many transmit large amounts of I/O during execution, resulting in unused CPU cycles. By sharing the CPU among multiple such jobs, the I/O cycles of one job can be *masked* by another. Another benefit virtualization solutions such as Xen and VMWare offer is the ability to precisely control the CPU allocation of each VM. Our scheduling methodology harnesses this functionality to ensure jobs are allotted enough CPU to meet their deadlines.

A drawback with virtualization is that it adds computational overhead [5,20-25]. For scientific applications in particular, virtualization has been shown to result in a significant performance penalty. The general consensus is that loosely coupled parallel applications (i.e. those whose parallel tasks do not communicate throughout their execution) are not significantly affected by this. Tightly coupled applications, whose workers must exchange data throughout their execution, do suffer a performance penalty, although the magnitude of the penalty reported in different studies has varied. Besides this drawback, multi-tenancy complicates deadline satisfaction since it is harder to predict execution time when only a portion of compute resources can be used. We address this with our prediction model described in the next chapter.

III.3 Executing Scientific Jobs on Virtualized and Non-virtualized Systems

To aid in our analysis, and for later cross reference in the development of our performance prediction model, we now describe the behavior of scientific applications, as observed by running them on *Mind*. First, we describe their general execution characteristics that apply in any environment. Then we describe details specific to their execution in virtualized environments, particularly when time sharing the CPU. Some issues regarding the modeling of application execution in these environments are also discussed.

III.3.1 Characteristics of Parallel Applications

The execution behavior of a parallel job naturally depends on its implementation and run time configuration. For this work, we separate parallel applications into two classes. First, there are *loosely coupled* parallel applications, in which all workers process unrelated data sets or separate portions of a single data set; there is little or no inter-process communication during execution of these applications (e.g. communication only occurs at the beginning and/or end of the job's execution.) The second type is the *tightly coupled* parallel application, in which the workers of the application communicate frequently. Tightly coupled applications can be further characterized as either *fine grained* or *coarse grained*, depending on how often the workers communicate. Tightly coupled parallel applications are iterative in nature. Each iteration consists of a computation phase, where a subset of the problem is solved, followed by a communication phase, where data is exchanged among workers. The point between the computation and communication phases is called the synchronization point. Once a worker reaches the synchronization point, it cannot proceed until it receives data for the next computation phase. As a result, the worker with the highest computational load limits the execution rate of the entire job. The size of the data transferred during the communication cycles depends on the application, the problem size, and the number of workers. Using commodity network infrastructure, a 60-80% duty cycle is common for tightly coupled jobs, depending on the job's problem size and on the granularity of communication. Some of the jobs we test in our experiments are as low as 40%.

The fact that communications consume a significant portion of the execution time of these jobs implies that CPU cycles are being wasted, unless other jobs time share the CPU and hence *mask* the otherwise idle communication cycles. For example, we took a sample of the first 800 jobs with CPU time requirements greater than 10 seconds from the Cornell Theory Center workload trace from the Parallel Workloads Archive [56] and calculated the mean ratio of CPU time to wall clock time for the jobs. We found that the mean CPU utilization was 84%. We ran a similar test with the San Diego State University trace, this time analyzing the first 2000 jobs requiring at least 100 seconds of CPU time and found the mean CPU utilization to be 79%. By overcommitting the CPU, it is possible to approach 100% utilization, resulting in a better return on investment. On the other hand, it is necessary to account for context switching penalty, and if jobs have deadlines, they must be taken into account. We analyze the context switching overhead of some applications later in this chapter and we address the deadline issue in Chapter V.

III.3.2 Application Execution on Virtualized Platforms

Our work focuses on Xen [5], although products like KVM [57] and VMWare [6] have similar functionality. In the case of Xen, the hypervisor is a thin layer and the control of VMs is accessed through a privileged virtual machine known as *Domain 0.* Guest VMs are referred to as *user* domains (or *Domain U*). I/O functions are handled by a *driver* domain, which allows regular device drivers to be used and ensures I/O is properly isolated. The drawback is the additional CPU time required for virtualization overhead that occurs during I/O operations to process individual sets of data in order to route them to the correct VM. This way, multi-tenant environments can be supported while ensuring that VMs cannot access each other's data.

Collocated virtual machines sharing a CPU are scheduled similar to processes in a multiprogrammed operating system. That is, the hypervisor's scheduler periodically monitors VMs' states and allocates a physical CPU to whichever VM is requesting it. When more than one VM is requesting it, it is assigned to the one with the highest priority for a given time quantum. In the version of Xen used for our experiments, a proportional share scheduler known as the Credit Scheduler is used. The scheduler *ticks* every 10ms and each tick is accounted to whichever VM is using the CPU. The default time quantum for each VM is set at 30ms. Further details about this scheduler can be found in [58-59].

There are two states in which each VM can be: working and non-working. In the working state, the VM is consuming CPU cycles. In the non-working state it is not, either because it does not have work to do, is waiting for I/O, or because it has a negative credit balance and a collocated VM is in need of the CPU. VMs build credits as long as they are waiting for the CPU. The time a VM spends in each state depends on the resource consumption characteristics of the processes it is running and its priority; the latter is based on its scheduling parameters. In Xen, these parameters consist of a *weight* and a *cap*. The *weight* parameter determines the share of processing power allotted to a VM when it competes with others. For example, if a physical machine has 3 VMs with weights of 1, 1, and 2, their shares are ¼, ¼, and ½, respectively. The share value ranges from 0 to 1. The *cap* is a hard limit on the percentage of the CPU capacity that a VM can use. Even if there are free CPU cycles available, the VM cannot exceed its *cap*. On the other hand, a VM can obtain more than its share if there is excess capacity available. In other words, imposing a cap turns the Credit Scheduler into a non-work-conserving scheduler. The *cap* ranges in value from 0 to 100. We refer to a VM's allocation after accounting for the *caps* of all VMs sharing the same CPU as the *net share.*

When two VMs running CPU-bound jobs share a CPU, each will proceed at a rate proportional to its net share. Since VMs accumulate credits while they are not in the working state, they each eventually get their fair share as long as they do not remain idle for longer than the scheduler's *reset* period [25-26], which is unlikely since small amounts of CPU are required while performing message passing. I/O related virtualization overhead is charged to the VM it pertains to.

III.3.3 Modeling Application Execution on Shared CPU VMs

Since tightly coupled parallel applications tend to have iterative steady-state execution behavior, their execution time under different CPU allocations can be estimated if their computation and communication requirements are known. The scheduling behavior outlined in the previous subsection dictates their execution rate. Since all tasks of a given tightly coupled application proceed in lockstep, the one with the lowest CPU share limits the execution rate of the others. Also, the computational load is not necessarily balanced among all the workers of the job, so that of each worker must be known for optimal prediction accuracy.

Our approach to modeling execution time relies on knowing the computation and communication requirements of the application in question. We address how these can be determined in the next chapter. If running in a VM, the virtualization overhead must be measured separately since it is external to the VM itself. Xen provides user space tools to obtain CPU accounting information. Using this information, the virtualization overhead can be accurately determined, since it is added to the Domain-0 CPU consumption statistics (as well as the user domain CPU consumption statistics). Subtracting the overall execution time from the computation time in the dedicated CPU case yields the I/O time.

III.4 Applications Used

III.4.1 Medical Image Processing

The medical image processing domain consists of applying complex, computationally intensive algorithms to large sets of images. As we will confirm, these applications work well in virtualized environments since there is little or no interprocess communication when processing images from separate studies on different resources.

Our main focus will be on brain magnetic resonance image (MRI) processing. All of the algorithms used are implemented in the FMRIB Software Library (FSL) [60]. MRI studies can be dichotomized into structural and functional studies [61]. Structural studies deal with the variability between adjacent brain tissues; we employ a segmentation tool called FMRIB Automated Segmentation Tool (FAST) [62] for these experiments. The algorithm it implements segments the basic tissues of the brain: gray matter, white matter, and cerebral spinal fluid. FAST is an iterative algorithm that depends on the within tissue variability while addressing problems arising from image noise, head motion artifacts and inhomogeneity in the magnetic field, all of which affect the performance and speed of the algorithm. On the other hand, functional studies deal with the temporal differences in the activation of neurons. We test an exploratory algorithm called Multivariate Exploratory Linear Optimized Decomposition into Independent Components (MELODIC), which consists of a pipeline of algorithms that can be summarized to motion artifacts correction, image registration [63] to high resolution MRI and to a standard brain image, and finally probabilistic independent component analysis [64].

Data from 66 patients from Miami Children's Hospital (MCH), the Children's Healthcare of Atlanta (CHOA), the Children's Hospital of Philadelphia (CHOP), the Children's National Medical Center (CNMC), and BC Children's Hospital (BCCH) were used for these experiments. FAST and MELODIC jobs have similar computational requirements for similarly-sized input data, so we only show results with a subset of the input data. Specifically, we use fMRI datasets from 20 patients, where each dataset (which consists of data from one medical study) consists of 14 slices of 64x64-pixel images and 150 time points in total. Each of the datasets also contains a 256x256 static MRI image. The algorithms are applied to each patient separately, so the algorithms themselves do not need to be implemented in parallel; instead, they are submitted as a *bag of tasks*, where each task consists of applying the segmentation or registration algorithm to a single dataset.

III.4.2 Tightly Coupled Parallel Applications

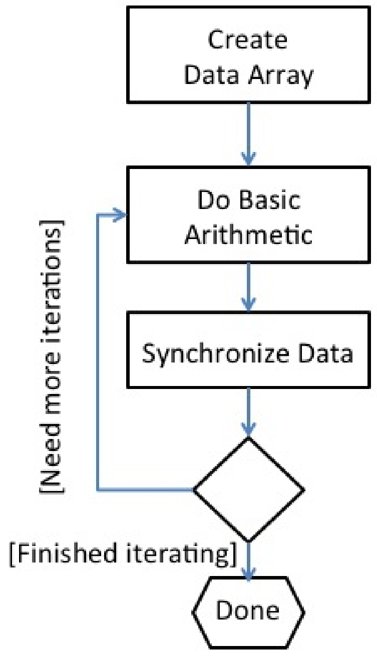
While our main focus is on medical applications, we also measure the performance impact of Xen on more general applications. The applications employed address a kind of parallel application that was not covered with the other applications: the tightly coupled parallel application. We later perform some experiments in which we schedule mixed parallel workloads and observe the performance of our scheduling algorithm. This gives us deeper insight on the performance of different kinds of tightly coupled parallel codes, which may be useful for the development of new medical imaging algorithms.

The first benchmark used is a simple tightly coupled application we call *Compcomm*. Its algorithm can be seen in Figure 1. It consists of iteratively performing a set number of arithmetic operations followed by data exchange between 2 nodes. Barriers are used after each iteration to measure performance fluctuations between iterations. The arithmetic consists of multiplying three integers. The *send* and *receive* calls use MPI functions for performing a blocking send and a blocking receive of an array of floating point variables created when the benchmark is started. The algorithm is ideal in the sense that during each iteration the CPU load is perfectly balanced and non-changing, and the size of the communications is always the same. By varying the number of computations, we can explore the effects of different computation to communication ratios on virtualization overhead. We also vary the message size and analyze its effects on virtualization overhead.

We then employ NASA's Numerical Aerodynamic Simulation Parallel Benchmarks (NPB). The NPB suite contains several benchmarks. Three of them replicate the computation and communications patterns of Computational Fluid Dynamics (CFD) and computational aerodynamics applications [65]. Specifically, they provide different kernels for solving Navier-Stokes parallel differential equations on a spatial grid or *mesh* of a given size. These algorithms iteratively perform the same solving routine until converging to a solution, or until a set number of time steps are reached.

```
data = populate_float_array()
for(i = 0 : NUM_ITERATIONS)
  do_arithmetic() // multiplication
  mpi_barrier() // synchronization
  mpi_send() // blocking send
  mpi_recv() // blocking receive
  mpi_barrier() // synchronization
```

(a)



(b)

Figure 1. Algorithm for *Compcomm* benchmark. (a) *pseudocode* and (b) flowchart.

There are two versions of the NPB, *original* and *multi-zone* (*MZ*). The computations in the original benchmarks exhibit fine grain parallelism [66], i.e. they perform multiple communications of data in each iteration of the solving stage. As a result, their performance is more sensitive to communication latency. The *MZ* versions take a parallelization approach that mimics different kinds of applications. They solve the same discretization problem, but using multiple meshes (or *zones*). The MZ benchmarks are designed to perform only coarse-grained parallelism at the message passing level. The MZ version is more sensitive to load imbalance than latency, so their performance should be less affected by virtualization. We employ both versions of the benchmarks to see how their performance impacts compare. The benchmarks come with five input problems of increasing size, which they refer to as *classes*. We use classes A, B, and C, whose properties are shown in Table 1. The NPB are well studied and often used to test the performance of HPC systems. The multi-zone versions have been shown to scale well on up to at least 16 processors, and possibly over 1,000, depending on the benchmark and runtime configuration [67]. To investigate even larger problem sizes, we use the weather research and forecasting (WRF) software for a few experiments, which is another kind of fluid dynamics application with tightly coupled execution behavior. We use a popular benchmark input, *jan00,* and a large input, *75x4*, which requires more computations than Class C of the NPB.

Table 1. Sizes of the three classes of NPB inputs used

Class	**Dimensions**	**Area**
A	128 X 128 X 16	262K
B	304 X 268 X 17	1.075M
C	480 X 320 X 28	4.3M

III.5 Virtualization Performance Impact in Dedicated-CPU Scenarios

We now discuss the observations made from the experiments. For the experiments carried out in this section, each VM has one or two dedicated CPUs. The values presented represent the average of at least three executions run under identical conditions. We observe the performance penalty due to virtualization overhead in terms of the computation and communication characteristics of each job when executed in virtualized and non-virtualized environments. By doing so, we can provide improved insight on the virtualization overhead of different scientific applications being applied to different input problem sizes. For tightly coupled applications, we distinguish between computation and I/O time. There is little disk I/O needed for the jobs we run, so we do not consider it necessary to separate it from I/O due to inter-process communication via network.

III.5.1 Terminology

We refer to the ratio of virtualized execution time to bare metal execution time as the *virtualization penalty*. The extra CPU time that the hypervisor requires for I/O operations is referred to as *virtualization overhead.*

III.5.2 Performance Analysis of Image Processing with FSL

One caveat with the image processing applications used is that their execution times vary due to random components in the algorithms. MELODIC executions vary more because the main algorithm is iterated until converging and the number of steps required to converge depends on a random initial variable. For example, we performed 20 executions of the same data set on the same physical machine and observed an 8% difference in execution time between the fastest and slowest execution. FAST times varied less than 2%, since the heuristics used are guaranteed to converge in only “a few iterations” [62]. The data sets used are discussed in Section III.4.1.

Image Segmentation

Since there is no communication of data involved when executing separate studies in parallel, we expected the bare metal and virtualized performance of Xen to be roughly the same. Surprisingly, the virtualized executions were 10-15% faster. We performed profiled executions using *Oprofile* to understand why. The execution profiles were similar for all data sets, so in describing this phenomenon, we focus on the first data set from MCH. In Figure 2, we show the execution time of the 7 most time-consuming functions (labeled A-E for brevity) in the VM (using circles) and BM (using squares) configurations. As can be seen, function *A,* which corresponds to the *convolution* function, has a disparity between the BM and VM executions. Furthermore, running the program through the GNU debugger (*gdb*) revealed that the function is only slowed down in the BM when processing about the k^{th} direction in the *i,j,k* space. This function is called 30 times and consists of 193 million additions and multiplications and 6.03 million assignments of a 3 dimensional local variable per call when processing a 256x256x190 image and using a 40x40x32 convolution kernel. According to the profiler, memory operations consumed the bulk of the time, suggesting that virtualization-related cache optimization is the reason for the speedup. This coincides with a similar observation made in [36] when the authors ran BLAST [37] jobs, in which they suggested that VM double caching caused the virtualized execution to be faster.

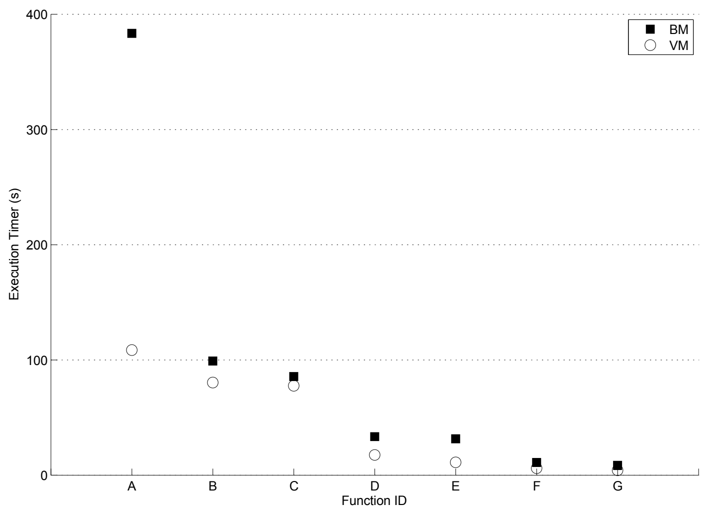


Figure 2. Execution times of the 7 most time-consuming functions of FAST.

Image Registration

Comparing the VM and BM performance of the MELODIC image registration experiments was not straightforward due to the aforementioned randomness in the algorithm. Specifically, The ICA algorithm does not terminate until it converges, and the number of steps required until it converges depends on the random initial value. We observed anywhere from 63 to 136 steps before converging for identical executions, hence there was some variation in the resulting execution times.

While this variation makes it difficult to measure the effect of virtualization, the results clearly showed that the VM executions were slightly slower when simultaneously processing 2 data sets per node. When only one data set at a time was processed on each node, the average overhead was negligible. When all data sets were submitted at once (but only allowing one CPU per job), the overall VM slowdown was 13%, 10%, and 13% for 1-, 2-, and 4-node executions, respectively.

Figure 3 compares the completion times of each data set from the MCH repository for the VM and BM experiments for single-node, single process (*solo*) and 4-node, 2-process-per-node (*4n*) executions. The relationship between the VM and BM executions is always the same, with the BM finishing slightly faster in the latter scenario.

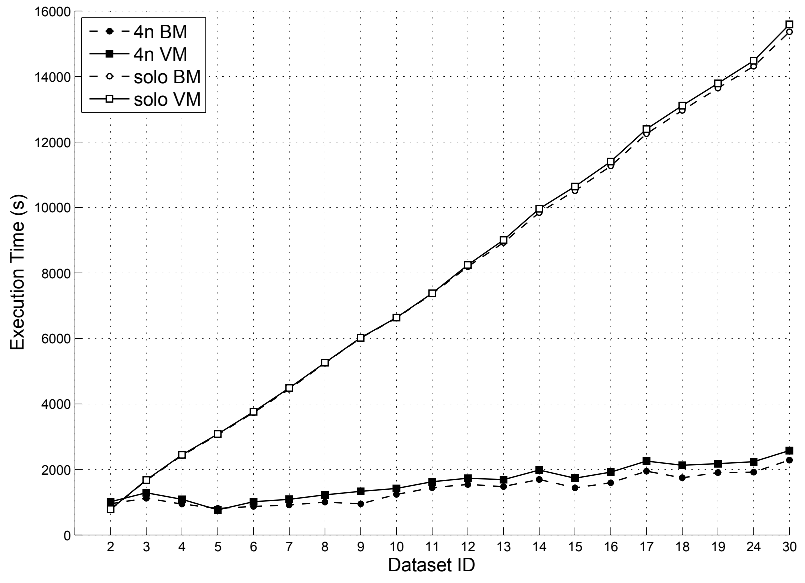


Figure 3. Execution time of MELODIC when run solo and when using 4 nodes.

III.5.3 Virtualization's Impact on Tightly Coupled Applications

We now discuss the virtualization impact on tightly coupled applications. Since the impact of virtualization on these applications varies so much depending on the characteristics of the job, it is more complex to describe, and thus we dedicate a relatively large amount of space to discussing it.

Compcomm

We begin the discussion on tightly coupled application performance with the *compcomm* benchmark, whose algorithm was shown in Figure 1. To gain insight on the relationship between computation ratio, message size, and virtualization overhead, we vary the number of computations per iteration and the message sizes. Computations per iteration values used are 25, 50, 100, and 200; Message size values used are 0.64, 1.28, 40, 8.75, 17.5, 35, 70 and 140 kB. In Figure 4, we plot the virtualization penalty (vertical axis) for different MPI message sizes as the duration of the computation cycle (horizontal axis) is increased. We observe an inverse relationship between computation cycle length and virtualization penalty. The figure shows that the penalty tends towards unity as the length of compute iterations is increased. For message sizes below 8.5kB, the virtualized executions are actually slightly faster. We attribute this to reduced operating system noise in the VM nodes as we observed that idle bare metal nodes experience more than 15 times as many interrupts as idle *vm-container* nodes.

Looking at the relationship between message size and overhead (keeping computation duration constant), we see a significant increase in virtualization penalty as the message size is increased, especially when the computation cycle duration is less than 0.2 seconds, because the communication time is a significant portion of the execution time. Only the executions with 140 kB remained at over 2% overhead when the computation duration reaches 0.67 seconds. It is observed that 140kB is large compared to the message sizes used by the applications we experimented with. Hence, we can deduce that the virtualization overhead is minor for well balanced tightly coupled applications as long as the problem size is not small.

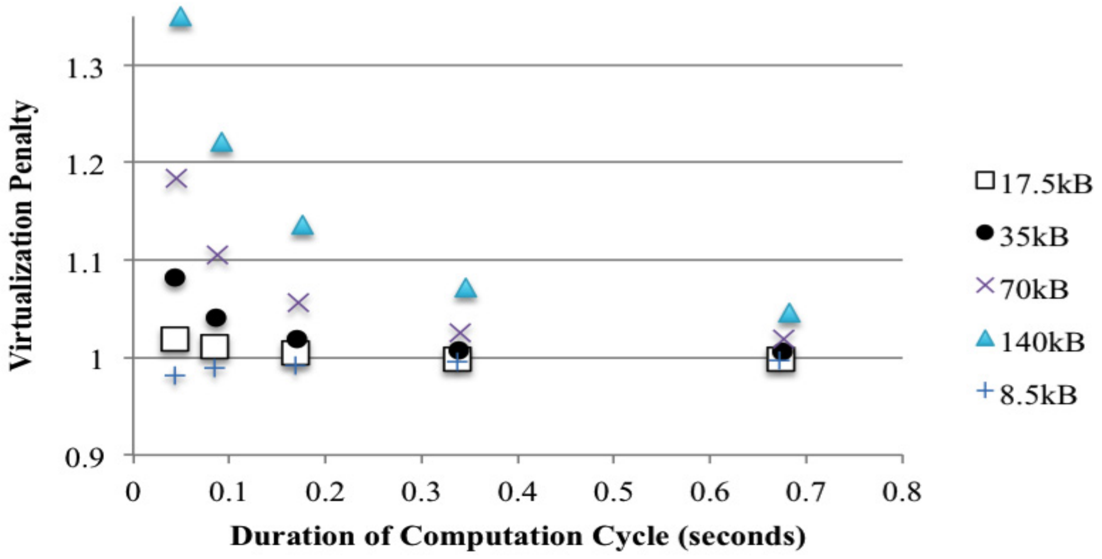
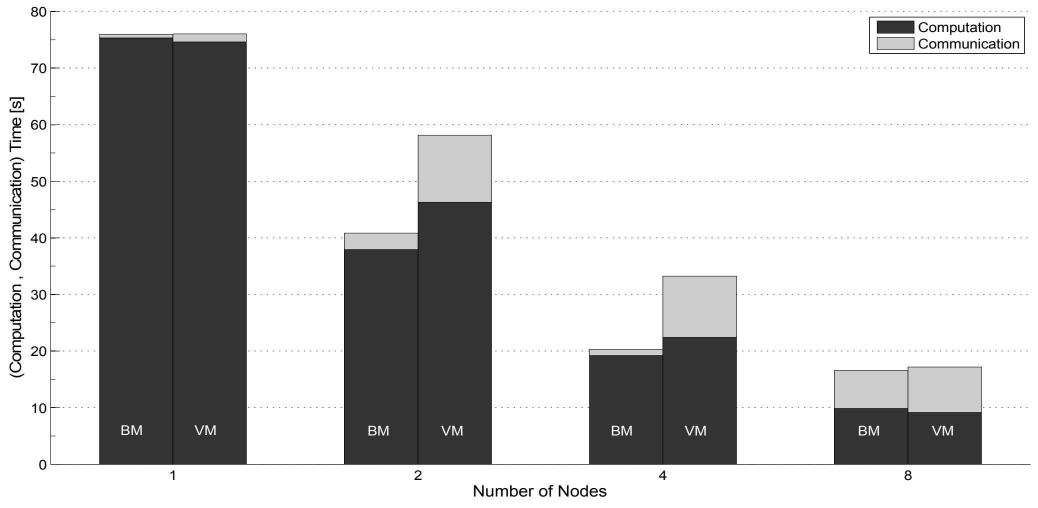


Figure 4. Effect of increasing computation cycle duration on virtualization penalty.

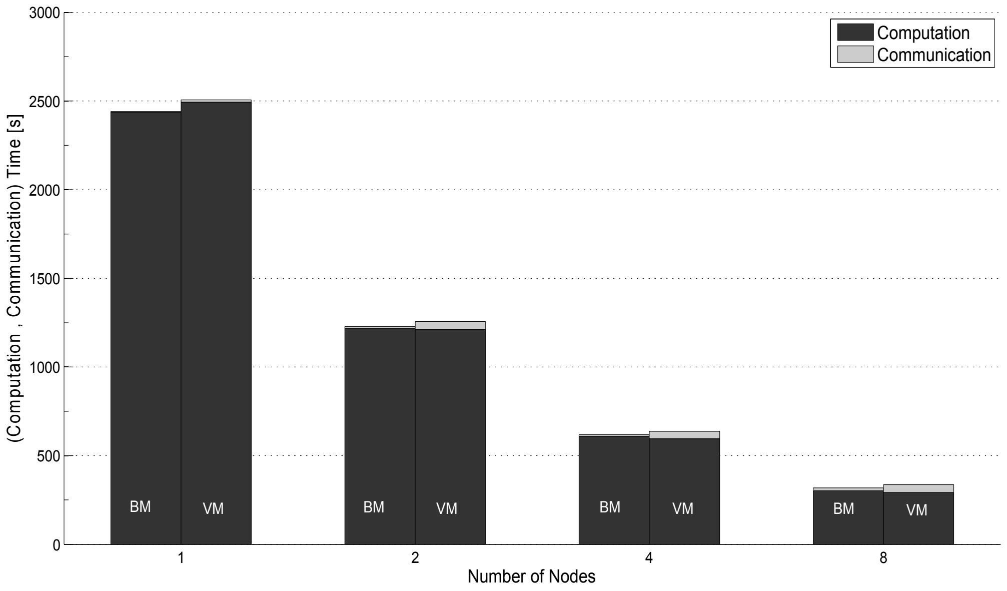
NPB LU and LU-MZ Benchmarks

We now look at how the relationship between computation ratio, message size, and virtualization overhead observed using *Compcomm* compares to that of actual applications for which communication requirements vary for each worker. Figure 5 shows the overall communication and computation times for 2-process-per-node LU-MZ executions with Class A (Figure 5a) and Class C (Figure 5b) for 1, 2, 4, and 8 nodes. The BM and VM times are shown in adjacent rectangles for each configuration. Since the performance penalty was below 6% for the 1-process-per-node executions, we do not show them. The times depicted in the figures were obtained from the timers built into the benchmarks. The communication times include physical communication as well as virtualization overhead. Comparing the figures, we can see that using a smaller input results in a larger performance penalty on multi-node executions compared to the larger input.

Using Xen's command line tools revealed that virtualization overhead was less than 2% larger for Class A, which does not explain the larger difference in performance penalty. We analyzed the communication pattern of the execution using the *Paraver* trace analysis tool [70], which revealed that when running Class A, the average duration of the computation cycles was only 72 milliseconds, which implies that there was high communication frequency. For Class C (Figure 5b), the duration is 520 milliseconds, resulting in a much smaller virtualization penalty. This coincides with the observations from the *compcomm* experiments, where we found that the virtualization penalty increases as computation rate decreases. However, the largest virtualization penalty with *compcomm* was 35%, compared to 62% for LU-MZ. The other culprits are load imbalance and contention between the processors when accessing the network interface.



(a)



(b)

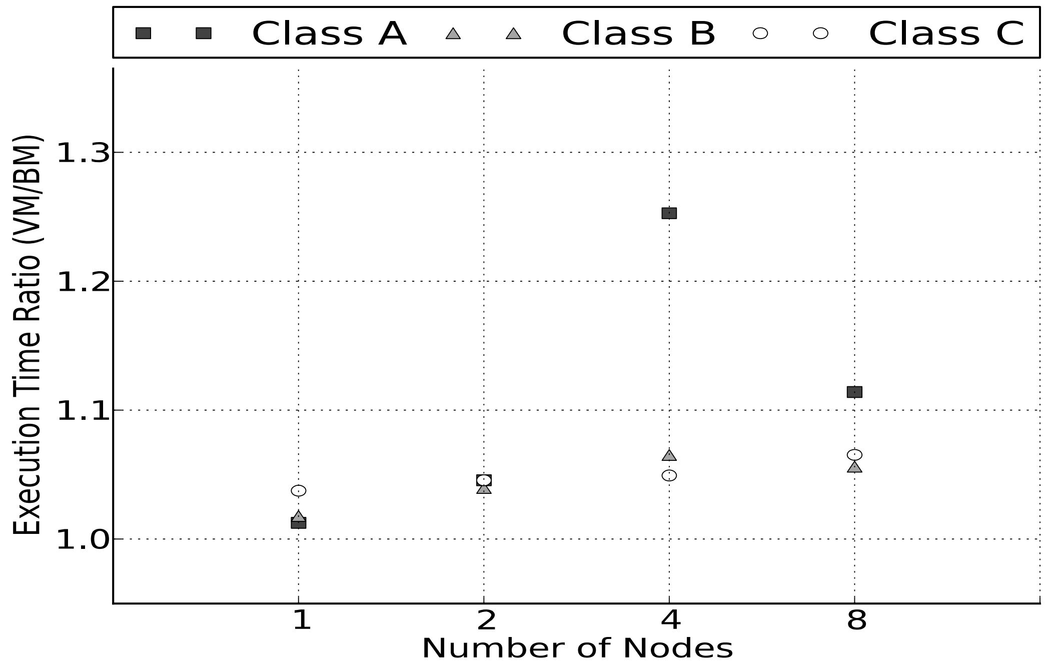
Figure 5. Communication and computation times for LU-MZ (a) Class A and (b) Class C, using 2 processes per node.

Another observation is that the overall execution times with 8 node Class A executions are roughly equal for the VM and BM. This is because the (non-virtualized) computation ratio is only 53%. Since the CPU spends so much time idle, the virtualization overhead has a negligible effect on overall execution time.

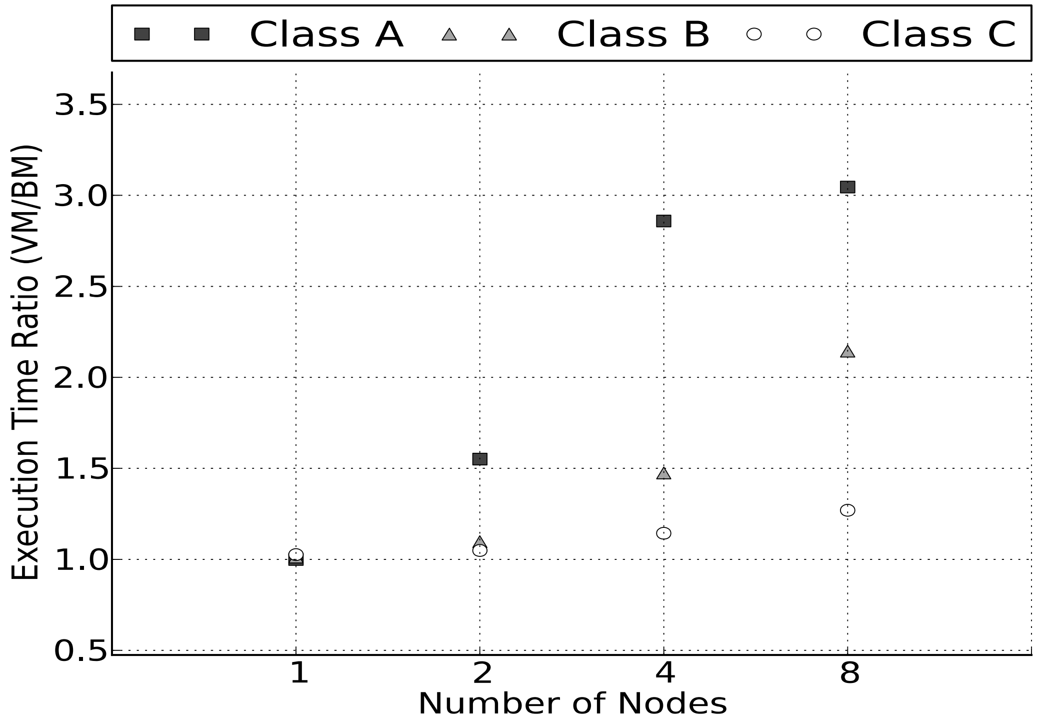
We repeated these experiments with the other coarse grained NPB CFD benchmarks (SP-MZ and BT-MZ), with similar observations. With 1 process per node, the virtualization penalty increased roughly linearly as a function of the parallelism and never surpassed 10%. With 2 processes per node, the pattern of the virtualization penalty was similar to LU-MZ.

Next, we repeated the experiments with the original (fine-grained) LU, SP, and BT benchmarks. As expected, the virtualization penalty was greater since the fine-grained implementation performs more frequent message passing (e.g. between 0.6 and 0.7 milliseconds between most messages for 8-node Class A runs, which is two orders of magnitude more frequent than with the MZ benchmark). Also, a larger amount of data is transferred during the execution; for example, a 4-processor execution of LU, Class A transfers a total of 122 megabytes of data with the fine-grained implementation but only 34 megabytes with the coarse-grained implementation. Looking back at Figure 2, we see that when the length of the computation cycles is below 100 milliseconds, quadrupling the message sizes results in a large virtualization penalty. Unlike the MZ benchmarks, the original benchmarks experienced significant overhead as can be seen in Figure 6 for both the 1-process-per-node (6a) and the 2-process-per-node (6b) executions. Both figures show the virtualization penalty (i.e. execution time on the VM divided by execution time on bare metal) as the number of nodes is increased. Again, we observe that using the smaller input data results in more overhead. Again, the penalty is attenuated when the (bare metal) computation ratio drops below 60%, as can be seen in the 8-node, 1 process-per-node Class A execution in Figure 6a. With 2 processes-per-node (Figure 6b), this occurs when the cluster is larger than 8 nodes, since the virtualization penalty stops increasing from 4 to 8 nodes. With 2 processes-per-node, virtualization causes additional latency multiplexing the network interface between the 2 processors, so the virtualization penalty is not attenuated despite the low computation ratios. Since the computation ratio is bigger with the larger problem sizes, the penalty monotonically increases with the number of nodes.

Our results thus far have given an overview of the performance impact of virtualization. To estimate a job's execution time, and to anticipate the *maskability* of its communication when it shares the CPU with other applications (assuming at least one is tightly coupled), we need to know its virtualization overhead. In Tables 2 and 3, we tabulate the virtualization overhead (in CPU percentage) for all the experiments carried out using 1-process-per-node and 2-processes-per-node executions, respectively.



(a)



(b)

Figure 6. Virtualization penalty for the original LU benchmark, running (a) 1- and (b) 2- processes per node.

Table 2. Percentage of CPU used for virtualization overhead running 1 process per VM

App (Input)	1 node	2 node	4 node	8 node
LU-MZ (A)	0.7	3.2	6.6	6.1
LU-MZ (B)	0.7	2.4	4.8	5.2
LU-MZ (C)	0.8	2.0	3.0	2.9
LU (A)	0.7	5.5	10.4	10.0
LU (B)	0.7	4.5	10.7	8.2
LU (C)	0.9	3.2	5.3	5.8
WRF (*jan00)*	0.9	5.1	7.0	8.5
WRF *(75x4)*	0.7	5.0	7.5	8.5

Table 3. Percentage of CPU used for virtualization overhead running 2 processes per VM

App (Input)	1 node	2 node	4 node	8 node
LU-MZ (A)	0.7	4.8	4.3	4.0
LU-MZ (B)	0.7	6.0	4.8	5.2
LU-MZ (C)	0.7	3.0	3.0	2.9
LU (A)	0.7	7.4	8.4	9.4
LU (B)	0.8	9.5	11.4	7.3
LU (C)	0.7	9.2	11.7	8.7
WRF (*jan00)*	0.6	5.5	6.8	8.7
WRF *(75x4)*	0.7	5.6	7.9	7.4

III.6 Performance Analysis With Shared-CPU Executions

III.6.1 Sharing CPU Among Loosely Coupled Jobs

We ran multiple simultaneous serial executions of WRF and FAST to measure the execution time impact due to CPU sharing. We found no significant slowdown compared to running the jobs sequentially. With WRF, we ran up to 8 multiplexed serial instances of the *jan00* domain, which takes 25 minutes to complete and uses 200 megabytes of RAM, and running simultaneously took roughly the same amount of time to finish all 8 as running sequentially. We ran a similar experiment with FAST to see if it would be affected more than WRF, since its more memory intensive, but we found that the makespan of 4 simultaneously-executed jobs was within 1% of the time it would take to run them sequentially, using the average FAST execution time as a basis. The relation for FAST can be seen in Figure 7, where we plot the completion time of all jobs as a function of the number of simultaneous jobs. A linear trend line is used to show that the relationship is roughly linear. As a result, we conclude that execution time prolongation due to CPU sharing for loosely coupled jobs can be accurately predicted as the product of the computation time and the inverse of the CPU allocation of the job. This model will work with a up to 8 jobs for WRF and up to 4 jobs for FAST. These are reasonable limits considering the memory requirements of each application.

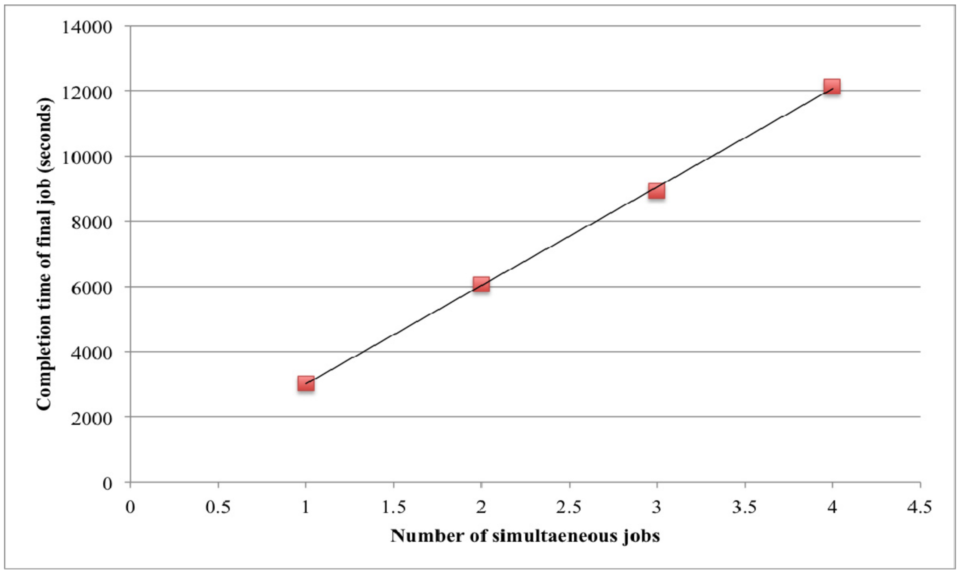


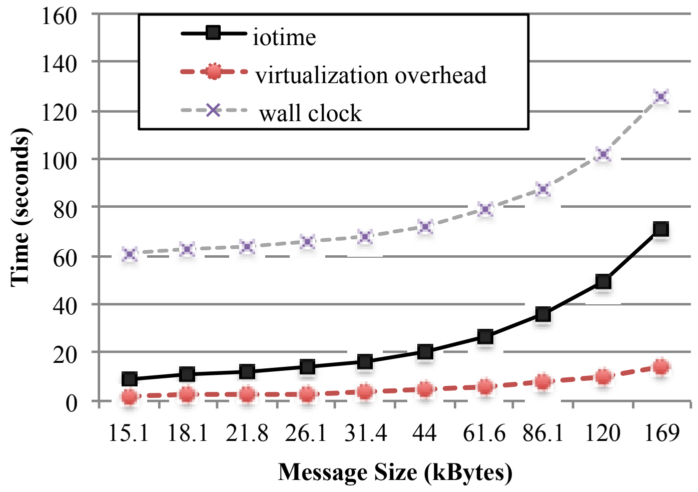
Figure 7. Effect of multiplexing up to 4 FAST jobs on one CPU on makespan.

III.6.2 Sharing CPU Among Tightly Coupled Jobs

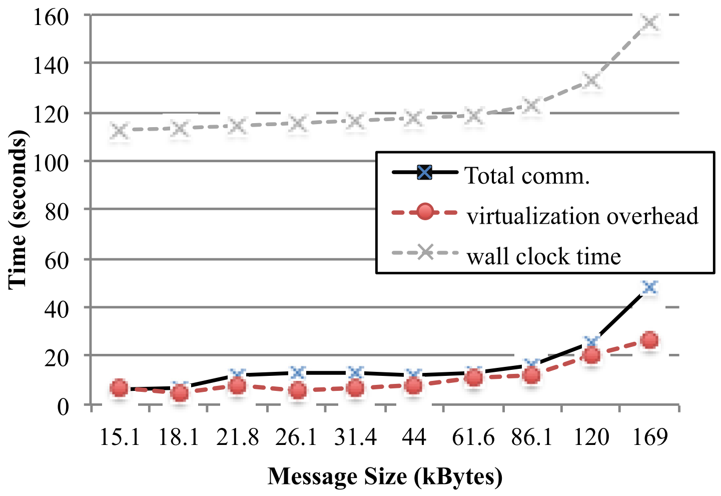
The execution time of shared-CPU tightly coupled jobs is harder to predict since a significant portion of their execution is spent performing I/O. We now study the shared-CPU behavior of the tightly coupled applications mentioned earlier, and compare our findings to the expected behavior described in Sections III.3.1 and III.3.2. We start with an observation that confirms that by collocating 2 parallel jobs and multiplexing the CPU, we reduce the makespan of the two jobs compared to running them sequentially, by virtue of communication masking, despite the virtualization penalty.

We start by running two instances of the 2-task *Compcomm* parallel benchmark, described in Section III.4.2, on 2 physical machines. One physical machine hosts the 2 master VMs and the other the 2 slave VMs. Each VM runs one parallel task. In each physical machine, both VMs multiplex the same processor. The non-multiplexed and multiplexed execution times, for different message sizes, are shown in Figure 8(a) and (b), respectively. Both figures show execution time (vertical axis) as a function of message size (horizontal axis). Three relations are plotted in each figure. The dark solid line shows the total communication time, including virtualization overhead. The lighter dashed line only shows the CPU time pertaining to the virtualization overhead. The light dotted line shows the wall clock time. In the multiplexed case, the latter is the makespan of the two jobs. Note that the difference between the *total communication* and *virtualization overhead* lines is the physical communication time. We find that the majority of the physical communication time can be masked as long as the computation ratio is above 50% or so. Also, comparing the wall clock execution times of the two figures we can infer that the makespan when multiplexing the two jobs is faster than running them sequentially, especially if message sizes are large. For the final data point, the execution time increased significantly because the computation ratio dropped below 50%, hence less communications could be masked.

Our initial expectation, under the assumption that messages can be transferred while collocated jobs are in the working state, was that the makespan would be roughly twice the application's *CPU time*, plus the virtualization overhead, and a small penalty for context switching. In the case of this benchmark, context switch overhead is minor since the memory footprint is small. Looking at Figure 8, it can be seen that there is some additional overhead beyond the virtualization overhead. For example, when using a message size of 44 kB, the expected makespan under this assumption is 113.2 seconds, whereas the measured makespan is 117.2 seconds. Using *Paraver*, we found the additional overhead was due to jobs' communication intervals occasionally overlapping, resulting in wasted CPU cycles. In other words, not all communications were masked by computations. This can be seen in Figure 9, where we plot a portion of the *Paraver* execution trace visualization. In the figure, we show the temporal execution pattern for several iterations of executions in the dedicated CPU (Figure 9a) and shared CPU (Figure 9b) cases. Each bar in the figure represents a worker; e.g. J(N,W) is the W^{th} worker of Job N. Black sections represent states where the worker is *running* (i.e. requesting or using CPU), dark gray sections represent states where the worker is *synchronizing* (with another worker), and light gray sections represent when a worker is *sending* data. There is overlap in the second communication iteration shown in Figure 9b, resulting in idle CPU cycles.



(a)



(b)

Figure 8. Overall I/O time, virtualization overhead, and makespan as a function of message size, with (a) a single *Compcomm* job and (b) 2 multiplexed *Compcomm* jobs.

The overlap of communications will affect the results of the performance model, since it is not possible to analytically determine when they will occur. However, we expect them to be rare, as they were for the experiment corresponding to Figure 9, with most tightly coupled jobs. Also, note that this would not be a problem when multiplexing a CPU-bound loosely coupled job with the tightly coupled job(s). Collocating these two kinds of jobs will help yield optimal utilization of the CPU in virtualized environments. In addition, since tightly coupled jobs execute at the rate of the worker with the least available CPU, it is possible for many physical machines to have underutilized CPU due to fragmentation. By collocating loosely coupled and tightly coupled tasks, this problem can be avoided as well.

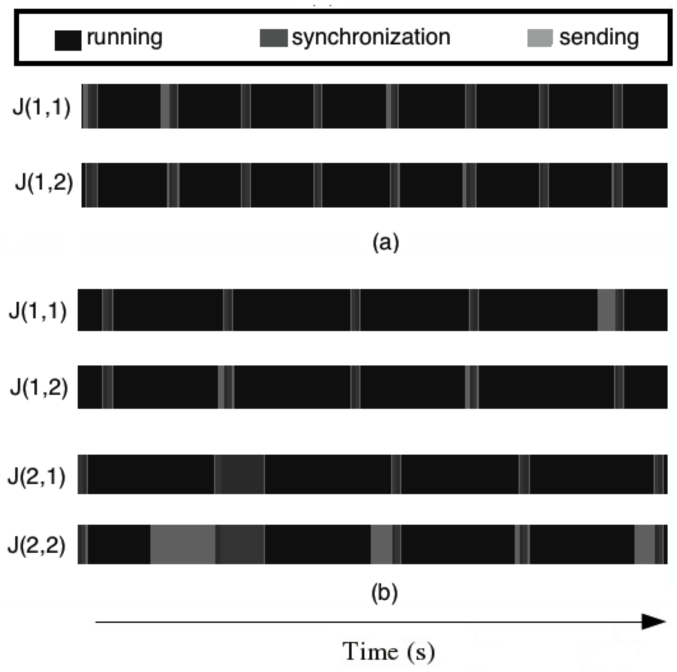


Figure 9. Execution trace of *Compcomm*. (a) dedicated CPU, (b) shared CPU.

CHAPTER IV

EXECUTION TIME PREDICTION METHODOLOGY

In this chapter, we discuss the performance prediction methodology to address two related problems: computation time prediction and scalability prediction. Computation time prediction refers to predicting how much CPU time an application requires to execute, given an input problem. Since we deal with shared-CPU environments, we also account for different CPU allocations. Scalability prediction refers to predicting how the execution time of an application will increase/decrease depending on the number and type of machines being used to run it.

In both cases, we rely on statistical prediction models that use historical job execution data as training data to extrapolate for future executions. This approach fits our scenario best for two reasons. First, a provider of medical image processing services should know basic execution-related characteristics about these applications. This information can be obtained by carrying out experiments similar to those presented in the previous chapter. Second, many statistical prediction methods are computationally simple, which is a requirement for our job scheduling methodology described in the next chapter, since it will be necessary to quickly perform one or more execution time predictions in order to make real-time job scheduling decisions.

IV.1 Overview of the Prediction Methodology

The methodology used can be described as a hybrid approach to execution time prediction, in the sense that the prediction model itself has no application- or domain-specific knowledge, but users may add this knowledge after determining the factors that affect performance. In other words, the model itself is oblivious to the application, but human knowledge about the application improves the model's accuracy. Some existing approaches to performance prediction have general and/or specific knowledge about application execution included in the prediction paradigm itself. A possible problem with these approaches is that they can be difficult to deploy; for example, some of these tools require the application used to be compiled with special tracing libraries. Conversely, approaches that are entirely oblivious to the application generally suffer worse prediction accuracy [72]. As we will show, we do not try to tailor our model to any specific application. Instead, we use knowledge of the application and execution platform to improve the model. We now summarize our performance prediction methodology.

Figure 10 depicts our multi-step, iterative performance modeling approach. The approach starts with Stage *A (Application/Code/ Platform inspection),* in which specific details about the application and/or execution platform are studied. The purpose of this step is to determine what parameters contribute to the execution time of the application. An example of a question that this step can answer is how the CPU of the execution platform affects the execution time. The depth of knowledge required for this step depends on the application. For example, some applications are I/O-bound, and increasing the CPU clock speed will not provide any performance improvement.

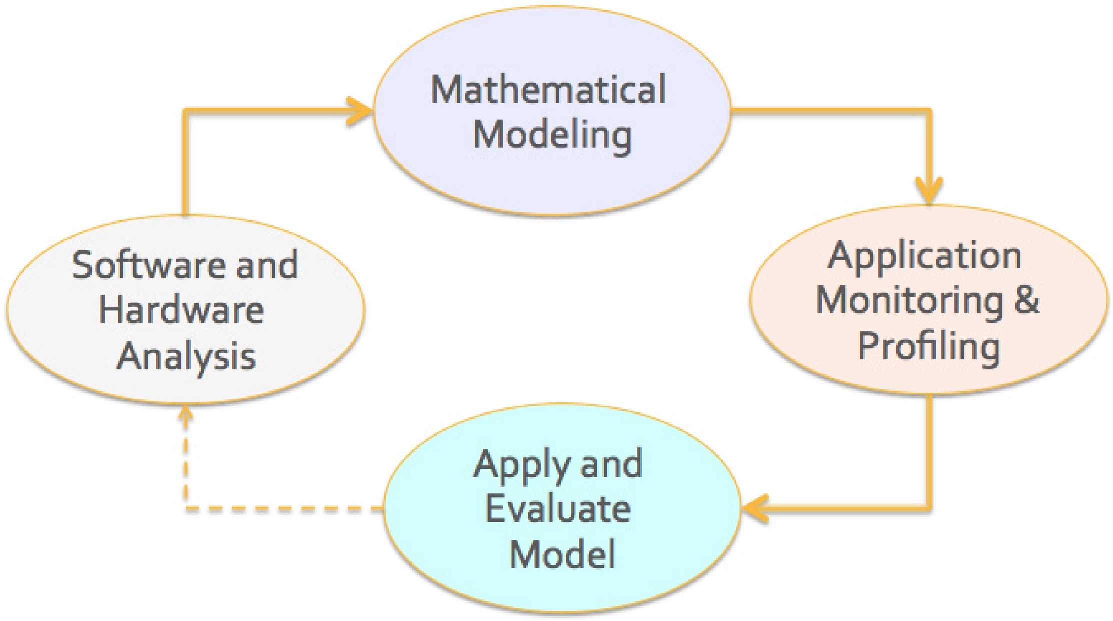

Mathematical Modeling
Application Monitoring & Profiling
Apply and Evaluate Model
Software and Hardware Analysis

Figure 10. Overview of the performance prediction methodology.

In Stage *B*, a mathematical model that relates execution time to other parameters, based on intuition and specific findings from Stage *A,* is devised. The main constraint in choosing a model is that it must be able to provide real-time execution time predictions in order to make fast scheduling decisions. The model is described in the next section.

In Stage *C*, we perform executions under different conditions and/or with different runtime configurations. We define a *runtime configuration* as the number of nodes and processes per node used for any single execution on a particular system. When we refer to a *data set* or *data series*, we are referring to a collection of execution statistics for a single instance of all possible runtime configurations (e.g. the execution times for all runs performed at a particular time with 8, 16, 32, and 64 nodes).

Each execution is *profiled,* i.e. the resources used for the execution are recorded. When all executions are finished, the acquired data is fed into the prediction model, which estimates the contribution of each parameter (Stage *D*). Based on these individual estimates, the total execution time is estimated for a target execution platform, and compared to the actual execution time. The iterations of the *A*-*B*-*C*-*D* cycle are repeated until an average prediction error of 15% or less is achieved. The time it takes to iterate through the cycle depends on the data being collected, but the tools were designed to provide fast results and use regular text files so that data can be easily added or removed using common text-processing tools.

IV.2 Prediction Model Overview

The model we use is implemented in a profiling tool, *Aprof,* described in [71], which was developed as part of the Latin American Grid partnership. In this section, we summarize the implementation of the model and how it was applied to our work. The model assumes that the execution time of an application can be expressed as the product of several *contributors* that affect a job's execution time. It determines the magnitude of their contributions with respect to execution time. Some contributors either vary too much between executions (e.g. the state of CPU registers) or they contribute a negligible amount to overall execution time, so the model relies on human intuition about the applications being run and the systems they are being run on, obtained in Stage A, to aid in its development. For example, the duration of an image processing job depends on the size of the input image(s) being processed, so image size should be a strong contributor to execution time. We find that using intuitive parameters, based on basic knowledge of the algorithm of a given application and the system(s) it runs on, yields predictions that are accurate enough for job scheduling.

The contribution parameters themselves may be polynomial equations of arbitrary length, which results in Equation (1), in which m is the number of parameters, m_i is the maximum polynomial degree of the current parameter, a_{ij} is the coefficient contribution of the i^{th} parameter, and z_i^j is the i^{th} parameter. Until now, we have had success with a simplified model in which the maximum polynomial degree of all parameters is equal to one (i.e. first-order polynomials).

$$T_{exec} = \prod_{i=0}^{m-1} \sum_{j=0}^{m_i} a_{ij} z_i^j \tag{1}$$

Based on this assumption, the model attempts to determine the contributions of each of these parameters (i.e. the a_{ij} values). The resource properties are all combined to form a sum-of-products, plus an error term to account for model inaccuracies and absent parameters. Regression analysis is used to determine the values of the coefficients.

IV.3 Applying the Model to Computation Time Prediction

In this section, we describe how we address the problem of predicting the computation time of a given application and input, based on historical execution data with different inputs. The main problem for this is determining the parameters that contribute most significantly to execution time in order to create a model. A constraint on the parameters chosen is that they must be programmatically obtainable (e.g. by reading header information of the input files) so that job scheduling decisions can be made in real time. Since this is application-dependent, we describe the approach for each application separately.

IV.3.1 Image Segmentation

We analyzed the execution time of FAST using data sets with different sizes and from different hospitals. We provide pertinent information in Table 4. We did not find a strong correlation between the 3 dimensions (X, Y, Z) of the data sets and their execution time requirements. Using the 2 dimensions (i.e. X and Y dimensions only) was actually better, but still would result in high error. Instead, we employed *aprof*, using the size of each dimension of the image as explanatory values and the execution time as the exploratory value. We obtained a mean execution time prediction error of 2.94%, max of 7.2% and min below 0.1%. These parameters can be read from the DICOM or NIFTI header of the image files, so they are suitable for our modeling approach. We repeated the test by predicting the execution time of all data sets, but using only two input data of different sizes, and the error remained below 10%.

IV.3.2 Image Registration

As mentioned earlier, MELODIC is subject to significant execution time variation since the duration of the algorithm performed before registering the images depends on the number of time steps required to converge, which in turn depends on a randomly-selected value. Using the same explanatory variables used for FAST in addition to the size of the temporal dimension and applying the model to the MCH data, we obtained a mean prediction error of 12.2%, a max of 29%, and a min of 0%. As a result, an extra “safety net” must be used when predicting execution times of MELODIC jobs in order to avoid deadline violations.

IV.3.1 LU Benchmark

The observations made in Section III.5.3 showed that the computational requirements of the NPB LU benchmark increase roughly proportionally to the input problem size. The relationship is not quite linear due to duplicate computations that occur with tightly coupled problems, which is a known problem. Since no new or interesting observations were made, we do not comment further on the computation time prediction for LU.

Table 4. Execution time and memory utilization for various FAST jobs

Dataset	dimX	dimY	dimZ	Exec. Time	Memory Utilization (Bytes)
CHOA_1	176	240	256	632.0	1.27E+09
CHOA_2	176	240	256	702.0	1.30E+09
CHOA_3	176	240	256	699.7	1.24E+09
CHOA_4	176	240	256	682.7	1.30E+09
CHOA_5	176	240	256	662.3	1.28E+09
CHOA_6	176	240	256	663.7	1.30E+09
CHOA_7	176	240	256	701.7	1.28E+09
CHOA_8	176	240	256	671.3	1.28E+09
CHOA_9	176	240	256	644.3	1.26E+09
CHOA_10	176	240	256	645.0	9.86E+08
CHOA_11	176	240	256	663.3	1.12E+09
CHOA_12	176	240	256	654.3	9.57E+08
CHOP_10	256	256	192	1464.0	1.50E+09
CHOP_11	256	256	192	1464.0	1.50E+09
CHOP_12	256	256	192	1466.0	1.50E+09
CHOP_13	256	256	192	1371.0	1.48E+09
CHOP_3	256	208	160	534.3	9.94E+08
CHOP_4	256	208	160	569.7	9.48E+08
CHOP_5	256	208	160	572.3	1.15E+09
CHOP_6	256	208	160	608.3	1.03E+09
CHOP_8	256	208	160	579.3	9.59E+08
CHOP_9	256	256	192	1549.7	1.48E+09
BCCH_30	211	288	288	1182.0	1.64E+09
BCCH_44	211	288	288	1081.7	1.94E+09

IV.4 Scalability Prediction

Since the historical execution data of a job may not have execution time requirements with currently-available resources, it is necessary to predict how the job will scale under different runtime scenarios. Hence, a scalability prediction model is needed to extrapolate this information.

For loosely coupled applications, scalability prediction is tractable. Multiple studies on loosely coupled applications have shown that the relationship between execution time and parallelism is roughly linear (e.g. for BLAST [73] and NAS EP [74]), so accurate predictions are obtainable. Figure 7 confirms this is the case with FAST as well. With *bags-of-tasks*, the computational requirement of the bag is simply the sum of that of each task. The challenge, therefore, is minimizing the makespan of all the jobs by optimally packing them among the available resources. Our algorithm for doing this is explained in the next chapter.

The scalability prediction of tightly coupled applications is complicated by their tendency to lose efficiency as the parallelism level increases due to redundant computations, load imbalance, and/or communication overhead. We mitigate this by using several prediction parameters and a large amount of training data.

We now discuss the scalability prediction approach taken. To limit the initial number of variables, we start with the dedicated-CPU case. The main challenges we address in this case are extrapolating for different combinations of CPU architectures and parallelism levels. For CPU architecture, we include such things as memory and network bandwidth, which can be affected by the bus and the number of CPU cores per machine. We ensure jobs are not placed on machines that cannot fit the problem into memory, since swapping to disk would result in a large execution time penalty that would be difficult to predict.

The remainder of this subsection details the work carried out in publications [71,75], which addressed scalability prediction for tightly coupled jobs.

IV.4.1 Overview of Challenges

We note three barriers to obtaining accurate predictions: the uncertainty of the CPU architecture's impact on the performance of the application, the distribution of nodes across machines and within the same machine, and the size of the data center. We now summarize these challenges and how we addressed them.

Extrapolating to Different CPU Architectures

Whereas CPU clock speed can be used to extrapolate performance among similar CPUs, as was shown in [75], different CPUs have much different characteristics, so another approach is necessary. A good example of this is the transition to more efficient CPUs after hitting the power wall with the Pentium-4 processor. Subsequent processors have achieved much better performance with lower clock speeds. To understand why a given processor is faster than another requires in-depth knowledge about its design. Such factors as pipelining, instructions/cycle, efficiency of internal components, etc. play an important role in this. A cycle-accurate simulator similar to the one implemented in [26] would yield accurate predictions, but the complexity of modern processors makes this difficult. Also, such an approach is not suitable for job scheduling, where real time execution predictions are needed.

An alternative to low-level modeling of the CPU is to find metrics that correlate well with execution time. For example, in [31] the authors ranked several metrics and found that for some applications, execution time correlated best with strided access to main memory, while for most other applications random access to L1 cache had better correlation. In [76], the authors found node bandwidth and latency to be the most significant parameters for the scalability of WRF. To properly evaluate the metrics with the highest contribution, it is necessary to measure several of them.

Benchmarking is an alternative that can give a good indication of CPU performance for different applications. The caveat with benchmarking is that, for best results, the benchmark needs to be representative of the application being modeled, which requires some knowledge of the application. Since CPU instruction patterns vary by application, the best benchmark for a particular application is the application itself. However, benchmarking the scalability of every application that is to run on a given system may not be ideal. A good tradeoff is to run a few benchmarks with resource consumption characteristics representative of a broad range of applications. Each application that is to be run on the system can be mapped to a particular benchmark. For example, WRF simulations involve solving differential equations and finite difference approximations, so the performance measured using a generic benchmark that ranks a CPU based on its performance executing these kinds of calculations should provide a good measure of the CPU's performance when running WRF jobs.

We use the product of the CPU clock speed and a constant, the *platform contribution*, to model the CPU parameter. The *platform contribution* is determined by benchmarking. It only needs to be calculated when performing predictions among systems with very different CPU architectures. Our results in [71] showed that clock speed is a good indicator of the performance of systems with the same or similar CPU architectures, so it is not necessary to measure separate *platform contribution* parameters for systems with similar CPUs but different clock speeds.

Challenges With Multicore Architectures

Multicore architectures have become commonplace for all types of computing systems, so we considered it necessary to accurately predict execution time on multicore systems. To determine the optimal parameters to introduce to the model in order to model execution on multicore systems, a closer look into multicore architectures is necessary. For parallel jobs in which only one core of each node is used and the system specifications are kept constant, speedup is affected by interconnection network performance and the application's parallelization ability. The latter is a combination of computational redundancy, synchronization requirements, etc. When multiple cores are used, several complications arise. For one, intra-node communication may take place. Since the bandwidth and latency of messages passed inside a processor/bus between processing cores is different from that of different nodes communicating through Ethernet, multiple communication factors need to be modeled. Furthermore, the cores need to share certain components, such as cache, main memory, network cards, etc. This introduces the possibility of contention occurring when accessing different hardware components, leaving less effective capacity for each core. For example, if a physical machine has a dedicated L2 cache of 1MB, but the arbitration logic is shared, memory bandwidth to each core is limited.

Here, again, knowledge about the application is helpful to determine what parameters to model. For example, it has been shown that WRF is memory-bandwidth and latency bound [76], so the model needs to account for memory bandwidth in order to provide accurate predictions. As a result, we added a memory bandwidth parameter to the model. The measured memory bandwidth value is divided by the number of CPU cores used in the execution, since it is shared by each of them.

The behavior of a multicore node itself is generally consistent as long as whatever instructions it is executing are constant, so an approach that relies on previous execution data is able to cope with the fact that sharing components amongst cores leads to non-trivial execution patterns. However, when combining several multicore nodes, prediction is complicated by the fact that communication speeds and latencies are much different for processors on the same physical machine compared to processors on separate physical machines connected via Ethernet. As a result, it is necessary to give the model separate parameters for number-of-nodes and cores-per-node. This is in addition to the memory bandwidth parameter described in the previous paragraph.

IV.4.2 Contribution Parameters Used in the Scalability Model

Accounting for all the challenges above, a set of contribution parameters were measured and added to the prediction model as explanatory variables. Memory bandwidth was measured using a tool that performs sequential reads and writes of different amounts of data. The read bandwidth for 16MB of sequential data is used as the *memory-read-bandwidth* (MBW_{RD}). The write bandwidth for 16MB of sequential data is used as the *memory-write-bandwidth* (MBW_{WR}). Since the network bandwidth is also shared by separate CPU cores, we also use a *network bandwidth* (*NBW*) parameter, which is the theoretical bandwidth of the underlying network switch.

Multiple steps of refinement were required to obtain acceptable accuracy with multicore experiments. The combination of parameters that best modeled the application was MBW_{RD}, MBW_{WR}, *number-of-nodes, total processors, network bandwidth, and cores-per-node*. When predicting across different systems, a *platform contribution* parameter was measured using a benchmark. Inserting these parameters into Equation (1) results in Equation (2). In the equation, Γ_x refers to the contribution of parameter *x* to the overall execution time.

$$T_{exec} = \Gamma_{MBW_{RD}} \times \Gamma_{MBW_{WR}} \times \Gamma_{nn} \times \Gamma_{nc} \times \Gamma_{p} \times \Gamma_{NBW} \times \Gamma_{pc} \qquad (2)$$

Using first-order polynomial equations for each parameter, the equations of the contribution parameters are as follows:

$$\Gamma_{MBW_{RD}} = A_0 + \frac{A_1 \times MBW_{RD}}{N_c}$$
$$\Gamma_{MBW_{WR}} = B_0 + \frac{B_1 \times MBW_{WR}}{N_c}$$
$$\Gamma_{nn} = C_0 + \frac{C_1}{N_n}$$
$$\Gamma_{nc} = D_0 + \frac{D_1}{N_c}$$
$$\Gamma_{p} = E_0 + \frac{E_1}{N_c \times N_n}$$
$$\Gamma_{net} = F_0 + \frac{F_1 \times NBW}{N_c}$$
$$\Gamma_{pc} = G_0 + \frac{G_1}{pc}$$

N_c is the number of cores-per-node, N_n is the number-of-nodes*,* NBW is the network bandwidth, and pc is the platform contribution. Note that for each parameter, there is a constant contribution, i.e. the X_0 factor, and a contribution due to the magnitude of the resource parameter, i.e. the X_1 factor. The memory and network bandwidth parameters are divided by the amount of cores used per node since each processing core in the node needs to share the memory bus and network card. Parameters that have an inverse relationship with execution time (e.g. number-of-nodes) are inversed in the formula.

IV.4.3 Model Creation and Profiling

The model is built using data obtained from historical executions of an application. In order to automatically generate this data, a system and application monitoring tool was developed. We call this tool *Amon,* which is short for *a monitoring tool.* The tool was also originally developed as part of the LA-Grid partnership and is described in [41]. It was rewritten for additional functionality needed by our scheduling methodology described in the next chapter, although for the purpose of performance modeling the functionality described in [41] is adequate. *Amon* performs two main functions, monitoring and reporting. In terms of monitoring, it collects resource consumption data for running applications by probing the Linux */proc* interface at discrete intervals. The resource consumption data collected include CPU time, memory, and network bandwidth. *Amon's* other function, reporting, is performed at two levels. When a job completes, a report of its overall resource consumption data is generated and recorded (e.g. to a text file). Reporting of instantaneous resource consumption data of a job in progress is also performed on a per-request basis. This is used by our job monitoring component to determine the progress and execution rate of a job, as described in the next chapter.

To automate the data collection stage, several shell scripts were created to run jobs with different configurations. Additional scripts were created for the evaluation of the model in order to test with several different input parameters and data set sizes.

IV.4.4 Model Evaluation

In addition to our infrastructure at CATE described in Section III.1, we used three additional systems, two of which were from large research data centers, which allowed us to test our scalability prediction at a much larger scale and to perform predictions across different CPU architectures. One is *Marenostrum,* from the Barcelona Supercomputing Center and the other is *Abe,* a Teragrid [4] cluster from the University of Illinois at Urbana Champaign. The specifications of all the systems used are tabulated in Table 5. The table shows the CPU used in each physical machine of each cluster, the number of such

CPUs/cores per machine, the maximum number of nodes used, and the interconnection technology.

Table 5. Systems used to test our performance prediction methodology

Host Name	**CPU**	**Cores per node**	**Max Nodes**	**Interconnect**
Mind	Xeon Netburst 3.6GHz	2	16	1 gigabit Ethernet
Abe	Xeon Clovertown 2.33GHz	8	64	10 gigabit ethernet
Marenostrum	Power 970MP 2.3GHz	4	128	Myrinet

In the benchmarking process, four sets of execution data were obtained for each configuration and the average execution time of each run was measured. In cases where an outlier was detected, it was discarded. To test our hypothesis that using a *platform contribution* parameter based on a relatively generic benchmark can model the CPU performance of similar applications, we use the NPB BT-MZ, Class A benchmark's reported operations-per-second value as the *platform contribution* parameter and use WRF as the test application. On *Abe* and *Marenostrum*, 8-, 16-, 32-, and 64-node execution data were used. On *Mind*, 4-, 8-, 12-, and 16-node execution data were used. For all systems, 1-, 2-, and 4-processes per node were used. On the large systems (i.e. *Abe* and *Marenostrum*) execution time can vary from run to run due to differences in node interconnection, so we worked around this as described in [75] to obtain consistent results.

Since CPU cores on separate nodes affect execution time differently than CPU cores on the same node, there is a non-linear relationship between execution time and the total number of CPU cores. Since we use a linear model, it is necessary to distinguish between processes running on separate nodes and processes running on separate processors/cores within a node. Figure 11 shows that when the number of cores-per-node is kept constant, the execution pattern is linear or semi-linear and predictable. A similar relation holds when keeping number of nodes constant while varying the number of cores. In the figures, we use the inverse of number-of-nodes, since the execution time is inversely proportional to the number of nodes (i.e. more nodes should result in lower execution time).

We summarize the results with an evaluation of prediction accuracy when using different architectures, numbers of nodes, and numbers of cores. Additional results when only varying a subset of these are shown in [75]. For this study, we used input data from *Abe* and *Mind* to predict first for *Abe* and then for *Mind*. Out of all the experiments performed, the maximum error observed was 10.12% and the mean was 6.74%. Figure 12 shows the actual versus predicted execution times. The error was obtained using Equation (3), where t_{actual} is the actual execution time and $t_{predicted}$ is the predicted time.

$$error = 100 \times \frac{\left| t_{acual} - t_{predicted} \right|}{t_{actual}} \tag{3}$$

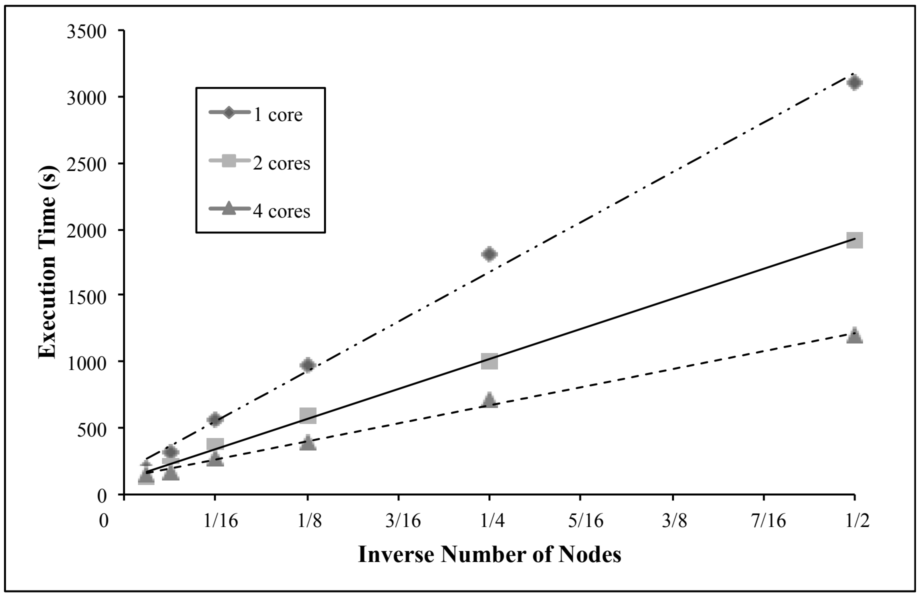


Figure 11. Execution time versus parallelism, keeping number-of-cores constant.

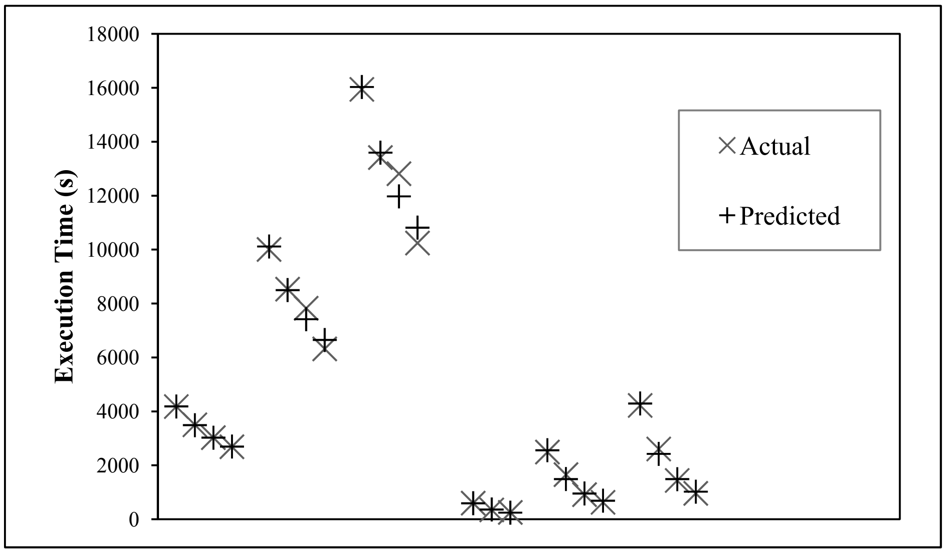


Figure 12. Actual versus predicted execution times for *Abe* and *Mind.*

IV.4.5 Extending the Prediction Methodology to Virtualized Platforms

We now show how the scalability prediction model was modified to account for virtualization, as presented in [77].

As we saw via the example in Figure 5, when run in a VM, a job's CPU time remains roughly the same, but its I/O time increases due to virtualization overhead. To address this, we modify the prediction methodology. Instead of modeling the overall wall clock execution time, we predict communication and computation times separately. For the computation time, the *user time* (i.e. CPU time spent in user space) collected by *Amon* was used. Communication time is not as simple to obtain using a lightweight monitor such as *Amon*. We use a simple estimator, t_{io} or simply *iotime*, which is the difference between wall clock time and user time, as shown in Equation (4).

$$iotime = t_{io} = t_{wall} - t_{cpu} \tag{4}$$

Before evaluating the revised model's ability to predict execution time, we test the efficacy of the values chosen to separate the CPU and I/O times by comparing them to the values of communication and computation time reported by the timers included with the NPB benchmarks. The computed correlation coefficients for all configurations of the VM executions of LU-MZ were 0.99 (computation) and 0.95 (communication). We consider this a good starting point for the model, hence, we use *CPU* time as the computation time estimate and *iotime* for the communication.

We measured the synchronous MPI bandwidth of the BM and VM configurations using a simple *ping-pong* test[i] that measures the bandwidth for transfers of different message sizes ranging from 8 Bytes to 1 MByte. The test was run 20 times and the average bandwidth of all runs was taken. The BM node was consistently about 40% faster throughout the range of message sizes evaluated. According to [78], the message sizes for the LU-MZ benchmarks range from approximately 220-350 kB for Class B to 600-950 kB for Class C, for systems with 2-16 processors. Since there is not much variation in the measured bandwidth for this range, the average of the 128, 256, 512, and 1024 kB measurements are used as the network bandwidth metric.

It was only necessary to evaluate the modified model with tightly coupled applications, since the other applications do not have significant I/O times. The resource consumption parameters used to estimate the computation times were: inverse number of nodes, inverse number of processes per node, and inverse memory bandwidth. To predict I/O time, the number of nodes, number of processes per node, and inverse of network bandwidth were used. The network bandwidth was adjusted according to the number of processes per node. Runtime configurations consisted of using 1, 2, 4, and 8 nodes and 1 and 2 processes per node, for a total of 16 data points per experiment. The overall error was calculated using Equation (5), in which *io* is the *iotime* and *u* is the CPU time.

[i]http://www.scl.ameslab.gov/Projects/mpi_introduction/para_pingpong.html

The actual and predicted computation and communication times for the LU and LU-MZ benchmarks with up to 8 nodes are shown in Figure 13. The predictions were performed separately for each class and for each implementation (i.e. *original* and *MZ*), for a total of 6 sets of experiments. The mean and median prediction errors were 13% and 4%, respectively.

$$error = \frac{|(io_{actual}+u_{actual})-(io_{estimated}+u_{estimated})|}{(io_{actual}+u_{actual})}100 \qquad (5)$$

The same experiments were repeated for WRF, using the same run time configurations and the *jan00* and *75x4* domains. The actual and predicted execution times are shown in Figure 14. The mean and median errors in this case were 9% and 6%, respectively. The mean error was more tolerable for WRF since using larger problem sizes results in less sporadic virtualization penalty. The NPB results were skewed due to the higher error of the Class A predictions.

IV.5 Modeling the Effect of CPU Sharing on Execution Time

In Section III.2 we described some of the reasons for using CPU sharing in multi-tenant, shared-CPU scenarios. In Section III.6.2, we demonstrated that multiplexing tightly coupled jobs with other jobs, such that during the communication cycles of one job the computation cycles of another can be performed, the makespan of the two jobs is reduced compared to running them sequentially. Hence, we deemed it necessary to implement a mathematical model for predicting execution time expansion due to CPU multiplexing.

We do this for both loosely coupled and tightly coupled jobs and assess the accuracy of the model empirically.

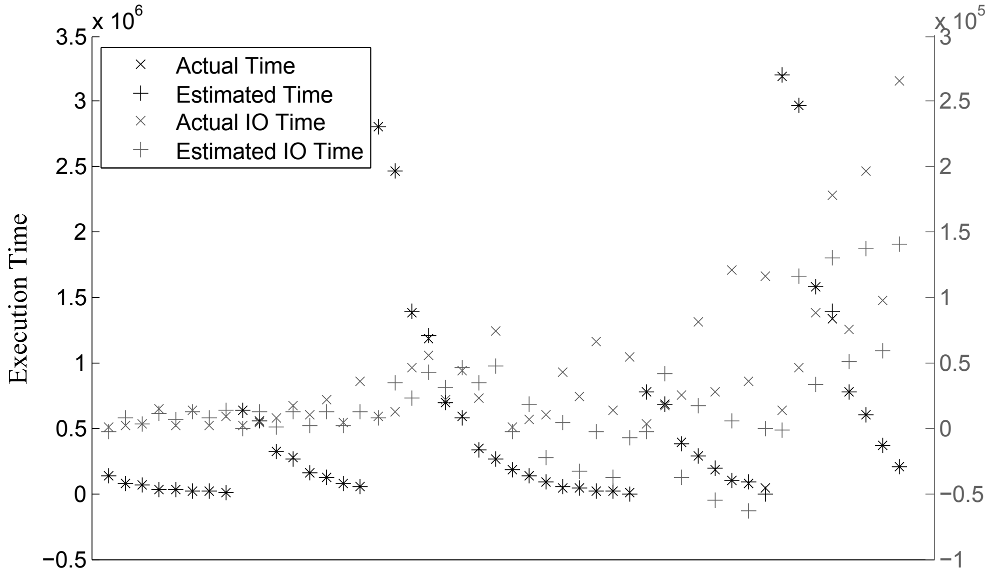


Figure 13. Predicted and actual CPU and I/O times for LU and LU-MZ.

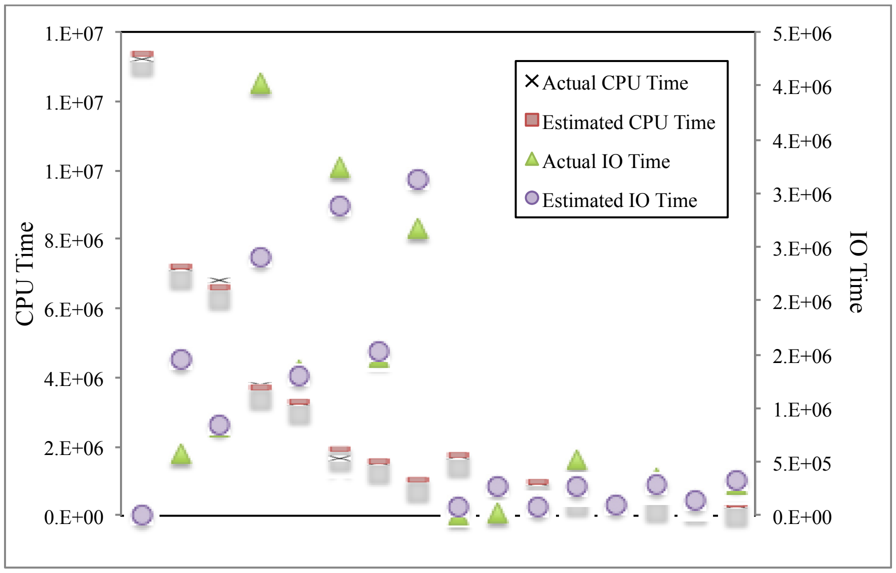


Figure 14. Predicted and actual computation and I/O times for WRF.

Before discussing the model itself, we discuss consistency and reproducibility issues that could hinder the accuracy of the model. In [79], the authors found that the Xen Credit

Scheduler can suffer from CPU allocation error, resulting in unfair load balancing when VMs are multiplexing the CPU. We did not encounter this particular problem when running identical parallel jobs, although we did note that the virtualization overhead caused by them was not included in the processor allocation decisions. For example, if a tightly coupled job requires 4% CPU for virtualization overhead and its multiplexing the CPU with a serial job, each job will only get roughly 48% of the CPU; the exact amount it gets is unpredictable. This needs to be accounted for in the model, as we discuss later.

A related issue is consistency. To determine if significant variation in execution time can be expected from multiplexed Xen executions, we ran 15 consecutive executions of *compcomm* and measured the durations of the computation and communication iterations. The number of iterations was set to 200. For the first set of tests, only one instance was run (i.e. no multiplexing). We then repeated it with a pair of 2-worker instances of *compcomm* running on 2 physical machines, so that in each physical machine, the 2 workers were multiplexing the CPU. Using analysis of variance (ANOVA), we found that communication cycles did not experience significant variation across runs or across iterations for both tests. The durations of the computation iterations were not normally distributed and we were unable to transform the data such that they would be, so ANOVA was not performed. Instead, we calculated the mean durations for the multiplexed and non-multiplexed executions. The mean remained at a consistent 67 milliseconds for non-multiplexed executions. When multiplexed, the mean varied between 86 and 93 milliseconds, since the amount of computation required before synchronizing can be different each time a job enters the working state. Considering the default 30 millisecond time slice used by Xen, the numbers seem reasonable.

An issue faced when predicting the multiplexed execution time of WRF and NPB is the fact that workers communicate at different frequencies and have different overall computation requirements. For example, the CPU time used by each worker of an 8-node WRF execution of the *jan00* domain varied between 150 and 220 seconds. To address this, we need to use the computational requirement of the worker(s) that are multiplexing the CPU, since the worker with the slowest execution rate will limit that of the others. This is depicted in Figure 15, where we show the CPU time required by each worker (using black circles) and the time required to execute the workload when one node is multiplexed (using gray asterisks). The figure shows the execution time for each of the eight possible multiplexed nodes. We see that the more CPU time the multiplexed worker needs, the more the overall execution time is prolonged, since the workers with less computational requirements must synchronize with them. Another issue is that the lengths of computation and communication iterations vary, but the variation was not significant enough to require the use of temporal requirements, i.e. the steady state execution pattern of all applications used was roughly constant with a small period of time (under a second).

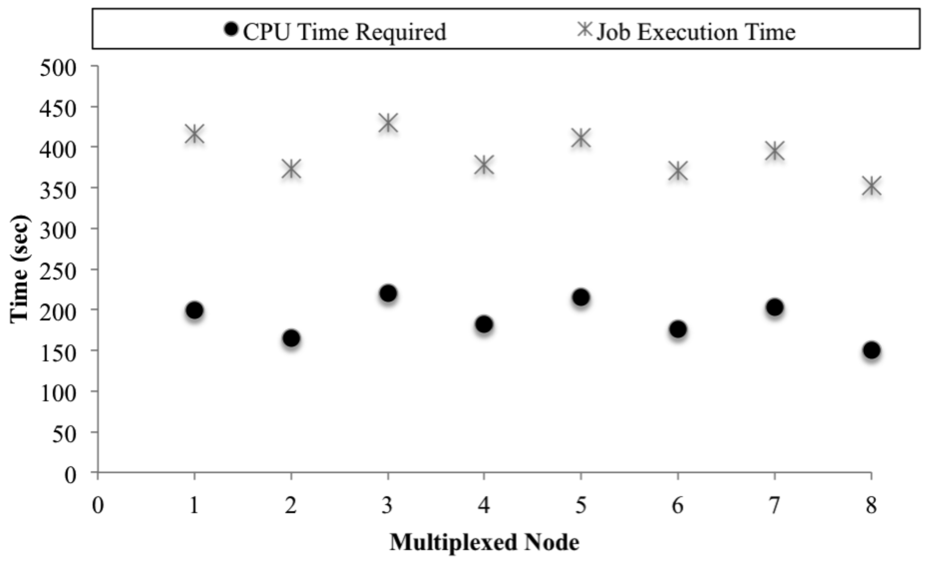


Figure 15. CPU time of different workers and execution time when multiplexing each one.

IV.5.1 Description of the Model

The model estimates the execution time of a job based on its computation and I/O requirements, the scheduling parameters of the VM it is executed on, its virtualization overhead, and the parameters of other VMs sharing the CPU. It assumes that the computation and I/O requirements are known from a previous execution and/or using a performance prediction model. We further assume that only coarse-grained computation and I/O requirements are known, since this data can be easily obtained using a lightweight resource monitor. An example of coarse-grained knowledge would be the total computation time of the job on a given system. Although it is possible to obtain relatively fine-grained resource consumption data with a lightweight performance monitor, the applications we studied have consistent steady state resource consumption behavior, so pursuing this task was deemed unnecessary.

The high level equation used to model execution time is shown in (6). We refer to the overall execution time as T_{exec} and separate it into non-collocated, non-multiplexed (*solo*) and collocated, multiplexed (*coll*) portions; the *solo* time is further separated into *active* and I/O portions. The *active* time is the portion of time in which the application can consume CPU cycles. The I/O portion is the time that it can only perform physical I/O because it is constrained by a *cap.* The computation portions include the time the application itself spends performing computation as well as the virtualization overhead. The I/O time includes the time spent physically transferring data. Note that the model assumes that while multiplexing, all I/O will take place while a job is in the non-working state. This is a safe assumption for the workloads used in the sense that their computation ratios are above 50% and their shares are never lower than 50%. One caveat is that there could be overlap of I/O cycles, as was shown in Figure 9b. This is not an issue when multiplexing with at least one serial job, since it can always use the CPU.

$$T_{exec} = T_{coll} + T_{solo,active} + T_{solo,IO} \tag{6}$$

We now describe the individual components of (6). For clarity, we describe the model for the case in which there are up to 2 multiplexed VMs per physical machine. In the formulas, we refer to the job being modeled as Job 1 (J_1) and the collocated job as Job 2 (J_2). In describing the equations, we assume that both jobs arrive at the same time, so they first execute collocated and when one of them completes, the other can then use the full CPU. This simplifies the explanation of the equations. In practice, jobs begin and end arbitrarily, so the formulas would use the remaining times instead of the overall times. The model assumes that communications can be performed during the non-working states and ignores the context switch overhead since we found it to be negligible in the experiments carried out in Section III.5.2.

First, we define the collocated computation rate (r_{coll}) in (7). Its value is the smaller of the *cap* of the VM the job is executed on and its net share relative to the collocated job. Its net share is the larger of its *cap* and its share, which in turn is based on its weight and that of the collocated job, as well as the collocated job's *cap.* For example, assuming both jobs have equal weight, if the collocated job is capped at ¼, the net share of the job is ¾.

$$r_{coll}^{j1} = \min\,[cap_{j1}, \max\left(100 \times \frac{weight_{j1}}{weight_{j1}+weight_{j2}}, 1 - cap_{j2}\right)] \times \frac{1}{100} \qquad (7)$$

T_{coll} is the wall clock time spent collocated; it is shown in (8), where t_{comp}^{jN} is the CPU time required for job N, including virtualization overhead. The equation assumes that a job will always have computations to perform when it is given the CPU, which implies that neither job has an r_{coll} larger than its *solo* computation rate. For example, if all jobs have equal weight, this implies that neither job has a computation rate below 50% for two jobs, 33% for 3 jobs, etc. This is a reasonable assumption given the computation rates of the jobs we tested in our experiments, as well as the data available in the CTC and SDSC workload traces of the parallel workloads archive [56]. Another caveat is that the formula assumes that communication overlap between the two jobs will not delay the execution. We observed only minor overlap, and did not expect this to affect accuracy significantly.

With these assumptions, the collocated time can be expressed as a function of the collocated computation rates (7) of the jobs and their computational requirements. The formula takes the lesser of the two jobs' computation times multiplied by their collocated execution rates.

$$T_{coll} = \min\left(t_{comp}^{j1} \times \frac{100}{r_{coll}^{j1}}, t_{comp}^{j2} \times \frac{100}{r_{coll}^{j2}} \right) \tag{8}$$

$T_{solo,active}$ is the time spent processing computations and communications while no other job is sharing the CPU. The equation is shown in (9) for job 1, where t_{comp} is the computation time of the job (excluding virtualization overhead), t_{virt} is the virtualization overhead, T_{coll} is the real (wall clock) time spent collocated, cap_{j1} is the cap of job 1, and r_{coll} is the computation rate (7) while collocated. For the virtualization overhead, t_{virt}^{j1} is the overhead observed executing job 1 and t_{virt}^{j1+j2} is the overhead observed executing jobs 1 and 2 with the same CPU. Basically, we subtract the computation portion of the job that was performed while collocated from the total known computation time, then account for additional virtualization overhead and additional execution time prolongation due to the *cap*.

$$T_{solo,active}^{j1} = \left[\frac{100}{cap_{j1}} \times \left(1 + t_{virt}^{j1}\right) \right] \times \left[t_{comp} - \frac{T_{coll} \times r_{coll}^{j1}}{1 + t_{virt}^{j1+j2}} \right] \tag{9}$$

$T_{solo,IO}$, shown in (10), is the additional time spent processing I/O while the job has a dedicated CPU. It accounts for cases where *cap*<100, in which some I/O can take place while the VM is forcibly put into the non-working state. The equation subtracts this time from the remaining wall clock time to determine the additional communication time required, if any. The remaining time is the difference between the (historical) communication time (t_{IO}) and the amount completed while collocated. The latter is the product of the job's communication ratio ($\frac{t_{IO}}{T}$) and the collocated time (T_{coll}). This value is then reduced by the idle time due to *cap*.

The CPU time required for virtualization overhead is small relative to the physical communication time of the parallel applications we experimented with (refer back to Tables 2 and 3), so we do not expect the *cap* to limit any communication from being performed during the non-working states since the VM will build credits while the physical transfer of the data is occurring.

$$T_{solo,IO} = \left(t_{io} - \frac{t_{io}}{T} T_{coll} \right) - \left(T_{solo,active} \times \frac{100 - cap}{100} \right) \tag{10}$$

IV.5.2 Model Evaluation

Table 6 shows the required CPU and I/O times for a number of empirical tests using WRF with the *jan00* input domain with different *cap* values, for 1 and 2 node executions. The prediction error, obtained using (11), is also shown. As observed in [19-20] for web workloads, virtualization overhead is predictable if the communication pattern is constant. We measured the mean virtualization overhead and applied it to (9) and (10) to determine the overall computation times.

$$error = \frac{PredictedRunTime - ActualRunTime}{ActualRunTime} \tag{11}$$

Executions with 1 and 2 nodes and with *cap* settings of 100 and 50 were performed. The execution times at 100% were used for the t_{IO} and t_{comp} values. The non-zero prediction error is due to sporadic virtualization overhead due to operating system noise. The results indicate that the model provides good estimates for the effect of different *cap* settings on execution time.

In Table 7, we show the times for multiplexed executions in which a node of a parallel job multiplexes the CPU with a serial job. Columns 1 and 4 indicate the physical nodes on which workers of the job executed. We vary the *cap* of each job, using values of 25, 50, 75, and 100. For the first 6 rows, the parallel job has 2 workers and the serial job is multiplexing the CPU with the first worker of the parallel job. The modeled execution time for all but five of these is within 4% of the actual time. These four outliers are due to the way Xen's Credit Scheduler distributes the CPU cycles. We found that it is biased towards the parallel job: it consumed 51% of the CPU *before* virtualization overhead (56% after the 5% consumed by the hypervisor for virtualization overhead) instead of the 50% it would be allotted if the scheduler distributed the capacity fairly. A similar problem was identified and a solution was given in [79], using Xen’s EDF scheduler. Since the model assumes each VM gets an equal share but the parallel job gets a larger share, its execution time is overestimated while that of the serial job is underestimated. Note that the most inaccurate estimates occur when there is no constraint on the capacity of the parallel job (i.e. *cap*=100). This is because when it does have a *cap*, the scheduler enforces the constraint for the job and virtualization overhead combined (i.e. the application itself gets less than its *cap*), which more accurately fits the model.

In each of the next 4 rows, the parallel job has 4 workers and each of these rows show the times for the case in which a different worker was multiplexing the CPU, starting with node 1 in row 7 and ending with node 4 in row 10. The estimated T_{coll} for all of these suffers due to the scheduler's allocation bias for the parallel job. The modeled T_{coll} accuracy for these executions varies significantly; when multiplexed on the first or third node, the modeled time is over predicted by 7 to 10% whereas when the second or fourth nodes are multiplexed, the predicted time is within 3%. This is because the second and fourth nodes have smaller computational loads than the others, so the bias for the parallel job is propagated less.

The last two rows show the results when job 2 has 8 workers. For the semi-last row, the first worker was multiplexed with the serial job and for the last row, the eighth worker was multiplexed. The results are similar to those of the 4-node experiment. When node 1 is multiplexing the two jobs, the estimate of the collocated execution time is accurate since the CPU allocation to the parallel job was 50% before virtualization overhead. However, the CPU allocation for the serial job was just 42% since 8% of the CPU was used for virtualization overhead, so the makespan of the serial job was underestimated by 11%.

To test the model while accounting for the bias towards parallel jobs, we modified the equation for the execution rate (7) to reflect our observation that the *Credit Scheduler* distributes the net capacity available after virtualization overhead, and accounting for the fact that the *cap* will be enforced including the virtualization overhead. The new (estimated) execution rate equation is shown in (12). Table 8 shows the results when using the updated formula. Note that the estimates for executions in which the *cap* is less than 100 are the same, so we indicate this by putting them in parenthesis. Now, when the parallel job has 2 workers we observe that the estimated times are almost all within 3%. The only exception is when the parallel job is capped at 75%, for which the overhead is off by 6%. With 4 workers, the estimates improve, although T_{coll} is still off by 6-7% when the second or fourth workers are multiplexed. With 8 workers, most times are underestimated significantly. This is because (12) is just a rough estimate of Xen's allocation. For example, we found that when there is significant virtualization overhead, the CPU capacity is not evenly distributed among the VMs.

$$r_{coll,adj} = \max\left[100 - cap, r_{coll} - (t_{virt}^{j1+j2} \times 100)\right] \tag{12}$$

Table 6. Model evaluation with 1 Job, no CPU sharing

#Nodes	Cap	CPU Time	I/O Time	Error(%)
1	100	1495	0	-0.18
1	50	1495	1559	-0.12
2	100	747	89	2.56
2	50	747	816	-1.55

Table 7. Model evaluation with 2 jobs sharing a CPU

Job 1 (Serial)			Job 2 (Parallel)		
Nodes	**Cap**	**Error(%)**	**Node(s)**	**Cap**	**Error(%)**
1	100	1.47	1,2	100	9.16
1	50	2.59	1,2	50	-2.69
1	50	-4.49	1,2	100	9.61
1	100	0.66	1,2	50	-2.97
1	25	-5.43	1,2	75	-5.94
1	75	3.50	1,2	25	-2.31
1	100	2.26	1-4	100	7.17
2	100	4.53	1-4	100	4.08
3	100	1.46	1-4	100	10.19
4	100	0.46	1-4	100	3.44
1	100	-9.22	1-8	100	3.40
8	100	1.31	1-8	100	-2.73

Table 8. Model evaluation with 2 jobs sharing the CPU and using the adjusted model

Job 1 (Serial)			Job 2 (Parallel)		
Nodes	**Cap**	**Error(%)**	**Node(s)**	**Cap**	**Error(%)**
1	100	1.40	1,2	100	2.02
1	50	(2.59)	1,2	50	(-2.69)
1	50	-1.35	1,2	100	2.44
1	100	(0.66)	1,2	50	(-2.97)
1	25	(-5.43)	1,2	75	(-5.94)
1	75	(3.50)	1,2	25	(-2.31)
1	100	1.24	1-4	100	-2.57
2	100	3.61	1-4	100	-4.29
3	100	0.45	1-4	100	0.17
4	100	-0.39	1-4	100	-4.82
1	100	-9.78	1-8	100	-9.30
8	100	0.94	1-8	100	-15.36

CHAPTER V

DEADLINE-DRIVEN DYNAMIC SCHEDULING

We now describe the job scheduling methodology used. As discussed in Chapter I, medical jobs often have deadlines. To know if a computing system can meet a job's deadline, it must be able to estimate whether the job can be scheduled such that it completes in time. In this chapter, we describe our job scheduling methodology, including our multi-objective scheduling algorithm that addresses the deadline satisfaction problem by harnessing the performance prediction methodology outlined in the previous chapter.

V.1 Design Overview

Our results in the previous chapter demonstrated that execution time predictions within 15% are possible when applying our prediction methodology to FAST and WRF. While more sophisticated models can be developed to reduce this error, a certain amount of error is unavoidable on modern systems due to their complex CPU architectures, distributed nature, etc. This creates a challenge for deadline satisfaction, so we went with a pragmatic approach when designing our scheduling methodology. Our system actively monitors a job's progress and when a deadline violation seems imminent under the current system state, additional resources are apportioned to the affected job(s) or it is migrated to a host with more free resources available.

We used a multi-objective scheduling approach. After deadline satisfaction, the next objective the scheduler satisfies is the maximization of resource utilization. The intent, in this case, is to allow as many jobs as possible so that the system is constantly loaded. The third objective is minimizing jobs' response times. Response time can be reduced by prioritizing short-duration jobs and by migrating tasks to maintain synchronized execution. However, maximizing throughput (in order to maximize utilization) tends to leave jobs running at just enough CPU allocation to finish before their deadline, negatively affecting response time.

V.2 Implementation

In order to satisfy all objectives and ensure that the system functions autonomously, several components were created to automatically determine if new jobs are schedulable, their optimal placement, availability of resources, and job status. We now discuss the implementation of these components and their interactions.

V.2.1 Tools

Amon and *Aprof* are used to monitor job status and predict resource requirements for new jobs. In addition, it is necessary to separately monitor the resource allocation of each VM, since virtualization overhead can result in a job receiving less net CPU capacity than it requires to complete and we observed non-intuitive CPU allocation with certain parallel applications. For this purpose, we developed another monitor, which we call *xhmon*, short for *Xen Hypervisor Monitor*. It is implemented as a Linux *daemon* that periodically monitors the resource utilization of VMs and listens for requests for specific utilization data. The information that can be requested from *xhmon* includes a VM's mean, median, minimum, or maximum CPU utilization, all of which are recalculated at discrete intervals.

V.2.2 Components

In this section, we discuss the steady state functionality of the four components of the scheduling methodology. Their names are *Predictor, Scheduler, Resource Manager,* and *Job Monitor*. All components are implemented as Linux *daemons*. The *Resource Manager* and *Job Monitor* update resource and job status parameters at discrete intervals. Since most jobs take several minutes to run, we use an interval of 60 seconds to maintain a reasonable monitoring overhead. The *Scheduler* is constantly listening for new job events, upon which it calls the *Predictor* to determine the job's computation requirements and subsequently whether or not it can be scheduled in time for its deadline. Now, we describe the individual components.

Predictor

The *Predictor* determines the resource requirements of new jobs. It can use either generic prediction parameters or application-specific parameters for improved accuracy. For the latter case, additional programming can be performed to extract pertinent information from the input data. The decision to use the application-specific parameters is made automatically when the name of the application pertaining to the job matches an application for which the *Predictor* has a specific prediction method. Based on our findings in Chapters III and IV, we bind the parameters *dimX, dimY, dimZ* for image processing applications to the *Predictor*. We use a third-party NIFTI library for reading these parameters from the input data programmatically. The *Predictor* distinguishes among the different parallel job types, such that for bags-of-tasks jobs each task's computation requirement is evaluated separately, whereas for tightly coupled jobs the overall job requirements with different levels of parallelism is predicted.

Scheduler

The *Scheduler* is responsible for matching jobs to resources in order to satisfy scheduling objectives. Resource requirements are queried from the *Predictor* and resource availability is queried from the *Resource Manager*. When there are multiple resources to choose from, different heuristics can be used to optimize scheduling performance. It also collects scheduling performance data, which include system utilization, deadline violation rates, and response times.

Figure 16a shows *pseudocode* for the two main functions carried out by the *Scheduler*, i.e. processing job arrivals and job completions. We defer describing the details of these functions until after describing the *Resource Manager* and *Job Monitor*, since they are involved in this functionality.

Resource Manager

The *Resource Manager* tracks the CPU and memory utilization of virtual machines and physical machines. It is also responsible for keeping a pool of VMs available on physical machines that can allocate new jobs without violating the deadlines of existing jobs, such that response times of new jobs can be decreased as described in Section III.2.

Figure 16b outlines the steady state functionality of the *Resource Manager*. This consists of 2 functions, VM *probing* and VM *deployment*. In the *probing* stage, the CPU consumption rates of VMs are probed using *xhmon.* Since virtualization overhead can impede a job's progress, each job's current and minimum execution rates are probed using the *Job Monitor'*s socket interface. Using this information, VM *slots,* which indicate points in time that VMs can be deployed and the amount of CPU capacity they can receive at these times, are created for each physical machine.

The concept of VM slots is depicted in Figure 17, where we show how the state of a physical machine initially running 3 jobs ($J_{1\text{-}3}$) on 3 VMs changes over time. The CPU allocation of each job is depicted using the height of the box it is enclosed in. The completion times of J_1 and J_2 at their current CPU allocation are $\widehat{T_1}$ and $\widehat{T_2}$, respectively. Initially, the full CPU capacity of the machine is required to ensure all remaining jobs finish before their deadlines. When J_1 completes, its share (of roughly 25%) becomes available, hence a *VM slot* of 25% CPU is created. Now, the slot can be used to run a new job or the other two VMs can use the excess capacity. Similarly, when J_2 completes at $\widehat{T_2}$, its slot of roughly 25% additional CPU capacity opens up. When J_3 completes at $\widehat{T}_3$, a slot with the full CPU capacity becomes available.

The current heuristic employed by the *Resource Manager* is as follows. If a job is receiving less CPU capacity than its minimum, it gets the available capacity in existing slots, up to its newly calculated minimum capacity. Theoretically, this should not happen, but in practice scheduling error can result in tasks getting less than their minimum. If there is still available capacity after accounting for this, a VM is created or migrated to the available slot so that later job arrivals can use it. This is what we refer to as the VM *deployment* functionality of the *Resource Manager*. If a job has exceeded its predicted computation time and is still running, all excess capacity is allocated to it, as this implies its execution time was underestimated and the possibility of a deadline violation is increased. The excess capacity is distributed among the running VMs.

Job Monitor

As its name implies, the *Job Monitor* keeps track of jobs' progress, particularly their CPU consumption progress and execution rates. It works with the *Resource Manager* to ensure jobs are getting enough resources to complete before their deadlines. The *Job Monitor* also attempts to minimize a job's response time. For example, for bags-of-tasks workloads, it attempts to balance jobs such that they finish at equal times.

An overview of the *Job Monitor's* functionality is shown using *pseudocode* in Figure 16c. At discrete intervals, each job's rate and CPU consumption progress is probed using

Amon's reporting interface. If a deadline violation is possible with the job's current *min* CPU allocation, the *min* value is increased. The *Job Monitor* merely updates this information; the *Resource Manager* is responsible for updating allocations based on the job's parameters. The job status can be queried by probing the *Job Monitor*'s socket interface, which returns the current and minimum execution rates for a given job.

V.2.3 Interaction Among Components

To better understand the scheduling methodology, we now discuss some additional details about the implementation in terms of how the components interact with each other. In Figure 18, we show a time line and the activities of each component from a job's arrival until its completion. The *Resource Manager* reallocates CPU to different VMs continuously at discreet intervals, based on updates from the *Job Monitor*, and updates slot availability accordingly. This is indicated in blue text in the figure. The first component to respond to a job arrival is the *Scheduler*. It obtains a prediction of a job's execution time from the *Predictor* (not shown). This requirement is sent to the *Resource Manager*, who updates its available slots and returns the list to the *Scheduler*. Assuming the job can complete before its deadline, it is assigned to a set of slots according to some scheduling heuristics. The *Resource Manager* is also responsible for allocating a VM on the physical machine, if necessary. This can be a new VM or a free VM can be migrated from another physical machine.

```
function job_arrival():
    predict_job_resource_requirements()
    determine_schedulability_from_available_resources()
    assign_resources_to_job()
function job_completion():
    unmap_resources()
    allocate_reserved_jobs()
```

(a)

```
function update_vm_slots():
    for each physical_machine:
        for each job_on_this_physical_machine:
            get_job_rate_and_min()
    if imminent_deadline_violation:
        reallocate_extra_capacity_to_job_in_danger
    update_slots_times_and_capacities()
    deploy_vms_for_available_slots
function probe_vms():
    for each VM:
        update resource utilization()
```

(b)

```
function monitor():
    probe_job_rate_and_progress()
    update vm_min_cpu()
```

(c)

Figure 16. Component overview: (a) Scheduler, (b) Resource manager, (c) Job Monitor

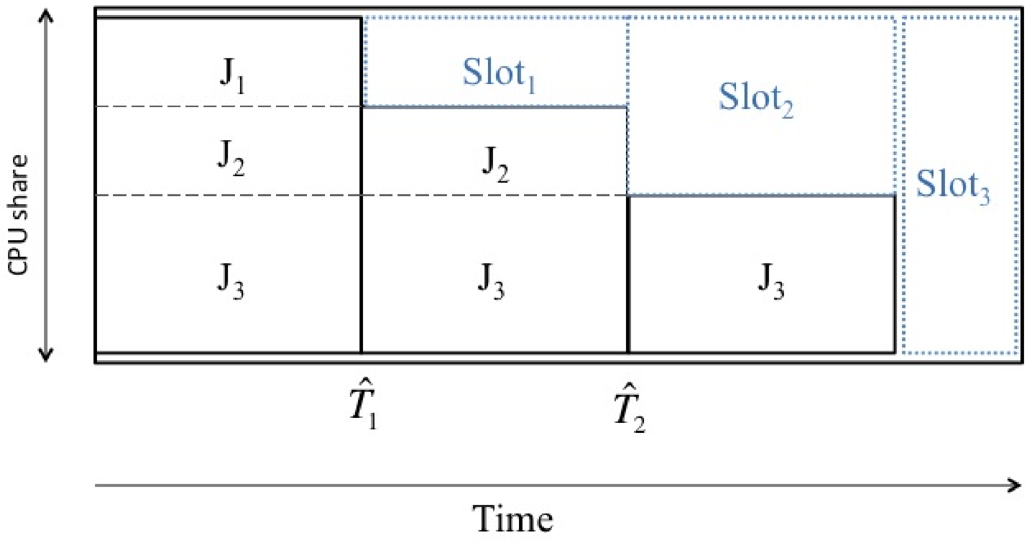


Figure 17. Slot availability at different times for a physical machine.

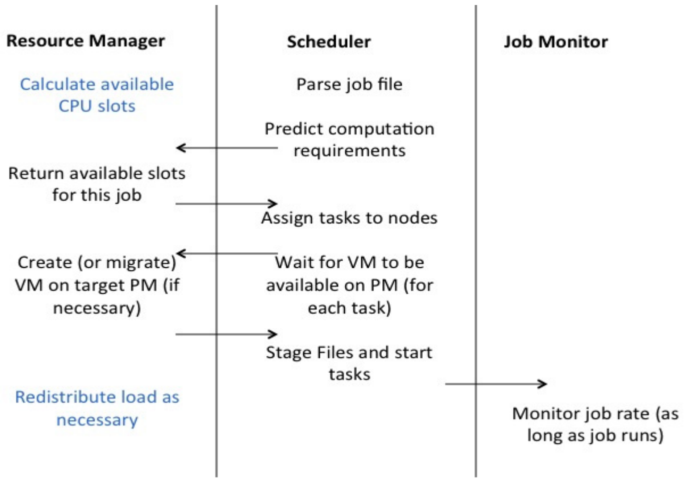

Resource Manager
Scheduler
Job Monitor
Calculate available CPU slots
Parse job file
Predict computation requirements
Return available slots for this job
Assign tasks to nodes
Create (or migrate) VM on target PM (if necessary)
Wait for VM to be available on PM (for each task)
Stage Files and start tasks
Redistribute load as necessary
Monitor job rate (as long as job runs)

Figure 18. Timeline: interaction of scheduling components during a job's lifecycle.

V.3 Scheduling Heuristics

The heuristics presented in this section are optimized for medical jobs, which resemble the *bags of tasks* model, where multiple tasks with different computation requirements are submitted together. Unlike tightly coupled jobs, the execution rates of each task are independent of each other. On the other hand, the results may not be useful until all tasks have finished so they benefit from synchronized execution.

Optimal job scheduling becomes computationally intractable as the number of tasks and machines increases, so heuristics must be employed to best meet the scheduling objectives. We use a best-fit based heuristic in making the job placement decisions, which places jobs on the resources that best fit its requirements. This can be visualized by thinking of tasks as moldable rectangles that need to be fit into different-sized bins. In Figure 19, we show how the execution time of an image segmentation task varies based on its CPU allocation/share. The dashed rectangles indicate three of the different shapes that the task can assume. Its height is equal to its CPU share and its width is equal to the time it takes to complete at a given CPU share. Since the task is CPU-constrained, the area of the rectangles is constant. The maximum width is the job's deadline. Looking back at Figure 17, we see a similar concept with slots. Hence, the objective of the best-fit algorithm is to match each task rectangle to the slot rectangle that provides the tightest fit.

By ensuring that the width of each task's rectangle does not exceed the job's deadline, we satisfy the first scheduling objective. Since we focus on CPU-bound medical applications, utilization is maximized by virtue of accepting as many jobs as possible, which implies strong execution time prediction accuracy. The third objective, minimizing response times, requires periodic monitoring of job execution rates, since even perfect initial job schedules can be disrupted by new job arrivals.

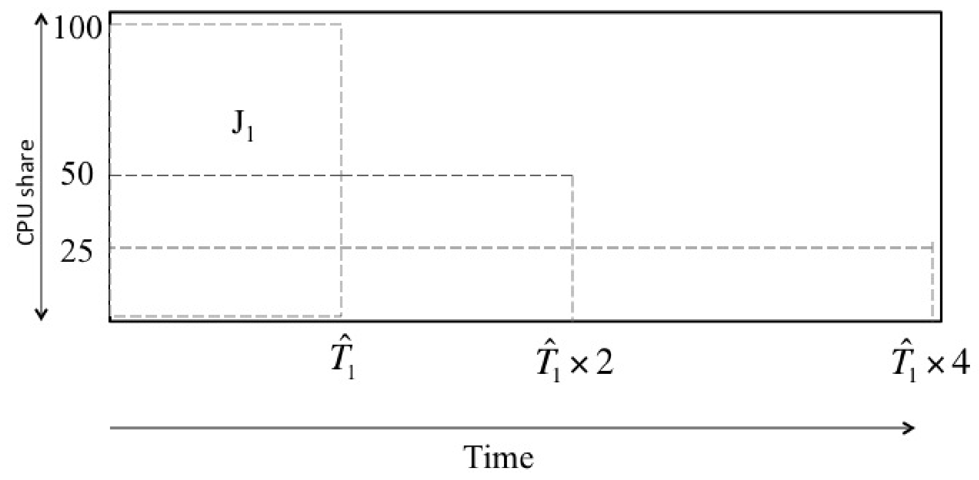


Figure 19. Assumed execution time model for image segmentation jobs.

V.4 Evaluation

The motivation for our scheduling methodology was to optimize scheduling behavior for incoming FSL jobs. While FSL provides the ability to automatically spread the workload using the Oracle *GridEngine* batch processing system, it does not provide any mechanisms for deadline satisfaction nor does it perform active load balancing as our approach does. Additionally, it does not automatically account for memory constraints, which resulted in out-of-memory errors when multiple image processing tasks execute on a single physical machine. Our baseline scheduler, therefore, is using *GridEngine* to process the workload. Doing so will compare how our overall scheduling methodology (i.e. virtualization and performance prediction) can improve scheduling performance. To compare our scheduling algorithm to another performance-aware scheduling algorithm, we use the popular first-come-first-serve-plus-backfill scheduling algorithm [15]. We use our *Predictor* to determine if jobs can be backfilled.

V.4.1 Workload

We use a workload consisting of 66 functional MRI data sets requiring image segmentation using FAST. Each of the 66 images is grouped into a separate job with a different arrival time and deadline. Jobs require between 4 and 10 tasks each. The times between arrivals of jobs also vary. We create random job arrival patterns and deadlines to mimic real world workloads. By using a relatively small workload, we can clearly explain the results obtained.

V.4.2 Scheduling Algorithms

- **GridEngine**: Uses FSL's built-in functionality to submit jobs via *GridEngine.* We do not use any of our scheduling components for this. Instead, we deployed *GridEngine* on *Mind*. We designate one VM as the *GridEngine* master and the rest as *GridEngine* execution hosts. Note that *GridEngine* is unaware of the underlying physical machine on which the VM runs.

- **FCFS**: Despite many advances in the scheduling literature, first-come-first-serve with backfill remains a popular choice for its simplicity and its balance of fairness and resource utilization. It works as follows. Jobs are processed in order of arrival. When a job requiring more nodes than are currently available arrives, it reserves a set of nodes able to run the job at a later time (provided that it can finish before its deadline). If smaller jobs arrive before this reservation time and can be scheduled such that the reservation of the large job is not violated, they are *backfilled,* i.e. they are scheduled before the large job. This algorithm does not consider partial CPU allocations, i.e. each task gets a full CPU to run on and physical machines running a job cannot be scheduled on until they complete the job. In order to determine if smaller jobs can be backfilled, FCFS traditionally depends on user-generated execution time estimates. For this test, we use our prediction methodology to predict the execution time. If a deadline cannot be satisfied, the job is skipped.

- **ElaDUR**: This is the scheduling algorithm we implement. Its name is short for Elastic-Deadline-Utilization-Response. It is based on the principles already discussed in this chapter: the resource allocation is elastic, such that multiple jobs can share a CPU as long as the deadlines of existing jobs are not jeopardized. Deadline-Utilization-Response is the list of scheduling objectives in order of priority.

Intuitively, using *GridEngine* would result in more deadline violations because it does not have any mechanisms for determining whether incoming jobs can complete before their deadlines. Also, it is absent of mechanisms to determine the physical CPU allocation that VMs have, so it may select VMs with less than a full CPU's capacity even if there are free CPUs available. This causes higher expansion factor and in turn a greater propensity to fail deadlines. On the other hand, *FCFS* does not allocate multiple VMs per physical machine, so if there are not enough idle physical machines available to schedule a job (either immediately or in the future) in time for its deadline to be met, it does not schedule it. *ElaDUR* affords more flexible allocations, which should result in more jobs being allowed into the system. Neither *FCFS* nor *ElaDUR* are expected to violate deadlines unless the execution time of a task is underestimated. To prevent deadline violations, we conservatively add 10% to the predicted execution time of each task, which is 3% more than maximum error observed in the experiment in Section IV.3.1, where we used our predictor to predict the computation time of FAST.

V.4.3 Results

The arrival time, deadline, and number of tasks of each job is shown in Table 9. The scheduling performance of each algorithm is shown in Table 10. The table shows the average utilization and expansion factors, as well as the number of deadline violations, the number of jobs processed, and the time elapsed between the first job arrival and the last job completion. The expansion factor is the ratio of the job's response time (completion time minus arrival time) to its computation time, i.e. it measures the job's response time relative to its computation time. We consider this a better measure of responsiveness than using only the response time, since longer jobs are less sensitive to response time delays.

The results align with the expectations summarized in the previous subsection. *GridEngine* processed all job, but in doing so violated the deadlines of 70% of them. The

VMs were initially deployed such that 6 physical machines were running one VM, 1 physical machine was running 2 VMs, and 1 physical machine was running 4 VMs. During the workload processing, for 2 jobs all 4 VMs on the latter physical machine was active, resulting in the tasks' execution times quadrupling while other physical machines remained idle. This was the culprit for the high average expansion factor and for some of the deadline violations.

FCFS had the lowest expansion factor because all tasks received a dedicated CPU, so only queuing delay contributes to the expansion factor and due to the mixture of job arrivals and deadlines, only one job could be queued with enough time left over to complete before its deadline.

ElaDUR only had to turn down 1 job, so it enjoyed a higher average utilization and job processing rate. Note that its performance corresponded with its scheduling objectives: there were no deadline violations, utilization was kept high, but expansion factor was higher than *FCFS* because certain jobs received a small amount of CPU in order to accommodate the deadlines of other jobs on the same physical machine.

Due to the relatively long gap in job arrivals between the 9th and 10th jobs, ElaDUR and FCFS finished the last job at roughly the same time. However, the cluster was idle for longer periods of time during the workload processing when using FCFS since ElaDUR processed more jobs.

Table 9. Parameters of jobs used for evaluating the scheduling algorithm

No.	**Arrival time (min.)**	**#Tasks**	**Deadline (min.)**
1	0	8	15
2	20	10	15
3	50	8	40
4	55	4	20
5	67	4	100
6	68	4	10
7	70	4	10
8	80	8	10
9	81	8	20
10	121	8	10

Table 10. Performance comparison of the 3 scheduling algorithms

Scheduler	**Utilization**	**Exp. Factor**	**Violations**	**Jobs processed**	**Completion of last job (minutes)**
GridEngine	53%	2.4	7	10	186
FCFS	55%	1.1	0	7	131
ElaDUR	72%	1.5	0	9	130

CHAPTER VI

CONCLUSION AND FUTURE WORK

The work discussed in this dissertation harnesses modern advances in virtualization technology to address the issue of deadline-driven job scheduling. Since the performance of a given job scheduling algorithm is dependent on the arrival patterns and applications of the workload being processed, we focused our work on a specific application that would benefit our medical collaborators as well as researchers in the lab. Throughout the dissertation, however, we provided additional insight into how the findings made throughout this work could be extended to other scientific applications (e.g. fluid dynamics). This insight was provided in the form of extensive performance analyses and performance models for these applications.

To this end, we looked into three pertinent issues. First, recognizing the need for performance modeling in order to satisfy scheduling deadlines, we started with an in-depth analysis of the performance of different scientific applications via empirical evaluation on a compute cluster. Since virtualization provides key benefits for resource provisioning, we went on to explore the effects of virtualization on scientific workloads. This included studying the overhead caused by the virtualization software itself as well as the impact of CPU sharing on application performance, since it is common to pack multiple virtual machines on the available physical machines.

Among our findings, we confirmed that the performance of typical medical image processing workloads consisting of a large amount of independent tasks is not affected significantly by virtualization. In terms of CPU sharing, the tasks scaled proportionally to the share of CPU they were given. By virtue of this, a linear scalability model could be used, which is ideal for making real time scheduling decisions, since this kind of model can be implemented using computationally simple algorithms. Using a performance prediction model based on regression analysis, we were able to predict the scalability of tightly coupled parallel applications with an average error of 15% and the computation time of individual image segmentation tasks to within 7% for different-sized images.

We then applied the performance prediction model to a deadline-driven job scheduling methodology. We developed several components to enable job scheduling on virtual machines combined with autonomous resource management to ensure deadlines are satisfied while maximizing utilization and minimizing response time. Through our collaboration with a consortium of hospitals, we obtained 66 sets of fMRI image data of different sizes to process in order to evaluate our scheduling algorithm. The scheduling algorithm was compared to a current solution for batch scheduling image processing jobs and to a traditional, but virtual machine aware first-come-first-serve scheduling algorithm. We found that our scheduling algorithm processed more jobs without jeopardizing any deadlines. It also utilized the available resources significantly better than the other two algorithms.

The fact that our algorithm performed better confirms the benefits of virtualization in terms of job scheduling discussed early on in this dissertation. We observed no performance impact from virtualization on the workloads used for the scheduling evaluation, in fact we found that virtualized executions can *outperform* regular executions, which suggests that further work should go into developing production environments for virtualization-aware scientific job scheduling. Our observations and models provide additional insight for doing this, which we consider an interesting direction for future work.

Additional future work could consist of further refinements to the prediction model itself and more optimizations to the scheduling algorithm. Specifically, migration can be harnessed to further improve resource utilization and/or other goals such as energy efficiency. Another direction would be to look into resource federation. Currently, our scheduler rejects jobs for which there are not enough resources to satisfy deadlines. An alternative is to allow federation of resources from other administrative domains to lease external resources when local resources are not adequate, as long as they can provide a performance guarantee.

LIST OF REFERENCES

[1] L. M. Vaquero, L. R. Merino, J. Caceres, and M. Lindner, “A break in the clouds: towards a cloud definition,” *In proc. SIGCOMM Comput. Commun.* Rev. 39, 1, pp.50-55, Seattle, WA, USA, Aug. 2008.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing". Technical Report EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[3] I. Foster, Y. Zhao, I. Raicu, S. Lu, “Cloud Computing and Grid Computing 360-Degree Compared,” *In proc. Grid Computing Environments Workshop (GCE '08),* pp. 1-10, Austin, Texas, USA, Nov. 2008.

[4] C. Catlett, “TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications,” *HPC and Grids in Action*, Ed. Lucio Grandinetti, IOS Press 'Advances in Parallel Computing' series, Amsterdam, 2007.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization.” *In proc. 19th ACM symposium on Operating Systems Principles (SOSP '03)*, pp. 164-17, Bolton Landing, New York, USA, Oct. 2003.

[6] VMware Inc. 2006. VMware ESX server: Platform for virtualizing servers, storage and networking;

[7] Google Inc. 2012, Google App Engine [URL], http://code.google.com/appengine/

[8] W. Gentzsch, “HPC in the Cloud: use cases from research and industry,” Presentation given at 4th annual Utility and Cloud Computing Conference, Dec. 2012.

[9] G. E. Moore (1965). "Cramming more components onto integrated circuits" (PDF). Electronics Magazine. p. 4.

[10] O. Sonmez, N. Yigitbasi, A. Iosup, and D. Epema, “Trace-based evaluation of job runtime and queue wait time predictions in grids,” *In proc. 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09),* pp. 111-120, Munich, Germany, Jun. 2009.

[11] D. Tsafrir and D.G. Feitelson, “Instability in parallel job scheduling simulation: the role of workload flurries,” *In proc. 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, pp. 73-73, Rhodes Island, Greece, Apr. 2006.

[12] J. K. Ousterhout, “Scheduling Techniques for Concurrent Systems,” *In proc. 3rd International Conference on Distributed Computing Systems (ICDCS)*, pp. 22–30, Fort Lauderdale, Florida, USA, Oct. 1982.

[13] M. Stillwell, F. Vivien, H. Casanova. “Dynamic Fractional Resource Scheduling for HPC Workloads.” *In proc. 2010 International Parallel and Distributed Processing Symposium (IPDPS '10)*, Atlanta, Georgia, USA, Apr. 2010.

[14] M. Stillwell, F. Vivien, H. Casanova, "Dynamic Fractional Resource Scheduling versus Batch Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 3, pp. 521-529, Mar. 2012.

[15] J. Skovira, W. Chan, H. Zhou, D. Lifka, “The EASY - LoadLeveler API Project,” *Proc. Job Scheduling Strategies for Parallel Processing Systems (JSSPP ’96)*, pp. 41-47, 1996.

[16] D. Wall, P. Kudtarkar, V. Fusaro, R. Pivovarov, P. Patil, and P. Tonellato. “Cloud computing for comparative genomics.” *BMC Bioinformatics*, vol. 11, no. 259, May 2010.

[17] R. Barga, D. Gannon, D. Reed, “The Client and the Cloud: Democratizing Research Computing,” *IEEE Internet Computing*, vol. 15, no. 1, pp. 72-75, Feb. 2011.

[18] Amazon Elastic Compute Cloud [URL]. Amazon (2012), http://aws.amazon.com/solutions/case-studies.

[19] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, “Profiling and modeling resource usage of virtualized applications,” *In proc. 9th ACM/IFIP/USENIX International Conference on Middleware* (Middleware '08), pp. 366-387, Leuven, Belgium, Dec. 2008.

[20] L. Cherkasova and R. Gardner, “Measuring CPU overhead for I/O processing in the Xen virtual machine monitor,” *In proc. USENIX Annual Technical Conference (ATEC '05)*, Anaheim, California, USA, Apr. 2005.

[21] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, “Paravirtualization for HPC Systems,” *Workshop on XEN in HPC Cluster and Grid Computing Environments*

(XHPC), held in conjunction with The International Symposium on Parallel and Distributed Processing and Application (ISPA ‘06), Sorrento, Italy, Dec. 2006.

[22] W. Huang, J. Liu, B. Abali, and D. K. Panda. “A case for high performance computing with virtual machines.” *In proc. 20th annual international conference on Supercomputing (ICS '06),* pp. 125-134, Queensland, Australia, Jun. 2006.

[23] A. Tikotekar, G. Valle, T. Naughton, H. Ong, C. Engelmann, and S. L. Scott, “An Analysis of HPC Benchmarks in Virtual Machine Environments,” *In proc. Euro-Par 2008 Workshops - Parallel Processing,* Gran Canaria, Spain, Aug. 2008.

[24] P. M. Papadopoulos, M. J. Katz, and G. Bruno. 2001. “NPACI Rocks Clusters: Tools for Easily Deploying and Maintaining Manageable High-Performance Linux Clusters.” *In proc. 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, London, UK, 10-11.

[25] N. Regola and J. C. Ducom, “Recommendations for Virtualization Technologies in High Performance Computing,” *In proc. 2nd International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 409-416, Athens, Greece, Dec. 2011.

[26] D. E. Lang , T. K. Agerwala , K. M. Chandy, “A modeling approach and design tool for pipelined central processors,” *In proc. 6th annual symposium on Computer architecture*, pp. 122-129, 1979.

[27] S. Kounev, “Performance Prediction, Sizing and Capacity Planning for Distributed E-Commerce Applications,” Technical report, Technische Universität Darmstadt, Germany, Jan. 2001.

[28] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni, “A regression-based analytic model for capacity planning of multi-tier applications. *In proc. 11th Cluster Computing*, pp. 197-211, Tsukuba, Japan, Sep. 2008.

[29] D. Wall, P. Kudtarkar, V. Fusaro, R. Pivovarov, P. Patil, and P. Tonellato. “Cloud computing for comparative genomics.” *BMC Bioinformatics*, vol. 11, no. 259, May 2010.

[30] W. Huang, M. Koop and D.K. Panda, “Efficient One-Copy MPI Shared Memory Communication in Virtual Machines,” *In proc. 11th IEEE Cluster, Tsukuba, Japan,* Sep. *2*008.

[31] T. Chen, M. Gunn, B. Simon, L. Carrington, and A. Snavely, "Metrics for ranking the performance of supercomputers," *Cyberinfrastructure Technology Watch Journal: Special Issue on High Productivity Computer Systems,* vol. 2. Feb. 2007.

[32] W. Smith, I. T. Foster, and V. E. Taylor, "Predicting application run times using historical information," *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 122-142, 1998.

[33] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris, "DiP: A parallel program development environment," *In proc. 2nd International EuroPar Conference on Parallel Processing*, pp. 665-674, Lyon, France, Aug. 1996.

[34] R. M. Badia, F. Escalé, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. S. Muller, "Performance prediction in a grid environment," *In proc. 1st European Across Grids Conference*, pp. 257-264, 2003.

[35] D. Katramatos and S.J. Chapin, "A Cost/Benefit Estimating Service for Mapping Parallel Applications on Heterogeneous Clusters," *In proc. 7th IEEE International Conference on Cluster Computing*, pp. 1-12, Osaka, Japan, Sep. 2005.

[36] H. Rasheed, R. Gruber, and V. Keller, "IANOS: An intelligent application oriented scheduling middleware for a HPC grid". CoreGRID Tech. Rep. TR-0110, 2007.

[37] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," *In proc. 18th Supercomputing Conference*, pp. 40–49, Seattle, Washington, Nov 2005.

[38] D. Schanzenbach and H. Casanova, "Accuracy and responsiveness of CPU sharing using Xen's cap values," Technical Report ICS2008-05-01, Computer and Information Sciences Dept., University of Hawai'i at Manoa, 2008.

[39] B. Urgaonkar , P. Shenoy , T. Roscoe, "Resource overbooking and application profiling in a shared Internet hosting platform," *ACM Transactions on Internet Technology (TOIT)*, vol. 9, no. 1, pp. 1-45, Feb. 2009.

[40] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Capacity Management and Demand Prediction for Next Generation Data Centers," *In proc. International Conference on Web Services (ICWS),* pp. 43-50, Salt Lake City, Utah, USA, Jul. 2007.

[41] D.G. Feitelson and L. Rudolph, “Gang Scheduling Performance Benefits for Fine-Grain Synchronization,” *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306-318, 1992.

[42] A. Gupta, A. Tucker, and S. Urushibara, “The impact of operating system scheduling policies and synchronization methods of performance of parallel applications,” *In proc. 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (SIGMETRICS '91), pp. 120-132, 1991.

[43] S. T. Leutenegger and M. K. Vernon, “The performance of multiprogrammed multiprocessor scheduling algorithms,” *SIGMETRICS Perform. Eval., Rev.* 18, 1, pp. 226-236, 1990.

[44] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, “Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing,” *IEEE Trans. on Parallel and Distributed Systems,* vol. 22, no. 6, pp. 931-945, Feb/Apr 2011.

[45] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, “Parallel job scheduling — a status report,” *In proc. 10th international conference on Job Scheduling Strategies for Parallel Processing (JSSPP'04)*, Springer-Verlag, Berlin, Heidelberg, pp. 1-16. 2004.

[46] D. Gupta, L. Cherkasova, R. Gardner, A. Vahdat, “Enforcing Performance Isolation Across Virtual Machines in Xen,” HP Laboratories Report No. HPL-2006-77, 2006.

[47] H. Casanova, M. Gallet, and F. Vivien, “Non-clairvoyant scheduling of multiple bag-of-tasks applications,” *In proc. 16th international Euro-Par conference on Parallel processing*, pp. 168-179, Caparica, Portugal, Sep. 2005.

[48] G. Katsaros, R. Kübert, G. Georgina, T. Wang, “Monitoring: A Fundamental Process to Provide QoS Guarantees in Cloud-based Platforms,” *In Cloud Computing: Methodology, System, and Applications*, CRC Press, Aug. 2011.

[49] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, “Are user runtime estimates inherently inaccurate?,” *In 10th Workshop on Job Scheduling Strategies for Parallel Processing*, 2004.

[50] D. Tsafrir, Y. Etsion, and D. G. Feitelson. “Backfilling using system-generated predictions rather than user runtime estimates,” *IEEE Trans. Par. Distr. Sys.*, 18:789–803, 2007.

[51] S. Pandey, W. Voorsluys, S. Niu, A. Khandoker, R. Buyya, "An autonomic cloud environment for hosting ECG data analysis services," *Future Generation Computer Systems,* Volume 28, Issue 1, pp. 147-154, ISSN 0167-739X, Jan. 2012.

[52] C. Vecchiola, X. Chu, and R. Buyya, "Aneka: A Software Platform for .NET-based Cloud Computing, High Speed and Large Scale Scientific Computing," pp. 267-295, ISBN: 978-1-60750-073-5, IOS Press, Amsterdam, Netherlands, 2009.

[53] H. Kim; M. Parashar, D.J. Foran, L. Yang, "Investigating the use of autonomic cloudbursts for high-throughput medical image registration," *In proc. 10th IEEE/ACM Conference on Grid Computing*, pp.34-41, Banff, AB, Canada, Oct. 2009.

[54] H. Kim, S. Chaudhari, M. Parashar, and C. Marty, "Online risk analytics on the cloud," *In proc. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (*CCGRID '09), pp. 484-489, Shanghai, China, May 2009.

[55] Distributed Systems Architecture Research Group: OpenNEbula Project [URL]. Universidad Complutense de Madrid (2009), http://www.opennebula.org.

[56] The parallel workloads archive, http://www.cs.huji.ac.il/labs/parallel/workload. 2012.

[57] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. "KVM: the Linux virtual machine monitor,“ *In The 2007 Ottawa Linux Symposium*, pp. 225-230, Jul. 2007.

[58] D. Chisnall, "The Definitive Guide to the Xen Hypervisor," (Prentice Hall Open Source Software Development Series), Prentice Hall PTR, Upper Saddle River, NJ, 2007.

[59] G. Dunlap, "Xen Scheduler Update," unpublished whitepaper, presented at Xen Summit Asia, Nov. 2009.

[60] S. M. Smith, M. Jenkinson, M.W. Woolrich, C.F. Beckmann, T.E.J. Behrens, H. Johansen-Berg, P.R. Bannister, M. De Luca, I. Drobnjak, D.E. Flitney, R. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. De Stefano, J.M. Brady, and P.M. Matthews. "Advances in functional and structural MR image analysis and implementation as FSL." *NeuroImage*. 23(S1), 2004, 208-219.

[61] P. Jezzard, P. M. Matthews, and S. M. Smith. "Functional MRI: An introduction to methods." England: Oxford University Press.

[62] Y. Zhang, M. Brady, and S. Smith. “Segmentation of brain MR images through a hidden Markov random field model and the expectation maximization algorithm.” *IEEE Trans. Med. Imag.*, January 2001, 20 (1), 45-57.

[63] M. Jenkinson, P. R. Bannister, J. M. Brady, and S. M. Smith, “Improved optimization for the robust and accurate linear registration and motion correction of brain images.” *NeuroImage,* 2002, 17 (2), 825-841.

[64] C. F. Beckmann and S. M. Smith. “Probabilistic independent component analysis for functional magnetic resonance imaging.” *IEEE Trans. on Med. Imag.*, 23(2): 137-152, 2004.

[65] D. Bailey, E. Barscz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinkski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, “The NAS Parallel Benchmarks,” (March 1994) NAS Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA.

[66] R. van der Wijngaart and H. Jin, "NAS Parallel Benchmarks, Multi-Zone Versions,” (July 2003) NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA.

[67] H. Jin, R. Van der Wijngaart, "Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks," *In proc. 18th International Parallel and Distributed Processing Symposium (IPDPS 04)*, Santa Fe, New Mexico, USA, Apr. 2004.

[68] A. Matsunaga, M. Tsugawa, and J. Fortes. “CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications,” *In proc. Fourth IEEE International Conference on eScience (ESCIENCE '08)*, pp. 222-229, Indianapolis, Indiana, USA, Dec. 2008.

[69] S. F. Altschul, W. Miller, E.W. Myers, and D.J. Lipman, "Basic local alignment search tool," *J Mol Biol,* vol. 215, no. 3, pp. 403–410.

[70] T. Cortes V. Pillet, J. Labarta, and S. Girona, “Paraver: A tool to visualize and analyze parallel code,” *In WoTUG-18*, pp. 17–31, Manchester, U.K., Apr. 1995.

[71] S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo, “A modeling approach for estimating execution time of long-running scientific applications.” *In Proc. 22nd IEEE International Parallel & Distributed Processing Symposium, the Fifth High-Performance Grid Computing Workshop,* Miami, FL, USA, 2008.

[72] Coregrid Network of Excellence Deliverable Number D.RMS.06, “Review of performance prediction models and solutions,” 2006.

[73] A. Darling, L. Carey, and W. Feng, “The Design, Implementation, and Evaluation of mpiBLAST,” *In proc. 4th International Conference on Linux Clusters: The HPC Revolution,* June 2003.

[74] C. Clémençon, K. M. Decker, V. R. Deshpande, A. Endo, J. Fritscher, P. R. Lorenzo, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B. J. N. Wylie, F. Zimmermann, “HPF and MPI Implementation of the NAS Parallel Benchmarks Supported by Integrated Program Engineering Tools,” *In proc. Parallel and Distributed Computing Systems (PDCS)*, Chicago, Illinois, USA, Oct. 1-4, 1996.

[75] J. Delgado, S. M. Sadjadi, H. A. Duran-Limon, M. Bright, and M. Adjouadi, “Performance prediction of weather forecasting software on multicore systems,” *In proc. 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS-2010), 11th Parallel and Distributed Scientific and Engineering Computing (PDSEC) workshop*, Atlanta, Georgia, USA, April 2010.

[76] J. Michalakes, J. Hacker, R. Loft, M. O. McCracken, A. Snavely, N. J. Wright, T. Spelce, B. Gorda, R. Walkup, "WRF nature run," *In. proc. 20th ACM/IEEE conference on Supercomputing*, pp. 1-6, Reno, Nevada, USA, Nov. 2007.

[77] J. Delgado, A. S. Eddin, M. Adjouadi, and S. Masoud Sadjadi. “Paravirtualiztion for Scientific Computing: Performance Analysis and Prediction,” *In proc. 2011 IEEE International Conference on High Performance Computing and Communications (HPCC ’11)*, pp. 536-543, Banff, AB, Canada, September 2011.

[78] M. Ben-Yehuda. “The Xen Hypervisor and its IO Subsystem,” Presentation given at the 2005 IBM Systems and Storage Seminar, Haifa, Israel, Dec. 2005. Available at: *http://www.research.ibm.com/haifa/Workshops/systems-and-storage2005/papers/xen-io.pdf.*

[79] L. Cherkasova, D. Gupta, and A. Vahdat, “When Virtual is Harder than Real: Resource Allocation Challenges in Virtual Machine Based IT Environments,” HP Laboratories Report No. HPL-2007-25, 2007.

VITA

JAVIER DELGADO

2004	B.S., Computer Engineering Florida International University Miami, FL
2007	M.S., Computer Engineering Florida International University Miami, FL
2011	Ph.D. Candidate, Electrical Engineering Florida International University Miami, FL

PUBLICATIONS

1. J. Delgado, L. Fong, Y. Liu, N. Bobroff, S. Seelam, and M. Sadjadi, “Efficiency Assessment of Parallel Workloads on Virtualized Resources,” *In proc. 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pp.89-96, Melbourne, Australia, December 2011.

2. J. Delgado, A. S. Eddin, M. Adjouadi, and S. Masoud Sadjadi. “Paravirtualiztion for Scientific Computing: Performance Analysis and Prediction,” *In proc. 2011 IEEE International Conference on High Performance Computing and Communications (HPCC ’11)*, pp. 536-543, Banff, AB, Canada, September 2011.

3. J. Delgado, S. M. Sadjadi, H. Duran, M. Bright, and Malek Adjouadi, “Performance prediction of weather forecasting software on multicore systems,” *In Proc. 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS-2010), 11th Parallel and Distributed Scientific and Engineering Computing (PDSEC) workshop*, Atlanta, Georgia, USA, April 2010.

4. J. Delgado and M. Adjouadi. "On the Efficacy of Present Grid Computing Software for Deploying a Medical Grid," *In proc. 2009 Richard Tapia Celebration of Diversity in Computing,* pp. 81-86, Portland, Oregon, USA, April 2009.

5. J. Delgado and M. Adjouadi, "Towards an Efficient and Extensible Grid-Based Data Storage Solution," *In proc. 22nd IEEE International Conference on Advanced Information Networking and Applications,* pp. 659-666, Okinawa, Japan, March 2008.

6. J. Delgado, M. R. Guillen, M. Lahlou, and M. Adjouadi, "MIND: A Tiled Display Visualization System at CATE/FIU," *In proc. IASTED International Conference on Graphics and Visualization in Engineering (GVE 2007),* Clearwater, Florida, USA, January 2007.

7. S. M. Sadjadi, S. Shimizu, J. Figueroa, R. Rangaswami, J. Delgado, H. Duran, and X. Collazo, "A Modeling Approach for Estimating Execution Time of Long-Running Scientific Applications," *In proc. Fifth High-Performance Grid Computing Workshop*, pp. 1-8, Miami, Florida, USA, April 14, 2008.

8. D. Villegas, N. Bobroff, I. Rodero, J. Delgado, Y. Liu, A. Devarakonda, L. Fong, S. M. Sadjadi, and M. Parashar, "Cloud federation in a layered service model," *Journal of Computer and System Sciences*, Available online, ISSN 0022-0000, 10.1016/j.jcss.2011.12.017, January 2012.

SUBMITTED

1. J. Delgado, A. S. Eddin, S. M. Sadjadi, and M. Adjouadi, "Issues Faced Running Scientific Applications on Multi-user Virtualized Systems," Submitted to IEEE Transactions on Parallel and Distributed Systems.