UNIVERSITY OF CALIFORNIA, SAN DIEGO

Automatic Resource Specification Generation for Resource Selection in Large-Scale Distributed Environments

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in

Computer Science

by

Richard Yu-Hua Huang

Committee in charge:

Professor Andrew A. Chien, Chair
Professor Francine Berman
Professor Henri Casanova
Professor Rene Cruz
Professor Jeanne Ferrante
Professor Tara Javidi

2007

The Dissertation of Richard Yu-Hua Huang is approved, and it is acceptable in quality and form for publication on microfilm.

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2007

For my parents and brother, who made this possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Andrew A. Chien for his continuous support and encouragements over the past four years and for his advice and insights on different research problems. I am also most grateful for my co-advisor, Professor Henri Casanova, who is always there for me every step of the way, whether discussing minute details of my experiments or collaborating on conference papers.

I also would like to thank Professor Francine Berman, Professor Jeanne Ferrante, Professor Rene Cruz, and Professor Tara Javidi for agreeing to serve on my doctoral committee and providing helpful feedback on my dissertation.

I am also grateful for everyone in the Concurrent Systems Architecture Group (CSAG) at UCSD. I want to thank those who were there before me and provided guidance for my doctoral work: Luis Rivera, Xin Liu, Ju Wang, Huaxia Xia, and Ryan Wu. I thank those who worked together with me on the Virtual Grid Application Development Software (VGrADS) project: Yang-Suk Kee, Kenneth Yocum, Jerry Chou, and Dionysios Logothetis. I also learned much from my fellow CSAG members Nut Taesombut, Justin Burke, Eric Weigle, Ryo Sugihara, Han-Suk Kim, and Jing Zhu. I also want to thank the CSAG staff members for their help over the years: Patricia Bladh, Alex Olugbile, Troy Chuang, Adam Brust, and Jenine Combs.

I was fortunate to have the opportunity to collaborate with many wonderful individuals on the VGrADS project from other institutions including Professor Ken Kennedy, Professor Keith Cooper, and Chuck Koelbel from Rice University; Professor Carl Kesselman from the USC/ISI; Professor Rich Wolski from the UCSB; Professor Dan Reed from the UNC; Professor Jack Dongarra from the University of Tennessee.

I would like to thank everyone who has helped me during my years of graduate studies at the University of California, San Diego.

I would like to thank Steven Lee for near daily chats on sports and running a startup. I would like to thank Lastly, I would like to thank Winnie Lee for keeping me company when I am not from doing research and making sure that I get my proper nutrients.

Chapter IV, in part, has been published as “Using Virtual Grids to Simplify Application Scheduling” by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006). The dissertation author was the primary investigator and author of this paper.

Chapter V, in part, has been published as “Generating Grid Resource Requirement Specifications” by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC 2007). The dissertation author was the primary investigator and author of this paper.

Chapter V, in part, has been submitted for publication and will appear as “Automatic Resource Specification Generation for Resource Selection” by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the ACM/IEEE International Conference on High Performance Networking and Computing (SC 2007). The dissertation author was the primary investigator and author of this paper.

VITA

1997	Bachelor of Science, Biology, Massachusetts Institute of Technology
1998	Bachelor of Science, Electrical Engineering and Computer Science, Massachusetts Institute of Technology
2005	Master of Science, Computer Science, University of California, San Diego
2007	Doctor of Philosophy, Computer Science, University of California, San Diego

PUBLICATIONS

Richard Huang, Henri Casanova, and Andrew A. Chien. Automatic Resource Specification Generation for Resource Selection., ACM/IEEE International Conference on High Performance Networking and Computing (SC 2007).

Richard Huang, Henri Casanova, and Andrew A. Chien. Generating Grid Resource Requirement Specifications, IEEE International Symposium on High Performance Distributed Computing (HPDC 2007).

Dionysios Logothetis, Kenneth G. Yocum, Richard Huang and Andrew A. Chien. Failure-Resilient Expectations for Federated Systems. UCSD Technical Report CS2006-0865 (2006).

Richard Huang, Henri Casanova, and Andrew A. Chien. Using Virtual Grids to Simplify Application Scheduling, IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006).

Yang-Suk Kee, Dionysios Logothetis, Richard Huang, Henri Casanova, and Andrew A. Chien. Efficient Resource Description and High Quality Selection for Virtual

Grids, In Proceedings of the IEEE Conference on Cluster Computing and the Grid (CCGrid 2005).

Richard Huang. Scheduling Compute Intensive Applications in Volatile, Shared Resource (Grid) Environments. Master's thesis, University of California, San Diego, 2005.

Andrew A. Chien, Henri Casanova, Yang-Suk Kee, Richard Huang. The Virtual Grid Descriptive Language: vgDL. UCSD Technical Report CS2005-0817 (2005).

Nut Taesombut, Richard Huang, Venkat Rangan. A Secure Multimedia System in Emerging Wirless Home Networks. Communications and Multimedia Security 2003: 76-88.

FIELD OF STUDY

Major Field: Computer Science

Studies in Large-Scale, High Performance Distributed Computing
Professor Andrew A. Chien, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Automatic Resource Specification Generation for Resource Selection in Large-Scale Distributed Environments

by

Richard Yu-Hua Huang

Doctor of Philosophy in Computer Science

University of California, San Diego, 2007

Professor Andrew A. Chien, Chair

With an increasing number of available resources in large-scale distributed environments, a key challenge is resource selection. First, we show why explicit resource selection is necessary to optimize application performance. Using both a simple and a more sophisticated scheduling heuristic, and for both a real application and a spectrum of randomly generated applications, we show that explicitly pre-selecting resources before running the scheduling heuristic always improved application performance.

With several middleware systems providing resource selection services, a user is still faced with a difficult question: "What should I ask for?" Since most users end up using naïve and suboptimal resource specifications, we propose an automated way to answer this question. We present an automated resource specification generator that given a workflow application (DAG-structured) generates an appropriate resource specification, including number of resources, the range of clock rates among the resources, and network connectivity. Our automatic resource specification generator is composed of a size prediction model, a scheduling heuristic prediction model, and a resource specification generator.

Our size prediction model employs application structure information as well as an optional utility function that trades off cost and performance. With extensive simulation experiments for different types of applications, resource conditions, and scheduling heuristics, we show that our model leads consistently to close to optimal application performance and often reduces resource usage. Further, we construct a model that predicts the optimal scheduling heuristic that can be used in conjunction with the size prediction model. Lastly, we show how our resource specification generator can be used in practice to generate resource specifications for three real-world resource selection systems and offer alternative resource specifications when the best resource request cannot be fulfilled.

I

INTRODUCTION

A clear trend in parallel computing over the recent years is the steady growth of the number of deployed clusters and of the sizes of the clusters. Advances in hardware and manufacturing allow cost-effective commodity clusters to be affordably deployed. In 1996, the National High Performance Cluster Computing Software Exchange (NHSE) [1] reviewed (in [2]) more than twenty Cluster Management Software (CMS) packages, some of which have become commercial products. The increasing availability of cluster management software and vendor options also contributes to the growth of deployed clusters.

Along with the growth of cluster computing and equally important to the growth of distributed computing, are advances in networking technology. The requirement for moving data across machine boundaries fueled networking research starting with the US Gigabit testbed program in 1990 to [3] to provide data rates on the order of 1Gbps to the endpoints of networks. More recently, networking advances in fiber optics have allowed networking bandwidth to reach 10-40 Gbps. High speed connections among increasing number of clusters across administrative domains and institutions help foster the establishment of large-scale distributed environments (LSDEs).

The emergence of LSDEs is at the same time fueled by advances in hardware (computing clusters and networking routers and fibers) and by demands from the scientific community. During the 1980s, multi-disciplinary teams started collaborating on Grand Challenge problems, key problems in science and engineering that require enormous amount of compute power. With the growing need to share data and resources across geographically diverse regions, we have

witnessed the establishment of more and more LSDEs as institutions are willing to share their resources in a collaborative effort. Notable examples of deployed LSDEs today include TeraGrid [4], OpenScienceGrid [5], Grid3 (formerly Grid2003) [6], and Grid5000 [7].

The establishment of LSDEs also brings new challenges. New software is required to execute applications across LSDEs. Before applications can run on compute resources in LSDEs, appropriate resources must be discovered, selected, and bound. When the application is running, monitoring software is required to monitor both the application and the set of resources on which the application is executing. Fortunately, these challenges are shared by all wishing to execute applications across LSDEs. Collectively, the infrastructure necessary to facilitate executing applications in LSDEs is known as *middleware infrastructure.* The Globus Alliance [8] was founded by a community of users and developers who both demanded and built the middleware infrastructure.

I.1 Motivation

One important challenge in executing applications across LSDEs is selecting the appropriate set of resources on which to execute different components of the application. This topic has been widely studied [9-19] and implemented in practice. Resource selection systems range from bilateral matching process [9] to constraint-solving systems [12, 13, 15, 20]. Others employ relational databases [16, 17] to organize the resources and apply nondeterministic queries [18] or other optimizations such as scoping or approximate queries [19] for faster searches.

Any resource selection system can, under different scenarios, return a good and even optimal set of resources given the appropriate inputs. Indeed, resource selection systems are developed so that given a resource specification, they can quickly find a set of resources that matches the resource specification well. Therefore, application developers (or users) can choose whichever resource specification best fit their application and expect resource selection systems

to satisfy this specification whenever possible. The problem is that it is not clear on what basis this choice would be made. The question of what the "best" resource specification, that is the specification that will ultimately lead to best application performance as perceived by the user, is elusive at best. Oftentimes, scientists or application developers can specify exactly the minimum requirements for memory and perhaps processor types but they do not know precisely or cannot even give a good estimate of the number of resources that would be optimal for their applications or the amount of resource heterogeneity their application could tolerate and or take advantage.

The key problem is that none of the systems that we are aware of can provide a good estimate for the number of resources that would be ideal for the application, or provide any guidance for the appropriate amount of heterogeneity among the resources that could optimize application performance. Further, none of these resource selection systems take into account the scheduling algorithms that might be employed once the resources have been acquired.

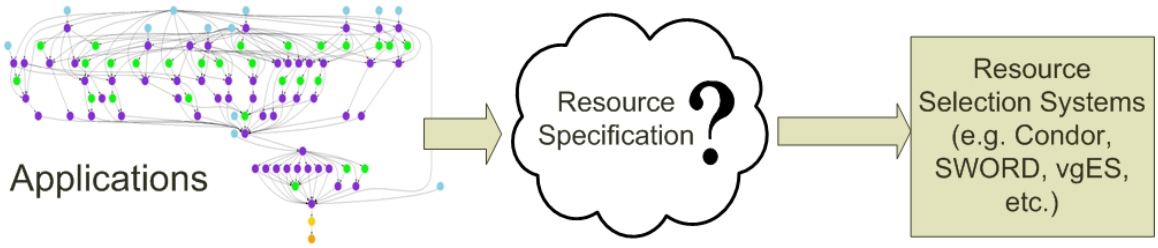

Figure I-1: A missing link exists between applications and resource selection systems

We believe there exists a missing link (illustrated by Figure I-1), between the available resource selection systems and LSDE users. To further illustrate this missing link, we make the following key observations:

- Application developers are experts in their domain but cannot always be counted on to provide accurate guidance for the types of resources that can lead to optimal application performance.

- Resource selection systems can often return a set of resources that closely matches what the application or user specify, but they do not provide guidance on what resource specification the application should provide in the first place.
- Resource selection systems are oblivious to the scheduling heuristics that are used for application execution. This is because the interdependence between application characteristics, resource configuration characteristics, and scheduling heuristics is extremely complex and not well understood.

Because of the difficulty in choosing a resource specification for an application, the commonly used practice consists of requesting the largest number of individual hosts that could be possibly used in concurrently by the application. Unfortunately, although intuitively unsatisfying, this practice is often vastly sub-optimal in terms of both application performance and cost, as we will demonstrate. Furthermore, the problem of choosing a resource specification is complicated because the best choice may depend on the scheduling heuristic used for scheduling the application. In general, the best resource specification for a given application depends on which scheduling heuristic is used.

I.2 Thesis and Approach

In this dissertation, we prove the following thesis statement:

Automatic resource specification generation is necessary and feasible to optimize large-scale distributed environment application performance in a cost-effective manner.

We prove this statement by first examining whether explicit resource selection is necessary and conducive to optimizing application performance. We compare application

performance for both explicit and implicit resource selection. For explicit resource selection, a resource selection system explicitly selects a subset of the resource universe, followed by a scheduling heuristic employed to schedule the application. For implicit resource selection, a scheduling heuristic is employed on the whole resource universe to schedule the application. For explicit resource selection, we also compare two different types of resource abstractions used by resource selection systems.

After we show that explicit resource selection is necessary to optimize application performance, we develop a solution for automatic resource specification generation. This solution consists of three components, illustrated by Figure I-2. First, we formulate an empirical model based on application characteristics to predict the best resource size given an application and a scheduling heuristic (denoted by 'Size Prediction Model' in the figure). We allow the flexibility of a utility function for applications or users to trade off application performance for cost. Second, we formulate an empirical model to predict the best scheduling heuristic for a given application (denoted by 'Heuristic Prediction Model' in the figure). Specifying a scheduling heuristic in conjunction with using the best set of resources to execute applications is necessary to optimize application performance. Third, we combine these two empirical models, along with our observations and assumptions about the resource environments to automatically generate resource specifications for different resource selection systems (denoted by 'Resource Specification Generator' in the figure).

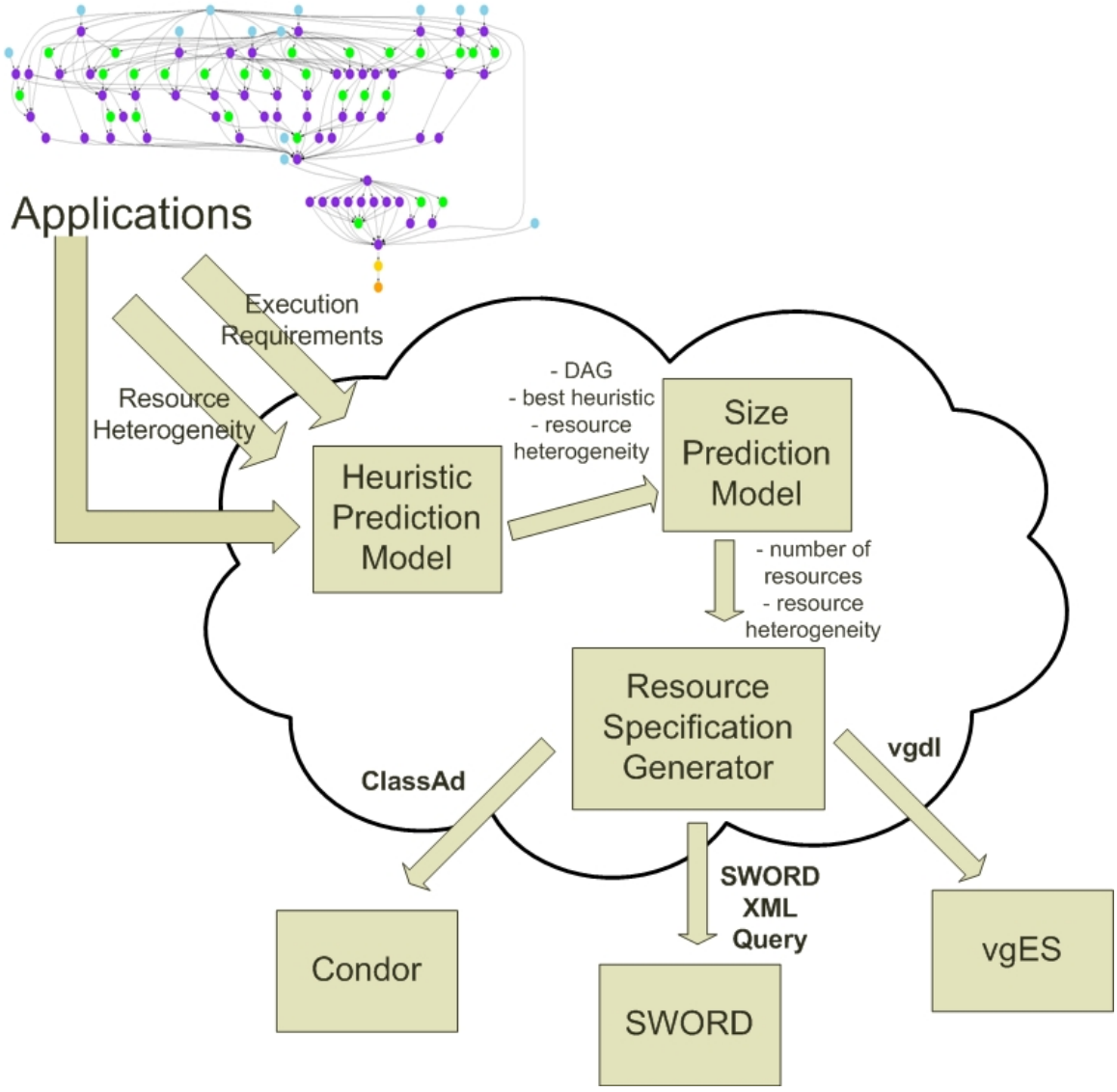

Figure I-2: Three part solution for a resource specification generator

I.3 Contributions

While much research efforts focused on resource selection, we are not aware of any work providing guidance for generating resource specifications. One of the goals of any resource selection system is to improve/optimize application performance; yet a major component in determining application performance, the resource specification, is left to educated guesses at best. Another major component in determining application performance, the scheduling heuristic,

is also left largely ignored by resource selection systems. Providing automatic resource specification generation serves two purposes: it removes the guesswork from users and with the appropriate resource specifications can optimize application performance.

Our contributions in this dissertation are as follows:

- We show that pre-selecting resources prior to running the scheduling heuristic always improved application performance, sometimes by several orders of magnitude.
- We show that when one pre-selects an appropriate set of resources, a simplistic scheduling heuristic can be employed to achieve similar to better performance than using a more sophisticated scheduling heuristic.
- We construct an empirical resource collection size prediction model based on relevant application characteristics. In extensive simulation over a wide range of application configurations, we show that our prediction model consistently allowed application to achieve performance within a few percent of optimal. When applied to a real application, we show that our prediction model leads to almost optimal performance.
- We construct an empirical scheduling heuristic prediction model to be used in conjunction with the resource collection size prediction model. We validated that using both of our prediction models achieved application performance very close to the optimal application performance.
- We incorporated both of our prediction models into a resource specification generator that generates the resource specifications for three different resource selection systems: the Virtual Grid Execution System, Condor, and SWORD. We analyzed the syntax and translated the output of our two prediction models into each of the three resource selection languages.

- We provide an algorithm for generating alternative resource specification if the original generated optimal resource specification cannot be fulfilled by a resource selection system.

I.4 Organization

This dissertation is organized as follows. In Chapter II, we discuss the background and motivation of this work by presenting some of the requirements and solutions for running applications on LSDEs. In Chapter III, we present our resource models, application models, and scheduling models for our experiments; we also provide a roadmap for our approach. In Chapter IV, we investigate why explicit resource selection is necessary to optimize application performance. In Chapter V, we formulate an empirical model to predict the best resource collection size (given input application). We validate that our model works for different resource conditions, scheduling heuristics, and assumptions. In Chapter VI, we formulate a model to predict the best scheduling heuristic given an input application. We validate that our model works in conjunction with the size prediction model. In Chapter VII, we present our automatic resource specification generator that takes the output of prior two models and generate resource specifications for three different resource selection systems. Additionally, we construct a heuristic that allows us to generate alternative resource specifications. We summarize our contributions and highlight directions for future work in Chapter VIII.

II

BACKGROUND

In this chapter, we provide background information on the requirements and solutions for executing applications in large-scale distributed environments. In Section II.1, we present a brief history of how the computational demands of applications forced the evolution from single processor computing to distributed computing on large scale distribute environments. We describe the necessary software and hardware developments, including the middleware infrastructure that provides the fundamental mechanisms for executing applications in distributed environments. We present the six steps necessary to execute applications on such environments in Section II.2, detailing the major challenges involved in each step and the typical solutions. In Section II.3, we present the Globus Alliance, a community formed to share the knowledge of the middleware infrastructure and the various middleware components. The collective software developed by members of the Globus Alliance is known as the Globus Toolkit. Executing applications involves more than the middleware infrastructure, so higher-level systems were developed to address issues not solved within the middleware infrastructure. We describe three systems that address the resource selection issue in Section II.4. Although these systems provide users and applications with the critical ability to select resources given arbitrary resource requirement specifications, generating appropriate such specification is still a challenging problem. We describe this problem in detail in Section II.5, as it is the main motivation for our work.

II.1 History of Large Scale Distributed Environments

Computing began as computation done on a single processor. Given increasing demands in computing power, around the 1980s, a single processor was no longer sufficient. As application demands outgrew the advances in processor speeds, it was necessary to build machines with multiple processors. These machines could share memory and increased the throughput of computation. Thus, computer scientists focused their efforts on algorithms, programs, and architectures that allowed applications to run simultaneously on more than one processor. Parallel computing refers to simultaneously executing the same task on multiple processors in order to compute a result faster. As computing demands grew, it was difficult to scale such machines to large number of processors. This provided the impetus for going across machine boundaries and building distributed-memory machines by connecting individual machines with a network. This was only possible with advances in networking technology.

With computation distributed across distributed machines, programming became more difficult than programming for a single machine. To alleviate these difficulties, message-passing libraries such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), as well as entire languages such as High Performance Fortran (HPF) were developed to support communications for parallel applications executing on multiple machines[21]. Often, these machines are connected by fast local area networks in one physical location. Collectively, these machines are known as a *cluster* and individually these machines are then known as *nodes*. Clusters are typically much more cost-effective than single shared-memory machines with comparable speed or reliability and are the most popular forms of distributed-memory parallel computing platforms today. With dropping hardware prices, commodity clusters can be built with increasingly cheaper commodity computers. Open-source cluster management tools such as ROCKS [22, 23] makes it increasingly straightforward to deploy powerful Linux clusters. At the

writing of this dissertation, 1153 ROCKS clusters are registered totaling 51,935 CPUs. Clusters have become the basic building blocks for distributed environments.

By *distributed computing* we refer to a computation that is distributed across different clusters, with the added implication that the clusters are some geographical distance apart. *Large scale distributed environments* (LSDEs) are distributed environments with large number of computing resources, i.e., large numbers of (large) clusters. Often, the resources within the computing environments are heterogeneous with respect to each other and in many cases can be composed of heterogeneous resources at one geographical location. Heterogeneity can refer to differences in operating systems, clock rates, memory, or any other characteristic of physical nodes.

The usefulness and the necessity of distributed computing environments became apparent during the 1980s, when multi-disciplinary teams of researchers started working on so-called Grand Challenge problems, which are key problems in science and engineering that require enormous amount of compute power. Such collaborative work necessitates the use of a large-scale computational infrastructure to achieve new scientific discoveries [24]. The interdisciplinary research teams often comprised of researchers from geographically distinct locations, thus requiring remote data transfers and coordinations. This requirement to move or use data from other sites fueled research starting with the US Gigabit testbed program in 1990 [3] to provide data rates on the order of 1Gbps to the endpoints of networks. More recently, projects such as OptIPuter [25] are using optical fibers to create "supernetworks" that are on the orders of 10-40Gbs. With higher bandwidth and the need for researchers in geographically distinct locations to collaborate on different projects, we have witnessed the establishment of more and more LSDEs as institutions are willing to share their resources in a collaborative effort. Notable examples of LSDEs include TeraGrid [4], OpenScienceGrid [5], Grid3 (formerly Grid2003) [6], Grid5000 [7].

Scientific applications deployed in LSDEs are typically compute intensive, that is requiring lots of computational resources, or data intensive, that is processing large amount of experimental data, with many applications falling in both categories. Specific domains that can benefit tremendously from LSDEs span most areas of science and engineering with well-known examples including physics, astronomy, climatology, or seismology. GriPhyN (Grid Physics Network) [26] is a good example of a scientific community (in this case, physicists) requiring LSDEs to enable Petabyte-scale data intensive science. A community of thousands of scientists distributed globally requires access to raw data as well as computationally intensive analyses of datasets that will grow from the 100 Terabytes to the 100 Petabyte scale in the next decade, according to current projections. The computing and storage requirements are distributed, and the data collections and analysis/visualization are distributed. This distributed nature of the data stems from the fact that data is captured from scientific instruments (particle accelerators, microscopes, or telescopes). Due to advances in computing and networking, scientific communities like GriPhyN can come together to share datasets and computing resources at a global scale.

Although LSDEs provide resources to execute applications at unprecedented scale, the logistics of application execution are complex, e.g., due to resource being in different administrative domains and under different access policies, due to resources being heterogeneous, and due to complex application requirements. To address this complexity, one needs a software infrastructure that provides the necessary basic mechanisms. This infrastructure is commonly termed *middleware.* More specifically, the middleware infrastructure provide the following functionalities: discovering locations of available resources, selecting the appropriate set of resources, acquire (bind) the resources on behalf of the application, scheduling and launching application components on different resources, as well as monitoring the progress of application components and the availability of resources.

The Globus Alliance [8] was founded by a community of users and developers who both needed and built middleware to allow applications to execute across machine and administrative domain boundaries. This collective effort resulted in the Globus Toolkit, where ongoing open source middleware development helps deploy applications in LSDEs.

LSDEs are becoming increasingly prevalent as a critical means for scientists to achieve new advances in their respective fields. Nevertheless, many challenges remain for LSDEs to deliver their true potential, especially as scale continues to increase. Many of these challenges represent opportunities for computer scientists to contribute novel systems and algorithmic ideas for helping other scientists to execute their applications on LSDEs a way that is both convenient and efficient.

II.2 Executing applications in LSDE

One big advantage of LSDEs is that scientists can share data and have access to more resources than available at a single institution, whether computing or other instruments. In order to take advantage of LSDEs, scientists must be able to execute their application in a convenient and efficient manner. Executing an application in an LSDE today typically entails 6 steps, which have been extensively studied, both in practice (middleware) and in theory. These six steps are:

1. Resource Discovery
2. Resource Selection
3. Resource Binding
4. Application Scheduling
5. Application Launching
6. Application Monitoring

We describe all six steps below, highlighting challenges at hand and current solutions.

II.2.1 Resource Discovery

When one shares computing resources with a few close collaborators, it is very easy to keep track of where and what resources are available. However, when hundreds and thousands of scientists join in the collective sharing of resources, it becomes a challenge to know which resources are available and where each resource resides. *Resource discovery* refers to the task of identifying the resources that are available in an LSDE. Before using a resource to execute an application, it is necessary to know the location and identity of said resource. Typically, this problem is solved by the implementation of an indexing service. Whenever a resource joins the LSDE, it needs to contact the indexing service to announce its existence. Conversely, the indexing service then can be contacted by potential resource users and inform these users of the existence of the resource. Some examples include the Globus Monitoring and Discovery Services (MDS) [27] and the Internet Scout Project [28]. The Globus MDS is a suite of web services that allows users to discover available resources considered part of a Virtual Organization (which can be considered an LSDE). The Internet Scout Project allows groups or organizations to share their knowledge and resources via the World Wide Web.

II.2.2 Resource Selection

As with resource discovery, when the number of sites increases to hundreds or thousands and the number of resources available at each site is one or two orders of magnitude more, choosing the appropriate resources on which to execute an application becomes a challenge. *Resource selection* refers to choosing a set of resources from the resource universe (i.e. an LSDE) to execute an application. Much research [9-13] has gone into resource selection. Resource selection systems range from the bilateral matching process called matchmaking [9] in Condor [14], to added economic elements in SWORD [15, 20] where resources are allocated based on

auctions, to constraint-solving systems such as RedLine [12, 13] where resources are selected based on the constraints of the resource characteristics. Others employ relational databases (such as the vgES system [16, 17] of the VGrADS project [29]) to organize the resources and apply nondeterministic queries [18] or other optimizations such as scoping or approximate queries [19] for faster searches.

All these systems define a resource description language by which users and applications can describe their resource requirements. Some resource description languages include ranking function by which one can specify that particular resource characteristics are favored. For example, inside a descriptive language, an application or user could describe a resource collection containing between 10 and 20 Xeon processors, while tolerating clock rate ranging from 2GHz to 3GHz. A ranking function might be defined in a view to favoring the faster resources, i.e., resources with higher clock rates.

II.2.3 Resource Binding

After an application consults a resource selector and the resource selector returns a desired set of resources, the application needs to "bind" the resource in order to execute components of the application on them. *Resource binding* refers to establishing application presence on a computing resource. Usually binding involves the application (or an agent for the application) negotiating with a local resource manager, either a human being or the more likely case of a service running on the computing resource. To complete binding, the local resource manager must agree for the application to execute tasks on the resources.

The biggest challenge here is the heterogeneity in local resource managers and the different resource management policies. Some resource managers might grant applications dedicated use of the resources immediately, or they might require the application to wait in a queue, or they might have an advance reservation system where the application can request slots

of time to execute their components. The solutions to such challenges typically involve interfaces to various types of resource managers and resource management policies. An example solution is the Globus Resource Allocation and Management (GRAM) [30] service that provides a single interface for requesting and using remote resources for the execution of application components. GRAM interfaces with various local resource management systems including schedulers, queuing systems, and reservation systems but provide one unified interface to all applications.

II.2.4 Applications Scheduling

Scheduling algorithms have been well studied since it was first formulated in the 1950s. However, LSDEs have only come into existence in recent years. Consequently, many researchers have been actively adapting existing algorithms or developing novel algorithms for application executing in LSDEs. One major constraint for scheduling algorithms when applied to application executing on LSDE is their execution time. The general scheduling problem is NP-complete. Since scheduling algorithm requiring exponential execution times cannot be used in practice, one typically develops heuristics that have polynomial complexity. For instance, many heuristics are available for the DAG-scheduling problem [31-34], which is arguably among the most general scheduling problems. However, because of the scale of LSDEs, even scheduling heuristics with polynomial complexity can take an unreasonable amount of time to compute a schedule. Therefore, a challenging tradeoff arises. Indeed, what mattes to the user in the end is the application turn-around time, which include both the execution time of the scheduling heuristic and the execution time of the application using the schedule. Therefore, using an effective but perhaps long running scheduling heuristic may result in longer turn-around time, making choosing the best scheduling heuristic to use very challenging.

II.2.5 Application Launching

Launching applications requires staging of executables and data, and starting of the application processes. In current solutions starting application processes is often combined with the Resource Binding process described above. Typical solutions to file staging involve protocols designed to transfer files efficiently to various distributed hosts. An example solution is GridFTP [35], a secure, reliable, data transfer protocol optimized for LSDEs.

II.2.6 Application Monitoring

Once the application has been launched and is executing on the different remote resources, it can be beneficial to monitor the application or the resources to ascertain the application progress. Two possibilities exist for application monitoring:

1. The application implements a built-in status monitoring capability and can report on its own progress directly.
2. The middleware infrastructure provides such monitoring capabilities.

Along with application monitoring, it may be necessary to monitor the resources instead because of resource overload or failure, both of which are unrelated to the application. In such a scenario, it would be beneficial for the application to migrate the work elsewhere to improve application performance. Resource monitoring is particularly relevant when resources are not dedicated.

The main challenge of monitoring resources on a LSDE is the large volume of resource data that needs to be collected and processed. Another challenging issue is determining the frequency of data collection. While collecting resource data very frequently can lead to more timely information, it also increases the volume of data that is collected and that needs to be processed. Data collection also impacts the load on any resource and more frequent data

collection means heavier load on the resource. Collecting data also means that data needs to be sent elsewhere to tally aggregate data. Frequent data collection would also mean bandwidth consumption by the monitoring software.

Another major challenge is to identify whether the application is behaving as expected. The problem arises because the application may not be utilizing any particular resource at all times. It is extremely difficult for a resource monitor to gauge when the application is not executing on a resource because it has finished its processing or because it is waiting for some other resource to send data needed for its processing. These two cases would be normal behavior whereas a faulty behavior would arise when the application is not executing because of software faults on the resource, overloading on the resource, or some other permission problems on the resource. Identifying what is expected behavior and what is faulty behavior remains a challenge. A monitoring system would need input from the application or user to define what is expected behavior and what is unexpected behavior.

As part of the Grid Application Development Software Project (GrADS) [36], Autopilot [37, 38] is a real-time adaptive control infrastructure which provides a flexible set of performance sensors, decision procedures, and policy actuators to realize adaptive control of applications and resource management policies. Autopilot assesses application progress using performance contracts. When a violation is detected, Autopilot works with other components of the GrADS environment to maintain reasonable application performance under current operating conditions. The Globus Toolkit also provide a monitoring component as part of the Globus Monitoring and Discovery System (MDS) [39, 40].

II.3 Middleware for LSDEs: the Globus Alliance

The Globus Alliance was formed to address the development of middleware to enable sharing of computing power, databases, instruments, and other online tools securely across

corporate, institutional, and geographic boundaries without sacrificing local autonomy [8]. The collective software developed by the members of the Globus Alliance is referred to as the Globus Toolkit [41, 42], an open source software toolkit comprising many of the middleware capabilities necessary for LSDEs. The Globus Alliance was formed when scientists across different disciplines realized that generalized solutions (middleware) for their needs were necessary to avoid having to repeat building similar software components. We give more details about how the Globus toolkit address, or fail to address, the six steps involved with executing an application on an LSDE.

II.3.1 Resource Discovery

With respect to resource discovery and resource monitor, Globus provides the Monitoring and Discovery System (MDS) [39, 40] as part of the Globus Toolkit Information Services. For resource discovery, MDS4 provides an index service which collects and publishes aggregated information about information sources. Users and applications can query the indexing service to discover the locations of desired resources as well as the availability (based on load) of the resources.

II.3.2 Resource Selection

The Globus Toolkit does not provide any components for resource selection. Instead, higher level systems such as vgES interface with other Globus Toolkit components to provide the resource selection functionality.

II.3.3 Resource Binding

The Globus Grid Resource Allocation and Management Service (GRAM) was created to solve one of the most fundamental requirements of executing applications on LSDE, namely,

applications negotiating with underlying resource managers for use of computing resources. GRAM itself is not a scheduler. It is an interface to different scheduling components such as PBS or LSF for remote job submission and control. It solves both the resource binding problem and launching the application by providing two-way file staging – bringing needed input files and takes out the application output files. It provides remote I/O redirection, job status monitoring, and job signaling (start, stop, kill, etc.). After the release of Globus Toolkit 4.0, GRAM is based on Web services interfaces.

II.3.4 Application Scheduling

The Globus Toolkit does not provide any scheduling heuristics. Instead, applications must provide their own schedulers or rely on a higher level system such as vgES that interfaces with other Globus Toolkit components to provide the application scheduling functionality.

II.3.5 Application Launching

Related to resource binding, the Globus Toolkit component GRAM interfaces with different scheduling components such as PBS or LSF for remote job submission and control. It solves both the resource binding problem and launching the application by providing two-way file staging – uploading necessary input files and downloading produced output files.

II.3.6 Application Monitoring

The Globus MDS4 provides application monitoring (in addition to providing resource discovery). For monitoring, MDS4 interfaces with various information sources such as cluster monitors like Ganglia [43, 44] or Hawkeye [45], or services like GRAM, RFT, RLS, or queuing systems like PBS (Portable Batch System) [46] or LSF (Load Sharing Facility) [47], translating

their diverse schemas into appropriate XML schemas based on standards such as the GLUE schema [48] whenever possible. The XML files then can be parsed to extract useful information.

II.3.7 Security

Although not a specific part of the 6 steps necessary to run applications in LSDE, the Globus Toolkit provides the Grid Security Infrastructure (GSI) to address security needs. The three major points GSI cover are the need for secure communications between hosts in an LSDE, the need to support security across organizational boundaries, and the need to support a single sign-on for a single user within a LSDE, including delegating authority for multiple resources and/or sites. A GSI utility generates a private key and certificate that is valid for a few hours. Each certificate also contains the identity of a Certificate Authority (CA) that certifies that both the public key and the identity belong to the subject. Mutual authentication can happen when both parties trust the CAs that sign each other's certificate.

II.4 Systems for Resource Selection in LSDEs

While the Globus Toolkit provided a fair number of middleware components to facilitate running applications in LSDEs, the idea of the Toolkit is to provide basic mechanisms that everyone can share and use. Choosing a subset of resources in a LSDE to run applications, or resource selection,cannot be solved via a simple mechanism but is in fact a challenging research problem. Fortunately, resource selection has been widely studied [9-19] and software solutions are implemented in practice. In this section, we examine three middleware systems that implement resource selection, as well as discuss the inputs to these systems.

II.4.1 Virtual Grid Execution System

The Virtual Grid Execution System (vgES) [16, 17] was designed and prototyped as part of the Virtual Grid Application Development Software Project (VGrADS) [29]. The VGrADS project was built on and informed by a four year effort to build development tools for adaptive grid applications, the Grid Application Development Software Project (GrADS) [36]. The vgES architecture was built on the key insight from GrADS that application participation is required to effectively manage performance in a LSDE.

The main contribution of VGrADS is the notion of a *Virtual Grid* (VG), a high-level, hierarchical abstraction of the resource collection that is needed and used by an application. This abstraction provides a clean separation of concerns between applications and the complexity of the underlying middleware infrastructure and the heterogeneity of the underlying physical resources. The application specifies its resource needs using a high level language, the Virtual Grid Descriptive Language (vgDL), and vgES finds and allocates the appropriate resources on the behalf of the application.

Resource selection plays a major role in determining the architecture of vgES because of the end goal of producing a virtual grid based on the user written vgDL specifications. Within vgES, the main component for resource selection is called the vgFAB (the "finder and binder"). The vgFAB performs integrated resource selection and binding which enables optimized resource choices in a high load resource environment. The vgFAB parses the input vgDL and performs the resource selection via queries to a relational database populated with resource information that is updated by a vgAgent component. The vgAgent component interfaces with middleware such as the Globus MDS or Ganglia to discover resources and populate the database with current dynamic resource information such as the load for other components to process the data. The resultant tuples of resources are sorted by a ranking function which the user or application can

specify via the input vgDL. The vgFAB then binds the resources by interacting with autonomous resource managers through the underlying Globus GRAM. After the resources have been bound, a virtual grid is returned as the output of vgES.

The vgES also provides a vgLaunch component which launches the application on the bound virtual grid according to scripts provided by applications. After the application is launched, a monitoring component called the virtual grid monitor (vgMON) monitors the virtual grid based on default expectations regarding the resources in the virtual grid. Users may specify additional expectations through a higher level language, the Expectation Definition Language (EDL) [49].

II.4.1.1 Input to vgES: vgDL

The input to vgES is a resource specification written in a high-level resource description language, the Virtual Grid Description Language (vgDL). The vgDL incorporated the RedLine [13] BNF for resource attribute constraints. The salient point of vgDL is the capability for applications to specify hierarchical resource aggregates and qualitative notions of network proximity between these aggregates. The three resource aggregates are distinguished by homogeneity and network connectivity:

1. *LooseBag* - a collection of heterogeneous nodes with possibly poor connectivity
2. *TightBag* - a collection of heterogeneous nodes with good connectivity;
3. *Cluster* - a set of well-connected nodes with identical (or nearly so) individual resource attributes.

The notion of "good" is defined in term of a network latency threshold. The implicit assumption is a positive correlation between low latency and high bandwidth. For instance, in vgDL, an application can request a Cluster of between 32 and 64 Opteron processors with clock rate higher than 2Ghz and more than 1GB of RAM that is "close" to a TightBag of 32 to 128 processors that have clock rates higher than 1Ghz. Figure II-1 shows the syntax of such a vgDL

specification. The tenet of the VGrADS project is that such simple and qualitative specifications fit the need of most applications in practice.

```
VG =
{
 ClusterOf(nodes) [32:64]
 {
   nodes =   [(Processor == Opteron) && (Clock>=2000) && (Memory >=
1024]
 }
 close
 TightBagOf(nodes2) [32:128]
 {
   nodes2 = [(Clock >=1000)]
 }
}
```

Figure II-1: Example vgDL resource collection specification

II.4.2 Condor

Condor [14] is a high throughput computing system developed at the University of Wisconsin to run applications on LSDEs. The focus is workload management for compute-intensive jobs. In addition to job queuing mechanisms, Condor also provides scheduling policies, priority schemes, resource monitoring, and resource management. Condor was originally designed to harness wasted computing cycles on idle workstations. When a machine is idle for some period of time, Condor tags the machine as available. Tasks from other users may be migrated to machines tagged as available and executed there as long as there are not keyboard or mouse inputs. Periodically, tasks are checkpointed and when the owner of the machine reclaims the workstation, checkpointed tasks are migrated off the machine and finished elsewhere.

To achieve the highest throughput possible, Condor provides two important functions. First, it makes available idle machines and thus limit wasted computing cycles. Second, it expands the resources available to users, by functioning in a distributed environment. Today, in addition to harnessing idle personal workstations, Condor allows the addition of clusters to the

list of resources. These new resources are often dedicated to tasks. Condor manages the newer resources in the same way that it managed the old workstations.

II.4.2.1 Input to Condor: ClassAds

In the Condor system, a bilateral matching process called matchmaking [9] is used for resource discovery and resource selection. Using Classified Advertisements (ClassAd's), both resource providers and requesters post "ads" for advertising resource availability or resource needs, respectively. A matchmaker (or a central clearinghouse) then attempts to match the ads from the resource providers and requesters. The drawback of bilateral matchmaking is that each resource requester is limited to one resource, precluding the possibility of more advanced resource management capabilities, such as resource co-allocation.

An extension to Matchmaking is Gangmatching [10], which supports a multilateral matching of a *gang* (or a group) of ClassAds. Specifically, it provides a new ability to relate and evaluate the properties of multiple candidate ClassAds through arbitrary constraint defined on candidate individuals or groups. The multilateral matching model allows multiple resources to be collectively matched with the needs of a single job, thus enabling resource co-allocation.

Gangmatching extends Matchmaking's bilateral constraints by replacing a single bilateral match imperative (defined in a ClassAd's *requirement* attribute) with a list of required bilateral matches (defined in a new attributed, called *port*). A port attribute defines the number of and characteristics of the matching candidate ClassAds for its associated ClassAd to be satisfied. Each port defines *Labels* that name the candidate bound to that port. To validly match a gang of ClassAds, all their ports must be bound with compatible ports (i.e., no conflict between them) of some other ClassAds in a group. For example, in a Gangmatch request, one can create a port specifying an Opteron Linux machine and another port specifying an Intel Linux machine while ranking both according to a function of CPU Flops and memory. Figure II-2 illustrate such a

request. In order for the matchmaker to match machines to the request, the machines must satisfy the constraints specified under each port.

```
[ Type    = "Job";
  // some common attributes
  Owner = "somedude";
  QDate = ' Mon Oct 30 12:23:45 2006 (PST) -08:00';
  Cmd    = "run_simulation";
  Ports = {
     [  // request first machine
        Label         = cpu;
        ImageSize  = 100M;
        Rank         = cpu.KFlops/1E3  + cpu.Memory/32;
        Constraint  = cpu.Type =="Machine" &&
                         cpu.Arch == "OPTERON"  &&
                         cpu.OpSys == "LINUX"
     ],
     [  // request second machine
        Label         = cpu;
        ImageSize  = 100M;
        Rank         = cpu.MFlops/1E3  + cpu.Memory/32;
        Constraint  = cpu.Type =="Machine" &&
                         cpu.Arch == "INTEL"  &&
                         cpu.OpSys == "LINUX"
     ]
  }
]
```

Figure II-2: A Gangmatch ClassAd request

A machine ClassAd is fairly straightforward as it advertises some static as well as dynamic attributes for the machine. For example, a machine could be idle for more than 15 minutes with a low load average and be available for claim. Figure II-3 shows an advertisement for such a workstation.

```
[ Type          = “Machine”;
  Activity      = “Idle”;
  KeybrdIdle = ‘00:22:35’;
  Disk          = 200 G;
  Memory        = 1000 M;
  State         = “Unclaimed”;
  LoadAvg       = 0.04345;
  Mips          = 104;
  Arch          = “INTEL”;
  OpSys         = “LINUX”;
  MFlops        = “

  Ports = {
     [  // request first machine
        Label       = requester;
        Rank        = 1/requester.ImageSize;
        Constraint  = requester.Type ==”Job” &&
                      requester.Owner == “valid_user”  &&
                      LoadAvg < 0.3 &&
                      KeybrdIdle> ’00:15’
     ]
  }
]
```

Figure II-3: Workstation Advertisement

II.4.3 SWORD

SWORD [15] is a scalable resource discovery service for wide-area distributed systems. The focus of SWORD is the set of resources on which users can deploy services (as opposed to executing a short-lived application). Thus, SWORD runs on Internet-scale infrastructure machines (such as the nodes of the PlanetLab [50] testbed). SWORD collects both static and dynamic resource information and selects resources based on user defined criteria. These criteria may be per-node (e.g. free memory, free disk space) or inter-node (e.g. inter-node latency).

Resource specifications center on the notion of *groups* that capture equivalent classes of nodes with similar characteristics. Users can describe the desired resources as a topology of interconnected groups with required intra-group, inter-group, and per-node characteristics. Additionally, users may specify a range of required and desired values of per-node and inter-node resource measurements, with varying level of penalties (costs) for selecting nodes that are within

the required range but outside the desired range. SWORD endeavors to locate the lowest cost resource configuration while meeting user requirements.

SWORD is designed with two usage scenarios. One scenario is the "best-effort" environment such as the PlanetLab. In such a scenario, SWORD simply returns a list of resources matching the description of the input query. Another scenario is one where SWORD is used in conjunction with a resource allocation or admission control mechanism. The resource allocation mechanism might be able to arbitrate the start, duration, and cost of usage for any of the resources. Currently, the deployment of SWORD has been in the "best-effort" environment of PlanetLab [50], with expectations to integrate with resource allocation tools such as SHARP [51] or SNAP[52] to support arbitrated usage scenarios.

II.4.3.1 Input to SWORD: XML file

SWORD takes two forms of input: Condor ClassAds and the SWORD query language. A SWORD query takes the form of an XML document with three sections. The first section describes the (optional) resource consumption constraints the user places on evaluating the query. In this section, the user can specify the desired trade-off between the "quality" of node selection for amount of network resource consumption in evaluating the distributed query and limit the running time of the optimization step in which candidate nodes are culled to a final approximately optimal set. For example, in Figure II-4, the user is allowing at most 30 nodes to be visited in processing the distributed query and at most 100 seconds of running time for optimization.

The second section of the SWORD query specifies the constraints on single-node and inter-node attributes of desired groups. The single node attributes are similar to those of vgDL and ClassAds. They can be static attributes such as operating system or dynamic attributes such as availability. Additionally, inter-node attributes such as pair-wise latency or bandwidth can be specified. One interesting aspect of the SWORD group is the attribute

'network_coordinate_center', which describes where the group should be located. Examples of such centers include broad locations such as North America or Europe.

The third section of the SWORD query specifies pair-wise constraints between individual members of different groups. For example, in Figure II-4, there must be at least one node in each group such that the latency between that node and at least one node in the other group is less than 100ms. The assumption here is that users have an idea of general inter-node measurements and thus can impose such constraints.

```
<request>
  <dist_query_budget>30</dist_quer_budget>
  <optimizer_budget>100</optimizer_budget>
  <group>
    <name>Cluster_NA</name>
    <num_machines>5</num_machines>
    <cpu_load>0.5, 0.1, 0.1, 0.0, 0.0</cpu_load>
    <free_mem>256.0, 512.0, MAX, MAX, 100.0</free_mem>
    <free_disk>500.0, 1000.0, MAX, MAX, 5.0</free_disk>
    <latency>0.0, 0.0, 10.0, 20.0, 0.5</latency>
    <os>
       <value>Linux, 0.0</value>
    </os>
    <network_coordinate_center>
       <value>North_America, 0.0</value>
    </network_coordinate_center>
  </group>
  <group>
    <name>Cluster_Europe</name>
    <num_machines>5</num_machines>
    <cpu_load>0.5, 0.1, 0.1, 0.0, 0.0</cpu_load>
    <free_mem>256.0, 512.0, MAX, MAX, 100.0</free_mem>
    <free_disk>500.0, 1000.0, MAX, MAX, 5.0</free_disk>
    <latency>0.0, 0.0, 10.0, 20.0, 0.5</latency>
    <os>
       <value>Linux, 0.0</value>
    </os>
    <network_coordinate_center>
       <value>Europe, 0.0</value>
    </network_coordinate_center>
  </group>
  <constraint>
    <group_names>Cluster_NA Cluster_Europe</group_names>
       <latency>0.0, 0.0, 50.0, 100.0, 0.5</latency>
  </constraint>
</request>
```

Figure II-4: Sample SWORD XML query

If the Condor ClassAd system is suitable for requesting a handful of distinct machines, the SWORD system is suitable for requesting groups of machines with similar characteristics. Furthermore, the intra- and inter-group network constraints allow users to clearly specify the desired network connectivity between groups of machines. The drawback seems to be long running selection times when analyzing different network constraints to determine the suitability of different machines in forming the "groups". Users also have the option of trading off the "quality" of their resource selection and by limiting the running time of the optimization step in which candidate nodes are culled to a final approximately optimal set. The drawback here is the difficulty any new user would face in choosing the appropriate tradeoff values that would return a sufficiently high quality set of resources.

II.5 Motivation

The resource selection capability in systems such as vgES, Condor, or SWORD require that a specification be provided that describes the number of types of resources desired by the user or application. Oftentimes, scientists or application developers can specify exactly the minimum requirements for memory and perhaps processor types but they do not know precisely or cannot even give a good estimate of the number of resources that would be optimal for their applications or the amount of resource heterogeneity their application could tolerate and or take advantage.

Any resource selection system can, under different scenarios, return a good and even optimal set of resources given the appropriate inputs. The key problem is that none of the three systems (or any other systems that we are aware of) can provide a good estimate for the number of resources that would be ideal for the application, or provide any guidance for the appropriate amount of heterogeneity among the resources that could optimize application performance.

Further, none of these resource selection systems take into account the scheduling algorithms that might be employed once the resources have been acquired.

We believe there is a missing link, or a gap, between the available resource selection systems and LSDE users. To illustrate this gape, we make the following key observations:

- Application developers are experts in their domain but cannot always be counted on to provide accurate guidance for the types of resources that can lead to optimal application performance.
- Resource selection systems can often return a set of resources that closely matches what the application or user specify, but they do not provide guidance on what resource specification the application should provide in the first place.

Resource selection systems are oblivious to the scheduling heuristics that are used for application execution. This is because the interdependence between application characteristics, resource configuration characteristics, and scheduling heuristics is extremely complex and not well understood.

Motivated by the points above, our goal in this work is two-fold:

1. Predict the best scheduling heuristic to use given an input application (while optimizing application or trading off performance for cost).
2. Generate best resource specification given best scheduling heuristic and the input application. The resource specification can optimize for application performance or some function of tradeoff between performance and cost.

III

MODELS AND METHODOLOGY

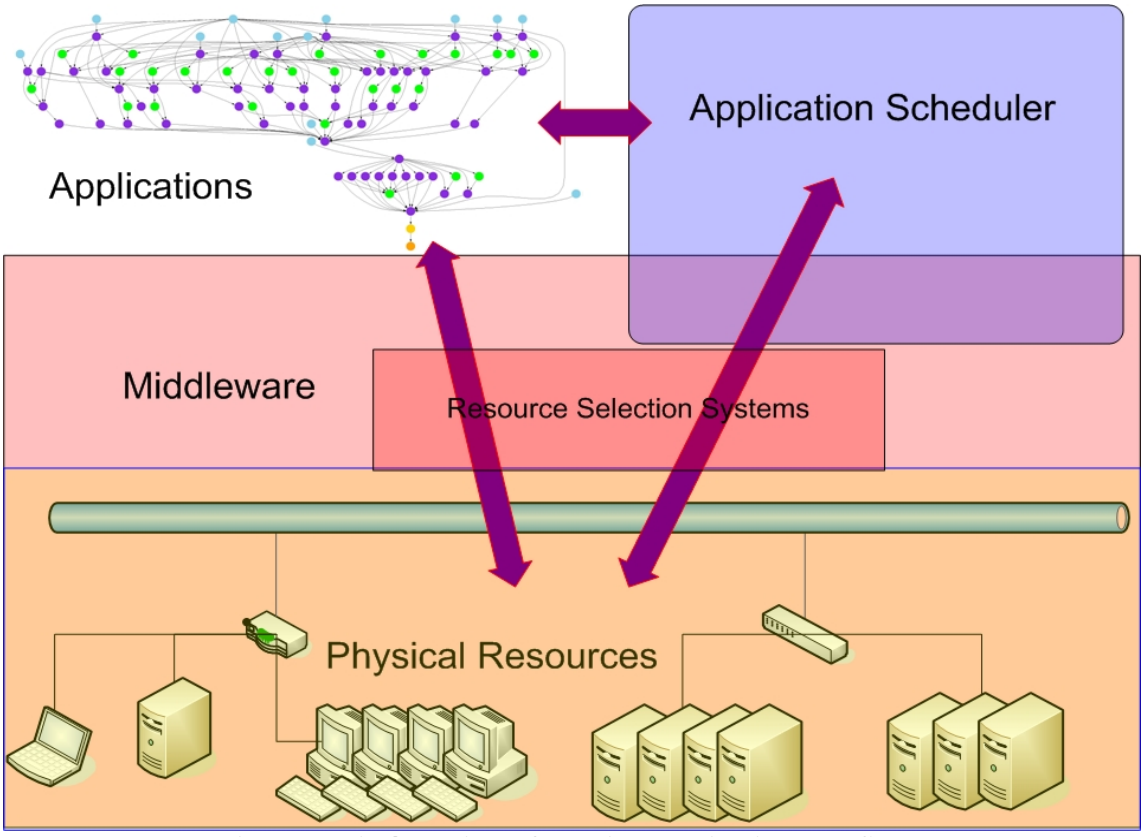

Figure III-1: Overview of running application on LSDEs

In this chapter, we develop models for the LSDE components that are involved when running applications and relevant to our work in this dissertation. These components and some of their interactions are depicted in Figure III-1. At the bottom of the figure are the physical resources that comprise the computing environment available to run applications. These resources may include compute devices, storage devices, network devices, as well as scientific instruments

and visualization devices. In this work, we only focus on the compute and network devices. LSDE resources are heterogeneous with regards to software, hardware, and access policies. A middleware infrastructure is used to hide and manage this heterogeneity, at least partially. In chapter II, we have discussed the functionalities provided by currently deployed middleware in today's LSDEs.

The application, depicted in the top left corner of Figure III-1, is comprised of potentially many compute tasks with different resource requirements. These tasks may also need to access and exchange significant amounts of application data. Our depiction of the application is purposely in the shape of a generic directed acyclic graph, which is the application model we consider in this dissertation.

In the top right corner of the figure is the application scheduler, which is responsible for mapping application tasks and data to resources, and which might be implemented entirely or partially in the application. The application can inform the scheduler about its characteristics, and the scheduler can use the middleware infrastructure to discover necessary resource information. Either the application or the application scheduler needs to specify the desired set of resources to the resource selection system [9-19], which then selects and acquires the desired set of resources from the resource universe. Note that we depict the scheduler as overlapping with the middleware. This is because all or part of the scheduler's functionalities could be implemented as part of a middleware infrastructure, which will be discussed further in this chapter.

Our goal in this chapter is twofold. First, we define realistic models for some of the LSDE components highlighted above. Section III.1 defines a generic and popular application model; Section III.2 defined a model for the underlying physical resources and for the way in which they are managed; and Section III.3 defines the scheduler-middleware interaction model. Second, based on these models, we present our methodology and identify the roadmap of our investigation to answer the questions and challenges discussed in Chapter II.

III.1 Application Model

A popular model for which scheduling heuristics have been developed is the "task graph" model, by which an application is represented as a weighted Directed Acyclic Graph (DAG). In this dissertation, we use the DAG application model. We do not consider data-parallel applications where one task can be separated into many sub-tasks which can be executed in parallel, and typically synchronously, on a cluster. Indeed, this problem can be intrinsically reduced to finding the most appropriate cluster. We also do not consider mixed-parallel applications, where each node in the DAG is a data-parallel task. For future work, we can expand the results of this dissertation to mixed-parallel applications by generating resource specifications requiring clusters instead of hosts for each node in the DAG.

III.1.1 Directed Acyclic Graph

The DAG application model is particularly relevant for *scientific workflows* [53]. The last few years have seen active development and deployment of many such workflows in various domains such as physics [54, 55], image processing [56], and astronomy [57]. These workflows require considerable amounts of computing power and are loosely coupled parallel applications. Therefore, it is natural to explore the possibility of executing them on LSDEs [58].

Formally, we define a DAG as (V,E) where V = {v1, v2, …, vn}is a set of nodes and E = {e1, e2, …, em}is a set of edges. A node in the DAG represents a task in the "task graph". A task is a set of indivisible executables or instructions that must be executed on one processor. We assume that tasks run on processors to completion without preemption. The computational costs for each task is denoted by wv(vi), in units of seconds on a reference CPU. Although the study of unrelated processors can be interesting, for the purpose of this work, we consider only uniform processors. This corresponds to "uniform" and "unrelated" processors as described in [59]. Others

[60] refer to the matrix of task/hosts execution times as "consistent" or "inconsistent". In our experiments, we consider uniform processors where the task/hosts execution times are consistent. This corresponds to the case in which all processors are of the same type, but differ in clock rates. Thus all tasks would run faster on a faster CPU and slower on a slower one, and we make the assumption that the ratio of each task's execution time on a CPU is directly proportional to its clock rate.

Each node can have multiple inputs, that is multiple edges pointing to it. An edge in the DAG represents the cost of sending intermediate files from one node to another. The communication costs for each edge e_k is denoted by $w_e(e_k)$, also in units of seconds as we calculate $w_e(e_k)$ by dividing file size by a reference bandwidth of 10Gbps. We choose 10Gbps as this represent an upper bound on the achievable bandwidth that might be available at different research institutions or LSDEs today, as on the TeraGrid [4]. Each directed edge e_k represents dependency between two tasks and implies that if $v_i \rightarrow v_j$, then v_i is the parent (denoted by $p(e_k)$) and v_j is the child (denoted by $c(e_k)$). Also, v_j cannot start to execute until v_i has completed and has sent its data to v_j. A task can start to execute only when all of its parents are done processing and have transferred all the required files to the physical host running the task. We denote the set of all vertices comprising the parents for a node v_i as $P(v_i)$ and the set of all vertices comprising the children of a node v_i as as $C(v_i)$. A node without any parents is called an entry node and a node without any child nodes is called an exit node. The *makespan* of the application is calculated by taking the difference between the start time of the earliest entry node and the end time of the latest exit node.

We define the following DAG characteristics which can play important roles in determining how to schedule the tasks in the DAG:

1. Dag size (n)
2. Dag height (h), or number of levels

3. Average number of tasks per level (τ)
4. Communication-to-computation ration (CCR)
5. Parallelism (α)
6. Density (δ)
7. Regularity (β)
8. Mean computational cost (ω)

The DAG *size* refers to the number of tasks in the DAG. It is defined above as *n*. A *level* of a node, denoted by level(v_i), is defined as the length of the longest path from an entry node to node v_i. Here, we consider the length of the longest path to be the sum of all the nodes along the path. For example, all entry nodes are level 0; any children of entry level nodes is level 1; and any grandchildren of entry nodes is level 2. Note that nodes in the same level cannot have any dependencies among themselves and thus can theoretically be processed in parallel. We define height of the DAG (*h*), as the longest path from an entry node to an exit node, in number of nodes. It also refers to the number of levels in the DAG. We denote the set of all levels L = $\{l_1, l_2, \ldots, l_h\}$ where l_1 contains the first entry node and l_h contains the last exit node. We define the function size(l_k) to denote the number of tasks in level *k*. We define the average number of tasks per level to be τ, where $\tau = n/h$.

The *Communication-to-Computation Ratio* (CCR) refers to the average ratio of work done transferring intermediate files between tasks and the actual processing of tasks on any given host or processor. Thus, CCR is defined as the average of the cost of each edge e_k divided by the weight of each $p_e(e_k)$ for all $e_k \in E$ (both are in units of seconds):

$$CCR = \frac{1}{m}\sum_{k=1}^{m}\frac{w_e(e_k)}{w_v(p_e(e_k))}$$

The *parallelism* (α) parameter is derived from the number of tasks per level in the graph. Intuitively, we want to equate low α with low parallelism and high α with high parallelism so that when the number of tasks per level is 1 (in the case of a chain), α is 0 and when the number of tasks per level is equal to the DAG size (and there is only 1 level in the DAG), α is 1. We define parallelism as:

$$\alpha = \frac{\log(\tau)}{\log(n)}$$

where τ is the number of tasks per level and *n* is the DAG size.

The DAG *density* (δ) characterizes the number of dependencies for each task. A density value of 0.5 would mean that each task depends on 50% of the tasks in the previous level. A density value of 1 would mean that each task depends on 100% of the tasks in the previous level. We define density as the average percentage of tasks in the previous level with which each task in the current level has a dependency:

$$\delta = \frac{1}{n}\sum_{i=1}^{n}\frac{|\,\mathrm{Prev}(v_i)\,|}{size(level(v_i)-1)}$$

where by convention we assume that *size(-1) = 1* (for root nodes).

The DAG *regularity* (β) characterizes the regularity of the number of tasks at each level in the DAG. It quantifies the distribution of the number of tasks per level in the DAG. We allow for values between 0 and 1. A regularity value of 1 would mean that all levels have the same number of tasks. The lower the regularity value the higher the variance in the number of tasks per level. We define regularity as:

$$\beta = 1 - \frac{\max_{i=1,\ldots,k} |\,size(l_i)-\tau\,|}{\tau}$$

where τ is the average number of tasks per level.

The *mean computational cost* refers to the mean task runtimes for the tasks in the DAG.

Mean computational cost = $\frac{1}{n}\sum_{i=1}^{n} v_i$

III.1.1.1 Example DAG

We constructed an example DAG in Figure III-2 to illustrate the different DAG characteristics. This simple DAG has 8 nodes, so the DAG *size* or *n* is 8. The number of levels is 4, so $h = 4$ and L = $\{l_0, l_1, l_2, l_3\}$ where size(l_0) = 2, size(l_1) = 3, size(l_2) = 2, and size(l_3) = 1. Note that v_5 belongs in level 1 because it only has one dependency and that dependency comes from an entry node; v_7 belongs in level 2 because the longest path from an entry node is two, either the path composing of v_1 and v_4, or v_2 and v_4. The average number of tasks per level is $\tau = 8/4 = 2$.

The $CCR = \frac{1}{11}\left(\frac{5}{10}+\frac{5}{10}+\frac{3}{12}+\frac{3}{12}+\frac{3}{12}+\frac{4}{8}+\frac{4}{12}+\frac{4}{12}+\frac{5}{10}+\frac{5}{10}+\frac{3}{9}\right) = 0.386.$

The parallelism $\alpha = \frac{\log(\tau)}{\log(n)} = \frac{\log(2)}{\log(8)} = \frac{1}{3} = 0.333.$

The density $\delta = \frac{1}{6}\left(\frac{1}{2}+\frac{2}{2}+\frac{1}{2}+\frac{2}{3}+\frac{1}{3}+\frac{3}{3}\right) = 0.667.$

The regularity $\beta = 1 - \frac{(3-2)}{2} = 0.5.$

The mean computational cost is $\frac{1}{8}(10+12+8+12+9+10+10+9) = \frac{80}{8} = 10.$

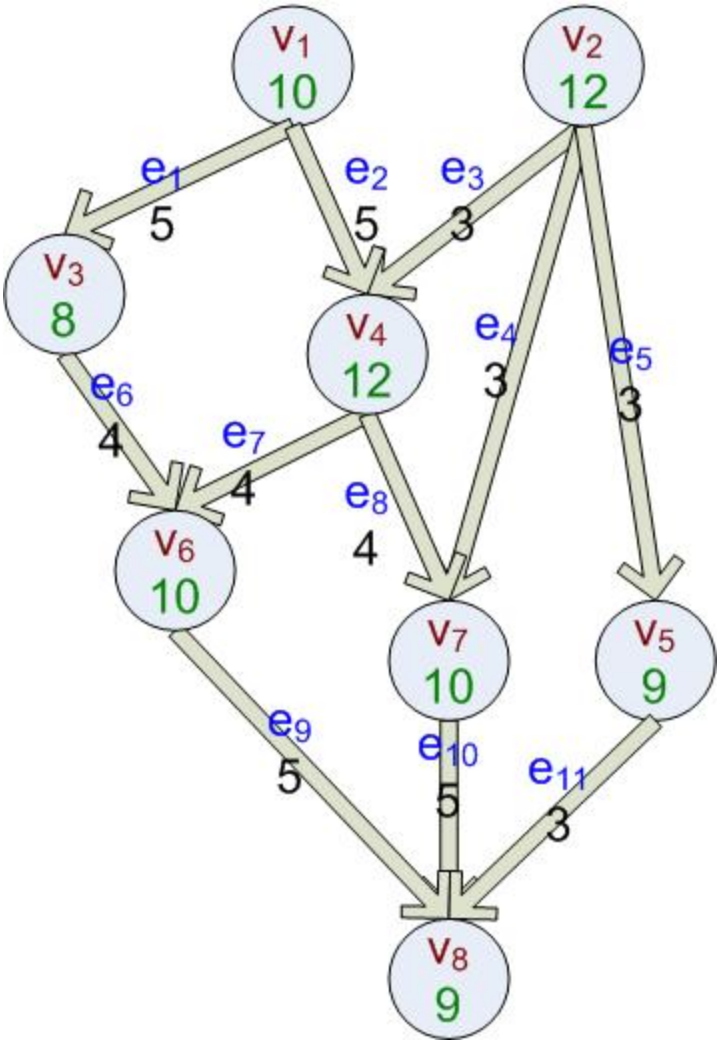

Figure III-2: Example DAG

III.1.2 Task Performance Models

The DAG characteristics defined above can only be computed with information about the application, namely, runtimes and data sizes. The results in this dissertation rely on such knowledge (sometimes referred to as *task performance models*). For some applications, this data is freely available. For instance, this is the case for the Montage application [61] because all tasks in Montage DAGs are operations that have been executed many times and researchers running Montage have constructed *performance models*. These models provide an accurate prediction of task execution time given a specification of the compute resource used. For other applications, the

task performance model might not be so readily available. However, in several instances, predictions are possible. For instance, predictive models are developed in [62]. When no information about the application is available, then scheduling is in some sense straightforward because there cannot be any sophisticated logic to choose one task over another task to run on a given resource. In this case, one typically resorts to some type of greedy scheduling algorithm.

III.2 Resource Model

Application tasks are executed on compute resources and application data are transferred using networking resources. We resort to using synthetic resources to simulate LSDEs for two reasons. First, we are interested in running experiments on very large scale distributed environments, larger than any such existing LSDE. Although these systems are not deployed today, they will in the near term future and we wish for our experiments to evaluate how our work will apply to these future systems. Second, even if these systems were already deployed today, running experiments on them would be expensive, time consuming, and most likely non-repeatable. Therefore, we resort to synthetic compute resource generators to simulate compute resources and topology generators to simulate network topologies. For full discourse on our use of simulation, please refer to Section III.4.1.

We formulate a compute resource model in section III.2.1 and a network model in section III.2.2. While models for physical resource are important, we also present a model for resource management in Section III.2.3.

III.2.1 Compute Resource Model

The trend we observe in recent years is one of steady growth for the number of clusters and number of CPUs. With dropping hardware prices for commodity computers, with several cluster vendors, and with the availability of open-source cluster management tools such as

ROCKS [22, 23], it is increasingly affordable and straightforward to purchase/deploy powerful Linux clusters. Figure III-3 shows the historical data for registered ROCKS clusters since its inception in 2003. The number of registered ROCKS clusters has increased by a factor of nine in just three years. Based on the recent trend, we model compute resources in an LSDE as thousands of clusters, with each cluster having as few as one or two processors and as many as thousands of processors.

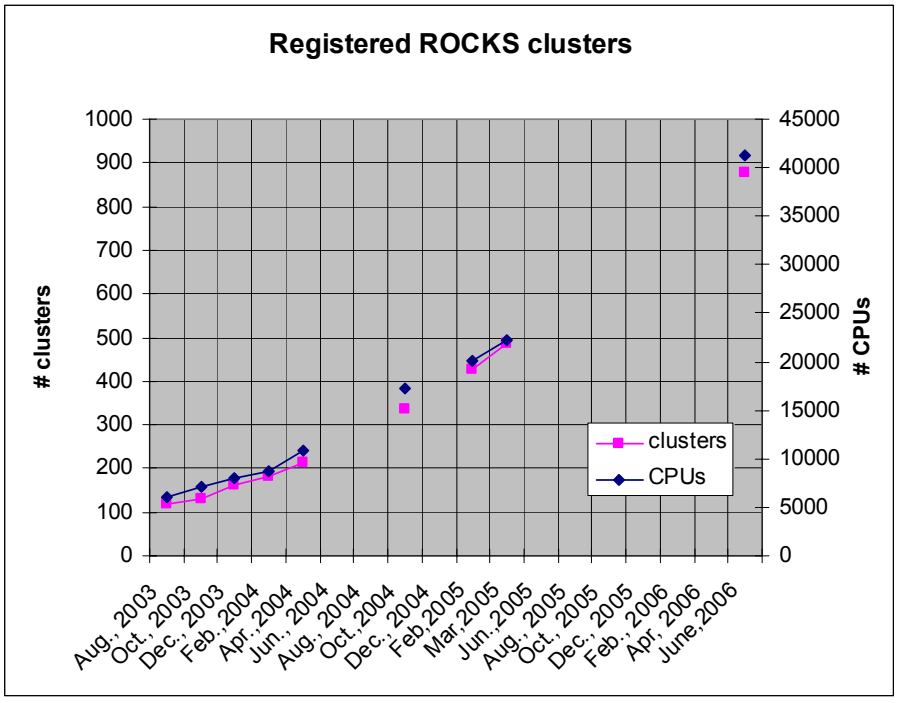

Figure III-3: Historical data for registered ROCKS clusters

To generate compute resources for our LSDE, we can reuse resource generators developed by other researchers in previous work [63, 64]. We need to decide which models/generators are more realistic for our LSDEs. Our requirements for choosing a compute resource generator include one that:

1. Provides realistic breakdowns of different clock speeds. For our work, we are ignoring the effects of heterogeneous processor architecture. The effects of heterogeneous processor architectures, while interesting, does not fundamentally change our results.

2. Generates resources structured as multiple cluster. Clusters are becoming more prevalent and are the most common high performance computing platforms today. Therefore we wish to generate multi-cluster LSDE synthetic platform configurations.
3. Realistically captures future technology trends, since we are interested in exploring the behavior of future LSDEs.

Kee, Casanova, and Chien [64] constructed a synthetic compute resource generator using statistical models for currently deployed resources and using the estimates for modeling characteristics of future LSDEs. One big advantage of using this compute resource generator is the flexibility in choosing the number of synthetic clusters and the ability to predict the composition of the clusters based on the desired year. The compute resource generator constructs a model for the computing resources in an LSDE by listing the different clusters in order with different characteristics for each cluster, such as the number of hosts and processors in the cluster, as well as the clock speed and the memory for each host in the cluster.

Other compute resource generators such as GridG [63] first generate a network topology, and then annotate network nodes with resources. GridG annotates nodes according to user supplied rules and empirical resource information. The rules capture potentially realistic correlations between number of processors, clock speed, memory size, and disk size and also operating system concentration within a local area network. However, such generators focus on the procedure of compute resource synthesis without evaluating the accuracy of their rules; thus the generated synthetic compute resources may or may not be representative of real world LSDEs. Another drawback is the lack of future forecast, which we deem important for larger LSDEs.

For our experiments, we choose Kee, Casanova, and Chien's compute resource generator because it fulfilled all of our requirements. The one drawback of this synthetic compute resource generator is the lack of topology or network connectivity between the clusters. Because of this,

we have a separate network model to model the topology of the LSDE and we merge the two models by mapping the nodes from our network model with the clusters generated by our synthetic compute resource generator.

III.2.2 Network Model

The field of realistic internet topology modeling, while not new, has no clear state-of-the-art or agreed upon model as the de facto method. The first network topology generator to become widely used was developed by Waxman [65] and it was based on probabilistic link creations between nodes. Later research on topology generators emphasized hierarchical structures, most notably Tiers [66]. A seminal paper in 1999 [67] brought to light the fact that degree distribution of router-level and AS-level Internet graphs exhibits power laws. GridG, as discussed in the previous section, is an extension of Tiers based on the power laws.

More recently proposed topology generators such as Inet [68] and BRITE [69] focus on the degrees of links between nodes instead of hierarchical structures. Even today, the debate goes on as to whether it is more important to follow the power laws for the node degree distribution or to model the Internet at a macroscopic level with the hierarchical structures.

For our experiments, we decided to use BRITE. Although it focuses on the power laws at the node level, it does include the option to create hierarchical structures at the macroscopic levels. BRITE assigns specific capacities to links based on current technologies. For example, router links could be OC3, OC12, OC48, 1Gb, or 10BaseT.

One thing BRITE (or any other topology generator) lacks is the modeling of contention on the links. Modeling the link contention is outside the scopes of this work, so we allow the bandwidth as specified by BRITE to be representative of bandwidth between the nodes. In reality at any given point in time, the bandwidth for any links might be somewhat smaller due to sharing and contention. Another factor in our decision is projects such as OptIPuter [25] which are

exploring protocols and application using optical fibers and very high bandwidth among nodes. We believe that the bandwidth as currently modeled by BRITE can be easily achieved in the near future even with link contention. Since for our experiments we are only interested in the communication-to-computation (CCR) ratio, we use a reference bandwidth to calculate the ratio and a contended link would imply a smaller reference bandwidth, but nevertheless a reference bandwidth. Therefore, higher or lower network contentions can be simulated by picking different ranges of the CCR value.

Although BRITE also assigns network latency between links, we ignore this latency in calculating the CCR because the latency is on the orders of milliseconds, negligible when both communication and computation are at least in the order of seconds. We acknowledge the fact network round-trip times, and thus network latencies, impact achievable bandwidth when using the TCP protocols. But again our use of a reference value for the bandwidth and our varying the CCR parameter allow us to explore a spectrum of relevant scenarios in our experiments.

III.2.3 Resource Management Model

Aside from how we model the physical resources, we need to consider resource management policies. One question is the cost of resources. Some systems such as the Grid Architecture for Computational Economy [70] considers resource economy; most other systems do not have resource economy as part of their motivation or goals. Typically, resource selection systems are more concerned with returning a set of resources quickly, or returning high quality resources, or both. The cost of obtaining a resource is typically considered as an additional constraint on the resource, and thus systems are not typically design to specifically consider resource economy.

In this dissertation, we are interested in the best way to produce the best application turn-around time, while also considering the cost of resource utilization. Application turn-around time

is the sum of the scheduler execution time and the application makespan (execution time of the application). We see two issues affecting our work regarding resource economy:

1. Paying for more resources as a possible way to speed up application turn-around time. The problem with this approach is that using more resource also increases the running time of the scheduler, thus increasing to the application turn-around time. Thus, using more resources does not always produce faster application turn-around time.
2. Request highly heterogeneous resources as a cost saving way to run applications. We investigate conditions for optimal application turn-around time by taking into consideration resource heterogeneity. Our work allows those with budgetary concerns to achieve the best application turn-around time given a certain level of heterogeneity among the resources (in the case that they can save money by using more heterogeneous resources).

Another issue with different resource managers is the idea of resource binding. Each administrative domain within a LSDE may have different resource managers, ranging from popular batch queue systems to those with advanced reservations to dedicated private resources. We do not address the issue of resource binding in this work. Instead, we make the assumption that the underlying Grid middleware can interact with each resource manager and bind the resources. From the point of view of each scheduling algorithm, the resources are either bound or not bound.

Although the Grid middleware can make repeated attempts to find resources when appropriate and available resources are found, one possible concern arises when the resource selection system cannot find available resources given the input resource specification. In this situation, the application or application user may need to degrade the resource specification so that the resource selection system can find available resources. We address this issue in Chapter VII.

For the purposes of this study, we assume that applications have dedicated access to bound resources. For time sharing resources, we can model the resource as available for only certain time slots and unavailable for the remaining slots. During the free slots, we assume that applications can gain dedicated access to the resource. For space sharing resources, we model the resource as being a fixed fraction of the capabilities of the actual resource. For example, for a processor with clock rate of 3.0 GHz that is being space shared by five virtual processors, we can model each virtual processor as having clock rate of 0.6GHz and any application using that virtual processor has dedicated access to the 0.6GHz processor. Examples of virtualization systems that enable the virtual processor concept, such as Xen [71] and ModelNet [72]. Additionally, we assume that task execution is non-preemptive on any given compute resource.

III.3 Application Scheduling Approaches on LSDEs

Along with an application model and a resource model, we need an application scheduler to assign tasks from the application to the compute resources. The application scheduler interacts with both the application and the resources. Applications provide the scheduler with relevant information regarding the DAG to execute. Depending on the scheduling heuristic employed, some or all of the application information may or may not be used. For example, a greedy scheduling heuristic may assign tasks to the first available resource without considering the cost of application data transfer.

After obtaining application information, the application scheduler can query the underlying middleware infrastructure to obtain information about the resources, such as resource availability and characteristics (e.g., clock rate). With both application and resource information, the scheduler can intelligently assign tasks to the resources by employing some type of scheduling heuristic.

The scheduler has two choices for selecting resources: *implicit* or *explicit* resource selection. With implicit resource selection, the scheduler uses a scheduling heuristic considering all LSDE resources to assign tasks from the application. It is implicit resource selection because the scheduling heuristic decides which resource is best for every single task and the scheduler merely selects and binds the resource assigned by the scheduling heuristic. The advantage of using implicit resource selection is that the scheduler can choose the best resource to execute each task and thus the application is likely to achieve as good performance as can be expected from the scheduling heuristic.

With explicit resource selection, the scheduler first narrows the scope of the LSDE to a smaller subset. The scheduler has the options of any number of resource selectors such as the vgES [16, 17] from VGrADS, Condor Matchmaker [9, 10], or SWORD [15] to execute the first step of resource selection. After the resource selection step, the scheduler employs a scheduling heuristic much similar to the scenario with the implicit resource selection.

With implicit resource selection, the majority of the resources are not used. With explicit resource selection, the scheduler can eliminate a lot of the resources unlikely to be assigned by the scheduling heuristic. The advantage is that the scheduling heuristic to assign tasks from the application can run much faster with a smaller resource set. The tradeoff is clear: potentially faster application makespan for implicit resource selection vs. potentially faster scheduling time for explicit resource selection. We aim to answer the interesting question of whether a resource selection step is necessary and/or preferred. We also aim to provide guidance for the resource selection step leading to the best application performance.

With or without explicit resource selection, the major component of the scheduler is the scheduling heuristic. For this study, we are interested in exploring a range of scheduling heuristics and how they affect application performance. We use a range of scheduling heuristics that reflect what is used in practice and also different representative classes of heuristics based on

how each heuristic treats the critical path of the applications. More details of each scheduling heuristic are discussed in Section V.6.

III.4 Methodology and Roadmap

In this section, we describe the methodology and the roadmap for running our experiments, as well as raise interesting questions we hope to answer with our experiments. We discuss our use our simulation for LSDEs in Section III.4.1 and describe the computing environment we use for running our experiments in Section III.4.2. In Section III.4.3, we describe the roadmap for running the experiments in this dissertation.

III.4.1 Simulation of LSDEs

In order for experimental results to be valid, the results need to be repeatable. When running experiments on real-world platforms where resources are often shared, it becomes extremely difficult to reproduce the exact same settings, thus results are often non-repeatable. Further, in any real-world platforms, different administrative domains have different fixed configurations and thus limit the range of possible experiments. Because the real-world platforms are in production, any experiments requiring modifications to the configurations would not be possible and any experiments would possibly disrupt users and perhaps cost money. Also, monetary issues arise because experiments can take a long time to run and monetary budgets may be limited. Lastly, there are few platforms today of the scale which we study in this work. We are interested in solutions that not only work today, but will work tomorrow when the platforms are larger.

Therefore, many researchers in the LSDE computing discipline resort to simulation for their experiments. Using simulations to artificially create large scale environments allow us to efficiently experiment with various types of resource heterogeneity in a repeatable manner.

Although we use simulated resource environments, we schedule applications using actual scheduling algorithms and instantiated based on real applications with performance models of task runtimes.

III.4.2 Computing Environment

As discussed in Section III.2.1 and Section III.2.2, we use a compute resource generator and a network topology generator to generate our LSDE. After we construct our LSDEs, we run different scheduling heuristics for our experiments. These scheduling heuristics are run on clusters from the Concurrent Systems and Architecture Group as well as the FWGrid Project, both at the University of California, San Diego. For our experiments, we use only machines with dual Intel Xeon processors with 2.80GHz clock rates running linux; thus the scheduling time reflect scheduling heuristics running on a 2.80GHz processor. Although our results reflect specifically clock rates of 2.80GHz, the general principles found in our results are applicable for faster clock rates, as one would simply adjust for the clock rate differences. In Section V.7, we study the impact of varying this reference clock rate of 2.80 GHz.

III.4.3 Roadmap

Here we present how we organize the experiments for this dissertation. The rest of the dissertation focuses on answering these two broad questions:

1. What is the best set of resources to use given an application and a scheduling heuristic?
2. How do we bridge the gap between applications and resource selection systems?

We decompose and refine these two broad questions as four more specific ones as follows:

Q1. What is the relationship between resource selection and application scheduling? (Chapter IV)

Q2. What is the best resource collection to use for best application performance? (Chapter V)

Q3. What is the best scheduling heuristic to use in conjunction with the best resource collection for best application performance? (Chapter VI)

Q4. How do we generate the best resource specification given the best heuristic and the best resource collection? (Chapter VII)

III.4.3.1 Role of Resource Selection

In Chapter IV, we answer Q1. Q1 can be expanded into two parts:

- How do we optimize application performance by resource selection?
- How can we simplify application scheduling by resource selection?

Addressing the first part of Q1, we want to determine whether explicit resource selection improves application performance by comparing three methods of resource selection:

- Implicit resource selection
- Explicit resource selection using naïve resource abstraction
- Explicit resource selection using more sophisticated resource abstraction

We use application performance from implicit resource selection as the baseline. Explicit resource selection is beneficial if applications achieve better performance than when using implicit resource selection. Explicit resource selection can be further broken down into resource selection using naïve resource abstraction or using more sophisticated resource abstraction. The naïve resource abstraction could be something such as “fastest CPUs”. If such a strategy always produces good performance, then no further sophisticated resource abstraction would be needed. However, applications have communication costs in addition to computational costs, so querying for the fastest CPUs may not always produce the best application performance. We need to

determine scenarios under which naïve resource abstractions would work and scenarios under which more sophisticated resource abstractions would work better.

Addressing the second part of Q1, we want to determine the type of resource abstraction used for resource selection that may simplify application scheduling. We want to compare using naïve and more sophisticated resource abstractions to determine whether using a more sophisticated resource abstraction can lead to simpler scheduling heuristics, without sacrificing application performance. Our hypothesis is that it may be possible that simpler scheduling heuristics can be used to achieve good application performance when given the appropriate resource collection. In fact, it may be possible that using simpler scheduling heuristic can lead to better application performance as long as the appropriate resource collection is used. Based on the results of Chapter IV, we formulate a plan in Chapter V-VII to generate the appropriate resource specification for explicit resource selection.

III.4.3.2 Best Resource Collection

In Chapter V, we address Q2. We construct an empirical model to predict the best resource collection size given an input application and a reference scheduling heuristic. We construct the prediction model systematically by the following steps:

1. Define the specifications for best resource collections.
2. Determine relevant application characteristics (from the set of characteristics defined in Section III.1.1) that influence the best resource collection.
3. Using the relevant application characteristics, construct a model to predict the best resource collection size assuming homogeneous resources.
4. Verify that application performance using specification predicted by our model matches closely with the optimal application performance using a reference scheduling heuristic.

We verify using arbitrarily generated application DAGs and DAGs instantiated based on real applications.

5. Expand the prediction model to include heterogeneous resources.
6. Conduct sensitivity analysis on our empirical model for different scheduling heuristics.
7. Conduct experiments determining the effect of using reference clock rates for the scheduler and for computational hosts.

Once we can predict the best resource collection size, the next step is to determine the best heuristic to use given an input DAG. With the best scheduling heuristic and the best size, along with analysis for clock rate heterogeneity, we can generate the best resource specifications for different resource selection frameworks.

III.4.3.3 Best Scheduling Heuristic

We address Q3 in Chapter VI. Application performance depends not only on the physical resource characteristics, but also on the scheduling heuristic. In Chapter V, we construct a prediction model to predict the best resource collection size given an input DAG and an input scheduling heuristic. In Chapter VI, we construct a predictive model to predict the best scheduling heuristic given an input DAG that we can use in conjunction with the prediction model from Chapter V. We choose different scheduling heuristics ranging from ones commonly used in practice to more sophisticated heuristics. All experiments in Chapter V (except for the sensitivity analysis) use the MCP scheduling heuristic, a relatively fast scheduling heuristic that also considers communication costs.

The most important goal in Chapter VI is to provide a comparison and recommendation for the best combination of scheduling heuristic and resource collection given any DAG application. Different users or applications may be constrained by the complexity of scheduling heuristic or by how much resource heterogeneity the application can tolerate. By addressing the

needs and constraints of each user or application, we hope to identify the best scheduling heuristic that can be used in conjunction with the best resource collection specification leading to the best application performance for each user or application.

A secondary goal is to answer the question of whether using appropriate resource collections can allow applications to employ simpler scheduling heuristics while achieving as good (if not better) performance. We view this as an important goal because it would enable application developers to focus their efforts on developing applications and resort to simpler scheduling heuristics to achieve similar application performance. We identify scenarios under which simple scheduling heuristics is preferred for better application performance.

III.4.3.4 Generating Resource Specification

We address Q4 in Chapter VII. Our overall goal is to bridge the gap between applications and resource selection systems. In Chapters V and VI, we construct models to predict the best resource collection to use in conjunction with the best scheduling heuristic. In Chapter VII, we combine the outputs from these models and some of our assumptions about the resource environment to automatically generate resource specifications for different resource selection systems.

IV

RESOURCE SELECTION AND APPLICATION SCHEDULING

In this chapter, we examine the role of resource selection in optimizing application performance. Resource selection is the process of finding the best set of resources to run an application. Fundamentally, resource selection is a part of scheduling. An application scheduler typically aims both at finding the best possible resources (resource selection), and ordering the execution of tasks on these resources.

Given the goal of minimizing application turn-around time, one possible improvement is through minimizing the scheduling time. A major portion of the scheduling time is due to resource selection. Any scheduling heuristic whose running time is a function of the size of the resource universe will take longer to run with an increasingly larger resource universe. By reducing the size of the resource universe, the scheduling time will also be reduced. Thus, one possibility to reduce overall application turn-around time is to explicitly reduce the number of resources given to the scheduling heuristic.

In this chapter, our goals are to answer the following questions about the role of explicit resource selection:

1. Is explicit resource selection beneficial? (Does is lead to faster application turn-around time?)
2. What types of resource abstractions are required/necessary to perform explicit resource selection?
    a. Can we naively reduce the size of the resource universe?

b. Are more sophisticated resource abstractions required?

3. How do resource abstractions affect the complexity of scheduling heuristics?
   a. Can we simplify scheduling when we give the scheduler an appropriate set of resources to work with?
   b. Under what conditions is such a simplification possible?

IV.1 Application Scheduling in LSDEs

Users of scientific applications, and in particular of scientific workflows, are increasingly faced with situations in which they have to select appropriate compute resources among a large number of potential resources distributed over the wide-area. This is due to two factors. First, with dropping hardware prices for commodity computers, with several cluster vendors, and with the availability of open-source cluster management tools such as ROCKS [22, 23], it is increasingly affordable and straightforward to purchase/deploy powerful Linux clusters. Therefore, an increasing number of users have access to an increasing number of clusters. Second, the development of the grid middleware infrastructure such as Globus [8] makes it straightforward for users to access a wide collection of resources uniformly and securely. Additionally, with projects exploring optical networks and providing high bandwidth among many clusters [25], there is a trend towards resource-rich environments with good network connectivity in which users can access many clusters in many institutions concurrently. Workflow applications can benefit from such environments because they are often loosely coupled and can utilize resources at multiple sites concurrently and efficiently.

IV.1.1 Challenges of Scheduling in LSDEs

With the explosion in the number of computing resources, one important challenge for scheduling workflows is scalability of the scheduling algorithm itself. Even when of polynomial

complexity, DAG scheduling heuristics may become impractical when considering large numbers of individual resources. More importantly perhaps, existing heuristics require information about individual resources and about their distances from each other over the network. Collecting and processing reasonably up-to-date such information may itself not be scalable. There is therefore a trade-off between the time spent computing a schedule (perhaps prohibitively high for a sophisticated heuristics, but low for a simple one) and the time spent executing it (arguably low for a sophisticated heuristic, but probably high for a simple one).

IV.1.2 Current Scheduling Approaches

As seen in [73], DAG scheduling heuristics that calculate and account for the "critical path" of the DAG are often the most effective. The critical path is essentially the longest path in the DAG (in terms of node and edge weights), and is thus a lower bound on the overall makespan. These heuristics attempt to lower this lower bound in the hope of lowering the makespan.

In practice however, for the purpose of scheduling grid workflows, these heuristics are not used. For instance, the Pegasus grid workflow framework [74, 75] implements only the simplistic random, round-robin, or min-min [76] heuristics for scheduling workflows of the Montage astronomy application [57, 77].

There are several reasons for the lack of acceptance of more sophisticated scheduling algorithms. First, these algorithms are more complicated to implement. Second, they often require more information about the application and/or the resources, which may be difficult to obtain scalably. Third, there has been no clear demonstration that they would improve application turn-around time in practice (i.e., achieve a good trade-off between the time to compute a schedule and the time to execute it).

In this chapter, we are interested in answering the question of whether simpler scheduling heuristics can be employed to achieve good application performance. One of our goals is to show

that although the use of sophisticated algorithms may be worthwhile, simplistic algorithms can achieve comparable or even better application turn-around time in many relevant cases, provided that resources are preselected appropriately.

IV.1.3 Resource Selection

Much research [9-13] has gone into resource selection without considering the impact on scheduling heuristics and the impact on application performance. Typically, the goals of resource selectors are to match the needs of the application with available resources by selecting the set of resources that best meet resource requirement specifications. While lacking any evidence, all known resource selectors make the assumption that the application (or user using the application) can supply the appropriate resource specifications to best optimize the application performance. We address the issue of generating the resource specifications that best meet each application in Chapters V-VII. In this chapter, we address the issue of whether explicitly selecting resources can indeed improve application performance.

IV.2 Experimental Approach

Our goals are to determine whether explicit resource selection is beneficial for application performance and whether more sophisticated resource abstractions are necessary (instead of using naïve resource abstractions) for better application performance. We perform the following experiments. We use DAGs from a real-world grid workflow application, Montage [77], as well as randomly generated DAGs to better understand the impact of DAG characteristics on our results. We consider a computing platform generated by a resource generator [64] (discussed in detail in Section III.2) that instantiates synthetic large-scale computing environments that are representative of current technology.

Using simulation we execute two different scheduling algorithms: a naïve greedy heuristic (which we call "simple") and a popular DAG scheduling heuristic (which we call "complex"). We execute these algorithms in three modes:

1. On the whole "resource universe" without pre-selection of resources.
2. Only on some pre-selected "top" fraction of the resources sorted by clock rate.
3. Only on pre-selected resources that have been obtained as part of a more sophisticated resource abstraction.

In Section II.4, we discuss three systems for resource selection in LSDEs. For this experiment, we use the Virtual Grid Execution System (vgES) [16, 17] to compose a Virtual Grid (VG) as our sophisticated resource abstraction. We obtain the VG by querying a vgES prototype, which has stored resource information corresponding to our synthetic computing environment. For the "top hosts", we also use vgES to return the fastest fraction of the resource universe. Therefore, we conduct 6 different types of experiments, as summarized in Table IV-1. We provide details on all the above in the following sections.

Table IV-1: Scheduling schemes in Grid environments

Scheduling Heuristic	Resources
Complex	Universe
Complex	Top Hosts
Complex	VG
Simple	Universe
Simple	Top Hosts
Simple	VG

IV.2.1 Real Application - Montage

Montage is an astronomy application that creates a mosaic image of a portion of the sky on demand. Figure IV-1 shows the structure of a small Montage workflow. All tasks on level k have a parent task on level k-1. The top-level tasks (level 1) are not dependent on any other tasks.

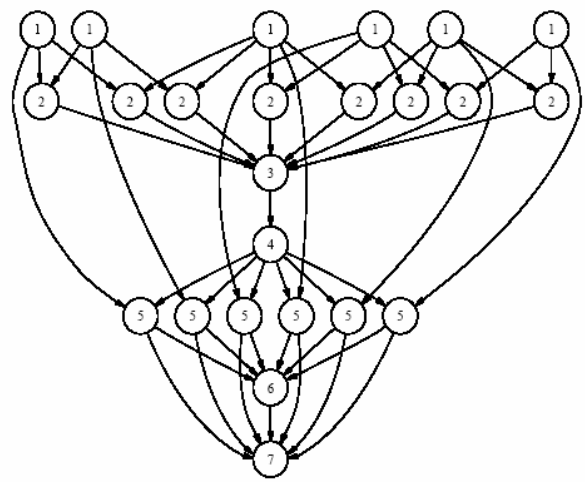

Figure IV-1: A small Montage workflow

Table IV-2: Runtime and number of tasks at various levels of a Montage workflow

Level	Task name	Task purpose	Number of Tasks	Runtime (in seconds)
1	mProject	Re-projection of images	892	8.2
2	mDiffFit	Calculating difference in images	2633	2
3	mConcatFit	Fitting images to common plane	1	68
4	mBgModel	Modeling background	1	56
5	mBackground	Background correction	892	1
6	mImgtbl	Adding images to get final mosaic	25	6
7	mAdd	Registering the mosaic	25	40

For our experiments, we consider a 4469-task Montage workflow used to create a five square degree mosaic of the sky centered at the M16 region of the sky. The M16 [78], also known as the Eagle Nebula in the constellation Serpens, is one of the most famous and easily recognized space objects. Table IV-2 shows the average runtimes of Montage tasks on a 1.5Ghz host as reported in [61]. We are interested in seeing how communication might affect scheduling. Therefore, for each Montage workflow, we vary the communication-to-computation ratio (CCR). We test ratios of 0.1, 0.5, 1.0, 2.0, and 10.0. A ratio of 1 implies equal amount of computation and communication. For each task, we calculate the size of its output file based on the computational cost, the CCR, and the maximum bandwidth in the network, which in our case is

10Gbps. For example, for a CCR of 1, we derive the appropriate file size such that the communication cost would be the same as the computational cost (e.g. 8.2 seconds for the level 1 task in Montage). In this case, the files size would be 152MB, as it would take 8.2 seconds to transfer this on the fastest link in our synthetic platform.

IV.2.2 Random DAGs

We also generate a collection of random DAGs, defined by the following characteristics. For each random graph, we vary its size, its mean computation cost (using a 1.5Ghz host as the reference), its communication-to-computation ratio (CCR), its parallelism, its density, its regularity, and its mean task computational cost. The parallelism characterizes the width of the DAG; density characterizes the number of edges; regularity determines the variance in the number of tasks at each level. (See Section III.1.1 for the full definitions of these DAG characteristics.) Table IV-3 summarizes the different characteristic and their corresponding values for the random DAGs we generate. While generating random DAGs, and so as to keep the number of DAG configurations tractable, we vary a single characteristic and set all other characteristics to the default values shown in the table.

Table IV-3: DAG characteristics and corresponding values for random DAG generation

DAG Characteristic	Values	Default Value
DAG size (tasks)	44, 447, 4469, 8938	4469
CCR	0.1,0.2,1,2,10	1
Parallelism	0.1,0.2,0.5,0.8,1	0.5
Density	0.1,0.2,0.5,0.8,1	0.5
Regularity	0.1,0.2,0.5,0.8,1	0.5
Mean comp cost	1,5,40,100	40

IV.2.3 Scheduling Heuristics

Among all the DAG scheduling algorithms surveyed and evaluated in [73] we choose the popular MCP (Modified Critical Path) heuristic [31] for the experiments in this chapter, as it is competitive according the results in [73]. MCP is our "complex" scheduling algorithm. The pseudo code for MCP is shown in Figure IV-2.

For our "simple" scheduling algorithm we use a greedy scheduling algorithm that assigns each task to a random available host as soon as the task's dependencies have cleared. The corresponding pseudo code is shown in Figure IV-3.

We expect that running a more complex scheduling algorithm such as MCP on the resource universe would produce the best makespan by taking into consideration all the resources. We hope that appropriate resource pre-selection would allow a simple scheduling algorithm to achieve better trade-off between the time to compute a schedule and the time to execute the schedule, thereby leading to better turn-around time.

```
CP = length of the longest path (in terms of node weights
     and edge weights) from the root node to the end
     node, including both these nodes
For each non-root node N_i in the DAG
  BL_i = length of the longest path (in terms of node
         weights and edge weights) from node N_i to the
         end node, including both these nodes
  ALAP_i = CP – BL_i
End For
For each node N_i
  L_i = list of the ALAP values of node Ni and all its
        descendents, in ascending order
End For
Sort all L_i lists in lexicographical order and
Re-Order the nodes according to this order
For each node N_i
  Schedule Ni on the host that would complete its
  execution soonest
End For
```

Figure IV-2: Modified Critical Path (MCP) Algorithm

```
While there are still some tasks to schedule
   For each node N_i whose predecessors, if any,
            have already been scheduled
      Schedule N_i on the host that would start its execution
            soonest
   End For
End While
```

Figure IV-3: Simple Greedy Algorithm

IV.2.4 Resources

We are interested in scheduling applications in LSDEs. For the reasons discussed in Section III.4.1, we use simulation of a synthetic resource pool. We use the synthetic resource generator described by Section III.2 to generate 1000 clusters for a total of 33,667 hosts.

When scheduling and simulating the execution of workflows we ignore the architectures of the hosts and use only clock rates to determine task runtimes. We scale the reference task runtimes (which are for a 1.5GHz host) to account for lower or higher clock rates.

IV.2.4.1 Naïve Resource Abstraction

To test whether more sophisticated resource abstractions are needed, we experiment with naïve resource abstractions. A simple naïve resource abstraction is "top hosts" denoting a certain number or fraction of the fastest nodes available when the nodes are sorted by clock rates. We choose the fastest 2633 hosts as the resource set returned by vgES. The widest part of the Montage DAG is 2633 tasks and choosing the same number of hosts ensures that maximum parallelism would be possible when the scheduling heuristic assigns each of the tasks in the widest part to a distinct host.

IV.2.4.2 Sophisticated Resource Abstraction

In general, one can expect that using an unlimited number of the fastest machines in a cluster will lead to the lowest application makespans. Unfortunately, the number of nodes in a

cluster is limited. In fact, the fastest clusters might not always be the biggest clusters. Furthermore, it may be best to use multiple clusters provided they are not too "far" from each other.

The above is exactly the sort of trade-offs that make scheduling difficult. For our experiments, we use the sophisticated Virtual Grid (VG) resource abstraction provided by the Virtual Grid Execution System. The VG abstraction allows users the luxury of asking for a TightBag (that is sets of heterogeneous hosts that are "close"), with a parameter to determine what "close" means. The vgES will identify such a TightBag quickly, even in large-scale environments [17]. Our approach focuses on finding an appropriate TightBag for a given DAG. For instance, for the Montage workflow described in Table IV-2, we can write the vgDL specification shown in Figure IV-4, which asks for a TightBag containing between 500 and 2633 hosts, where hosts have clock rates higher than 3Ghz. We choose 2633 as the upper bound on the number of hosts in the VG as this represents the widest portion of the Montage DAG, using the same rationale as in Section IV.2.4.1. The [rank = Nodes] statement just means that a larger TightBag is preferable. (Section II.4.1.1 discusses vgDL in more detail.) When the resource platform does not contain the number of resources we want (2633) for a TightBag, we can specify the willingness to accept fewer resources (in this case as few as 500). In our synthetic resource environment such a request happens to returns a VG containing 924 hosts.

```
VG = TightBagOf(nodes) [500:2633]
[rank = Nodes] {
   nodes =   [ (Clock>=3000) ]
}
```

Figure IV-4: vgDL used for the Montage workflow

IV.3 Results

The main results from our experiments is that explicit resource selection is always preferable to allowing the scheduling heuristic to implicitly select resources. This is especially true when using a simple naïve greedy scheduling heuristic that does not select the best resource to execute a task. When a more sophisticated resource abstraction was used, the simple greedy scheduling heuristic was able to achieve better application turn-around time than the more sophisticated MCP scheduling heuristic in some cases. We discuss below specific results for Montage and randomly generated DAGs. We compute a lower bound on application makespan by assuming all tasks run on hosts as fast as the fastest available host and that all data transfers take place on network links as fast as the fastest network link available.

IV.3.1 Montage Results

Figure IV-5 and Figure IV-6 show results for the Montage workflow using the MCP and the simple greedy scheduling heuristic. Results include the time to compute the schedule, the application makespan resulting from the schedule, the time to obtain a VG when applicable, and the total application turn-around time including all of the above.

The results in Figure IV-5 are for the actual Montage communication costs. The intermediate files generated by different stages ranged from 300 bytes to 4 megabytes, so communication costs were relatively low. The conclusion from these results is that running the greedy heuristic on a VG achieves the best application turn-around time overall (within 8% of the ideal lower bound), if not the best makespan. The best makespan is achieved when running MCP on the whole resource universe, but this makespan comes with a prohibitive scheduling cost. Running on Top Hosts (fastest hosts) gives good performance (if not best) because communication costs are low. Interestingly, running the greedy algorithm on the whole resource

universe still outperforms running MCP on the whole universe in spite of poor makespan since the time to compute the MCP schedule is so high.

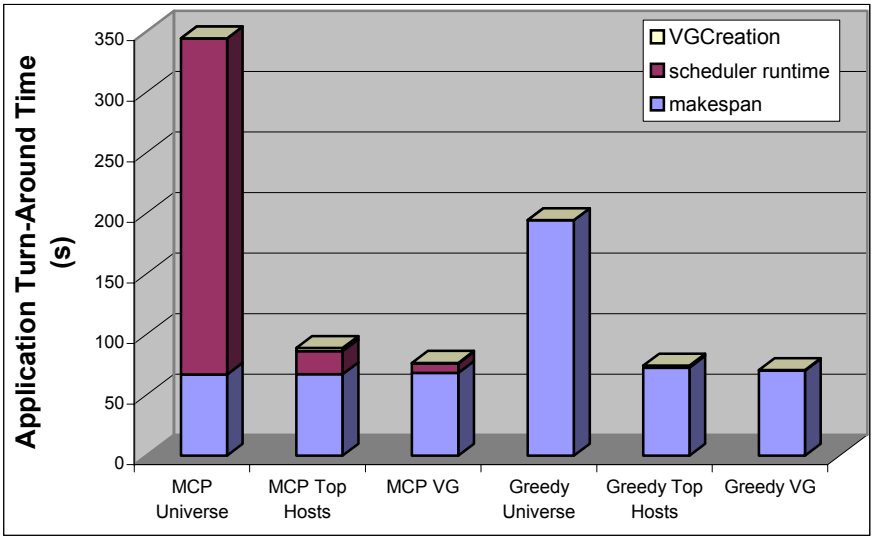

Figure IV-5: Running Montage Workflow with actual communication costs

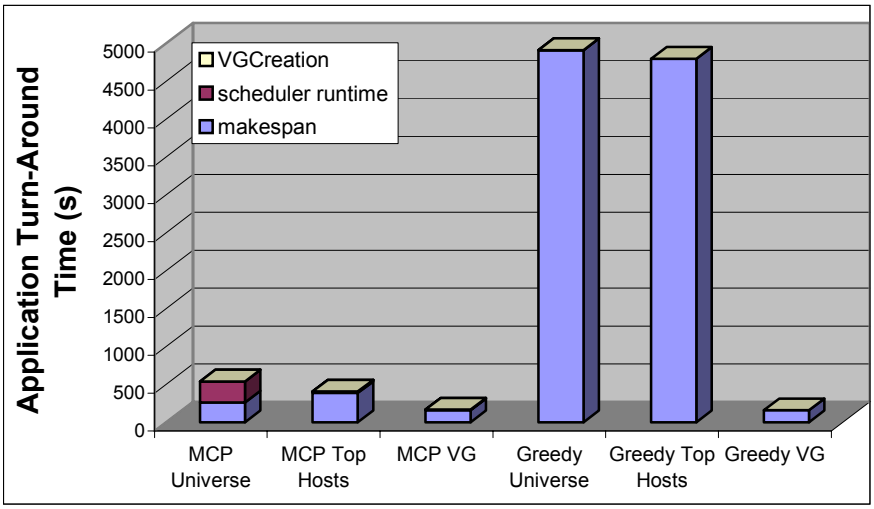

Figure IV-6: Running Montage workflow with equal communication and computation costs

Figure IV-6 shows similar results for a CCR value of 1, which is balanced communication and computation cost. Here, it is not enough to simply schedule tasks on the fastest machines as communication costs matter, and the benefits of using a VG are plain. Surprisingly, running the greedy algorithm on a VG produces a better makespan than running MCP on the resource universe. This is because MCP is merely a heuristic with no guarantees. It

makes greedy decisions based on the relations between tasks and the critical path, disregarding possibly harmful effects due to task dependencies. More sophisticated scheduling algorithms may or may not lead to better makespans in our experimental setting. At any rate, using a simple greedy scheduling algorithm is as effective once resources have been pre-selected.

IV.3.1.1 Varying CCR

Figure IV-7 shows the ratio of Montage makespans as compared to running MCP on the universe, for increasing CCRs. One striking result is that when the CCR is increased, either algorithm running on the VG can construct schedules with much shorter makespans than the schedule MCP can construct on the whole resource universe.

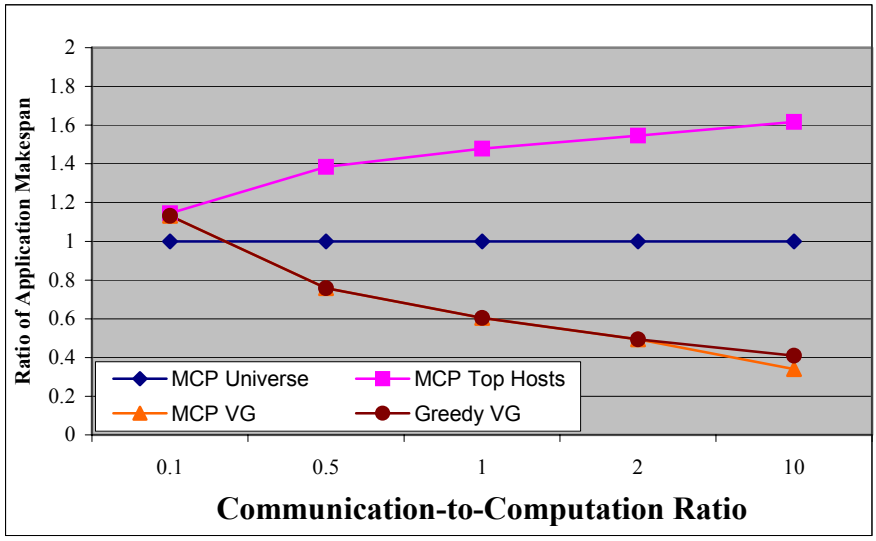

Figure IV-7: Ratio of Montage makespan compared to running MCP on universe while varying CCR

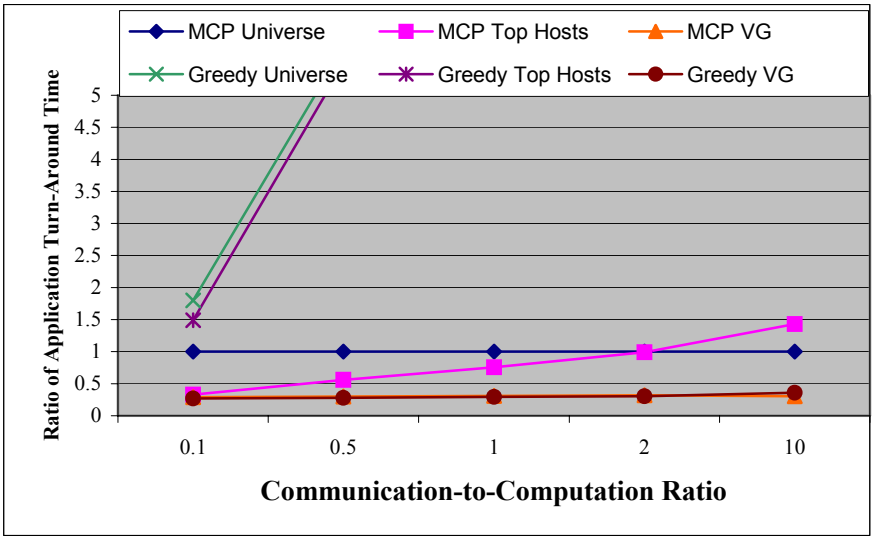

Figure IV-8: Ratio of Montage makespan compared to running MCP on universe while varying CCR

For most CCRs, when using the VG, no differences exist between using the greedy algorithm and MCP. Only when the CCR is very high do we notice a slight improvement in performance when MCP is used. The makespan for the greedy algorithm running on either the top hosts or the universe were 6 to 23 times longer than the MCP on universe makespan. We contend that this such high CCR values are not likely for most workflow applications intended to run on LSDEs.

We show Figure IV-8 to highlight the definite advantage of using a more sophisticated resource abstraction (VG in this case). When taking the scheduling time into consideration, using either algorithm running on the VG achieves application turn-around time less than 30% of the turn-around time needed to run MCP on the universe.

IV.3.2 Random DAGs

We generate random DAGs according to the characteristics in Table IV-3. When varying a single characteristic all other characteristics take the default values shown in the table. In some cases the application turn-around time for running the greedy heuristic on the resource universe

were so large that we left them out of the figures. Each data point is averaged over 10 distinct instances of random DAGs. The coefficients of variation for these samples were all within 3%, except for the case of running MCP on the universe, which ranged from 1% to 73%.

IV.3.2.1 Varying DAG size

As we vary the DAG sizes, we needed to vary the corresponding vgDLs to create different VGs for each DAG size (that is larger VGs for larger DAG widths). Expectedly, the scheduling time for running MCP increases as the DAG sizes increased. However, because of the relative small sizes of the VGs compared to the universe, this increase was only marginal. The bulk of the application turn-around time when using MCP on the whole resource universe is due to the application makespan. We also observed no significant makespan differences between running the MCP scheduling heuristic on VG and running the greedy scheduling heuristic on VG. Figure IV-9 shows the ratios of the application turn-around times compared to running the greedy heuristic on VG. For bigger DAGs, one can see that there is little difference between running the greedy heuristic and running the MCP heuristic on VG in terms of turn-around time. For smaller DAGs, because of smaller turn-around time, the difference between using the MCP heuristic and using the greedy heuristic on the VG is magnified.

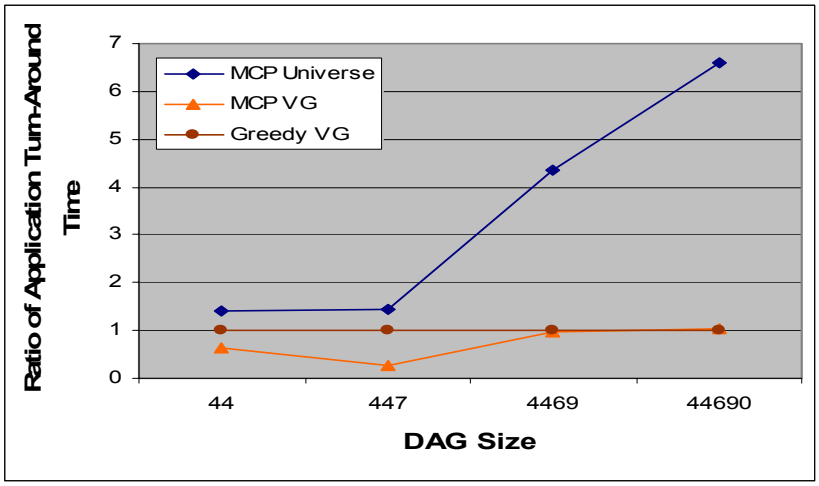

Figure IV-9: Varying DAG sizes for random DAGs

IV.3.2.2 Varying CCR

As with Montage, we wanted to investigate whether the greedy on VG approach would tolerate high-communication scenarios. Figure IV-10 shows that running the greedy on VG is within only 4% of results for running the MCP on VG for all CCR values. The performance of running the greedy on the universe was between 16 and 62 times the application turn-around time for running the greedy on VG.

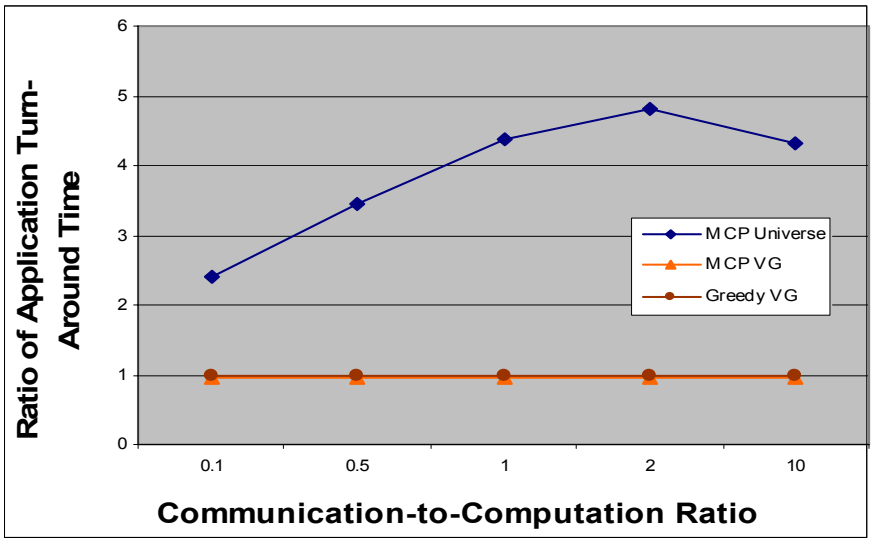

Figure IV-10: Varying CCR for random DAGs

IV.3.2.3 Varying Parallelism

When the parallelism of a DAG (as defined in Section III.1.1) is 0, then the DAG is just a chain of tasks where each task depends on the previous task. Scheduling consists of finding the fastest host. When the parallelism is 1, all of the tasks can be run in parallel and scheduling consists of finding the fastest N hosts for each of the N tasks in the DAG.

Figure IV-11 shows results for varying DAG parallelisms. We see that at parallelism of 0.5 or higher, running the greedy heuristic on the VG has comparable performance to running the MCP heuristic on the VG. For parallelism of 0.8, running the greedy heuristic is actually preferable to running MCP due to MCP taking more time to compute the schedule because of the

increased number of tasks at each level. However, we see the limitation of using the VG as a means for good performance when the parallelism is below 0.5. (A value of 0.5 implies that the number of tasks per stage is equivalent to the square root of the total number of tasks in the DAG.)

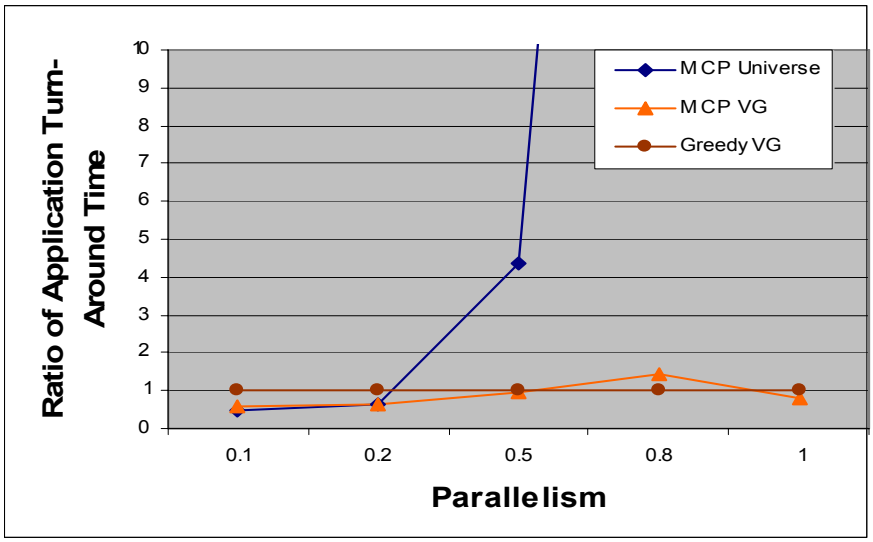

Figure IV-11: Varying parallelism for random DAGs

The poorer performance while running the greedy heuristic for less parallel DAGs is due to increased communication costs, or rather, the lack of opportune communications savings. Whereas MCP actively seeks to minimize communication costs by calculating the tradeoff between scheduling two tasks on the same host sequentially, which would lead to longer computational time but zero communication costs, the greedy algorithm would greedily schedule the two tasks on separate hosts whenever the second host becomes available. Of course, note that a minor modification of our greedy algorithm could alleviate this deficiency (e.g., always try to reuse a host that has been used before). Nevertheless, while the implication of Figure IV-11 is that when workflows are not highly parallel our approach is not effective, it is reasonable to expect that many applications will in fact have parallelism higher than 0.5 and thus not mandate anything more sophisticated than our greedy heuristic.

IV.3.2.4 Varying Density

The density of a DAG determines the number of dependencies among the tasks. A density of 0.5 means that each task depends on 50% of the tasks in the previous level. Here again we found that scheduling on a VG greatly outperforms scheduling on the whole universe of resources. The application turn-around time for running the MCP heuristic on the universe is 3 to 15 times larger than running the greedy heuristic on a VG, depending on the density of the DAG. Figure IV-12 shows that running the MCP heuristic on a VG outperforms running the greedy heuristic on a VG in most cases. For densities higher than 0.5 the difference is below 4%, but it is up to 15% for a density of 0.1.

MCP was able to achieve better application performance as the number of dependencies decreased because it was able to schedule some of the tasks on the same hosts as their parents, particularly tasks that have one parent task. As the number of dependencies decreases, unlike the greedy heuristic, MCP can increasingly optimize the communication costs.

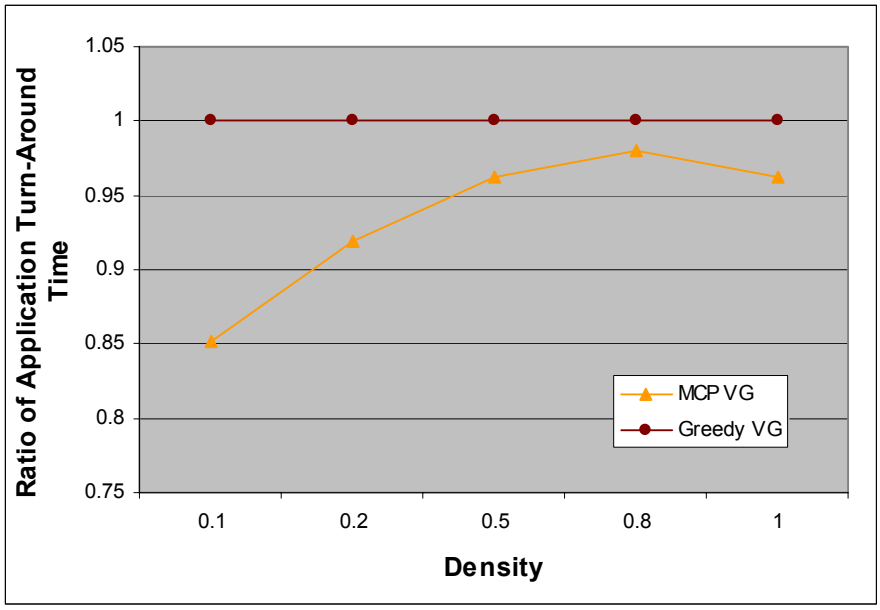

Figure IV-12: Varying density for random DAGs

IV.3.2.5 Varying Regularity

Regularity quantifies the distribution of the number of tasks per level in the DAG. A regularity of 1 means that all levels have the same number of tasks. The lower the granularity the higher the variance in the numbers of tasks per level. Here again, using a VG is preferable to using the whole resource universe. Figure IV-13 shows that with the appropriate VG, running a greedy heuristic can create a schedule with makespans more than ten times shorter than running the MCP scheduling heuristic on the universe when the DAG is highly irregular. Performance is more than fifty times better (not shown) when compared to greedy running on the whole universe of resources. We see that for any regularity type, the greedy algorithm running on the VG performs within 3% of MCP running on the VG.

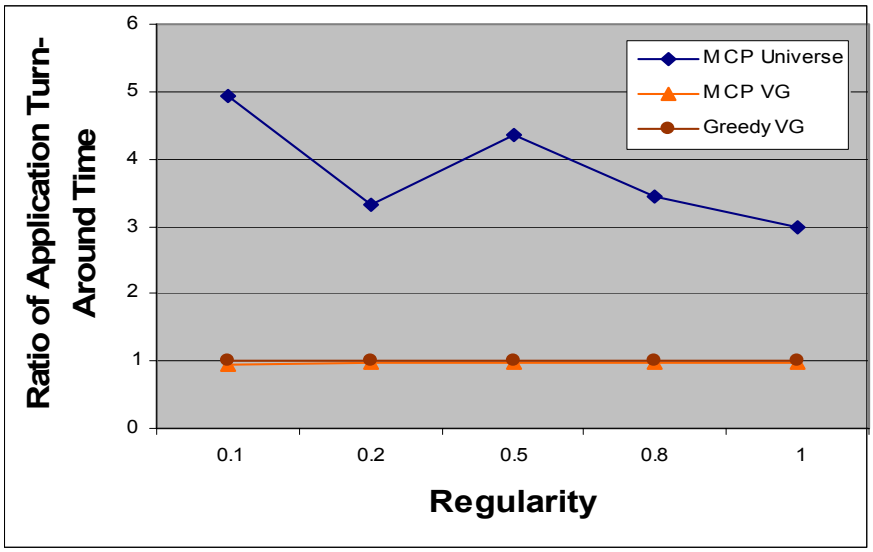

Figure IV-13: Varying regularity for random DAGs

IV.3.2.6 Varying Mean Computational Cost

The mean computational cost refers to the mean execution times for the tasks in the DAG. Varying the mean computational cost makes very little difference between running the greedy heuristic or running the MCP heuristic on the VG, as seen in Figure IV-14. For any mean

computational cost, the greedy algorithm running on the VG performs within 4% of MCP running on the VG. Here again, using a VG greatly outperforms using the whole resource universe.

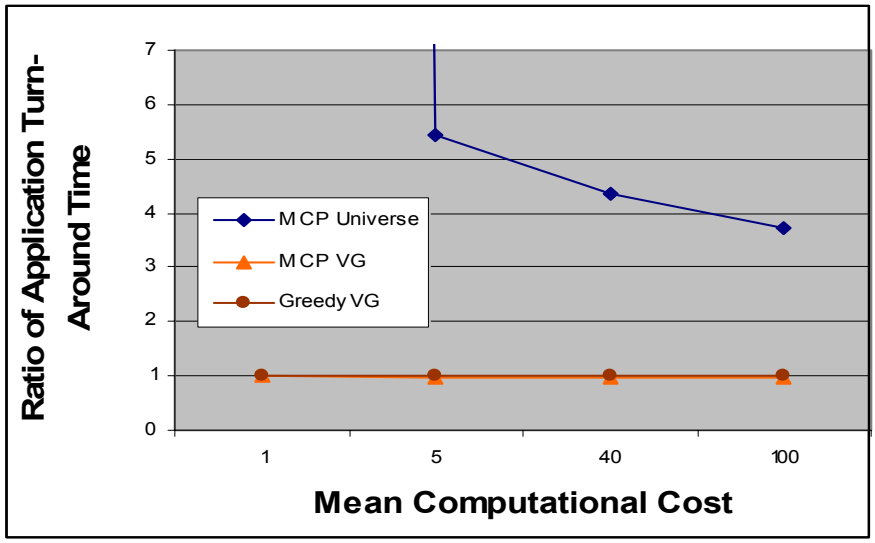

Figure IV-14: Varying mean computational costs for random DAGs

IV.4 Conclusion

Our number one goal in this chapter was to determine whether explicit resource selection is beneficial to application performance. Using both a simplistic greedy scheduling heuristic and a more sophisticated MCP scheduling heuristic, we have shown that for both the Montage application and a spectrum of randomly generated DAGs, explicitly pre-selecting resources before running the scheduling heuristic on the subset of the resource universe always improved application performance, sometimes by several orders of magnitude. This held true with both types of resource abstractions we used – the simplistic "top hosts" and also the VG from vgES.

Our second goal was to determine whether (and under what conditions) more sophisticated resource abstractions such as the VG is necessary. We found that the naïve resource abstraction, which does not account for networking information, does not perform as well when communication costs are not insignificant. For an application such as Montage where the

communication costs are minimal, using the naïve resource abstraction led to similar application performance results as using more sophisticated resource abstractions.

Our third goal was to determine how resource abstractions affect the complexity of the scheduling heuristics. We found that using a sophisticated resource abstraction such as the VG enabled a simple greedy scheduling heuristic to achieve better application turn-around time than a more sophisticated scheduling heuristic (MCP) for the Montage application and some of the randomly generated DAGs. In almost all of the scenarios we tested, the greedy heuristic running on the VG performs within 4% of the MCP heuristic running on the VG. The only limitations we found for using the greedy heuristic on the VG occurs when the DAG is very sparse, either due to low parallelism or low number of dependencies among the tasks.

What we have shown is that under most conditions, when one explicitly selects an appropriate resource collection (such as a VG), a simplistic scheduling heuristic can be employed to achieve similar to better performance than using a more sophisticated scheduling heuristic. What is not clear is how to compose such an appropriate resource collection. In terms of the vgES, what is not clear is how to generate the vgDL so that the vgES can return a VG. In our experiments, we generated the vgDL based purely on the width of the DAG, while allowing the vgES flexibility in the returned resulting VG by specifying a range of desired nodes in the VG. Our next step (in Chapters V) is to determine the optimal point (or range) in the size of the resource collection, as well as the various characteristics of the optimal resource collection, so that we can compose good resource collection specifications for resource selection frameworks in existence today [14-17, 20] as well as future ones. We will see that in fact, in spite of the seemingly good results obtained in this chapter with a simple resource collection specification, much better performance can be achieved by constructing more intelligent specifications.

IV.5 Acknowledgement

Chapter IV, in part, has been published as “Using Virtual Grids to Simplify Application Scheduling” by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006). The dissertation author was the primary investigator and author of this paper.

V

DERIVING BEST RESOURCE COLLECTION SPECIFICATION

A resource collection (RC) is a set of computing hosts on which the users can execute their applications. The choice of a resource collection affects application performance as well as the choice of a scheduling heuristic. When the RC contains faster or more hosts, applications are more likely to run faster than if the RC contains slower or fewer hosts. When the RC is homogeneous with respect to clock rates and with respect to the network connectivity among the hosts, a simpler scheduling heuristic is likely to achieve good application performance when compared to using a more sophisticated scheduling heuristic. Therefore, using appropriate resource collections to run applications has two main advantages:

1. Achieve good application performance.
2. Allow simpler scheduling heuristics to achieve good application performance.

While the benefits of using appropriate resource collections are clear from Chapter IV, what is not clear is how to compose such collections in resource-rich large-scale distributed environments. Most application developers focus on optimizing their application or providing better user interfaces. Most developers of resource selection services (for middleware or resource management software) focus on faster resource selection heuristics and optimizing some metric of goodness for resource collections. The missing link between applications and resource selection services is that the application users or the application itself need to specify or provide guidance to the resource selection services to define the type of resource collection the resource selection service should return to the application. We are not aware of any work that

quantitatively analyzes the properties of a good resource collection nor are we aware of any work that provides guidance for applications to request resource collections from resource selection services.

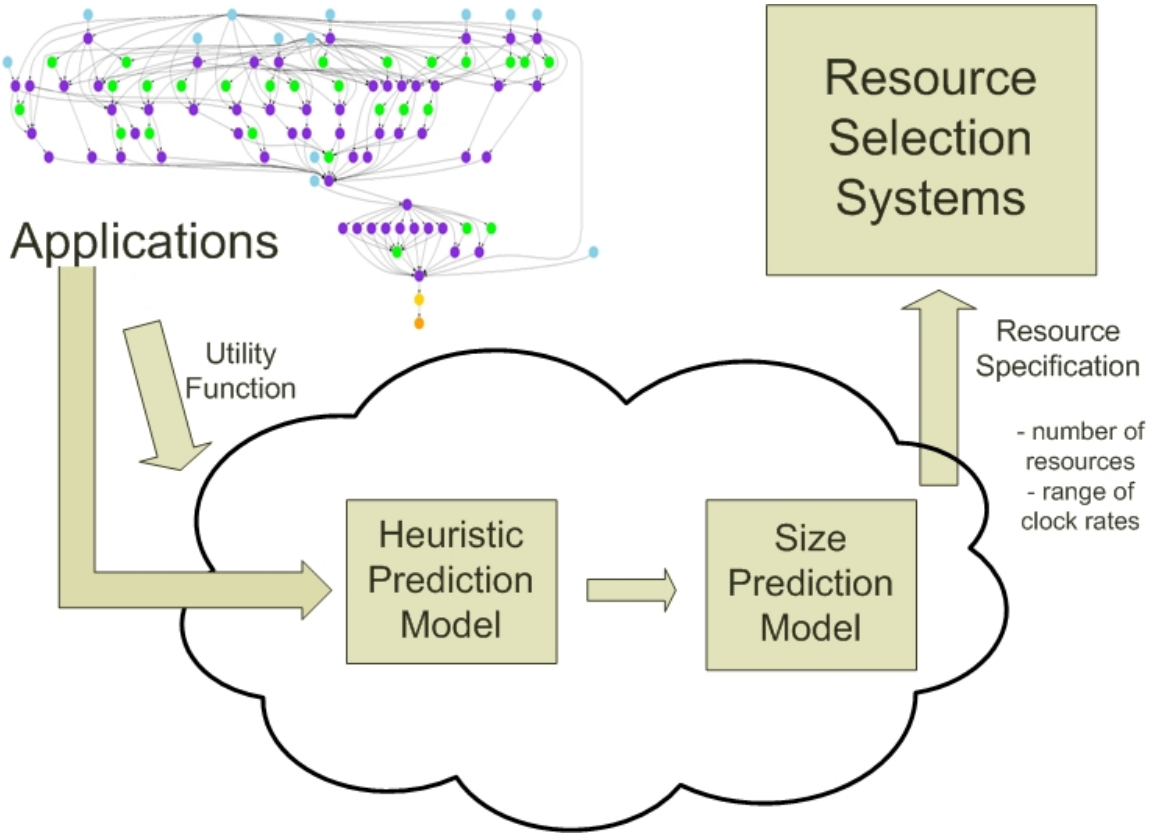

Figure V-1: Resource Specification Predictor

Our vision, depicted in Figure V-1, is that of a resource specification prediction model that takes as input the target DAG and an optional utility function that the user can specify to trade off high performance for low cost. The output is a resource specification that the user can use as the input to different resource selection systems to acquire the best set of resources for their particular application. The prediction model is composed of two parts: the *heuristic prediction model* determines which scheduling heuristic should be employed to optimize application turn-around time; and the *size prediction model* which predicts the best RC size based on application (DAG) characteristics, the optional utility function, and the predicted scheduling

heuristic. In this chapter, we focus on the size prediction model. We address the heuristic prediction model in Chapter VI.

In Section V.1, we define what constitutes the best resource collection specification with regards to good application performance. In Section V.2, we derive an empirical model to predict the best RC size for RCs with homogeneous clock rates and homogeneous network connectivity among the hosts. We validate the accuracy of our predictive model using randomly generated DAGs and DAGs from real applications in Section V.3. Then, we examine the impact of clock rate heterogeneity within the RC in Section V.4 and the impact of network heterogeneity among the hosts in Section V.5. We use a reference scheduling heuristic, the Modified Critical Path (MCP) [31] in the first five sections of this chapter and address the effects of using different scheduling heuristics in Section 0. We discuss how to choose the most appropriate scheduling heuristics in Chapter VI. In Section V.7, we investigate the effects of varying the reference scheduling and computational clock rate ratios. In Section V.8, we summarize the results of this chapter.

V.1 Best Resource Collection Specifications

RCs can vary by size, clock rate heterogeneity within the RC, and networking heterogeneity among hosts in the RC. A larger sized RC implies more host choices for a scheduling heuristics to assign tasks in a DAG, and more possibilities for tasks to be executed in parallel. The tradeoff is that most sophisticated scheduling heuristics take longer to run with larger sized RCs, along with possible cost/penalty for unused hosts. Greater heterogeneity in host clock rates within the RC implies faster (and slower) hosts within the RC. More sophisticated scheduling heuristics running on RCs with greater clock rate heterogeneity could improve application performance by scheduling tasks to execute on faster hosts. Similar logic applies to greater network heterogeneity among hosts in the RC as more sophisticated scheduling heuristics

can better reduce communication costs by scheduling tasks on hosts with higher bandwidth connections. The question is: For a given DAG, what is the best resource collection specification?

We define the best resource collection specification in terms of application performance. Recall that the metric we use for application performance is application turn-around time, which is the sum of scheduling time and application makespan. The scheduling time is the execution time for the scheduling heuristic. For this work, we run the scheduling heuristics on a 2.80 GHz Intel Xeon CPU. The application makespan refers to the time between the start of the first task to execute and the completion time of the last task to complete.

We define the best *resource collection specification (RCS)* as the description for a RC that minimizes the application turn-around time for a given DAG and a given scheduling heuristic. To derive the best RCS, we specify RCs by three characteristics:

1. size
2. clock rate heterogeneity within RC
3. network connectivity heterogeneity among hosts in RC

Varying any or all of these characteristics can impact the scheduling time and the makespan of an application. Most of the currently developed resource selection services allow users to specify each of these three characteristics. By varying these characteristics, we introduce tradeoffs in the application performance. For example:

1. Increasing RC size
    - Potentially increases scheduling time for scheduling heuristics whose running time depends on the number of hosts in the RC.
    - No effect on scheduling time for simpler scheduling heuristics whose running time is independent of the number of hosts in the RC.

- Potentially decreases makespan because more hosts can allow more potential parallelism among tasks. More hosts also allow more sophisticated scheduling heuristics more (and potentially better) choices to assign each task in the DAG.

2. Increasing clock rate heterogeneity within RC
    - Potentially increases scheduling time and scheduling complexity. For sophisticated scheduling heuristics that consider the earliest finishing time for a task on each host in the RC, increasing the clock rate heterogeneity would also increase the scheduling time. With no clock rate heterogeneity within the RC, a simpler heuristic choosing the earliest available host would be sufficient.
    - Decreases makespan for sophisticated scheduling heuristics that use clock rate information as well as resource availability information (such as MCP). Faster (and available) hosts can be used to execute tasks, which decreases the makespan.
    - Potentially increases makespan for simpler scheduling heuristics (such as First-Come-First-Serve (FCFS) or random) that do not use clock rate information. The scheduling heuristic might choose slower hosts that increase the makespan.
3. Increasing network connectivity heterogeneity among hosts in RC
    - Potentially increases scheduling time for more sophisticated scheduling heuristics that evaluate costs of transmitting intermediate files. Instead of considering only computational costs of the tasks in a homogeneous network, scheduling heuristics need to evaluate tradeoffs between executing the child task on the same host as the parent task or transmitting an intermediate file to another host and executing the child task there.
    - Potentially decreases makespan for more sophisticated scheduling heuristics that evaluate costs of transmitting intermediate files.

- Potentially increases makespan for simpler scheduling heuristics that do not evaluate costs of transmitting intermediate files.

The optimal RC size for each DAG might differ according to the scheduling heuristic, different clock rate heterogeneity within the RC, and different network heterogeneity among hosts in the RC. Because of the tradeoffs listed above, it is difficult to compose an optimal RC for any given DAG. Our goal this chapter is to derive an empirical model that predicts the best RC size. Because of the difficulties in the tradeoffs for the three RC characteristics due to the choice of scheduling heuristic, we use a reference scheduling heuristic in the first five sections of this chapter – Modified Critical Path [31] and examine the effects of using different scheduling heuristics in Section 0.

V.2 Deriving Best RC Size

Our first goal is to predict the number of resources that should be requested in the best resource specification, i.e., the RC size. To predict RC size in this section we assume homogeneous resources, with homogeneous bandwidth, and we use the reference scheduling heuristic, MCP. In Section V.4 and Section 0, we evaluate the sensitivity of our predictive model to resource heterogeneity and to different scheduling heuristics, respectively.

Our strategy for formulating an empirical prediction model of RC size is as follows:

1. Determine relevant DAG characteristics that are likely to impact the choice of resources for running an application (Section V.2.1).
2. Define what the best RC size should be (Section V.2.2).
3. Execute scheduling heuristic on an observation set of DAG configurations while varying the relevant DAG characteristics (Section V.2.3).
4. Derive a model from the observation set results that predicts the best RC size (Section V.2.4).

After we construct the empirical model that predicts the best RC size, we validate that our model leads to accurate predictions with workloads of randomly generated DAGs and DAGs from real applications in Section V.3.

V.2.1 Relevant DAG characteristics

We listed different DAG characteristics in Section III.1.1. Our goal here is to determine a subset of thse DAG characteristics that are relevant for constructing a model to predict the best RC size. We can separate DAG characteristics into those that have obvious effects on the choice of RC size, those that can be subsumed by other DAG characteristics, and those that most likely do not affect the choice of RC size.

The DAG characteristics that have obvious effects on the choice of RC size are:

1. DAG size – It is very clear that size is a factor. A bigger DAG would often benefit from a bigger RC and a smaller DAG would not need as big a RC. For a bigger DAG, a bigger RC would allow more DAG tasks to be parallelized, leading to a shorter makespan and shorter application turn-around time.
2. Communication-to-computation ratio – We know that with higher communication costs, a good scheduling heuristic would schedule tasks to run on the same host rather than incur higher communication costs that would lengthen the application makespan. Thus, a DAG with higher CCR would not benefit from a larger RC (because scheduling tasks on more hosts would lengthen the makespan with the additional communication costs to each additional host added to the RC). The combination of high CCR, naïve scheduling heuristics, and a bigger RC would lead to longer makespan (and application turn-around time) because using more compute hosts in the bigger RC also means higher communication costs. Correspondingly, a DAG with a smaller CCR would benefit from a larger RC as more

compute hosts can be utilized without incurring too much excess communication costs. Having a sufficiently large RC would lead to makespan minimization (depending on the scheduling heuristic) which possibly leads to faster application turn-around time.

3. Parallelism – Recall from Section III.1.1 the definition of parallelism: $\alpha = \frac{\log(\tau)}{\log(n)}$, where τ is the average number of tasks per level and *n* is the DAG size. Clearly, a DAG with higher parallelism would benefit from a bigger RC to maximize its parallelism during execution, leading to shorter application makespans. Correspondingly, a DAG with lower parallelism can maximize parallelism with a smaller RC.
4. Regularity – Recall from Section III.1.1 the definition of regularity: $\beta = 1 - \frac{\max_{i=1,..,k} | size(l_i) - \tau |}{\tau}$. Regularity reflects how the number of tasks per level varies from level to level. Along with parallelism, regularity determines the DAG width: parallelism determines the mean number of tasks per level while regularity determines dispersal from that mean. A DAG with higher irregularity (thus lower regularity) is more likely to have a larger DAG width, which is a larger maximum number of tasks in any level. A larger DAG width would necessarily require a bigger RC to maximize parallelism to minimize application makespan. Conversely, a DAG with a high degree of regularity can maximize parallelism (and minimize application makespan) with a smaller RC.

The DAG characteristics already subsumed by other DAG characteristics are:

1. DAG height or number of levels – DAG height is a function of DAG size and the average number of tasks per level. Thus, DAG height is subsumed by the DAG size and the average number of tasks per level.

2. Average number of tasks per level – This information is subsumed in the parallelism. The more highly parallel the DAG, the larger the average number of tasks per level.

The DAG characteristics that most likely do not affect the choice of RC size are:

1. Density – It is unclear whether tasks having more dependencies would impact the choice of RC size. A DAG with low density might have tasks with single dependencies. For example, say task n_i depends only on n_j. A sophisticated scheduling heuristic could merge tasks n_j and n_j into one task n_{ij}. Such reduction of the DAG is in essence reducing the DAG size while increasing the mean computational cost. It is unclear how one can generalize this DAG reduction to apply to all DAGs.
2. Mean computational cost – It is unclear how the mean computational cost would affect the best RC size. With all else being equal (and in particular the CCR), two DAGs that differ only in the mean computational cost would likely require the same RC size.

From our discussions above, we decide to construct our model based on the following relevant characteristics: DAG size, CCR, parallelism, and regularity. DAG height and the average number of tasks in the DAG are subsumed by the four relevant DAG characteristics, so we can leave them out of the model. Because it is unclear whether density or the mean computational cost can affect RC choices, we are also leaving them out of our model construction. From our results, we will see that the model we construct is accurate even without taking into account these two characteristics.

V.2.2 Best RC Size

Earlier, we have defined the best RCS as the description of an RC that minimizes the application turn-around time. Of the three characteristics we can specify for an RCS, we focus on

the first characteristic in this section – RC size. We fix the other characteristics by considering homogeneous clock rates within the RC and homogeneous network connectivity among the hosts in the RC. Thus, we want to construct a model that predicts the best RC size, which is the RC size that leads to the minimal application turn-around time.

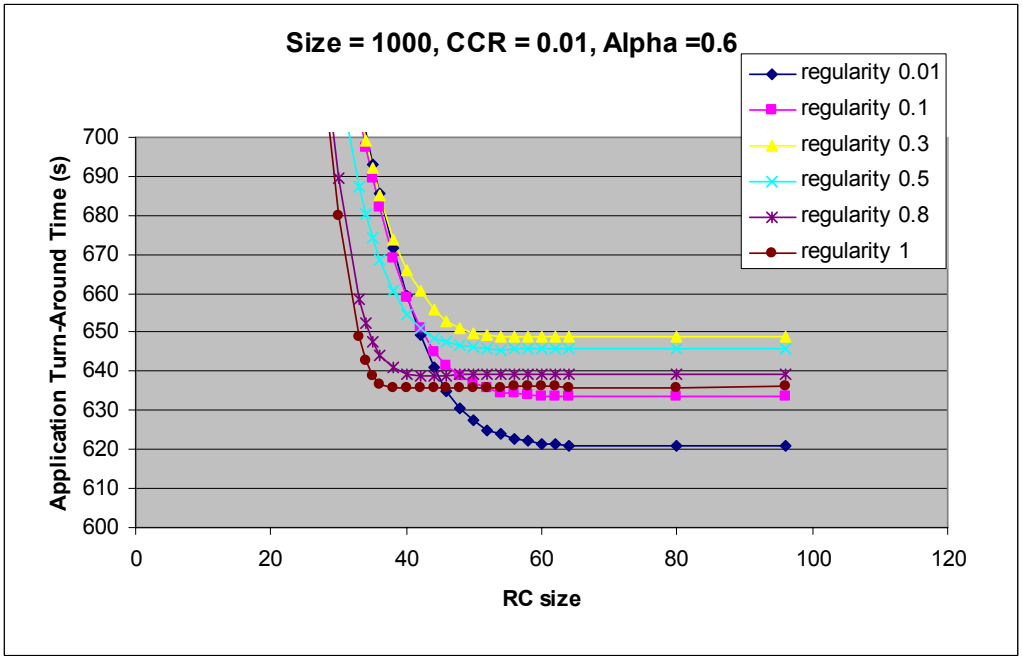

Figure V-2: Application turn-around time as function of RC size for DAG with size 1000, CCR of 0.01, and parallelism of 0.6 for various regularity values

To illustrate the notion of the best RC size, we show Figure V-2 as an example. Figure V-2 shows the application turn-around time as a function of the RC size for DAGs of size 1000, CCR of 0.01 and parallelism (α) of 0.6. Each of the points in the figure represent the average application turn-around time for ten distinct instances of DAGs with the same DAG characteristics. Using the same ten DAGs, we schedule the tasks on resource collections of increasing size. Each point on the plot represents the turn-around time as the result of using a particular RC size. Each of the lines represents DAGs with different regularity values. As we can see from the figure, increasing the RC size improves application turn-around time up to a certain

point. Beyond a certain threshold, the best application turn-around time can be achieved by a range of RC sizes.

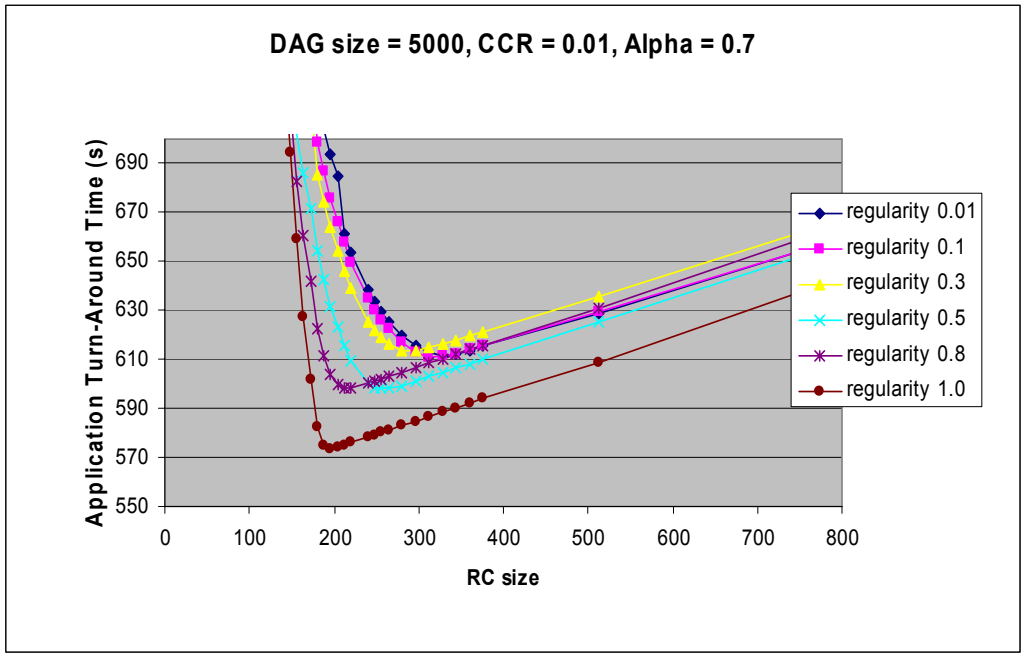

Figure V-3: Application turn-around time as function of RC size for DAG with size 5000, CCR of 0.01, and parallelism of 0.7 for various regularity values

In Figure V-3, we show the results of a very similar experiment as the results from Figure V-2. This time, the results are from the averages of running the scheduling heuristics over ten different distinct DAGs with size of 5000, CCR of 0.01, and parallelism (α) of 0.7. We want to show that for some other DAGs such as the ones used for the results shown in Figure V-3, the range of RC sizes where the best application turn-around time can be achieved is quite small (or it could be a single RC size). As we can see from Figure V-3, beyond a certain RC size, the application turn-around time actually increases with more hosts in the RC. The increase in application turn-around time is entirely due to increased scheduling time. Beyond a certain RC size, the application makespan remains the same yet the scheduling time slowly increases because the running time of MCP is a polynomial function in the number of hosts in the RC. This result in the increased application turn-around time with increasing RC size beyond the best RC size. For

other scheduling heuristics where the running time is a higher order polynomial function in the number of hosts in the RC as compared to MCP, the increase in the scheduling time could be much steeper than MCP and consequently the range of RC sizes where the best application turn-around time can be achieved would be very small, and likely occurring at exactly one size.

As the two figures illustrate, two scenarios exist for best RC sizes:

1. A range of RC sizes where the application turn-around time is minimized (as shown in Figure V-2). This is because for a small range of RC sizes, the variance among the execution time for the scheduling heuristic is negligible (within milliseconds).
2. A single RC size where the application turn-around time is minimized (as shown in Figure V-3). This is because the best application makespan achievable by the scheduling heuristic is achieved at a certain RC size, and an increase in RC size only adds to the scheduling time.

To have a unifying definition for both of these scenarios, we define the best RC size as the smallest RC size such that a bigger RC size would improve turnaround time by less than a threshold of 0.1%. We call this value the "knee" value. We define the application turn-around time as having achieved the best performance when increasing the RC size does not improve performance by more than 0.1%. It is necessary to pick a threshold value slightly greater than 0% because of the experimental nature of the process used to determine the application turn-around times. By choosing 0.1% as the threshold, we are ensuring that the knee is not the result of experimental fluctuations in running the scheduling heuristics. Had we used 0% as the threshold for the best application turn-around time, the knee may have been artificially inflated to a higher value when the scheduling heuristic ran faster (by milliseconds) for a slightly bigger RC size.

V.2.3 Observation Set

To construct a predictive model for best RC size given arbitrary DAGs, we need to examine how the best RC size varies as a function of the various relevant DAG characteristics. We can then formulate a function that includes each relevant DAG characteristic to predict the best RC size. Our strategy is to use arbitrary DAGs as an observation set and derive a function that follows closely with what we observe from the DAGS in the observation set.

For our observation set we choose our sample values at as evenly-spaced intervals as possible for each of the relevant DAG characteristics. We choose DAG sizes ranging from 100 tasks to 10,000 tasks as this represents the range of the interesting DAGs scientists run today. For CCR, we want to cover an interesting range where the DAG is not dominated by either communication or computation; we choose a range between 0.01 and 1.0. With extreme communication intensive applications (CCR much greater than 1), a RC with a single host can eliminate all the communication costs. When the computational costs is more than one hundred times the communication cost (as is the case when CCR is below 0.01), the communication costs becomes negligible and we can use the value of 0.01 for CCR to predict the RC for any smaller CCR.

When parallelism is low, the average number of tasks per level becomes small and a small RC would be sufficient to achieve good performance. Similarly, when the Parallelism is 1, all the tasks can be parallelized and the best RC size would be equal to the DAG size. Thus, we choose the range of parallelism between 0.3 and 0.9. For regularity, we choose a range of values ranging from 1.0 to 0.01. A regularity value of 1.0 means that the DAG is perfectly regular (all levels have the same number of tasks). A regularity value of 0.01 implies that the maximum dispersal from the average number of tasks per level is 99%. Thus we are examining DAGs with

number of tasks at any given level ranging from 1% to 199% of the average number of tasks per level. The DAG characteristic values are summarized in Table V-1.

To determine best RC sizes for sample DAGs, we schedule DAGs onto varying RC sizes. We generate arbitrary DAG configurations, corresponding to the cross-product of the relevant DAG characteristic listed in Table V-1 for a total of 1260 configurations. For each configuration, we instantiate ten distinct DAGs; thus we have a total of 12,600 DAGs. Our results reflect the average of the ten sample DAGs for each of the 1260 DAG configurations. Then we use our reference scheduling heuristic, MCP, to schedule randomly generated DAGs onto the RCs, calculating the application turn-around time in each case.

Table V-1: Relevant DAG characteristic and sample values

DAG Characteristic	Values
DAG size (# of tasks)	100,500,1000,5000,10000
CCR	0.01, 0.1, 0.3, 0.5, 0.8, 1.0
Parallelism (α)	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
Regularity (β)	0.01, 0.1, 0.3, 0.5, 0.8, 1.0

V.2.4 Model Formulation

Table V-2: Knee values for DAGs with size 5000 and CCR of 0.01

$\alpha\backslash\beta$	**0.01**	**0.1**	**0.3**	**0.5**	**0.8**	**1.0**
0.3	34	32	22	18	14	14
0.4	52	36	28	24	22	20
0.5	80	62	58	50	56	42
0.6	136	140	128	112	94	128
0.7	328	312	280	248	212	196
0.8	464	456	448	448	448	432
0.9	496	496	440	440	432	392

After we determine the knee for the observation set of DAGs, we need to formulate a model based on four variables (representing each of the four relevant DAG characteristics). Because of the difficulty of dealing with four variables at once, we can further simplify the formulation by first considering DAGs of a fixed size and fixed CCR; thus first we consider

parallelism (α) and regularity (β). For example, Table V-2 below shows the RC knee values for DAGs with size 5000 and CCR of 0.01.

From our results, we notice the trend of an exponential increase in knee values as a function of α. Our hypothesis is that for a given DAG size and CCR, the knee can be predicted by the following formula:

$$\text{Knee} = 2^{(a\alpha + b\beta + c)}$$

In the formula, α refers to the parallelism, β refers to the regularity, with unknowns *a*, *b*, and *c* to be determined. Figure V-4 plots the logarithm of the knee values of the various parallelism and regularity values for DAGs of size 5000 and CCR of 0.01. For all combination of DAG sizes and CCR values, we observe very similar planar shapes. It turns out that a planar fit to the data can be done in all cases with mean relative error of at most 16% for a DAG size of 5000.

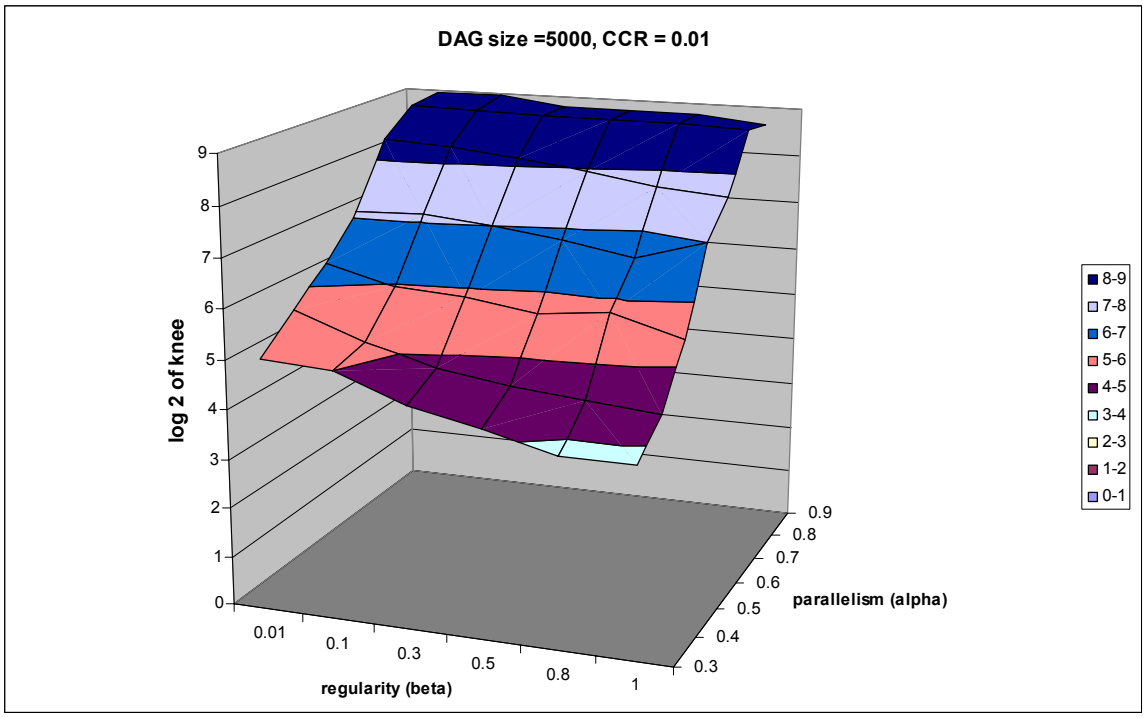

Figure V-4: Log2 of knee values when DAG size = 5000 and CCR = 0.01

Because of the planar shapes of the surface plots, one can use linear regression by fitting a plane through the different logarithm of knee values. By applying the logarithm function to both sides of the equation, we have 42 equations of the form:

$$\log_2(knee) = a\alpha + b\beta + c$$

For all equations i = {1, 2, …, 42},

Let z_i = experimental values for $\log_2(knee_i)$.

Let $x_i = \alpha_i$.

Let $y_i = \beta_i$.

We can minimize the mean squared error of the 42 equations by taking the partial derivatives to the equation:

$$\sum_{i=1}^{42} (z_i - (ax_i + by_i + c))^2$$

This results in the following three equations:

$$\sum_{i=1}^{42} ax_i^2 + \sum_{i=1}^{42} bx_i y_i + \sum_{i=1}^{42} cx_i = \sum_{i=1}^{42} z_i x_i$$

$$\sum_{i=1}^{42} ax_i y_i + \sum_{i=1}^{42} by_i^2 + \sum_{i=1}^{42} cy_i = \sum_{i=1}^{42} z_i y_i$$

$$\sum_{i=1}^{42} ax_i + \sum_{i=1}^{42} by_i + c * 42 = \sum_{i=1}^{42} z_i$$

We obtain a 3x3 linear system which can be easily solved for the 3 unknowns. We can solve for the three unknowns *a*, *b*, and *c* by rearranging the equations in the following matrix form:

$$\begin{pmatrix} \sum_{i=1}^{42} x_i^2 & \sum_{i=1}^{42} x_i y_i & \sum_{i=1}^{42} x_i \\ \sum_{i=1}^{42} x_i y_i & \sum_{i=1}^{42} y_i^2 & \sum_{i=1}^{42} y_i \\ \sum_{i=1}^{42} x_i & \sum_{i=1}^{42} y_i & 42 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{42} z_i x_i \\ \sum_{i=1}^{42} z_i y_i \\ \sum_{i=1}^{42} z_i \end{pmatrix}$$

By solving for *a*, *b*, and *c*, we have constructed a model that can predict the best RC size given a fixed DAG size and a fixed CCR value.

The last remaining step is to reconcile the two remaining application characteristics: DAG size and CCR. We show a representative plot of varying knee values as a function of DAG sizes in Figure V-5 for DAGs with fixed CCR of 0.01 and fixed parallelism at 0.7 for different regularity values. Other CCR and parallelism values show similar trends. We also show a representative plot of varying knee values as a function of CCR in Figure V-6 with fixed DAG size of 5000 and fixed regularity at 0.01. Again, other DAG sizes and other regularity values show similar trends. Unfortunately, although these curves at first glance look logarithmic and exponential, it is difficult to find a simple model with a good fit. Therefore we resort to an empirical approximation based on interpolation between experimental data points on these curves. We hypothesize that linear interpolations based on the two closest sample points provide sufficiently good results. We interpolate in both axis when both DAG size and CCR value fall between two sample values.

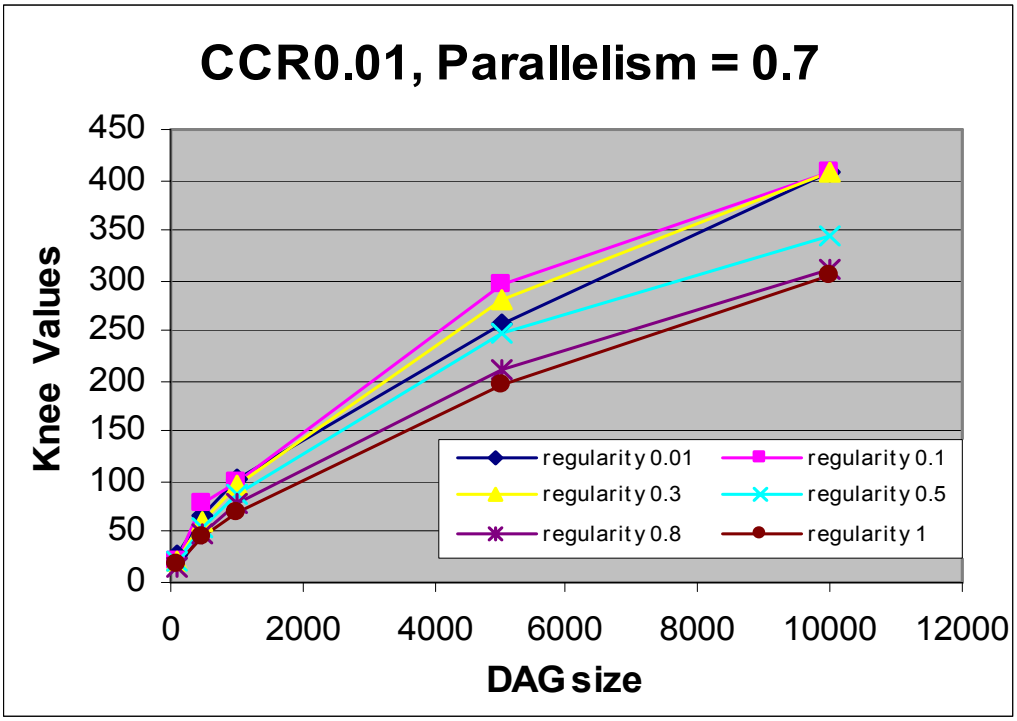

Figure V-5: Knee values as function of DAG size with fixed CCR at 0.01 and fixed parallelism at 0.7 for various regularity values

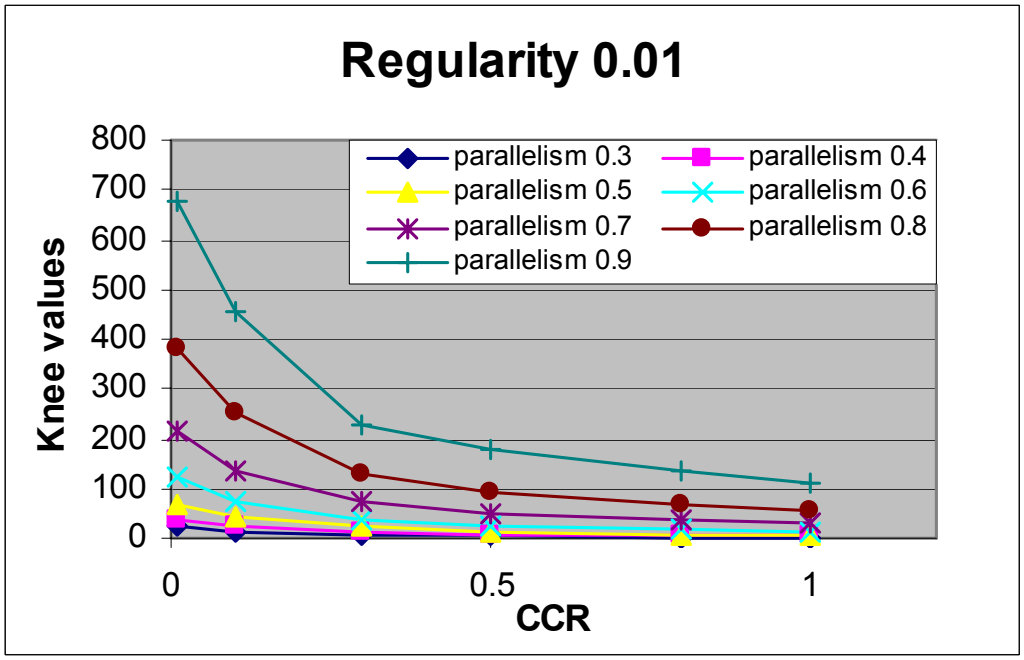

Figure V-6: Knee values as function of CCR with for DAGs with size 5000 and fixed regularity at 0.01 for various parallelism values

V.3 Predictive Model Validation

After constructing a predictive model to estimate the best RC size, we need to validate the accuracy of the model. We use two workloads to validate our predictive model for best RC sizes:

- Randomly generated DAGs
- DAGs from real applications

For each workload, we validate our prediction model by comparing the application turn-around time achieved using the predicted RC size and the actual optimal RC size (approximated by a time-consuming algorithm described in Section V.3.1). Further, we compare the application performance achieved when using our model to that achieved with the current practice of using maximum parallelism as the RC size.

V.3.1 Heuristic to Derive Actual Optimal RC Size

One way to derive the actual optimal RC size is by brute force. However, since we are interested in testing our prediction model over a wide variety of DAG characteristics (over 10,000 DAGs with varying characteristics for each DAG size), using a brute force method to derive each actual optimal RC size would take many CPU years to complete. Instead, we use a (still time-consuming) heuristic to derive the actual optimal RC size.

Our heuristic uses the predicted RC size as the starting point. From the predicted RC size we create RC sizes that vary from the predicted RC size by 10%-50% in 10% intervals in both directions (bigger and smaller RC sizes). Then we try RC sizes that are 2 times, 2.5 times, and 3 times the predicted RC sizes. Lastly, we try geometrically decreasing RC sizes by a factor of 2, starting with the predicted RC size and ending when the new RC size reaches 1. The RC sizes used by our heuristic are listed in Table V-3, along with 2 examples of predicted RC sizes, one at

100 and one at 300. In Table V-3, x is the predicted best RC size. Note that this heuristic requires knowledge of the best RC size a priori and still requires many CPU hours to compute the optimal RC size for larger DAGs; thus, one would not be able to employ this heuristic to derive the optimal RC size unless there is a good starting guess and even in such a scenario, many CPU hours would be needed to derive the optimal RC size.

Table V-3: Heuristic for deriving actual optimal RC size

Test Values for RC size	**Example 1**	**Example 2**
x	100	300
x ± 0.1x	110,90	330,270
x ± 0.2x	120,80	360,240
x ± 0.3x	130,70	390,210
x ± 0.4x	140,60	420,180
x ± 0.5x	150,50	450,150
2x	200	600
2.5x	250	750
3x	300	900
Repeated ½ function	50,25,13,7,4,2,1	150,75,38,19,10,5,3,2,1
RC sizes attempted	1,2,4,7,13,25,50,60,70,80,90,100, 110,120,130,140,150,200,250,300	1,2,3,5,10,19,38,75,150,180,210, 240,270,300,330,360,390,420,450, 600,750,900

V.3.2 Validation with Randomly Generated DAGs

First we validate our model with a workload of randomly generated DAGs. To have a thorough testing of our predictive model for RC sizes, we test all of the 1260 DAG configurations composing our observation set as well as the 840 DAG configurations containing the midpoint of the two DAG characteristics for which our model computes the best RC. For each DAG configuration, we generate ten distinct random DAGs. We summarize the values we choose for our validation suite in Table V-4. We expect the application turn-around time using predicted RC sizes from the observation set DAG configurations to be the closest match to the application turn-around time achievable using the actual optimal RC size. When we use the midpoint between two

observation set DAG characteristic values, we expect bigger loss of application performance because of the use of linear interpolation between two sample points as an approximation.

Table V-4: DAG characteristic values for validation suite

DAG characteristics	Observation set DAG characteristic values	Midpoint between two observation set DAG characteristic values
DAG Size	100, 500, 1000, 5000, 10,000	300, 750, 3000, 7500
CCR	0.01, 0.1, 0.3, 0.5, 0.8, 1.0	0.05, 0.2, 0.4, 0.65, 0.9
Parallelism	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
regularity	0.01, 0.1, 0.3, 0.5, 0.8, 1.0	0.01, 0.1, 0.3, 0.5, 0.8, 1.0
Total Number of Configurations	1260	840

V.3.2.1 Performance and Cost Metrics

We use three metrics to measure the accuracy of our predictive model. The first metric is the average predicted size difference. This metric tells us the normalized distance between the model-predicted RC size and the heuristic-derived optimal RC size. If our model can accurately predict the optimal RC size, then the average predicted size difference would be small.

The second, and perhaps the most important metric, is the average performance degradation. This metric measures the degradation of application turn-around when using the RC configuration predicted by our model compared against using the (approximated) optimal RC configuration. Ultimately, users of this predictive model would be most interested in achieving the best application turn-around time and this metric allows us to quantify the performance degradation.

The third metric is one of cost. While the application turn-around time is a common metric, defining a metric for cost is more difficult. Rather than coming up with an arbitrary metric, we chose to use the same one as an existing production system that charges users consistently: Amazon's Elastic Cloud [79]. In this system, each "instance", that is a (virtual) 1.7GHz x86 processor machine, is $0.10 per hour. We simply scale this cost by our simulated resources clock rates and compute total cost for application executions. We use a "relative cost"

metric. The relative cost is the cost when using the predicted size versus the cost of using the size for optimal application turn-around time. A positive value for relative cost indicates the prediction model predicted a size greater than the size for the optimal application turn-around time and thus costing more than using the size for optimal application turn-around time. A negative value corresponds to a smaller size and thus a cheaper execution.

V.3.2.2 Validation Results for Randomly Generated DAGs

Our results (summarized in Table V-5) show that our predictive model performs quite well over the range of DAG characteristics we tested. The top part of the table shows results for DAGs with DAG size values identical to DAG sizes in the observation set. The bottom part of the table shows results for DAGs with DAG sizes corresponding to midpoint between observation set DAG sizes. The left part of the table shows results for DAGs with CCR values corresponding to observation set CCR values, whereas the right side of the table shows results for DAGs with CCR values corresponding to exactly midpoint between observation set CCR values. Before our experiments, we expect that the results DAGs with characteristics corresponding to our observation set should have the highest performance, and results from for DAGs with characteristics corresponding to midpoint between observation set values would have worse performance. Thus, we expect the top left quadrant in Table V-5 to have the best performance and the lower right quadrant in Table V-5 to have the worst performance.

For the average predicted size difference, our results showed that in all DAG configurations tested, our predictive model predicted a RC size that is on average between 9%-15% from the optimal RC size.

For performance degradation, our results showed that for the observation set DAG sizes and observation set CCR values, the performance degrades as the DAG size increases, ranging from 0.18% (for the smallest DAGs) to 1.82% (for the biggest DAGs) performance degradation.

When the model is applied to DAG sizes that are exactly midpoints between two observation set DAG sizes, the performance is slightly worse, but even in the worst case is only 1.93%.

For the results from interpolating CCR values, we had expected performance degradation similar to the degradation experienced by the model when interpolating between DAG sizes; however, we observed the opposite. With all other DAG characteristics being equal, we observed better performance with the interpolated CCR values rather than the observation set CCR values. Across different DAG sizes (for both observation set DAG sizes and interpolated DAG sizes), the average application performance degradation resulting from using interpolated CCR values were approximately half of the performance degradation from using the observation set CCR values!

A closer look at individual DAGs revealed that the predictive model underestimates the best RC size for DAGs requiring more hosts to achieve maximum parallelism (and better performance). These DAGs have lower CCR values and higher parallelism (α) values. The higher parallelism values would lead to a DAG with bigger DAG width and consequently more hosts would be preferable to achieving maximum parallelism. The lower CCR values for these DAGs implies that using more hosts does not hurt the application makespan as much because the communication costs are minimal. From Figure V-6, we see that linear interpolation between two observation set CCR values actually predicts a bigger RC size than if we had fit a smooth curve between all the observation set points. The end result is that the error we had expected from linear interpolation overcompensates for the RC size underestimation of our model for DAGs requiring bigger RC sizes, leading to better application turn-around time (lower degradation) compared to the observation set CCR values. Note that these results could suggest an ad hoc improvement to our model. Indeed, one could artificially increase the predicted RC size for DAGs with low CCR and high parallelism values.

In terms of cost, we observed that for all of the DAG configurations we tested our predictive model yields a negative relative cost. The negative numbers implies that our predictive

model is on average underestimating the optimal RC size and therefore providing "savings" over using the optimal RC configuration.

During the process of constructing the predictive model, we observed that for DAGs with higher CCR and lower parallelism values (regardless of the range of regularity values we tested), the best application turn-around time can be achieved by using 1 host only. Adding additional hosts to the resource collection simply increased the application turn-around time. This is expected as the cost of file transfer outweighs the benefit of adding additional hosts to the resource collection. For that reason, we have excluded DAGs with high CCR and low parallelism values from the construction of our model and have thus also excluded these DAGs from our validation results. For the vast majority of the excluded DAG configurations, using a resource collection of size 1 would have produced the best result. We listed a column in Table V-5 to reflect the number of DAG configurations within the range of our predictive model.

Table V-5: Validation Results when using Predictive Model

DAG Size	**Observation Set CCR**				**Midpoint CCR**			
Obsv. Set DAG Sizes	**Average Predict Size Diff.**	**Average Perf. Degrad.**	**Relative Cost**	**DAG Config. Within Range of Model**	**Average Predict Size Diff.**	**Average Perf. Degrad.**	**Relative Cost**	**DAG Config. Within Range of Model**
100	9.59%	0.18%	-6.75%	144/252	11.04%	0.16%	-3.66%	120/210
500	11.49%	0.22%	-5.29%	198/252	11.11%	0.13%	-0.81%	156/210
1000	9.62%	0.32%	-4.32%	204/252	10.10%	0.13%	2.34%	162/210
5000	13.27%	0.77%	-4.72%	222/252	10.65%	0.34%	-0.40%	180/210
10,000	14.53%	1.82%	-2.94%	230/252	12.56%	0.86%	-6.97%	188/210
Mid-point DAG Sizes								
300	13.41%	0.34%	-11.31%	144/252	10.89%	0.19%	-6.76%	120/210
750	11.85%	0.29%	-5.59%	198/252	9.42%	0.14%	-0.42%	156/210
3000	14.97%	1.08%	-9.98%	204/252	11.95%	0.50%	-4.48%	162/210
7500	16.71%	1.93%	-5.28%	144/252	12.69%	0.98%	-1.12%	120/210

From Table V-5, we see that the experiments with DAG configurations containing observation set DAG sizes preserved application performance better than midpoint DAG sizes. The natural questions to ask are:

- Does the midpoint DAG size value represent the worst case scenario for using our size prediction model?
- Does varying DAG sizes between two observation set DAG sizes result in a smooth curve between the two expected lower observation set application performance degradations?

Table V-6 shows the effects on application performance for various DAG sizes at 1000 task intervals from size 1000 to size 5000, two sample points in our model. We see that the midpoint DAG size does represent the worst performance and the performance degradations from the other two DAG sizes falls in range between the best performance degradation of the observation set DAG sizes of 1000 and 5000 and the midpoint DAG size of 3000. One interesting note is that the performance degradation for DAG size 2000 is much closer to the worst observed case of DAG size 3000 rather than being midway between the performance degradation of DAG size 1000 and DAG size 3000. Thus, we might expect that performance degradations for any DAG size would be at least as great as the larger degradation of the two observation set values (on which the interpolation is based).

Table V-6: Experiment showing effects of varying DAG size

	Observation Set CCR				**Midpoint CCR**			
Varying sizes	**Average Predict Size Diff.**	**Average Perf. Degr.**	**Relative Cost**	**DAG Config. Within Range of Model**	**Average Predict Size Diff.**	**Average Perf. Degr.**	**Relative Cost**	**DAG Config. Within Range of Model**
1000	9.62%	0.32%	-4.32%	204/252	10.10%	0.13%	2.34%	162/210
2000	15.19%	0.99%	-10.47%	204/252	11.80%	0.47%	-5.60%	162/210
3000	14.97%	1.08%	-9.98%	204/252	11.95%	0.50%	-4.48%	162/210
4000	13.44%	0.87%	-7.85%	204/252	10.90%	0.41%	-2.06%	162/210
5000	13.27%	0.77%	-4.72%	222/252	10.65%	0.34%	-0.40%	180/210

V.3.2.3 Performance-Cost Tradeoff

So far we have only presented results about optimizing turnaround time. However, different users may have different notions of utility and different economic constraints. We enhance our model by allowing the user to specify simple notions of utility for trading off performance for lowering cost. We accomplish this by generating predicted sizes based on various thresholds for defining the knee values. Previously, we only used a threshold of 0.1%. Now our model also uses thresholds of 0.5%, 1.0%, 2.0%, 5.0%, and 10.0%.

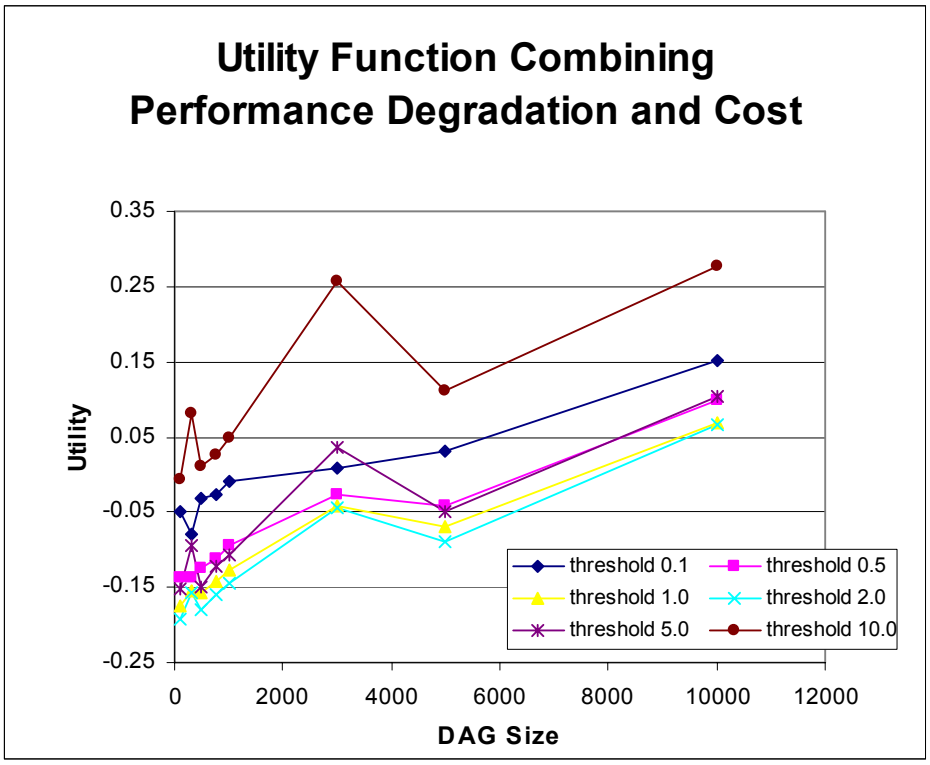

Figure V-7: Utility vs. DAG size for various threshold values

As an example, a user may wish to trade off a 1% decrease in performance for a 10% decrease in cost. Figure V-7 shows the utility for different thresholds. With the 1% / 10% utility above, our prediction model would use a threshold value of 2.0% as the curve minimizes the combination of degradation and cost. Alternatively, users can input the budget for running the

application and our model can choose the threshold corresponding to the best performance degradation while staying within budget.

V.3.3 Comparison with Current Practice

Aside from comparing with the optimal application performance achievable by an optimally sized RC, another way to assess the quality of our predictive model is to compare it with the performance of current practice. The natural and current practice of predicting the optimal RC size for any given DAG is to use the DAG width as the RC size. The DAG width represents the maximum number of tasks that could be executed at any given point in time; thus, using the DAG width as the RC size ensures that every task in the widest level of the DAG could be executed in parallel (assuming the scheduler is intelligent enough to assign tasks in the widest level of the DAG to different hosts if it is at all possible).

Conversely, the DAG width represents an upper bound on the optimal size of the RC. At no point during the execution of the DAG would the application require more hosts than the number of tasks in the widest level of the DAG. However, it is conceivable that not all tasks in the widest level of the DAG are ready to be executed at the exact same time. Thus, it might be possible that some tasks in the widest level of the DAG finish executing before some other tasks in that level. In this scenario, the optimal number of hosts in the RC would be lower than the DAG width.

For each of the ten instances of random DAGs generated for each of the DAG configurations listed in Table V-5 that are within range of our predictive model, we also calculated the application turn-around time for resource collections based on the DAG width. We take the average over the ten instances for the relative difference between the optimal RC size and the DAG width and also the relative difference between the optimal application turn-around time

and the application turn-around time achieved by using the DAG width as the RC size. Table V-7 summarizes our results.

Table V-7: Results using DAG width as the RC size

DAG Size	Observation Set CCR			Midpoint CCR		
Observation Set DAG Sizes	**Average Predicted Size Difference**	**Average App-turn-around Time Difference**	**Relative Cost**	**Average Predicted Size Difference**	**Average App-turn-around Time Difference**	**Relative Cost**
100	96.17%	0.50%	144.84%	130.70%	0.33%	209.16%
500	249.00%	0.20%	425.70%	285.22%	0.10%	437.26%
1000	470.17%	0.45%	562.94%	487.07%	0.36%	586.82%
5000	644.47%	22.66%	998.10%	694.20%	22.45%	1007.88%
10,000	883.53%	130.93%	3360.89%	855.57%	133.21%	3417.52%
Midpoint DAG Sizes						
300	166.68%	0.30%	219.20%	216.75%	0.15%	307.62%
750	415.19%	0.26%	503.04%	442.17%	0.18%	528.00%
3000	636.84%	6.80%	759.76%	657.60%	6.60%	782.24%
7500	493.50%	81.22%	1429.82%	429.03%	73.31%	1212.23%

The first prominent numbers that provide stark contrast to the numbers for our prediction model are the relative costs. Using the DAG width as the RC size incurs enormous costs. As the DAG size increases, the relative costs of the current practice of using the DAG width to choose the RC size grows to be 10 times more expensive for a 5000-task DAG. The predicted size difference confirms that using the DAG width as the basis of the RC size is grossly overestimating the necessary size of the RC to achieve optimal performance.

As one can expect, for smaller DAGs, the performance degradation is not very noticeable and is very comparable to the performance degradation suffered when using our predictive model. The main reason is that the application makespan achieved is equal to the application makespan achievable for an optimal sized RC when both are using the MCP scheduling heuristic. The only difference in performance degradation is in the scheduling time. While the application performance is similar, the average predicted size difference is an order of magnitude greater when using the DAG width as the predictive model to predict the RC size.

When the DAG sizes are bigger, the bigger RC sizes predicted by using the DAG width are contributing to longer scheduling times, thus contributing to worse performance when the DAG size is larger than 1000. The application turn-around time worsens at least by polynomial factors (and possibly exponentially) for DAG sizes greater than 1000.

In summary, our predictive model can achieve equal or better (or much better for larger DAGs) application performance when compared to the current practice of using the DAG width as the RC size while achieving such performance at a fraction of the costs of such practice.

V.3.4 Validate with Real Applications

In addition to validating our predictive model with randomly generated DAGs, we also validate the usefulness of our predictive model by applying it to DAGs from real applications. Some applications that are computationally intensive, such as EMAN [80], do not require use of our predictive model. For those applications, choosing the DAG width as the RC size would yield the best application turn-around time. For other applications such as DAGs from the Southern California Earthquake Center (SCEC) [81], our predictive model is also unnecessary due to the specific structure of the applications. For example, the SCEC DAGs are composed of parallel chains. For such DAGs, the optimal size would equal the number of chains in the DAG. Below, we validate our predictive model with applications that would benefit from such a predictive model and compare the application turn-around time using the predicted RC size from our model and using the DAG width as the RC size.

V.3.4.1 Validate with Montage

Recall from IV.2.1 that Montage is an astronomy application that creates a mosaic image of a portion of the sky on demand. The size of the Montage DAG corresponds to the size of the mosaic. We test two Montage sizes – 1629 tasks and 4469 tasks. The 1629-tasks DAG

corresponds to a three square degree mosaic and the 4469-tasks DAG corresponds to a five square degree mosaic. Table V-8 summarizes the number of tasks in each level for the two Montage DAGs.

Table V-8: Number of tasks in each level for two Montage DAGs

Level	Task Name	Number of Tasks (1629)	Number of Tasks (4469)
1	mProject	334	892
2	mDiffFit	935	2633
3	mConcatFit	1	1
4	mBgModel	1	1
5	mBackground	334	892
6	mImgtbl	12	25
7	mAdd	12	25

Montage DAGs are different from our observation set of DAG configurations in two ways:

1. Low regularity: Recall that DAG configurations in our observation set have regularity values between 0 and 1. Both of these Montage DAGs have negative regularity numbers. (Our predictive model accepts negative numbers of regularity values.)
2. Low CCR values: Montage DAGs have small intermediate files ranging from 200 bytes to 8 Mbytes. We choose a low CCR value of 0.01 for the two DAGs.

Table V-9 summarizes the results of applying our predictive model to the Montage DAGs. Recall from Section V.2.2 that we used a threshold of 0.1% as the "knee" value to determine the best RC size. Here, we vary the threshold from 0.1% to 10%. The performance degradation is the difference in application turn-around time from the optimal application turn-around time. One might reasonably expect that the varying degrees of threshold would correspond to the degree of performance degradation. Thus, a 1% threshold might lead to an application performance degradation of 1%. As we can see from Table V-9, that is not the case. Our prediction model predicted sizes suffered less performance degradation than the thresholds would indicate. For a user with a simple utility function to minimize the sum of performance

degradation and relative cost, the 10% threshold would be the choice for these two Montage DAGs.

For these Montage DAGs, using the DAG width as the RC size (the naïve model) can have similar application turn-around time as using a RC size predicted by our model. However, the relative cost of using the DAG width as the RC size are 89.08% and 195.9% for the 1629-task DAG and 4469-task DAG, respectively.

Table V-9: Applying predictive model to Montage DAGs

	1629-task DAG				**4469-task DAG**			
	Predictiveve Model		**Current Practice**		**Predictive Model**		**Current Practice**	
threshold	**Perf. Degrad.**	**Relative Cost**	**Perf. Degrad.**	**Relative Cost**	**Perf. Degrad.**	**Relative Cost**	**Perf. Degrad.**	**Relative Cost**
0.1%	0.08%	11.20%	0.00%	89.08%	0.00%	0.00%	6.53%	195.9%
0.5%	0.04%	7.52%	-	-	0.00%	-2.40%	-	-
1%	0.01%	0.62%	-	-	0.00%	-4.03%	-	-
2%	0.89%	-13.38%	-	-	1.35%	-21.21%	-	-
5%	0.75%	-30.80%	-	-	1.81%	-30.41%	-	-
10%	4.18%	-48.22%	-	-	4.67%	-50.98%	-	-

V.4 Impact of Clock Rate Heterogeneity

In this section, we address the impact of clock rate heterogeneity within the RC. Clock rate heterogeneity is important from the perspectives of both the application and the resource selection system. Some applications may be able to tolerate more clock rate heterogeneity among the resources in the RC while other users or applications may be economically constrained and higher clock rate heterogeneity among resources could potentially be cheaper to obtain. For most resource selection systems, specifying resource heterogeneity (or specifying a range of clock rates) allows potentially more resources to be considered and more resources to be returned to the application. When a resource selection system cannot fulfill a request for a given resource specification, modifying the specification so that it allows for more resource heterogeneity is common solution. Indeed, with more heterogeneity allowed the resource selection system has more choices.

In this section, we are interested in how the clock rate heterogeneity impacts:

- performance and cost of our predictive model,
- optimal RC size and application turn-around time

V.4.1 Impact on Performance and Cost of Predictive Model

The main issue with introducing clock rate heterogeneity is the question of how our predictive model would be affected. Recall that in Section V.2.4 we formulated a predictive model for resources with homogeneous clock rates. We apply the same methods in formulating predictive models for different clock rate heterogeneity. First, we define clock rate heterogeneity as the coefficient of variance of the host clock rate for the resources among the resources in the RC. For this work, we use clock rate heterogeneity values of 0, 0.01, 0.05, 0.1, 0.2, and 0.3. We maintain a constant mean clock rate in our experiments.

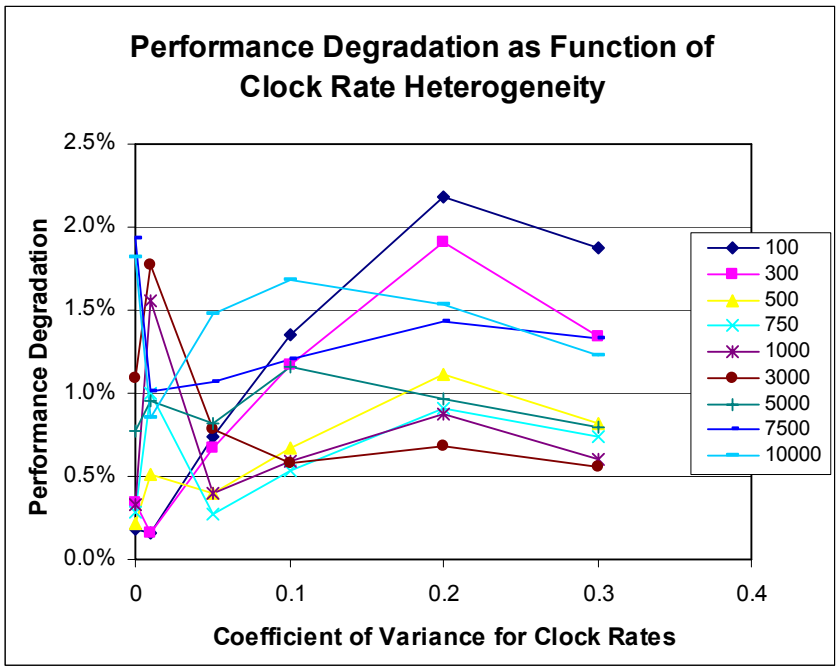

Figure V-8: Performance degradation as function of clock rate heterogeneity for various DAG sizes

Figure V-8 shows the degradation from best turnaround time versus clock rate heterogeneity for DAG configurations with observation set CCR values only (recall that these performed worse). Each of these points represents the average degradation of 252 DAG configurations (we use ten distinct DAG instantiations for each configuration). Each line represents different DAG sizes. We see that higher coefficient of variance among the clock rates did not affect the application turn-around times by using RC configurations predicted by our model. We observed that as the DAG sizes increase, the difference in predicted RC sizes from optimal decreases for larger clock rate heterogeneity. This suggests that it is feasible to increase the DAG size and increase the clock rate heterogeneity without suffering higher performance degradation.

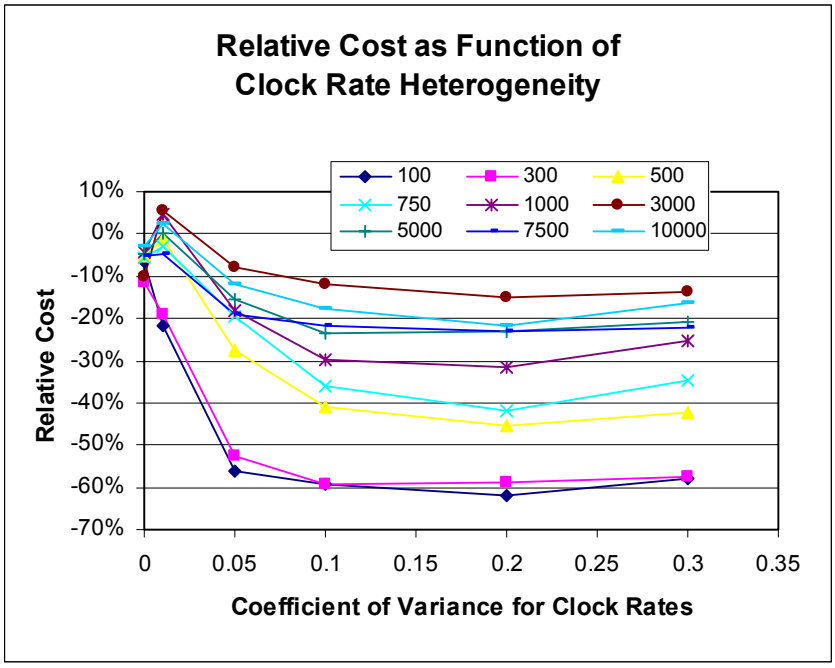

Figure V-9: Relative cost as function of clock rate heterogeneity for various DAG sizes

Figure V-9 shows the relative cost of using our model (line representing different DAG sizes) as a function for clock rate heterogeneity. Again, each point represents the average degradation of 252 DAG configurations (and 2520 total instantiation of DAGs). We see that for

all DAG sizes, as the coefficient of variance increases for clock rates, the costs for the resources decreases. The conclusion from Figure V-8 and Figure V-9 is that our model still leads to performance close to optimal at reduced costs even as heterogeneity increases.

V.4.2 Impact on Optimal RC Size and Application Turn-Around Time

Throughout our experiments, we maintain constant average clock rate when we consider various resource heterogeneity. Thus, when we increase the clock rate heterogeneity, we are adding some faster hosts, along with some slower hosts (to maintain the same average clock rate). With the introduction of clock rate heterogeneity into resource collections, we are interested in answering the following:

- How does the optimal RC size change as a function of the clock rate heterogeneity? Does having some faster hosts allow the scheduler to utilize fewer hosts (thus lowering the optimal RC size)?
- How does the optimal application turn-around time change as a function of the clock rate? Intuitively, one can assume a scheduler that considers clock rates can achieve faster application makespan (and thus faster application turn-around time) with the introduction of faster hosts into the RC.

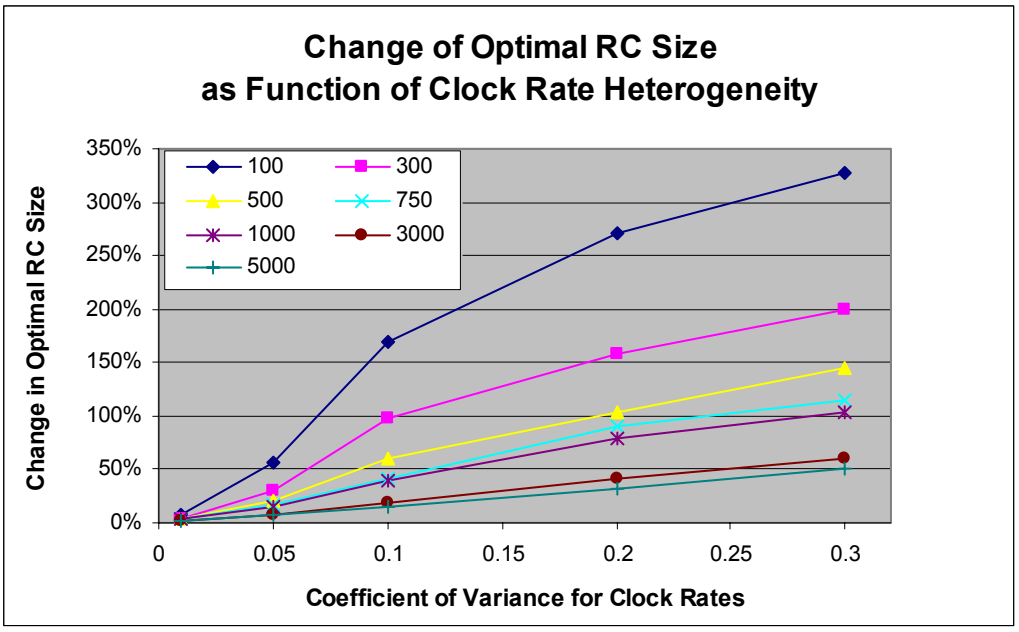

Figure V-10: Change of optimal RC size as function of clock rate heterogeneity

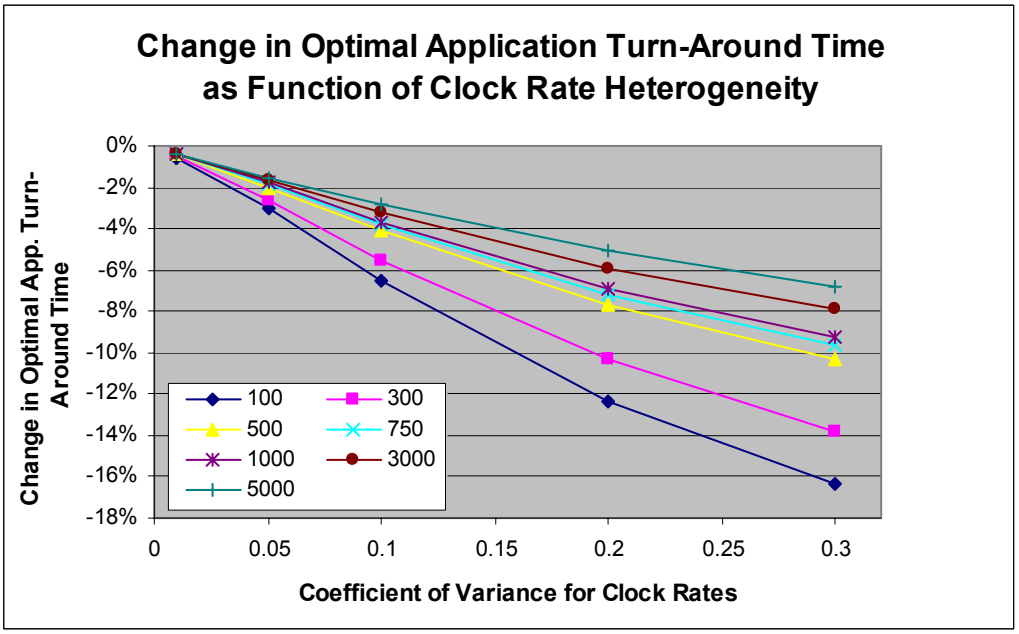

Figure V-11: Change in optimal turn-around time as function of clock rate heterogeneity

Figure V-10 shows the change in optimal RC size for DAGs with varying sizes as a function of resource heterogeneity. The baseline for comparison is the optimal RC size for a homogeneous resource environment. Figure V-11 shows the change in optimal application turn-

around time for different DAG sizes as a function of resource heterogeneity. The baseline for comparison is the optimal application turn-around time for a homogeneous resource environment. Each of the points in Figure V-10 and Figure V-11 represents the average degradation of 252 DAG configurations (we use ten distinct DAG instantiations for each configuration). Each line represents different DAG sizes.

From Figure V-10 and Figure V-11, we see the effects of clock rate heterogeneity on both optimal RC size and optimal application turn-around time as roughly linear for the range of DAG sizes in our experiments. From Figure V-10, we see the trend that as the DAG sizes increases, the relative change in optimal RC size decreases. With increased clock rate heterogeneity, increasing the RC size means that there are more faster hosts in the RC. For smaller DAGs, fewer faster hosts are sufficient for faster application makespan. Because the RC size is small to begin with, by doubling or tripling the RC size, the scheduling time is not significantly impacted while the makespan can be made faster. For bigger DAGs, doubling or tripling the RC size impacts the scheduling time and offsets the faster application makespan achievable by the presence of faster hosts in the RC. Thus for bigger DAGs, the optimal RC size does not change as much compared to smaller DAGs for increased clock rate heterogeneity.

From Figure V-11 we see faster application turn-around time for increased clock rate heterogeneity due to the presence of faster hosts in the RC. Because smaller DAGs do not incur as much scheduling penalty, by increasing the RC size, we are able to achieve faster application turn-around times for smaller DAGs for increasingly heterogeneous resource collections. Although increasing the resource heterogeneity increased the number of faster hosts and decrease the overall application turn-around time, the impact on bigger DAGs is less than the impact on smaller DAGs. This is most likely because more fast hosts are required to improve the overall performance of bigger DAGs.

From Figure V-8, Figure V-9, and Figure V-11, we draw the conclusion that by using our prediction model, for applications capable of tolerating clock rate heterogeneity (most workflow applications are), users can achieve better application performance at cheaper costs than when using homogeneous resources.

V.5 Impact of Network Heterogeneity

With projects such as OptIPuter [25] researching higher bandwidth, we believe that higher bandwidth will be prevalent in the short- and medium-term future. Further, we believe that connections between research institutions (where our target scientific applications are likely to be deployed) will be the first to achieve higher bandwidth and the heterogeneity among the different links should be low. We have already mentioned that network latency does not have much impact on the execution of loosely synchronous applications such as workflows. Furthermore, our model already accounts for two orders of magnitude in CCR values. Therefore, we feel that a study of network bandwidth heterogeneity is not critical and we do not address this issue in this dissertationImpact of Using Different Scheduling Heuristics

All of the results from the previous sections have assumed a reference scheduling heuristic, MCP. We are also interested in seeing whether our model can be applied to other types of scheduling heuristics. Because of the time-consuming nature of running comprehensive experiments on all of the different heuristics, we perform a sensitivity analysis on a subset of the DAG configurations.

V.6 Scheduling Heuristics

We apply the same method for constructing the predictive model described in Section V.2.4 for MCP to construct models for three other scheduling heuristics. We choose the DLS

(Dynamic Level Scheduling) algorithm [82], as both MCP and DLS are popular and competitive according the results in [73]. We then use two other simpler heuristics: Fastest Compute-host Available (FCA) and First-Come-First-Serve (FCFS).

The MCP heuristic operates by first sorting the tasks in the DAG according to their ALAP (As-Late-As-Possible) values (also known as t-levels). The ALAP values are computed by first computing the length of the critical path, which is the length of the longest path, and subtracting the *b-level* (bottom-level) of the task from it. The b-level of a task n_i is the longest path from n_i to an exit task in the DAG (including both n_i and the exit task). The length of the path includes both computation and communication costs. We use the reference computation cost for each task for this calculation and a reference communication cost for each necessary file transfer between the host executing the predecessor of n_i and the host executing n_i. After sorting the tasks in ascending order of ALAP, the MCP heuristic assigns tasks on the static list one by one such that a task is scheduled on a processor that allows the earliest start time. Figure V-12 shows the pseudo-code for the MCP heuristic.

```
CP =    length of the longest path (in terms of node weights and edge weights)
        from the root node to the end node, including both these nodes
For each non-root node Ni in the DAG
  BLi = length of the longest path (in terms of node weights and edge weights)
        from node Ni to the end node, including both these nodes
  ALAPi = CP – BLi
End For

For each node Ni
  Li = list of the ALAP values of node Ni and all its descendents, in ascending order
End For

Sort all Li lists in lexicographical order and
Re-Order the nodes according to this order
For each node Ni

  Schedule Ni on the host that would complete its execution soonest
```

Figure V-12: Pseudo-code for the Modified Critical Path (MCP) Heuristic

The DLS heuristic operates similarly to the MCP heuristic, with the exception of sorting according to an attribute called the *dynamic level* (DL) instead of the t-level. The DL is the

difference between the *static level* (SL) of a task and its earliest start time on a processor. The SL level of a task is the length of the longest path from task n_i to an exit node where the length of the path includes the computational costs and excludes the communication costs. Figure V-13 shows the pseudo-code for the DLS heuristic.

```
For each non-root node N_i in the DAG
   SL_i =  length of the longest path (in terms of node weights only)
           from node N_i to the end node, including both these nodes
   TL_i =  length of the longest path (in terms of node weights only)
           from the root to node N_i, including both these nodes
   DL_i = SL_i – TL_i
End For

For each node N_i
   L_i = list of the DL values of node Ni and all its descendents, in ascending order
End For

Sort all L_i lists in lexicographical order and
Re-Order the nodes according to this order
For each node N_i
   Schedule Ni on the host that would complete its execution soonest
End For
```

Figure V-13: Pseudo-code for the Dynamic Level Scheduling (DLS) Heuristic

The FCA (Fastest Compute-host Available) heuristic operates by keeping a sorted queue (sorted according to clock rate) of available compute hosts. Whenever all of the predecessors of a task n_i have finished executing (n_i is then considered a ready task), FCA assigns n_i to the first host in the queue. This heuristic does not consider communication costs, but is very fast to execute. Figure V-14 shows the pseudo-code for the FCA heuristic.

```
While there are still some tasks to schedule
   For each node N_i whose predecessors, if any, have already been scheduled
      Schedule N_i on the fastest host that would start its execution soonest
   End For
End While
```

Figure V-14: Pseudo-code for the FCA Heuristic

The last heuristic is FCFS, which is the simplest heuristic. FCFS operates by assigning any ready tasks randomly to any available compute host in the resource collection. It does not

consider communication or computational costs, but is very fast to execute. Figure V-15 shows the pseudo-code for the FCFS heuristic.

```
While there are still some tasks to schedule
   For each node N_i whose predecessors, if any, have already been scheduled
      Schedule N_i on the host that would start its execution soonest
   End For
End While
```

Figure V-15: Pseudo-code for the FCFS Heuristic

V.6.1 Sensitivity Studies for Different Heuristics

We perform experiments with DAG configurations with CCR values of 0.01 and 1.0, parallelism values of 0.4 and 0.8 and keep regularity constant at 0.5 (because regularity impacts our model the least). For each DAG configuration, we create ten distinct DAG instantiations with the same DAG characteristics.

We present Figure V-16 and Figure V-17 as the worst-case scenarios for our predictive model (a single DAG configuration with parallelism value of 0.8 and CCR value of 0.01) for different scheduling heuristics under different resource conditions. Each of the points represents the average of the ten DAG instantiations.

Figure V-16 shows the performance degradation for different heuristics. The two more complex heuristics, MCP and DLS, maintain application turn-around times within 3% of the approximated best times for different clock rate heterogeneity. The two simpler heuristics, FCA and FCFS suffer more variability across the different clock rate heterogeneity. In the worst case for all heuristics, FCA suffers performance degradation of less than 7.3% for resource heterogeneity between 0.1 and 0.2.

Figure V-17 shows the relative costs of each heuristic over different clock rate heterogeneity. As with performance degradation, MCP and DLS maintained stable cost savings at around 40% for most of the range of clock rate heterogeneity. FCA suffers the most performance

degradation but yielded the most cost savings at around 65% for the range of clock rate heterogeneity. As expected, the FCFS heuristic is more random and performance degradation increases with higher heterogeneity among the resources and costs are more unpredictable.

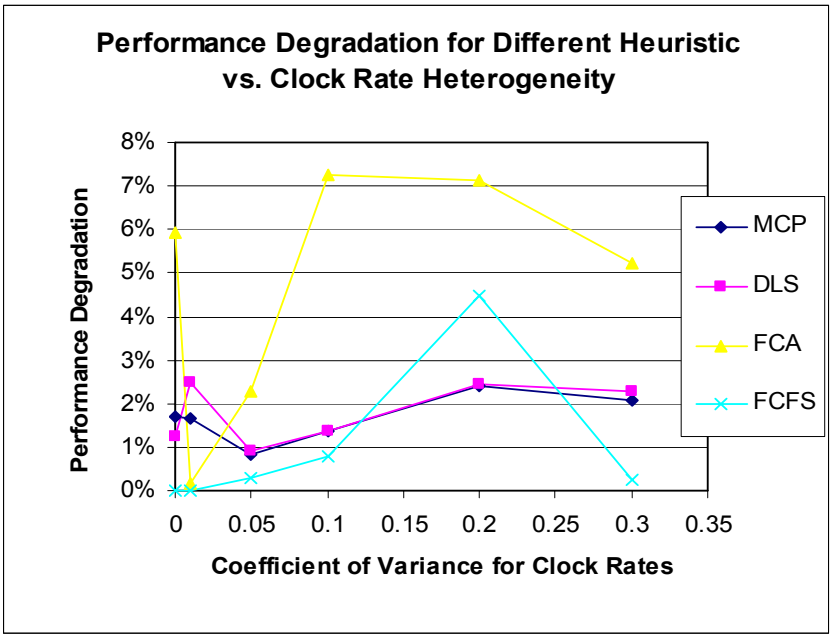

Figure V-16: Performance degradation for different heuristics and resource conditions

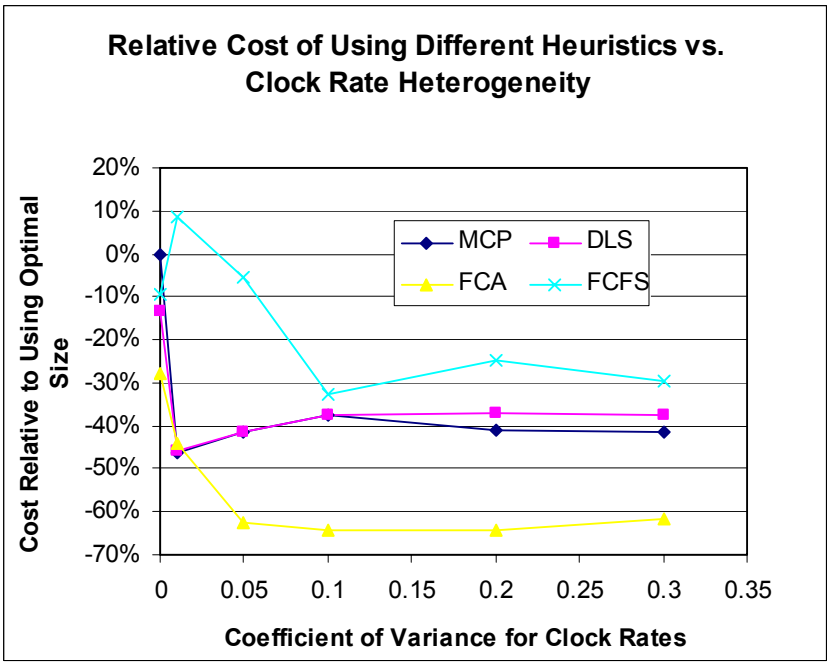

Figure V-17: Relative costs of using different heuristics over different resource conditions

Our results indicate that our RC size prediction model is robust and can be applied across different levels of resource heterogeneity and for different scheduling heuristics. For all of our sample space, any heuristic running in RCs with any resource heterogeneity achieved application performance within 7.3% of approximated best performance. Additionally, using our prediction model consistently provided cost savings ranging from 40% for the more sophisticated MCP and DLS to 65% for FCA for broad range of resource heterogeneity.

V.7 Effects of Scheduling and Computational Clock Rate Ratios

One possible concern about the applicability of our resource specification predictor is our usage of reference clock rates for both the scheduling clock rate and the average computational clock rate within the resource collection during the construction of the size prediction model. Indeed, it is clear that our predicted RC size could be inaccurate for different scheduling and/pr computational clock rates. One option is to simply re-construct our predictive model for the clock rates at hand, which is straightforward if perhaps time-consuming.

An alternative to re-constructing the size prediction models is to examine the effects of varying the scheduling-to-computational clock rate ratio (SCR). We have two goals for this section:

1. Identify what DAGs would be affected by changing the SCR.
2. Derive formulas predicting the best RC size for DAG affected by changing the SCR.

V.7.1 Identifying DAGs Affected by Varying SCR

Throughout our experiments, we used reference scheduling clock rate of 2.8GHz and reference computational clock rate of 3.5 GHz. Our strategy for identifying the DAGs affected by varying the SCR is to re-compute the predicted RC size for all the DAGs in our observation set, while varying the SCR. We choose SCR value of 0.1, 0.5 1.0, 2.0, 5.0, and 10.0 to represent the

range of possible scheduling CPU-to computational CPU-ratios. At the extremes, either the scheduling or the computational CPU could be faster or slower by a factor of 10. Our observations from the RC size prediction re-computations are the following:

1. For scheduling heuristics where the scheduling time is negligible, as in the case of heuristics FCA and FCFS, changing the SCR does not affect the knee values at all. Thus for all SCR, our size prediction work for FCA and FCFS.
2. For more complex scheduling heuristics MCP and DLS, the changes to the knee values due to varying SCR are very similar, so we show only the results for MCP hereafter, but they can be applied to scenarios where DLS is employed.
3. For smaller DAGs (size 1000 and smaller), varying the SCR does not change the predicted RC sizes in most cases, except for when the parallelism is high (when parallelism = 0.9). This makes sense because DAGs with higher parallelism have larger optimal RC sizes. Figure V-18 shows an example plot of predicted size change due to varying SCR for small DAGs over a range of parallelism and SCR values. It is for DAGs of size 1000, CCR of 0.01 and regularity of 0.5. The z-axis show the change in predicted RC size relative to the SCR value of 1. This plot is for a CCR value of of 0.01, but the plots for all the CCR values show very similar trends. For smaller DAGs, when the scheduling CPU is significantly slower than the average computational CPU, then the predicted RC size is decreased to minimize the application turn-around time. (A smaller RC size implies faster scheduling time.)

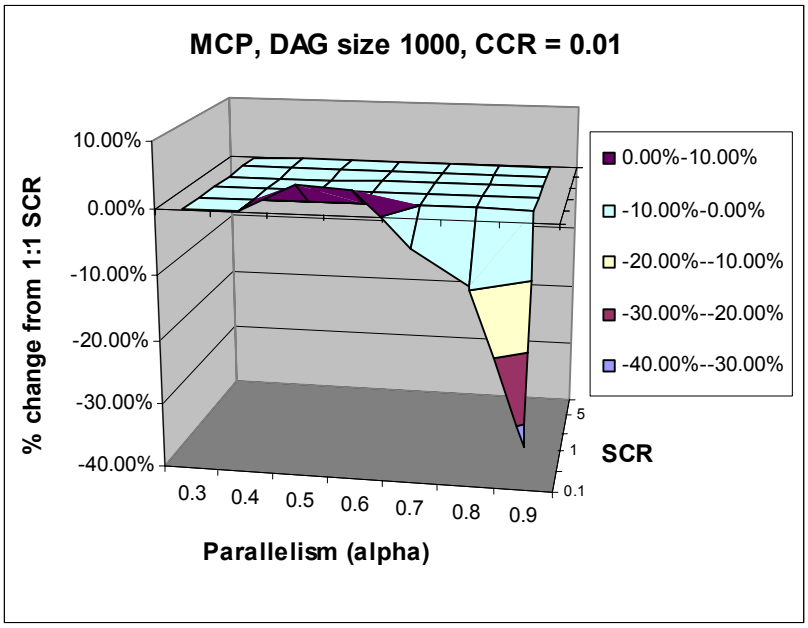

Figure V-18: Example plot of predicted RC size change due to varying SCR for small DAGs

For bigger DAGs and homogeneous resources, only the highly parallel DAGs are affected by changing the SCR. Figure V-19 shows an example plot of larger DAGs for homogeneous resources. It is for DAGs with size 5000, CCR of 0.01, and regularity of 0.5. The DAGs mostly affected are the ones with parallelism of 0.9 and to a lesser extent parallelism of 0.8. Different CCR values show similar shapes as the plot in Figure V-19. In Figure V-20, we show a similar plot by fixing the parallelism to 0.9 and varying the CCR. We see that DAGs with lower CCR values are affected more than the DAGs with higher CCR values. This is expected as with the lower CCR values DAGs have larger optimal RC sizes. By varying the SCR, we expect the impact to be greater for DAGs with bigger RC sizes.

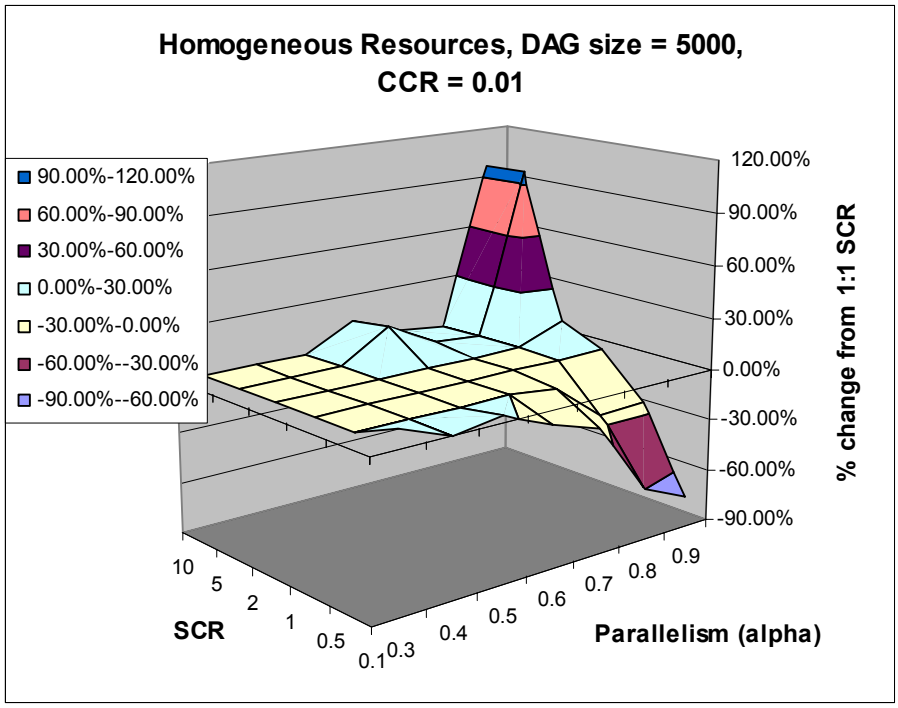

Figure V-19: Example plot of predicted RC size change due to varying SCR and parallelism for larger DAGs in homogeneous resource environment

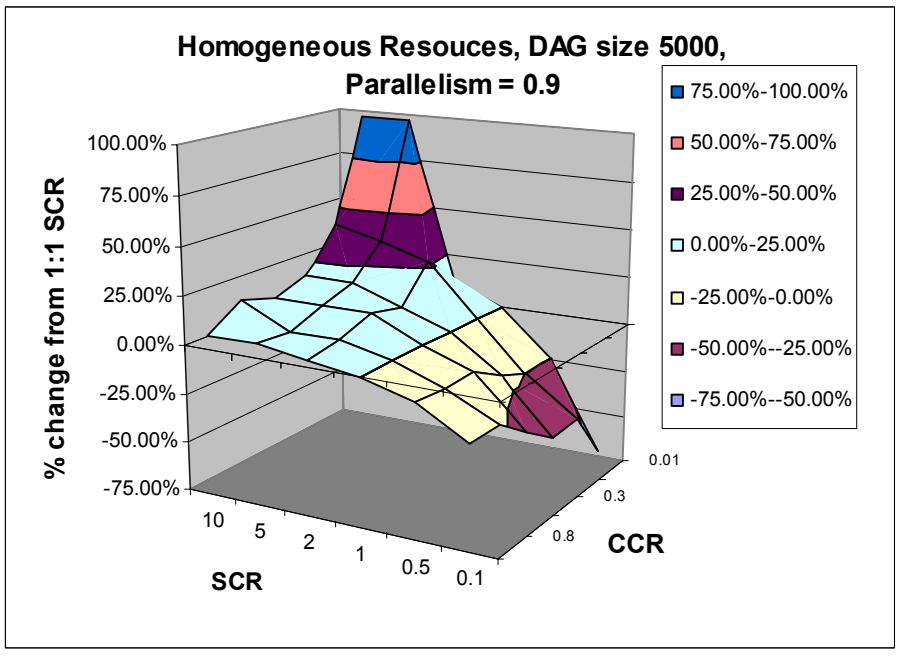

Figure V-20: Example plot of predicted RC size change due to varying SCR and CCR for larger DAGs in homogeneous resource environment

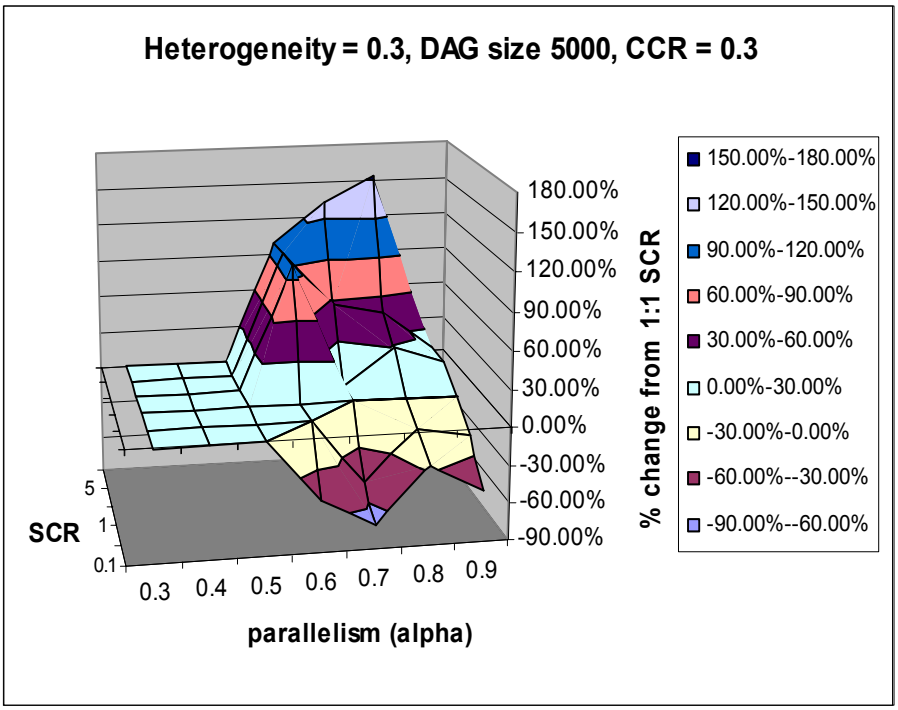

Figure V-21: Example plot of predicted RC size change due to varying SCR and parallelism for larger DAGs in heterogeneous resource environment

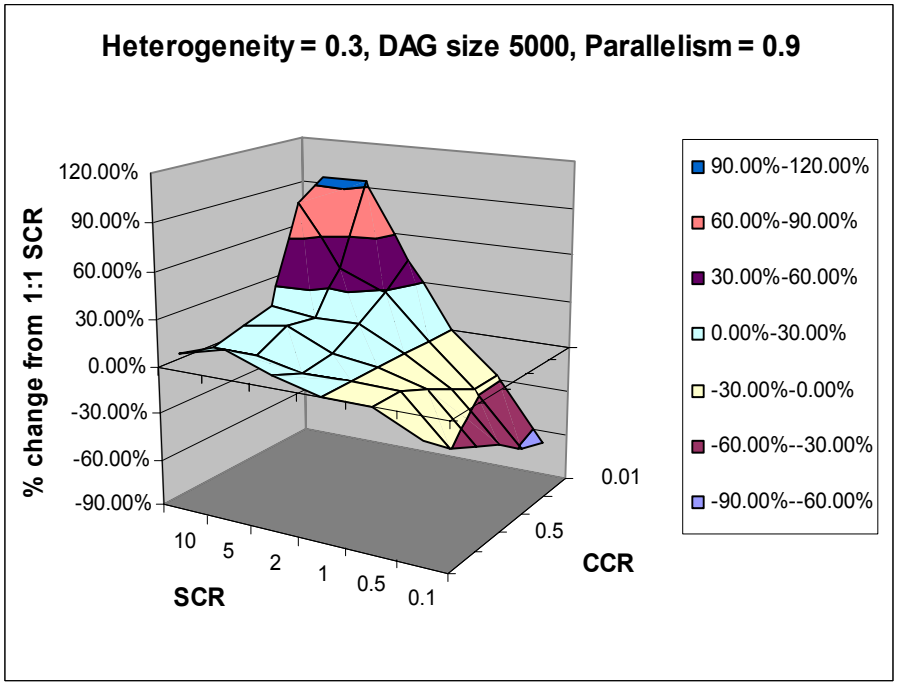

Figure V-22: Example plot of predicted RC size change due to varying SCR and CCR for larger DAGs in heterogeneous resource environment

As we increase the resource heterogeneity, we observe more DAGs affected by the SCR. Figure V-21 shows an example plot of predicted RC size change due to varying SCR and parallelism for DAGs with size 5000, CCR of 0.3, regularity of 0.5, and resource heterogeneity of 0.3. For DAGs with other CCR values, we see similar trends. Figure V-22 shows a similar plot by fixing the parallelism to 0.9 and varying the CCR. Again, we see that DAGs with lower CCR values are affected more than the DAGs with higher CCR value.

From our observations, we draw the conclusions that smaller DAGs are not particularly affected by SCR. Larger DAGs are affected when they exhibit high parallelism and low CCR. With increasing resource heterogeneity, the range of affected parallelism and CCR increases. For example, for homogeneous resources, DAGs of size 5000 are affected only for parallelism values of 0.9 and to a smaller extent, 0.8. With resource heterogeneity of 0.3, the affected parallelism extends to 0.6.

V.7.2 Modifying RC Size Predictions

Although there is nothing fundamentally wrong with the re-construction of the size prediction model based on new scheduling CPU speed or new mean computational CPU speed, in this section, we explore another way to modify the predicted RC size based on the reference scheduling and CPU speeds for various scheduling-to-computational CPU speed ratios. From the previous section, we can identify the DAGs where a modification is necessary. The remaining step is to derive formulas to predict new RC sizes based on SCR and the size prediction model derived earlier in this chapter. From Figure V-20 and Figure V-22, we can fit logarithmic regression lines to predict the changes in predicted RC size as a function of SCR. Figure V-23 and Figure V-24 show logarithmic formulas to predict changes in predicted RC size as functions of SCR for homogeneous resources and resource heterogeneity of 0.3, respectively. Both figures are for DAGs with sizes of 5000, parallelism of 0.9 and regularity of 0.5. The different lines

represent different CCR values. As we can see, the lower the CCR values, the bigger the changes to predicted RC sizes based on changes to SCR. Note that the R squared values for all of the fits are greater than 0.9, indicating decent fits.

From these two plots (and others like them), we can derive formulas to modify the predicted RC size as a function of the scheduling-to-computational clock rate ratios. The high R squared values suggest that the adjusted predicted size would be close to the optimal RC size for the varying SCR. We draw the conclusion that our RC size prediction model can be applied over arbitrary scheduling and computational clock rate ratios.

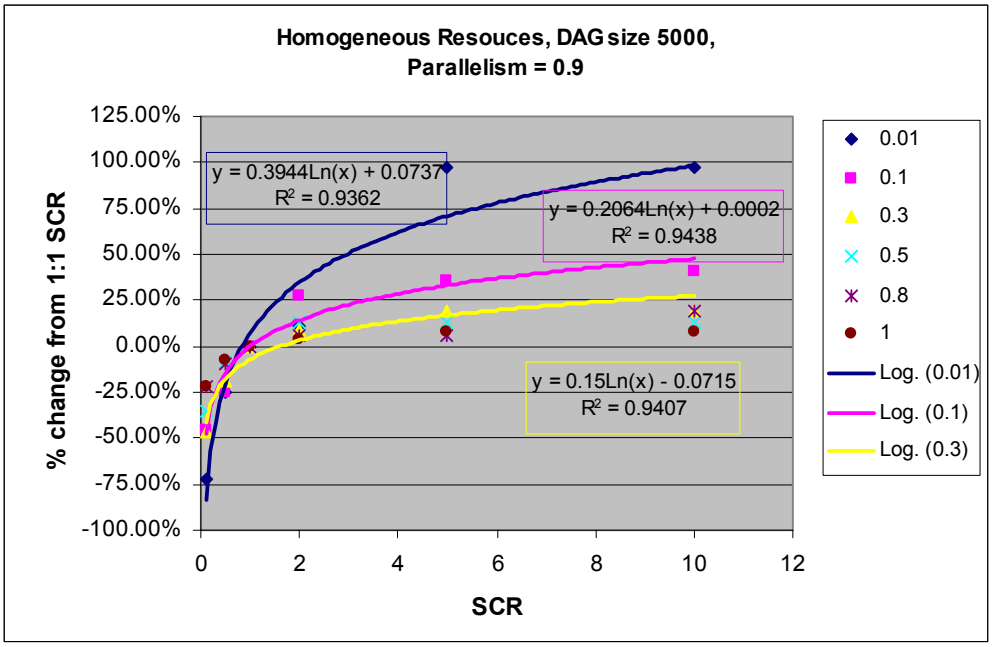

Figure V-23: Formulas predicting changes in predicted RC sizes as functions of SCR for DAGs with size 5000, parallelism of 0.9, with homogeneous resources

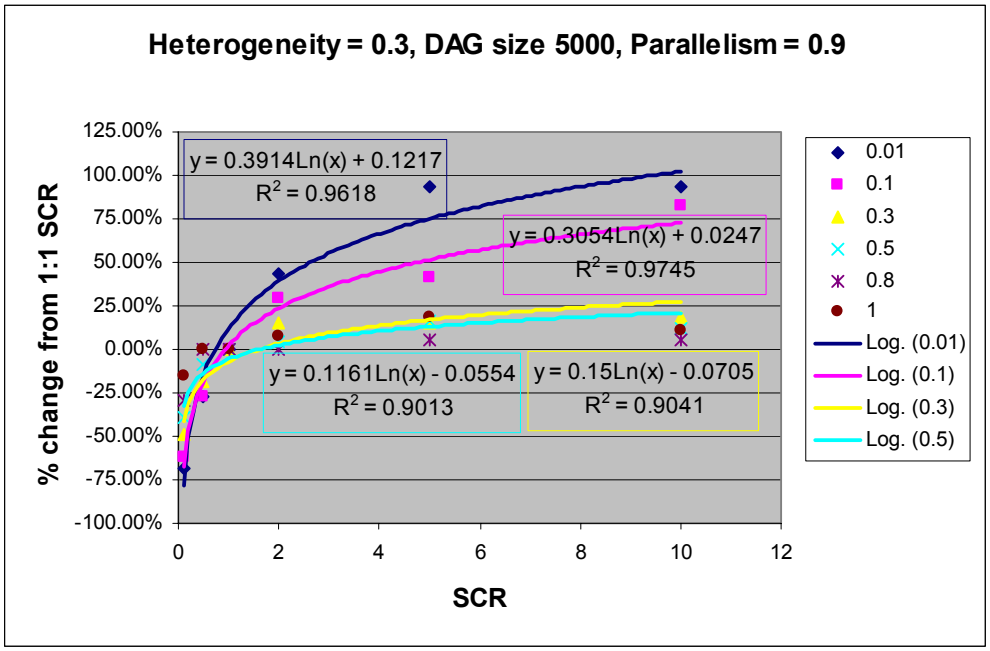

Figure V-24: Formulas predicting changes in predicted RC sizes as functions of SCR for DAGs with size 5000, parallelism of 0.9, with resource heterogeneity of 0.3

V.8 Conclusion

In this chapter, we have provided part of the missing link between applications and resource selection systems. We have constructed a prediction model based on relevant application characteristics, in our case scientific workflows, to output the best resource specification to optimize application performance (or some other utility that the user may specify). In this chapter, we focused solely on the component of the prediction model that predicts the best number of resources to use (given a scheduling heuristic) in resource requirement specifications sent to a resource selection service, while considering performance and cost tradeoffs. In extensive simulation over a wide range of workflow configurations, we showed that our prediction model consistently allowed workflows to achieve performance within a few percent of optimal. When applied to a real application, we showed that our prediction model lead to almost optimal performance. Furthermore, when comparing the usage of our prediction model with current

practice, we have found that using our model is far more cost effective while achieving better performance.

Our above validation was done for a set of homogeneous resources and a reference scheduling heuristic. A sensitivity analysis was required to show that our model can maintain accuracy over different resource heterogeneity and over different scheduling heuristics. Reproducing comprehensive experiments over all ranges of resource heterogeneity and all scheduling heuristics such as we did for running MCP over homogeneous resources would have been extremely time-consuming and near impossible. Instead, we sampled the resource heterogeneity space and chose four scheduling heuristics as a sensitivity analysis. We found that our model can be applied to different scheduling heuristics over resources with different levels of resource heterogeneity and different scheduling heuristics.

While we have validated the accuracy of our RC size predictor for different scheduling heuristics over different resource heterogeneity, we use a reference clock rate for the scheduler and an average CPU speed for resource collections. We investigated the effects of using different reference scheduling-to-computational clock rate ratios (SCR) and found that smaller DAGs are not particular affected by the SCR. Larger DAGs are affected when they exhibit higher parallelism and lower CCR. The ranges of affected parallelism and CCR are extended with increasing resource heterogeneity. Although our techniques for deriving the size prediction model can be employed to fully re-construct new prediction models based on a new scheduler CPU or a different average computational CPU, we also derived formulas to show how our predicted RC sizes can be adjusted to reflect newer SCR. Many of the formulas are based on logarithmic fits with R squared values greater than 0.9.

While this chapter focused on predicting the best size for a resource collection given the workflow application and a desired scheduling heuristic, a natural next step is to suggest to the application user the best scheduling heuristic in conjunction with the best resource specification

to provide the optimal application turn-around time at the lower cost. In the next chapter (Chapter VI), we use similar techniques as those employed in this chapter to construct a heuristic prediction model that predicts the best heuristic to use given the input workflow application. In Chapter VII, we combine the heuristic prediction model and the size prediction model and add a third component called the resource specification generator to automatically generate resource specifications for different resource selection systems.

V.9 Acknowledgement

Chapter V, in part, has been published as "Generating Grid Resource Requirement Specifications" by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC 2007). The dissertation author was the primary investigator and author of this paper.

Chapter V, in part, has been submitted for publication and will appear as "Automatic Resource Specification Generation for Resource Selection" by Richard Huang, Henri Casanova, and Andrew A. Chien in the proceedings of the ACM/IEEE International Conference on High Performance Networking and Computing (SC 2007). The dissertation author was the primary investigator and author of this paper.

VI

DERIVING THE BEST SCHEDULING HEURISTIC

Application performance depends not only on the physical resource characteristics, but also on the scheduling heuristic used to assign application tasks to resources. Recognizing the need for both the best set of resources, but also the best scheduling heuristic to optimize application performance, our vision of automatically generating resource specifications is composed of two models: one that predicts the best RC size and one that predicts the best scheduling heuristic for a given application. In Chapter V, we formulated an empirical model that predicts the best RC size given a scheduling heuristic along with the application and an optional utility function trading off application performance and cost. In this chapter, we focus on formulating a model that predicts the best scheduling heuristic given an application and additional performance-cost tradeoff constraints.

A single scheduling heuristic is not likely to yield best application performance for all resource collections because of the following:

1. Resource collections can vary in size. As the RC size increases, more complex scheduling heuristics that considers all the possible resources in the RC will take longer to run. Longer scheduling time contributes to longer application turn-around time.
2. Resource collections can vary in the heterogeneity of clock rates among the resources in the RC. More complex scheduling heuristics that consider task execution time on each resource can take advantage of faster resources and potentially reduce the application makespan. Reducing the application makespan contributes to shorter application turn-around time.

The problem we address in this chapter is how to formulate a model to predict the best scheduling heuristic for a given application. The input to the model is the application itself, with optional utility function. As seen from the two points above, application performance depends on the resources on which the scheduling heuristics are assigning tasks. Thus, we define the problem as formulating an empirical model to predict the best heuristic *assuming that the best set of resources are used for each heuristic* (e.g., by using a combination of the RC size predictor from Chapter V and accounting for resource heterogeneity).

Our proposed approach (similar to our construction of the RC size predictor in the previous chapter) is as follows:

1. Construct an observation set of DAG configurations spanning relevant DAG characteristics.
2. For each scheduling heuristics (MCP, DLS, FCA, and FCFS), run experiments over all the DAG configurations and over different types of resource environments to determine the best possible application performance for each (DAG configuration, heuristic) pair.
3. For each DAG configuration, compare the best possible application performance from each of the four scheduling heuristics (they will have different optimal resource collection sizes).
4. From the observation set of DAG configurations, delineate regions where one heuristic work better than other heuristics.
5. Construct a prediction model for all DAG configurations based on the above delineated regions.
6. Validate this model.

VI.1 Observation Set of DAG Configurations

We simulate different scheduling heuristics over an observation set of DAG configurations in an attempt to formulate a model to predict the best scheduling heuristic given any arbitrary DAG. We hope to derive trends from the results of our simulation to form our prediction model. Similar to how we choose the observation set for predicting the RC size (in the previous chapter), we choose our sample values at as evenly-spaced intervals as possible for each of the relevant DAG parameters. We choose DAG sizes ranging from 100 tasks to 10,000 tasks as this represents the range of the interesting DAGs scientists run today. For both CCR and parallelism, we maintain the same observation set DAG characteristic values as before: ranging from 0.01 to 1.0 for CCR and 0.3 to 0.9 for parallelism. For regularity, we choose a value of 0.5. From our experiences with the RC size predictor, changes in regularity did not affect the application makespan significantly and thus we do not expect that two DAGs with identical characteristics except for regularity would require two different scheduling heuristics to achieve their respective best application turn-around time. Table VI-1 summarizes the values we choose for each of the DAG characteristics in the observation set.

Table VI-1: DAG characteristics used for the observation set to derive a model for heuristic prediction

DAG Characteristic	Values
DAG size (# of tasks)	100,500,1000,5000,10000
CCR	0.01, 0.1, 0.3, 0.5, 0.8, 1.0
Parallelism (α)	0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
Regularity (β)	0.5

VI.2 Identifying Trends from Observation Set

We simulate four scheduling heuristics (MCP, DLS, FCA, and FCFS, which are described in V.6) scheduling each of the 710 DAG configurations (generated by the combination of the different DAG characteristics listed in Table VI-1) over resource collections of various

sizes and various resource heterogeneity. For each DAG configuration, we randomly generate ten distinct DAG instances (similar to how we generate the observation set of DAGs for the RC size predictor). For each (DAG configuration, heuristic) pair we determine the best possible application performance over different resource collection sizes and over different resource heterogeneity (using coefficient of variance values of 0, 0.01, 0.05, 0.1, 0.2, and 0.3) within the resource collection.

The trends we observed were the following:

1. *We can eliminate FCFS from the heuristic prediction model*. For DAG size 500 or greater, the optimal application turn-around time is achieved by MCP, DLS, or FCA, but never FCFS. Even in DAG configurations of size 100 where FCFS had the best application performance, the relative improvement in performance over other scheduling heuristics is less than 0.1%. We can thus leave FCFS out of any heuristic prediction model as other scheduling heuristic can achieve application performance that is no worse than 0.1% in all cases.

   Table VI-2 lists all of the best application turn-around times (denoted as "best time") for the four scheduling heuristics for DAG configurations in the observation set with CCR values of 0.8 and 1.0 and parallelism between 0.3 and 0.9. For each scheduling heuristic, we list the best application turn-around time for that particular DAG configuration and the resource heterogeneity which enabled that best time. Additionally, for each DAG configuration, we list the best overall scheduling heuristic and the resource heterogeneity which enabled that best time.

   We see that all four scheduling heuristics have very comparable performance in the cases where zero resource heterogeneity leads to the best application performance (i.e., CCR

0.8, parallelism from 0.4 to 0.8 and CCR 0.9, parallelism from 0.4 to 0.7). In some cases, using FCFS leads to the best application performance, but the performance gain is less than 1 second (less than 0.01%) over the other scheduling heuristics. Thus we can eliminate FCFS from the model because for cases where using FCFS leads to best performance, another scheduling heuristic can be used to achieve very comparable performance.

Table VI-2: Application Turn-around times for DAG size 100

		MCP		DLS		FCA		FCFS		Overall Best	
CCR	Parallelism (α)	Best time (s)	Res Het	Best time (s)	Res Het	Best time (s)	ResHet	Best time (s)	Res Het	Heuristic	Res. Het.
0.8	0.3	1303	0.3	1217	0.3	1691	0	1691	0	DLS	0.3
	0.4	1707	0	1707	0	1707	0	1707	0	FCA	0
	0.5	1705	0	1705	0	1705	0	1705	0	FCFS	0
	0.6	1713	0	1713	0	1713	0	1713	0	FCA	0
	0.7	1711	0	1711	0	1711	0	1711	0	FCA	0
	0.8	1528	0.3	1518	0.3	1691	0.3	1707	0	DLS	0.3
	0.9	1176	0.3	1178	0.3	1240	0.3	1260	0	MCP	0.3
1.0	0.3	1505	0.3	1398	0.3	1691	0	1691	0	DLS	0.3
	0.4	1707	0	1707	0	1707	0	1707	0	FCA	0
	0.5	1705	0	1705	0	1705	0	1705	0	FCFS	0
	0.6	1713	0	1713	0	1713	0	1713	0	FCFS	0
	0.7	1711	0	1711	0	1711	0	1711	0	FCFS	0
	0.8	1707	0	1707	0	1707	0	1707	0	FCFS	0
	0.9	1457	0.3	1460	0.3	1535	0.3	1556	0	MCP	0.3

2. *Given our experimental methodology, higher resource heterogeneity leads to lower application makespan.* For any DAG sized 500 or greater, the optimal application performance can be achieved with the highest resource heterogeneity in our experiments (0.3). This is due to the presence of faster machines in the resource collection. Recall that in all resource collections, we maintain a constant mean clock rate, so higher resource heterogeneity means some faster machines and some slower machines. Three of the scheduling heuristics (MCP, DLS, and FCA) considers the clock rate of the machines when scheduling, thus faster machines led to better application performance. Only for DAGs of size 100 does there exist some DAG configurations for which homogeneous

resources enabled better application performance. In Table VI-3, we show the application performance degradation using MCP and DLS in a resource collection with a resource heterogeneity level of 0.3 instead of using FCA or FCFS in a homogeneous resource environment. Out of the 710 DAG configurations we tested, these 9 configurations are the only ones for which using a more heterogeneous resource collection did not improve application performance. In the worst case, the performance degradation is still under 20% for these 1.3% (9 out of 710) DAG configurations for which using a homogeneous resource collection is better. Thus, we proceed with the assumption that given constant mean clock rate, higher resource heterogeneity leads to better application performance.

Table VI-3: Performance degradation using 0.3 instead of 0 for resource heterogeneity

CCR	Parallelism	MCP	DLS
0.8	0.4	18.9%	10.1%
	0.5	16.0%	19.0%
	0.6	16.0%	19.0%
	0.7	16.1%	18.0%
1.0	0.4	18.9%	18.7%
	0.5	16.0%	19.0%
	0.6	16.0%	19.0%
	0.7	16.1%	19.1%
	0.8	8.7%	9.2%

3. *Using FCA optimizes application performance only for bigger DAGs with lower CCR and/or higher parallelism.* Intuitively, one would expect this trend as FCA is most likely to perform better for DAGs with higher computational costs and for highly parallelized DAGs because FCA assigns tasks to the fastest compute resources available. A low CCR means less communication and thus FCA can be expected to perform better. In terms of parallelism, a DAG that has 100% parallel tasks would perform best by simply assigning each task to the fastest compute resource available. Using MCP or DLS would degrade application performance because of the extra scheduling costs.

4. *For all DAGs sized 1000 or smaller, either MCP or DLS led to best performance.* For smaller DAGs, the better application makespan achieved by the more complex MCP and DLS more than offset any of the scheduling costs.
5. *Similar performance between MCP and DLS.* Across all DAG configurations and different resource heterogeneity, we observe that MCP and DLS in almost all cases performed within 5% of each other. Other than similarity in performance, there is no discernible pattern for conditions under which one heuristic would perform better than the other. For example, at CCR 0.3 and parallelism of 0.6, MCP outperforms DLS for all tested DAG sizes except for 100. Yet when we change the parallelism to 0.7, DLS outperforms MCP when the DAG size is 500 but for no other DAG size.

VI.3 Heuristic Prediction Model Construction

Based on the trends we observed, the problem of predicting the best heuristic reduces to one of deciding when to use one of MCP or DLS and when to use FCA. We choose to use only MCP over DLS because for 85% of the DAG configurations with sizes 5000 or 10,000, MCP performed better. For smaller DAGs, that percentage decreases to less than 50% (DLS performed better more than half the time). However, even when DLS outperforms MCP, the overall turn-around time achieved by MCP is still within 5% of DLS. Thus, our problem further reduces to one of deciding when to use MCP and when to use FCA.

We know that for DAG size 5000 or 10,000, FCA performs better for DAGs with low CCR and high parallelism, but when the DAG size is 1000, MCP performs better. Thus, we draw the conclusion that there exists a DAG size between 5000 and 10,000 where using MCP or FCA would lead to the same performance. In order to determine the threshold (in terms of DAG size) where MCP ceases to produce the best application turn-around time and instead FCA produces the best application performance, we resort to linear interpolation to predict application turn-

around times between two observation set points. For example, Figure VI-1shows the linear interpolation between points in the observation set of DAG configurations for DAGs with CCR of 0.3 and parallelism of 0.9. The different lines represent the optimal turn-around times achieved by the different scheduling heuristics. From Figure VI-1 and many other plots of varying CCR and parallelism values, we observe that for smaller DAGs, MCP perform better and for bigger DAGs, FCA perform better. In all the plots, we define the threshold as the intersection between the MCP line and the FCA line. In Figure VI-1, the DAG size threshold is 1700. For CCR of 0.3 and parallelism of 0.9, at DAG size of around 1700, we would expect the MCP and FCA would have similar performance and when DAGs are much bigger than 1700, we would expect that FCA would have the best performance.

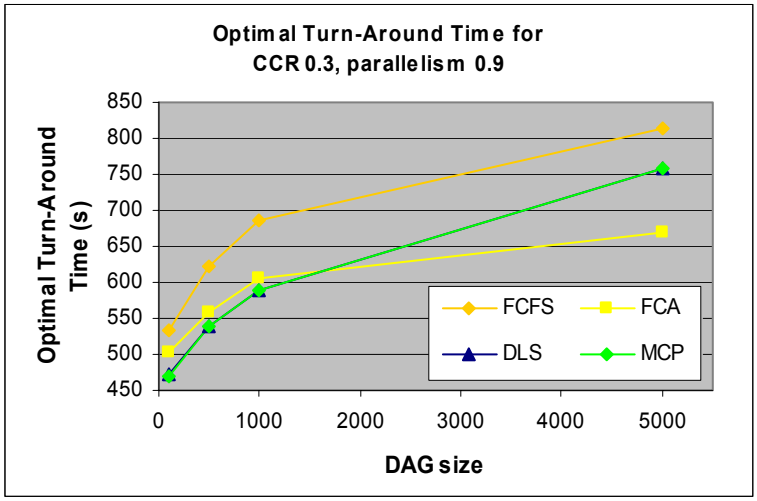

Figure VI-1: Optimal application turn-around time for different heuristics as function of DAG size

We can finish the construction of our prediction model by deriving the thresholds of all (CCR, parallelism) pairs. We can then interpolate between the different (CCR, parallelism) pairs for any arbitrary DAG that falls outside the observation set of DAG configurations. Figure VI-2 shows the surface plot of the threshold values of all (CCR, parallelism) pairs. Any input DAG above the surface should use FCA to optimize application performance and any DAG below the surface should use MCP as the scheduling heuristic to optimize performance. Because our

observation set of DAG configurations include only DAGs with sizes up to 10,000, the threshold values above 10,000 in Figure VI-2 is the intersection of extrapolation of the MCP and FCA lines using size 5000 and size 10,000 points.

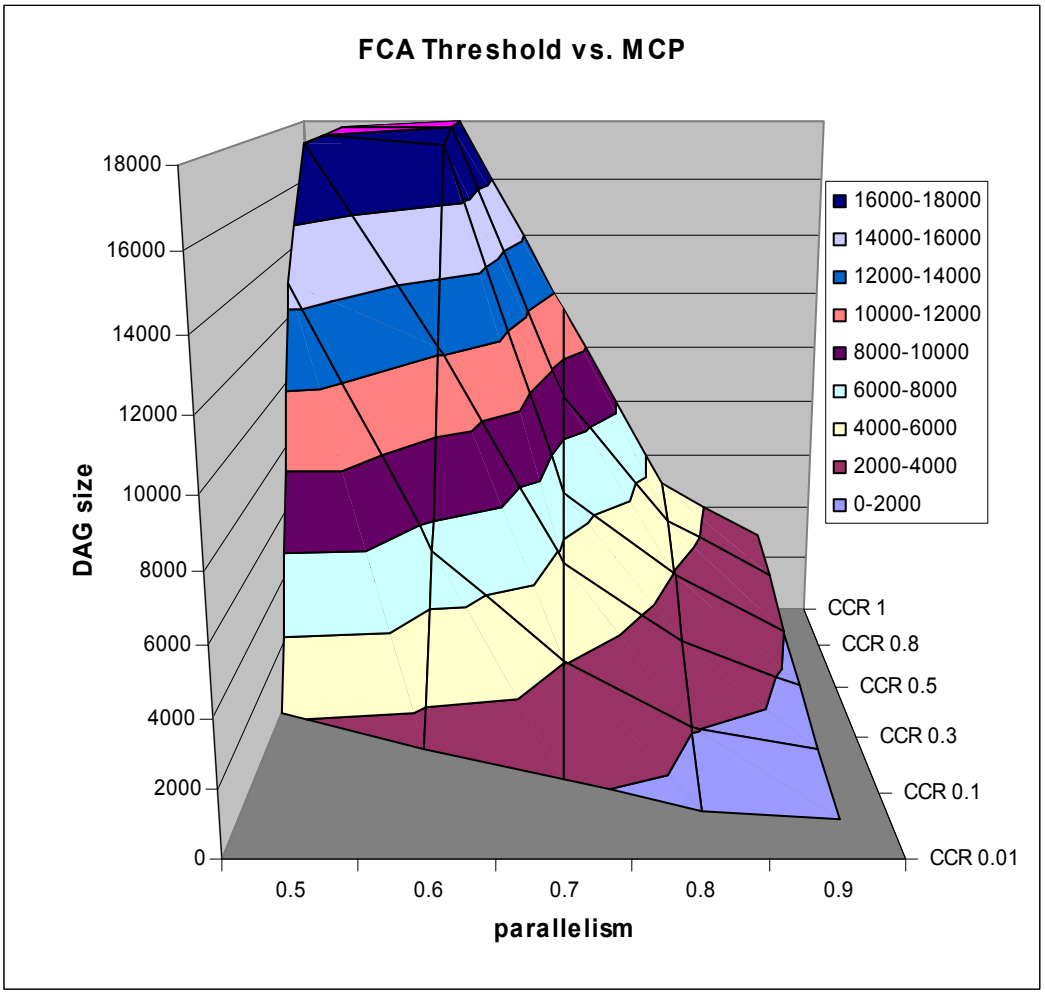

Figure VI-2: Surface plot for deciding when to use MCP and when to use FCA

VI.4 Model Validation

Our main goal for the heuristic predictor is for it to complete our broad vision for our automatic resource specification predictor, as seen in Figure VI-3. The heuristic predictor takes as input a DAG and an optional utility function and outputs the best scheduling heuristic for the size predictor (discussed in Chapter V). Now that we have formulated the heuristic predictor in

Section VI.3, we need to validate the accuracy of the model. Further, we validate the usefulness of the combination of the heuristic prediction model and the size prediction model as a whole.

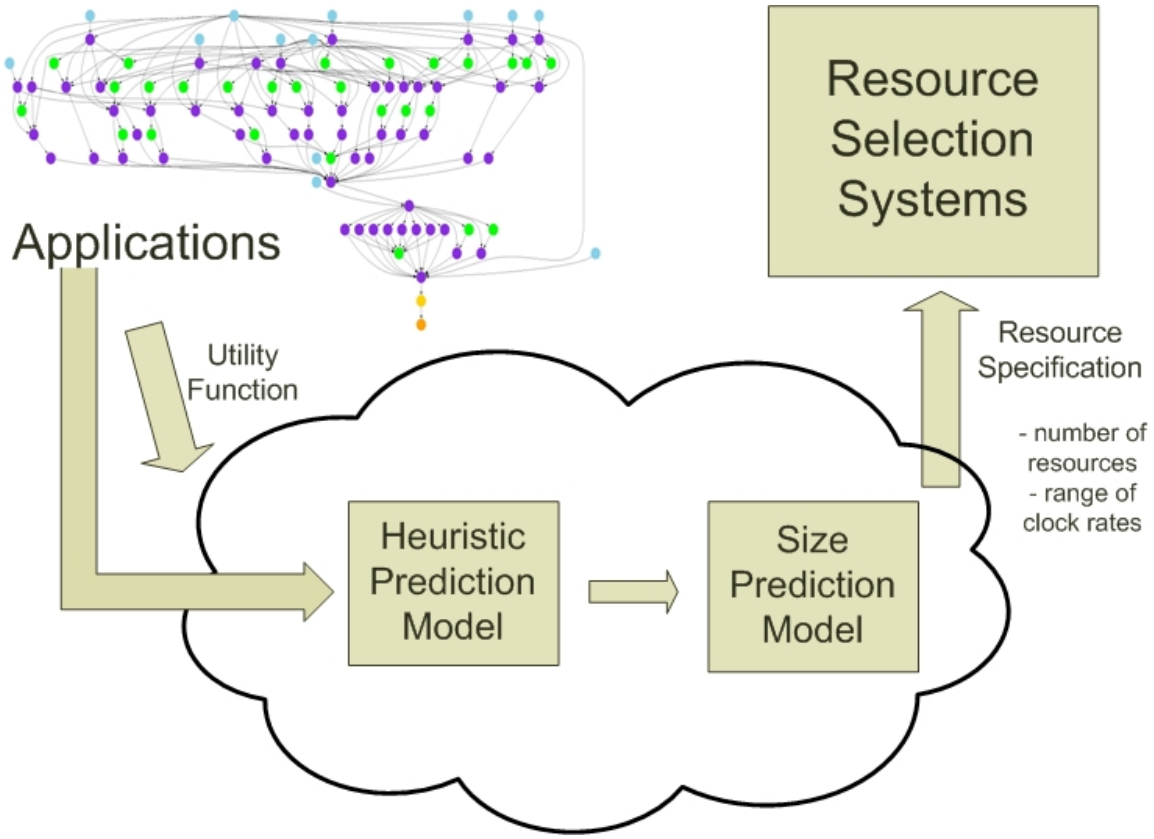

Figure VI-3: Overview of Resource Specification Predictor

Our strategy for validating the prediction model is as follows:

1. We choose a set of 16 points on the surface of the plot in Figure VI-2. For each of the points on the surface, we choose 3 points above and three points below the surface 1%, 10%, and 20% away from the surface for a total of 96 points. Because our size prediction model is limited to predicting RC sizes for DAGs of at most 10,000 tasks, we are limited to choosing points that are below the 10,000 DAG size limit so we can validate the overall application performance of both the heuristic prediction model and the size prediction model. We choose CCR and parallelism values evenly spaced apart to cover all portions of the surface plot below the 10,000 size limit. Table VI-4 lists all the points we choose from the surface as well as the 96 points we choose for validation. These

points represent the DAG sizes with the corresponding parallelism and CCR values and a regularity value of 0.5. For points above the surface, the heuristic predictor predicts using FCA as the best scheduling heuristics and for points below the surface, the heuristics predictor predicts using MCP as the best scheduling heuristic.

Table VI-4: Points chosen to validate the heuristic prediction model

parallelism	CCR	Surface DAG size	Above surface			Below surface		
			1%	10%	20%	1%	10%	20%
0.8	0.01	1389	1403	1528	1667	1375	1250	1111
0.8	0.1	2052	2073	2257	2462	2031	1847	1642
0.8	0.3	3157	3189	3473	3788	3125	2841	2526
0.8	0.5	3964	4004	4360	4757	3924	3568	3171
0.8	0.8	4626	4672	5089	5551	4580	4163	3701
0.8	1.0	4890	4939	5379	5868	4841	4401	3912
0.9	0.01	1156	1168	1272	1387	1144	1040	925
0.9	0.1	1333	1346	1466	1600	1320	1200	1066
0.9	0.3	1700	1717	1870	2040	1683	1530	1360
0.9	0.5	2006	2026	2207	2407	1986	1805	1605
0.9	0.8	2606	2632	2867	3127	2580	2345	2085
0.9	1.0	2864	2893	3150	3437	2835	2578	2291
0.7	0.01	2300	2323	2530	2760	2277	2070	1840
0.6	0.01	3140	3171	3454	3768	3109	2826	2512
0.5	0.01	4172	4214	4589	5006	4130	3755	3338
0.7	0.1	4055	4096	4461	4866	4014	3650	3244

2. After we choose the set of points to test, we input the different DAG configurations into the RC size prediction model to obtain the best RC size. We use resource heterogeneity of 0.3 for all of these points. Each of these 96 points will have different RC sizes.
3. We run FCA for all the DAG configurations above the surface (from Table VI-4) and MCP on all the DAG configurations below the surface (from Table VI-4) on the RC size returned by the size predictor. From this we obtain the application turn-around time for the DAG configurations using both of our prediction models.
4. Using a semi-brute force method, for each of the 96 DAG configurations, we determine the best turn-around time for *all* the scheduling heuristics in our experiments.

Our results are obtained by first running the heuristic prediction model and then running the RC size prediction model and can fall into one of four categories, summarized by Table VI-5.

We define the heuristic model as having predicted accurately when the predicted heuristic (in conjunction with using the best RC size determined by the semi-brute force method) achieves best performance; when another heuristic (in conjunction with using the best RC size determined by semi-brute force method) achieves the best performance, then we define the heuristic model as having predicted incorrectly.

Since we have a RC size prediction model, we can use it instead of the semi-brute force approach to predict the best RC size. When the heuristic prediction model accurately predicts the right heuristic, the RC size prediction model can have one of two possible outcomes:

1. The RC size prediction model accurately predicts the RC size, therefore enabling the predicted heuristic to achieve the best performance on the predicted RC size.
2. The RC size prediction model predicts the RC size inaccurately, thereby allowing another heuristic to achieve better performance (also with a predicted RC size for the other heuristic).

Given that we use the RC size prediction model in conjunction with the heuristic prediction model, when the heuristic prediction model predicts inaccurately, one of the following two outcomes is possible:

1. The predicted heuristic achieves best performance using the predicted RC size. This is only possible because the RC size prediction model predicted inaccurately the best RC size for the best possible heuristic.
2. Another heuristic achieves better performance using the RC size predicted by the RC prediction model.

Table VI-5: Possible outcome of validation results

Heuristic Model predicts accurately AND Coupled with Size Prediction Model leads to best performance	Heuristic Model predicts inaccurately But Coupled with Size Model leads to best performance
Heuristic Model predicts accurately But another heuristic performed better coupled with the Size Prediction Model	Heuristic Model predicts inaccurately AND another heuristic performed better coupled with the Size Prediction Model

Figure VI-4 shows the breakdown of the four scenarios listed in Table VI-5. We see that for 69.57% of the DAG configurations, the heuristic predictor was accurate in predicting the best heuristic for the given DAG configuration. For 36.96% of the DAG configurations, even though the heuristic prediction model accurately predicted the heuristic that can achieve the best performance, errors from the RC size prediction model allowed another heuristic to achieve better performance. For 55.43% of the DAG configurations, the combination of both the heuristic prediction model and the RC size prediction model enabled the application to achieve the best turn-around time. Only for 7.61% of the DAG configurations did the heuristic model predict the wrong heuristic and the combination of the two prediction models do not yield the best application performance.

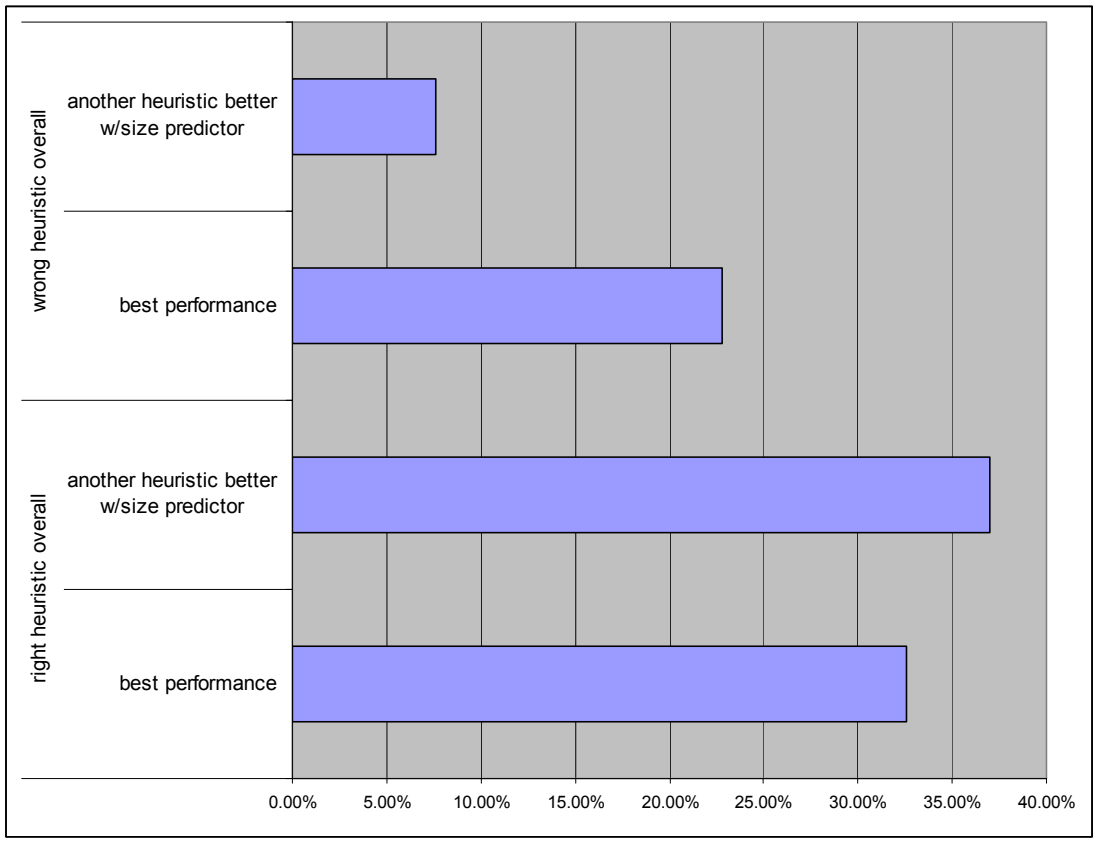

Figure VI-4: Breakdown of validation results

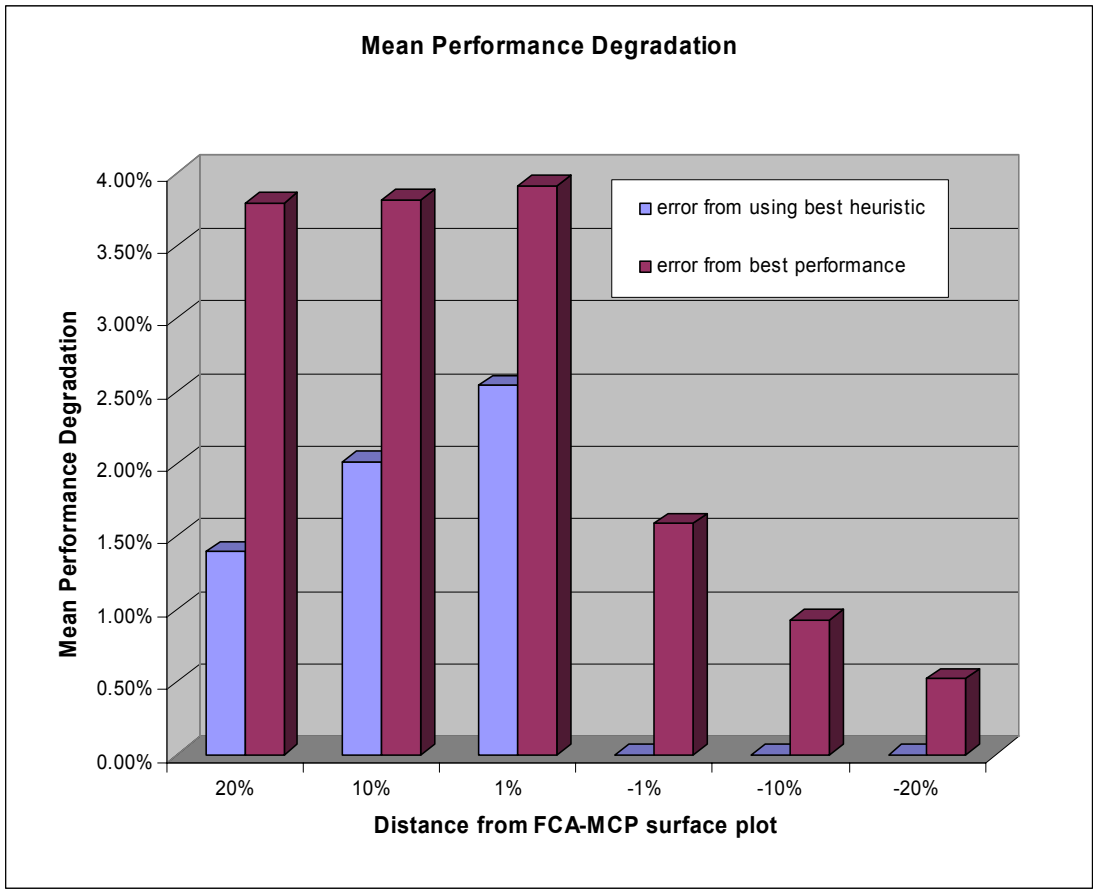

Figure VI-5: Mean performance degradation from best possible application turn-around time

While it is important to see the breakdown in percentage of DAG configurations for which the combined prediction models succeeds or not, it is equally important to examine the effects of the errors resulting from mispredictions of the models. Figure VI-5 shows the mean performance degradation from the best possible application turn-around time. The lighter bars represent the error in predicting the best heuristic. This represents the amount of performance lost because the prediction model chose the wrong heuristic. We see that the prediction model correctly identifies MCP as the best scheduling heuristic for all DAG configurations below the surface of the FCA-MCP plot. The darker bar represents the mean combined error of both of the prediction models. We see that for all of the DAG configurations we tested, using both of our prediction models achieved application turn-around time that is less than 4% from the optimal

turn-around time, approximately half of the performance degradation can be attributed to each of the prediction models.

VI.5 Summary

Application performance depends on both physical resource characteristics as well as the scheduling heuristic. In Chapter V, we constructed an empirical prediction model to predict the best RC size. We also examined the robustness of the RC size predictor for different resource heterogeneity and different scheduling heuristic. However, either the application or the user would still need to specify the scheduling heuristic. In this chapter, we construct a prediction model to suggest to the application/user the best scheduling heuristic given the application. We take an empirical approach similar to Chapter V by using an observation set of DAG configurations spanning different relevant DAG characteristics. By running the four scheduling heuristics on the observation set of DAG configurations, we can compare the best possible application performance from each of the four scheduling heuristics and delineate regions in the DAG configuration space where one heuristic work better than other heuristics.

From our results, we concluded that we can remove FCFS from the heuristic prediction model because in all cases FCA can perform better or no worse than FCFS. We saw that using FCA optimizes application performance only for bigger DAGs with lower CCR and/or higher parallelism. For smaller DAGs or DAGs with higher CCR and/or lower parallelism, using MCP or DLS led to best application performance.

Constructing our heuristic prediction model consists of deciding when to use MCP/DLS and when to use FCA. Based on the trends we observed, we choose to use only MCP over DLS because for 85% of the DAG configurations with sizes 5000 or 10,000, MCP performed better. Even when DLS outperforms MCP, the overall turn-around time achieved by MCP is still within 5% of DLS. We finish the construction of our prediction model by deciding when to use MCP

and when to use FCA. We use linear interpolation to predict the application turn-around times between two observation set points. We observe MCP performing better for smaller DAGs and FCA performing better for bigger DAGs. From the linear interpolation, we can determine the threshold where MCP ceases to have better performance than FCA as the DAG size is increased. We finish the construction of our model by deriving the thresholds of all (CCR, parallelism) pairs.

We validated both the heuristic prediction model and the size prediction model from Chapter V. We see that for 69.57% of the DAG configurations, the heuristic predictor was accurate in predicting the best heuristic for the given DAG configuration. For 36.96% of the DAG configurations, even though the heuristic prediction model accurately predicted the heuristic that can achieve the best performance, errors from the RC size prediction model allowed another heuristic to achieve better performance. For 55.43% of the DAG configurations, the combination of both the heuristic prediction model and the RC size prediction model enabled the application to achieve the best turn-around time. Only for 7.61% of the DAG configurations did the heuristic model predict the wrong heuristic and the combination of the two prediction models do not yield the best application performance. We see that for all of the DAG configurations we tested, on average, using both of our prediction models achieved application turn-around time that is less than 4% from the optimal turn-around time, approximately half of the performance degradation (as compared to the optimal turn-around time) can be attributed to each of the prediction models.

VII

RESOURCE SPECIFICATION PREDICTION IN PRACTICE

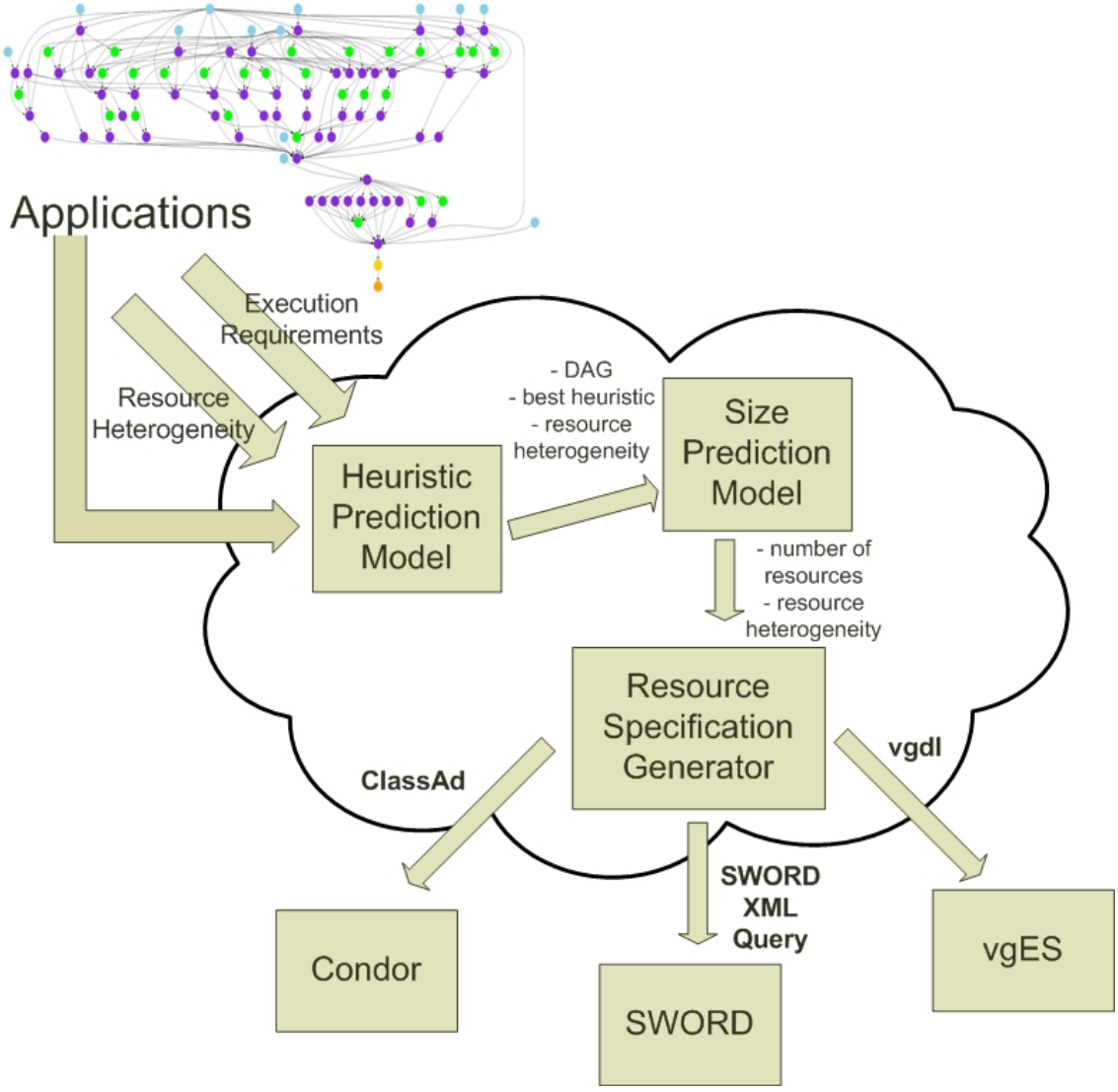

Figure VII-1: Generating resource specifications from heuristic prediction and size prediction models

The overarching goal of the work in this dissertation is to bridge the gap between workflow applications and resource selection systems. We address the question of how an application (or application user) can request the best set of resources with the notion of a resource specification predictor. Our resource specification predictor is composed of a resource collection size predictor, which is developed in Chapter V, and a heuristic predictor, which is developed in Chapter VI. In this chapter, we address how to use the resource specification predictor in practice by adding a component to generate resource specifications.

Figure VII-1 depicts our vision for generating resource specifications for three resource selection systems. Workflow applications characteristics, along with optional designation of resource heterogeneity and execution requirements are sent to the heuristic prediction model. The optional designation of resource heterogeneity allows the application to specify the level of heterogeneity the application can tolerate. By default, the resource specification predictor assumes applications can tolerate the highest level of resource heterogeneity. The optional execution requirements may include operating system requirements or memory requirements necessary to execute the application. The optional execution requirements are not used by the heuristic prediction model or by the size prediction model but passed to the resource specification generator.

The heuristic model determines the best heuristic for the given DAG and the given resource heterogeneity and passes its inputs along with the best heuristic to the size prediction model. The size prediction model determines the best RC size for the input DAG, the input resource heterogeneity, and the input scheduling heuristic. It outputs the best RC size, along with the resource heterogeneity, to the resource specification generator. The resource specification generator generates a generic resource specification based on the RC size, the resource heterogeneity, as well as the reference mean clock rate of the resource collection and the assumptions about the network connectivity. The resource specification generator maps the

generic resource specifications to the specific resource specifications of the target resource selection system.

In Section VII.1, we describe the general strategy for a mapping between the output of our resource specification predictor and a generic resource specification language. In Sections VII.2-VII.4, we show the mapping between the generic resource specification language described in Section VII.1 and actual resource specification languages employed by Condor (Section VII.2), SWORD (Section VII.3), and the Virtual Grid Execution System (vgES) (Section VII.4). In Section VII.5, we discuss how our resource specification generator can generate alternative resource specifications when the initial best resource specifications cannot be fulfilled by the resource selection system. In Section VII.6, we summarize how our resource specification predictor can be used in practice.

VII.1 Resource Specification Generator

The resource specification generator is composed of two parts: 1) mapping inputs into generic resource specifications and 2) mapping the generic resource specification into specific resource specifications for any resource selection system. We describe the latter in Sections VII.2-VII.4. In this section, we describe the former by first describing the inputs to the resource specification generator, then describing our generic resource specification, followed by describing the mapping from the input to the generic resource specification.

The resource specification generator takes its input from the size prediction model. The input consists of two parts: the RC size, and the resource heterogeneity. The default resource heterogeneity is coefficient of variance of 0.3. Before generating specific resource specifications, we consider a generic resource specification describing the ideal set of resources. A resource specification is composed of two major components: compute nodes and network connectivity among the compute nodes. Compute nodes can be either described individually or as a set sharing

similar characteristics. Our resource specification generator is designed for arbitrary applications; thus we do not generate distinct specifications for individual compute nodes. Instead, we focus on the resource collection as a whole and a single specification for the desired characteristics in compute nodes. To summarize, a generic resource specification for compute resources is composed of the following:

1. Size of the resource collection. This information is provided by the size predictor.
2. CPU requirements. We generate CPU requirements from our reference CPU and the resource heterogeneity. For the default resource heterogeneity, we translate the CPU speed requirement to be a range of 30% on both sides of 3.5GHz, for a range between 2.45GHz and 4.55GHz. We can further simplify the requirement by removing the upper bound of 4.55GHz and simply use 2.45GHz as the minimum CPU requirement. This bound could be further lowered when the resource selection system cannot select a set of resources with this bound. See Section VII.5 for a full discussion.
3. CPU load. Since we make our assumption of dedicated resources, we require the load to be 0.
4. All other resource characteristic (such as memory, architecture, processor type, etc.). We can use the execution requirements passed in by the application. If no requirements are passed in, we do not consider other resource characteristics. If a resource selection system requires other resource characteristics, we arbitrarily assign the required values, while maintaining the same types (e.g., processor architecture) across all resources.

Because no network connectivity information is provided to the resource specification generator, we depend on our assumptions regarding the network to generate the necessary resource specifications. Recall that in Chapter V, we assumed a resource environment with high bandwidth (justified by existing deployment and ongoing researchfor increasing and deploying higher bandwidth). Such a high network connectivity environment is likely to exist across

research institutions where our target scientific workflows are likely to be deployed. In our experiments, we use a reference bandwidth of 10Gbps, but we can change this depending on the CCR of the DAG. Since our observation set in construction of the size prediction model has a range of 100 for CCR, the reference bandwidth can be changed by factors of up to 100; thus, we can adjust the reference bandwidth down to 100Mbps if necessary. If we know a priori the bandwidth conditions of the resource environment, we can adjust the network connectivity specifications accordingly. Assuming high bandwidths are available, we use our reference bandwidth of 10Gbps. Since latency is negligible, we use an arbitrary value of 100ms. We also require that the network is fully connected as we do not consider partially (or fully) fragmented networks in our models. To summarize, a generic resource specification for network connectivity is composed of the following:

1. Bandwidth. By default, we set this to 10Gbps, but this is a tunable parameter that can be adjusted according to the resource environment.
2. Latency. By default, we set this to 100ms. For scientific workflows, this value is negligible; however, some resource selection systems require specific values for latencies, so we use a default value.
3. Connectivity among compute nodes. By default, we require a fully connected network. We have this specification for any resource selection specification requiring detailed connection information between compute nodes.

VII.1.1 Example Application: Montage

To demonstrate the generation of resource specifications from applications, we use the Montage application. Recall from IV.2.1 that Montage is an astronomy application that creates a mosaic image of a portion of the sky on demand. Figure VII-2 shows a small Montage workflow. All Montage workflows are similarly structured and are composed of seven levels. The size of the

Montage DAG corresponds to the size of the mosaic. Table VII-1 shows the runtime and the number of tasks at various levels for a Montage workflow with 4469 tasks.

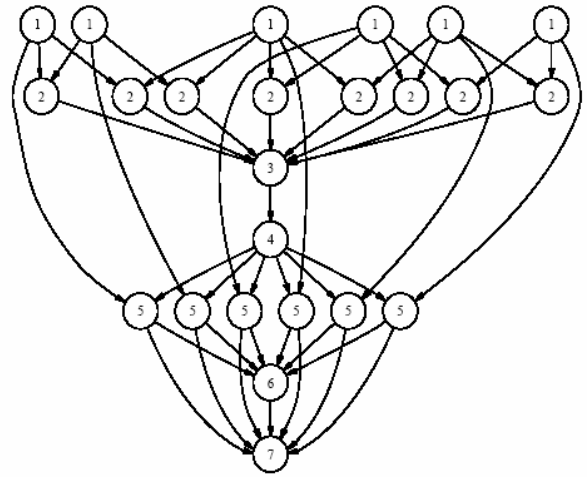

Figure VII-2: A small Montage workflow

Table VII-1: Number of tasks at various levels of a Montage workflow

Level	Task name	Number of Tasks
1	mProject	892
2	mDiffFit	2633
3	mConcatFit	1
4	mBgModel	1
5	mBackground	892
6	mImgtbl	25
7	mAdd	25

When this 4469-task Montage DAG is passed to the resource specification predictor, along with specifying that the the MCP scheduling heuristic is used and without specifying any resource heterogeneity, the RC size predictor uses the default resource heterogeneity of 0.3 and predicts a RC size of 599. This information is passed to the resource specification generator, which generates the following generic resource specification:

Resources:

1. RC size: 599
2. CPU > 2.45

3. load = 0

Network Connectivity

1. bandwidth = 10Gbps
2. latency = 100ms
3. connectivity = mesh

The above generic resource specification is then translated into specific resource specification languages. We consider three specifications languages in the following sections.

VII.2 Condor

Recall from II.4.2, that Condor [14] is a high throughput computing system that focuses on workload management for compute-intensive jobs. Applications or users can request resources by specifying their requests in a high level language called ClassAds.

VII.2.1 Converting to Condor ClassAds

ClassAds require users to specify a list of bilateral requirements called *ports*. A port attribute defines the number of and characteristics of the matching candidate ClassAds for its associated ClassAd to be satisfied. Each port defines *Labels* that name the candidate bound to that port. To validly match a gang of ClassAds, all their ports must be bound with compatible ports (i.e., with no conflict between them) of some other ClassAds in a group.

The strength of ports representing separate resources lies in the flexibility and expressiveness of the ClassAd system in describing distinct resources for each port, each with possibly different requirements. This system works well when a handful of distinct resources are required; however, when the number of resources desired reaches hundreds or thousands, it becomes cumbersome to specify a port for each machine, given that many machines can share the

same exact specifications. Another potential weakness in the ClassAd system is the lack of ability to specify relationships between compute nodes. One rationale is that the target applications are compute intensive; therefore network connectivity is not a big issue. A second rationale is the assumption that all of the resources within a Condor pool are within close proximity and most likely within some acceptable limits for bandwidth and latency.

The strategy for converting from the generic resource specification generated by our resource specification generator is straightforward. Because of the lack of network specifications, we need to focus solely on the resource component. In a ClassAd, we create a number of ports equal to the RC size predicted by the size predictor. All ports have identical syntax. For each port, we need to specify a label of "cpu" to correspond to the desire for a CPU. We indicate preference for faster CPUs by using the keyword "Rank" and indicate constraint on the speed of the CPU by the keyword "Constraint". Figure VII-3 shows the ClassAd generated by the resource specification generator to run the 4469-task Montage from Section VII.1.1. The resource specification generator specifies a single architecture and operating system for all compute nodes to eliminate discrepancies in the task execution times from their performance models due to different architecture or operating system.

```
[ Type    = “Job”;
  Owner = “Montage user”;
  QDate = ‘ Mon May 30 12:24:56 2007 (PST) -08:00’;
  Ports = {
     [  // request first machine
        Label        = cpu;
        Rank         = cpu.Speed;
        Constraint  = cpu.Type ==”Machine” &&
                      cpu.Arch == “OPTERON”  &&
                      cpu.OpSys == “LINUX” &&
                      cpu.Speed > “2.45GHz”
     ],
     [  // request second machine
        Label        = cpu;
        Rank         = cpu.Speed;
        Constraint  = cpu.Type ==”Machine” &&
                      cpu.Arch == “OPTERON”  &&
                      cpu.OpSys == “LINUX”
                      cpu.Speed > “2.45GHz”
     ]
  …
     [  // request 599th machine
        Label        = cpu;
        Rank         = cpu.Speed;
        Constraint  = cpu.Type ==”Machine” &&
                      cpu.Arch == “OPTERON”  &&
                      cpu.OpSys == “LINUX”
                      cpu.Speed > “2.45GHz”
     ]
   }
]
```

Figure VII-3: ClassAd generated by the resource specification generator to run the Montage DAG

VII.3 SWORD

Recall from II.4.3, that SWORD [15, 20] is a scalable resource discovery service for wide-area distributed systems. The focus of SWORD is the set of resources on which users can deploy services (as opposed to executing a short-lived application). Thus, SWORD runs on Internet-scale infrastructure machines (such as the nodes of the PlanetLab [50] testbed). SWORD takes two forms of input: Condor ClassAds and the SWORD query language. We focus on generating queries that use the SWORD query language. The generation of Condor ClassAds is addressed in the previous section.

VII.3.1 Converting to SWORD XML

To be compatible with SWORD, our resource specification generator can generate the Condor ClassAd described in Section VII.2.1; however, that specification lacks the network connectivity information and we can express this information in a SWORD XML. Since we cannot request an arbitrary number of groups without any further insight into the input application, our strategy is to request one group, which ensures best performance regardless of whether the application can be executed over multiple groups. Conveniently, SWORD XML allows the specification of the number of machines in the group. Using the 4469-task Montage from Section VII.1.1, the resource specification generator uses 599 for this value. We specify the required CPU speed by denoting the low and high end of the speed in brackets (e.g., [2.45, MAX]). The value "MAX" is used to denote the maximum speed in the system. Based our on assumption of dedicated machines, we require the load to be less than 0.02, in case of some routine processes running on any particular machine that is using 1% of the CPU. Similar to the required CPU speed, we denote the required all pairs latency and bandwidth to match our assumptions about the resource environment. Because the resource consumption constraints the user places on evaluating the query (i.e., the first section of the XML query) is optional, the resource specification generator leaves it out of the SWORD XML query. Also, because we require only one group, we leave out the third section of the SWORD XML query, which describes the inter-group constraints. Figure VII-4 shows the SWORD XML query generated by our resource specification generator. We show the modified XML format for clarity.

```
Group RC
   NumMachines 599
   Required CPU_speed [2.45, MAX]
   Required Load [0.0, 0.02]
   Preferred Load [0.0, 0.0] penalty 100.0
   Required AllPairs Latency [0.0, 100.0] (ms)
   Required AllPairs Bandwidth [10.0, MAX]
(Gb/s)
```

Figure VII-4: XML query generated by the resource specification generator to run the Montage DAG

VII.4 The Virtual Grid Execution System

Recall from II.4.1 that the Virtual Grid Execution System (vgES) [16, 17] was designed and prototyped as part of the Virtual Grid Application Development Software Project (VGrADS) [29]. The main contribution of VGrADS is the notion of a *Virtual Grid* (VG), a high-level, hierarchical abstraction of the resource collection that is needed and used by an application. Resource selection plays a major role in determining the architecture of vgES because of the end goal of producing a virtual grid based on the user written vgDL.

VII.4.1 Converting to vgDL

The input to vgES is a resource specification written in a high-level resource description language, the Virtual Grid Description Language (vgDL). The salient point from vgDL is the resource aggregate TightBag, which is a collection of heterogeneous nodes with good connectivity. The strategy for the resource specification generator in generating vgDLs is straightforward. First we verify that the default latency value for the TightBag threshold is set to 100ms. This ensures the network connectivity portion of the generic resource specification is satisfied. Given this, requesting a resource collection can be done by simply requesting a TightBag. Using the 4469-task Montage from Section VII.1.1, the resource specification

generator generates a vgDL for a TightBag of 599 nodes, where each node has the requirement of CPU speed greater than 2.45GHz and a load of 0.

```
VG =
{
  TightBagOf(nodes) [599:599]
  {
    Nodes = [(Clock > 2400) && (Load == 0)]
  }
}
```

Figure VII-5: vgDL generated by the resource specification generator to run the Montage DAG

VII.5 Alternative Resource Specification Generation

When our resource specification generator is used in practice, one interesting question arises: What if the resource selection system cannot fulfill the resources specified by the resource specification? In this section, we modify our resource specification generator to offer alternative resource specifications when the initial best resource specification request cannot be fulfilled.

When a resource request cannot be fulfilled, one of two courses of action is possible: 1) request fewer resources or 2) request slower resources. We do not consider requesting faster resources because presumably, the original request already included requesting the fastest resources possible. It is clear that generating alternative resource specifications by strictly requesting fewer resources or strictly requesting slower resources will not yield the optimal application performance. The problem of generating alternative resource specifications reduces to one of determining when to request fewer resources and when to request slower resources.

VII.5.1 Experimental Setup

Our approach to determining when to request fewer resources and when to request slower resources (similar to the approach in V.2) is to take an observation set of DAG configurations and

observe how the application turn-around time varies as a function of the computational clock rate and a function of RC size. From the application turn-around times generated by the observation set of DAG configurations, we hope to determine when requesting fewer resources is preferable to requesting slower resources (and vice versa). In general, we compose our observation set of DAG configurations by choosing combinations of DAG characteristics that would result in a bigger optimal RC size. Our reasoning is that bigger RC sizes predicted by our size prediction model would be more likely lead to resource specifications that cannot be fulfilled by a resource selection system. Table VII-2 summarizes the experimental setup values for the different DAG characteristics as well as resource heterogeneity and the different computational clock rates. We use the reference scheduling heuristic MCP for this set of experiments.

Table VII-2: Experimental setup values for determining alternative resource specifications

DAG characteristics	**Values**
Size	1000, 5000
CCR	0.01, 0.1, 0.5, 1.0
Parallelism (α)	0.7, 0.8, 0.9
Regularity (β)	0.5
Resource Heterogeneity	0.0, 0.3
Computational clock rate (GHz)	2.0, 2.5, 3.0, 3.5

We choose DAG sizes of 1000 and 5000 because bigger DAGs require bigger optimal RC sizes, thus increasing the likelihood of requiring alternative resource specifications. For CCR values, we sample a subset of our original observation set of CCR values. For the parallelism value, we choose values of 0.7, 0.8, and 0.9 as DAGs with higher parallelism requires bigger RC sizes. For regularity values, we choose only one value of 0.5 based on our experiences in Chapter V that regularity values does not affect the choice of RC sizes as much as other DAG characteristics. For the resource heterogeneity, we choose the two extremes of 0.0 and 0.3 as we are interested to see how heterogeneous resources would affect the choice of alternative resource specifications. Finally, we choose three other computational clock rates of 2.0GHz, 2.5GHz, and

3.0GHz in addition to our reference computational clock rate of 3.5GHz because they represent regular discrete intervals in computational clock rates.

VII.5.2 Experimental Results

We observed similar plots for all our experiments as shown in Figure VII-6. Figure VII-6 shows the application turn-around time as functions of the computational clock rates and decreasing RC sizes for DAGs with size of 5000, CCR of 0.01, parallelism of 0.8, and resource heterogeneity of 0.

From Figure VII-6, we observed that the application turn-around time for a band of smaller RC sizes for a faster computational clock rate can be achieved by a band of larger RC sizes for a slower computational clock rate. For example, in Figure VII-6, an RC of 264-280 3.5GHz hosts has application turn-around times of 429.58s-442.75s. This is similar to the application turn-around time achieved by 384-512 3.0GHz hosts, which achieved application turn-around times of 432.97s-444.64s. If the resource selection system cannot find 264 hosts with 3.5GHz to compose the RC, then it would be possible to achieve similar performance by utilizing a larger number of smaller hosts. Note that the best application turn-around achievable by a RC of the faster 3.5GHz hosts cannot be achieved by any number of slower 3.0GHz hosts because the application turn-around time at the knee value for 3.5GHz RCs is better than the application turn-around time at the knee value for 3.0GHz RCs.

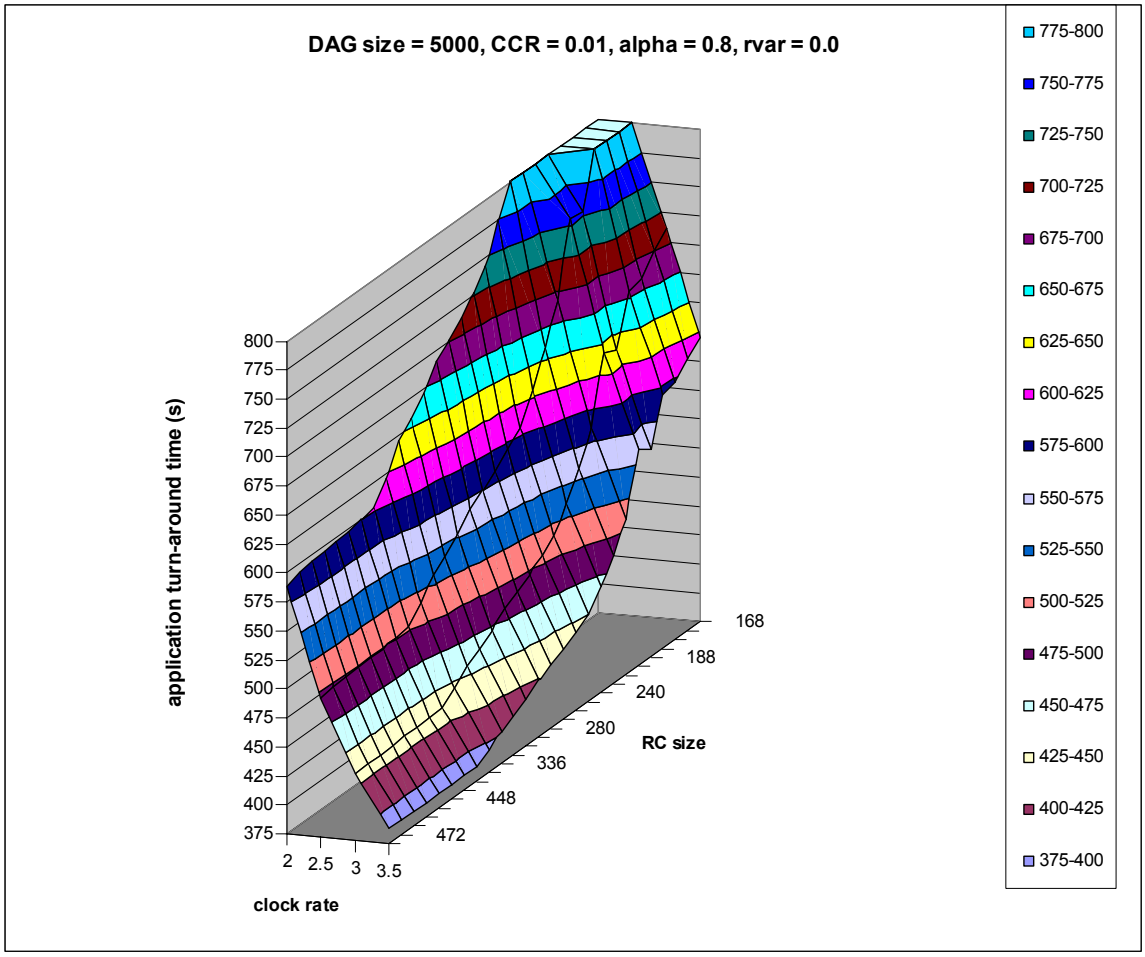

Figure VII-6: Application turn-around time as a function of computational clock rates and RC sizes

VII.5.3 Generating Alternative Resource Specifications

We make a key observation that for RCs with slower computational rates, the best RC size is typically the same of slightly bigger than for faster computational rates. From Figure VII-6 (and others like it), we also make the observation it is preferable to use fewer hosts than the optimal RC size at the 3.5GHz clock rate than to use the optimally sized RC at the 3.0GHz (or slower) clock rate. At a certain threshold, using hosts at the slower clock rate becomes preferable than using fewer hosts at the faster clock rate.

Our goal in Section VII.5 is to determine when to request fewer resources and when to request slower resources. Based on our observations, it seems clear that requesting fewer

resources at the faster computational clock rate is preferable until a certain threshold is reached, at which point using the best RC size at the slower computational clock rate would be equivalent.

Our next step is to determine the thresholds and determine whether any patterns exist for different DAG sizes, CCR values, and parallelism (α) values. Figure VII-7 shows the relative threshold value for moving from an RC composed of 3.5GHz hosts to an RC composed of 3.0GHz hosts. The different lines represent different parallelism (α) values. We observe that requesting RCs composed of 3.0GHz hosts can match the performance of using RCs composed of 3.5GHz hosts when the RC size is decreased to 70% of the best RC size as predicted by our size prediction model for DAGs with size 5000, parallelism value of 0.9, and CCR value of 0.01. As the CCR value is increased, the optimal RC size is decreased, thus increasing the importance of each host in the RC. Correspondingly, we observe from Figure VII-7 that the threshold moving from 3.5GHz hosts to 3.0GHz hosts decreases as the CCR is increased. We observe very similar threshold values going from 3.0GHz to 2.5GHz RCs and also similar threshold trends when we increase the resource heterogeneity to 0.3. When we change the DAG size, we also observe similar trends.

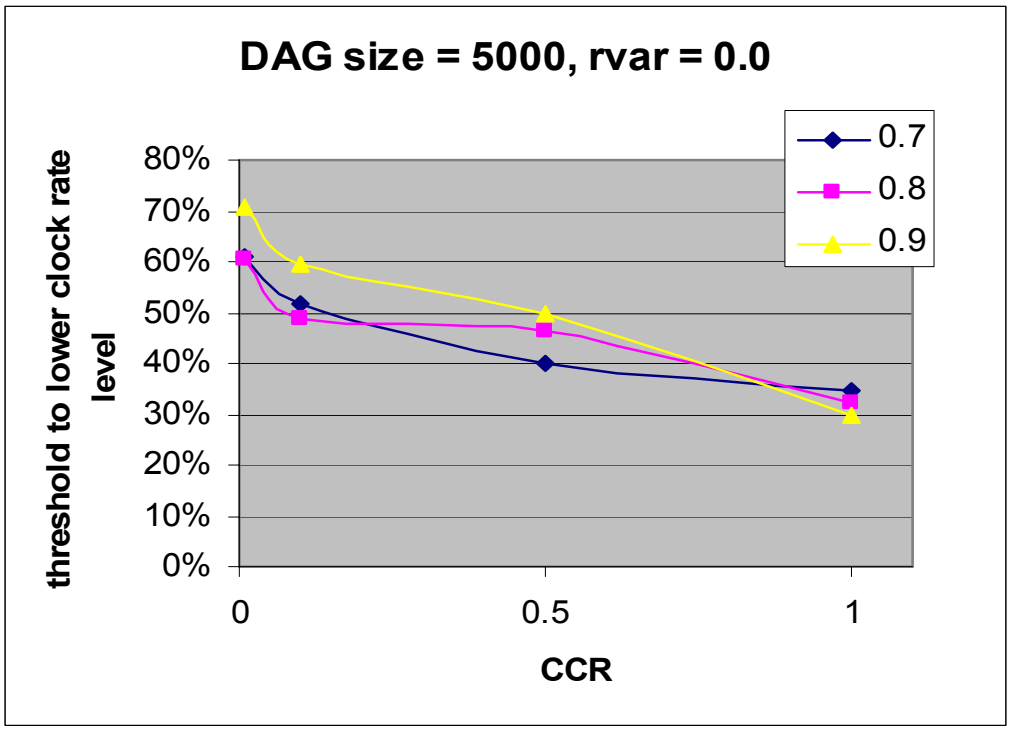

Figure VII-7: Relative RC size threshold for moving from 3.5GHz RCs to 3.0GHz RCs for DAGs with size 5000 and homogeneous resources

From Section V.2, we have an empirical model to predict the best RC size. Based on Figure VII-7 and others like it, we have an empirical model of generating alternative resource specifications. The heuristic we use is to look up the (DAG size, CCR) pair and determine the threshold when fewer resources should be used and when (more) slower resources should be used. For example, given a DAG of size 5000, CCR of 0.01 and parallelism of 0.9 and our size prediction model predicted an optimal size of 100, our resource specification generator would first generate a resource specification requesting between 70 and 100 (greater than or equal to 3.5GHz) hosts, while expressing a preference for more hosts. If that request cannot be fulfilled, then the next alternative request would be simply lowering the threshold for computational clock rate to 3.0GHz. The alternative request would still be requesting between 70 and 100 hosts, this time the constraint on clock rate would be 3.0GHz or greater, again expressing the preference for more hosts. In this fashion, we can lower the computational threshold in discrete steps, while

expressing a preference for RC size closer to the best RC size at each discrete computational clock rate.

VII.6 Summary

In this chapter, we showed how our resource predictor can be used in practice. First, we showed how the input workflow application can be processed by the heuristic predictor and the size predictor to generate the input to the resource specification generator. Then we showed how the resource generator can generate generic resource specifications based on the inputs it receives. At the last step, we show how the resource specification generator maps the generic resource specification into input languages of three different resource selection systems: Condor ClassAds, SWORD XML query, and vgES vgDL. We use a 4469-task Montage as an example application to show how each component of our resource specification predictor function to generate the respective inputs to the three resource selection systems.

With the resource specification generator in place, a natural question that arises is the question of what happens when the resource specification cannot be fulfilled by the resource selection system. The question reduces to one of when to request fewer resources and when to request slower resources. To answer this question, we compose a heuristic based on the application performance as a function of both the RC size and the computational clock rate. Our heuristic in effect traverses the resource size space and the computational clock rate space in a zigzag manner to ensure the best application performance given available resource constraints.

VIII

CONCLUSION

Over the recent years, the number of deployed clusters and the sizes of these clusters have grown due to dropping hardware costs and increasing availability of cluster management software. The emergence of large-scale distributed environments (LSDEs) is at the same time fueled by advances in hardware (computing clusters and networking routers and fibers) and by demands from the scientific community. With the growing need to share data and resources across geographically diverse regions, we have witnessed the establishment of more and more LSDEs as institutions are willing to share their resources in a collaborative effort.

The establishment of LSDEs brings new capabilities but also new challenges for executing applications. One important challenge is selecting the appropriate set of resources on which to execute different application components. This topic has been widely studied [9-19] and implemented in practice. The key observation that motivates the work in this dissertation is that there is a missing link between resource selection systems and applications. Resource selection systems are designed to return quickly and as closely a match to a resource specification whenever possible. However, most application users are interested in optimizing application performance. The missing link is that the application user has no sound basis for building the resource specification, that is the one that would return a set of resources that would in turn lead to best application performance. The question of what the "best" resource specification, that is the specification that will ultimately lead to best application performance as perceived by the user, is elusive at best.

In this dissertation we set out to prove the thesis statement that automatic resource specification generation is necessary and feasible to optimize LSDE application performance in a cost-effective manner. Further, we recognize that application performance is also dependent on the scheduling heuristic. Thus, we also want to provide guidance for the best scheduling heuristic in addition to the automatic resource specification generation.

VIII.1 Dissertation Contributions

The first question we addressed in this dissertation is how resource selection affects application scheduling. Resource selection is a part of scheduling, whether implicit or explicit. Using both a simplistic greedy scheduling heuristic and the more sophisticated MCP scheduling heuristic, we have shown that for both the Montage application and a spectrum of randomly generated DAGs, explicitly pre-selecting resources before running the scheduling heuristic on a subset of the resource universe always improved application performance, sometimes by several orders of magnitude. This held true when using either a naïve or a more sophisticated resource abstraction for resource selection.

We have shown that under most conditions, when one explicitly selects an appropriate resource collection, a simplistic scheduling heuristic can be employed to achieve similar to better performance than using a more sophisticated scheduling heuristic. A natural question was to ask how to compose such an appropriate resource collection. Our solution was an empirical model based on relevant application characteristics to predict the appropriate resource collection for any application. First, we constructed a model that predicts the best size for a homogeneous architecture resource collection. This model is based on an input scheduling heuristic and a utility function to tradeoff performance and cost. In extensive simulation over a wide range of workflow configurations, we showed that our prediction model consistently allowed workflows to achieve performance within a few percent of optimal. When applied to a real application, we showed that

our prediction model leads to almost optimal performance. Furthermore, when comparing the usage of our prediction model with current and typical practice of using the maximum application parallelism as the resource collection size, we found that using our model is far more cost effective while achieving better performance. We then performed a sensitivity analysis of our model by using different resource heterogeneity and different scheduling heuristics. We found that our model can be applied to different scheduling heuristics over resources with different heterogeneity. Finally, we investigated the effects of using different reference scheduler to computational clock rate ratios. Although our techniques for deriving the size prediction model can be employed to re-construct new prediction models based on a new scheduler clock rate or a different average computational clock rate, we also derived formulas to show how our predicted RC sizes can be modified to reflect arbitrary scheduler or computational clock rates.

The natural next step was to suggest to the application user the best scheduling heuristic in conjunction with the best resource specification to provide the optimal application turn-around time at the optimal cost. We constructed another empirical model to predict the best scheduling heuristic based on an input application and utility function. For all of the randomly generated workflow applications we tested, we found that using both of our prediction models achieved application turn-around time that is very close to the optimal turn-around time, approximately half of the performance degradation can be attributed to each of the prediction models.

Finally, we incorporated our two prediction models into an automatic resource specification generator. With the outputs of the two empirical models, our resource specification generator automatically generated resource specifications for three resource selection systems: the Virtual Grid Execution System, Condor, and SWORD. We analyzed the syntax and translated the outputs of our empirical models into each of the three resource selection languages. Our last contribution was an analysis of the scenarios under which our generated resource specifications does not return any resources from the resource selection systems. We answer the question of

when to request fewer resources and when to request slower resources when the original best resource specification cannot be fulfilled. We construct a heuristic that zigzags the resource size space and the computational clock rate space to ensure the best application performance given available resource constraints.

VIII.2 Future Directions

In this dissertation, we have made the following simplifying assumptions in constructing our resource specification generator:

i. Homogeneous network connectivity.

ii. Using dedicated resources.

iii. Application performance strictly as a function of the scheduling time and the application makespan.

iv. Available and accurate performance models for each of the tasks in the workflows.

v. DAGs of a certain size (i.e., between 100 and 10,000 tasks).

In future work we could explore ways for relaxing some of these assumptions.

VIII.2.1 Homogeneous Network Connectivity

One assumption we made is the homogeneous and high network connectivity among the compute resources. Although our range of CCR values in our prediction models allows the prediction model can maintain accuracy over lower connectivity networks, one interesting area to address is the heterogeneity in network connectivity. We expect that it is possible to construct similar empirical prediction models for highly heterogeneous networks. The main challenge would be to accurately model such a network. Currently, we are not aware of any de facto modeling of highly heterogeneous networks.

One potential interesting implication in delineating optimal network connectivity is the added constraints on the resource selection systems. For systems such as VGES where the network connectivity is treated as a binary value of high or low bandwidth (as implied by the latency values), it might not be possible to express quantitively the optimal network connectivity. For systems such as Condor where network connectivity is not expressible even in binary form, deriving the optimal network connectivity would not have added benefits. However, for systems such as SWORD where users (or the automatic resource specification generator) can specify the exact network requirements, potentially application performance can be improved, at the expense of added complexity stemming from trading off time spent searching for the optimal set of resources with the desired network connectivity and the benefits of having as close as possible to such a set of resources with optimal network connectivity.

VIII.2.2 Using Shared Resources

Another assumption that can be relaxed is the dedicated use of resources in executing applications. Due to increasing number of clusters along with increasing sizes for clusters, dedicated usage of compute resources should be prevalent; however, there remains compute resources which are shared. The best solution for optimizing application performance in a shared resource environment is to have a good application monitor and application tasks that can be migrated to other resources when any single resource becomes too overloaded. In the context of our prediction models, there exist two possibilities for the application. The first possibility is that the application has checkpointing or migrating capabilities for individual tasks. Either the application or the grid middleware needs to provide the appropriate monitoring software to detect when a particular task needs to be migrated due to overloading on a particular compute resource. In such a case, the adjustment to our empirical model in predicting the RC size would be proportional to how often any task would require migration. The adjusted best RC size would be

a size where everytime a task needs to migrate from a compute resource, another compute resource would be available and either lightly or not loaded. The second possibility is that the application does not have checkpointing or migrating capabilities. In such a case, the appropriate research direction is investigating a heuristic that can predict when and how often a particular resource may become overloaded. The strategy in optimizing application performance is to use this heuristic as part of the scheduling heuristic and avoid scheduling tasks on resources that are likely to become overloaded.

VIII.2.3 Other Factors in Determining Application Performance

In this dissertation, we have defined the application performance strictly as the sum of the scheduling time and the application makespan. However, other factors may be contributing to the overall application turn-around time. For example, the time required for staging each task on a compute host may not be negligible. Additionally, time spent initializing and authenticating permissions for one single task on different compute resources may not be negligible and may be in fact heterogeneous. An interesting research direction would be to model these non-negligible task staging/authentication times into the overall application turn-around time. The empirical models constructed in this dissertation can be adjusted to take into account these additional factors in the application performance.

VIII.2.4 Available and Accurate Performance Models

In this dissertation, we make the assumption that for any given application, an associated performance model is provided. For scientific workflows where each tasks has been executed many times, this assumption is realistic. However, for some other applications, performance models for each of the tasks may not be readily available. One interesting area of study is to predict task runtimes without the benefit of executing the task first.

Another assumption we make is the accuracy of the performance models. With any performance model, one can reasonably expect some variance from the predicted task runtime. One interesting research direction is to investigate the effects of the task runtime variance on the empirical prediction models in this dissertation.

VIII.2.5 Identifying Optimal DAG size

Another area of potentially interesting research is one of identifying the optimal DAG size relative to scheduling costs. The Montage workflow has tasks that are the results of unrolling some loops in a bigger "task". The loops in Montage are unrolled in an attempt to maximize parallelism. In this dissertation, we treat the results of the complete unrolling of loops as one DAG. Yet different sized DAGs are affected by scheduling costs in different ways. It may be possible to divide any bigger "task" of an application into an optimally sized DAG with regards to scheduling costs.

If it is possible for applications to be arbitrarily divisible into variable sized DAGs, it would be interesting to find an optimal size where parallelism can be maximized and the costs of running the scheduling algorithm can be minimized. One related scheduling heuristic is Dominant Sequence Clustering (DSC) Algorithm [83]. DSC works by merging ("cluster") tasks to optimize communication costs. However, the end result of running DSC is a series of clusters, each of which should be executed on one physical host. Instead of merging tasks, we are interested in expanding tasks such that parallelism can be maximized without incurring excessive scheduling costs.

REFERENCES

1. *National HPCC Software Exchange: http://www.nhse.org/index.htm*.

2. *Cluster Management Software Review: http://nhse.cs.rice.edu/NHSEreview/CMS/*.

3. *CNRI, Corporation for National Research Initiatives, Gigabit Initiative Final Report, December 1996, http://www.cnri.reston.va.us/gigafr/index.html*.

4. *TeraGrid. http://www.teragrid.org/*.

5. *Open Science Grid: http://www.opensciencegrid.org/*.

6. *Grid3: http://www.ivdgl.org/grid2003/*.

7. *Grid5000: https://www.grid5000.fr/*.

8. *Globus http://www.globus.org*.

9. Raman, R., M. Livny, and M. Solomon. *Matchmaking: Distributed Resource Management for High Throughput Computing*. in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. 1998. Chicago, IL.

10. Raman, R., M. Livny, and M. Solomon. *Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching*. in *Proceedings of the Twelfth IEEE International Symposium on High-Performance Distributed Computing*. 2003. Seattle, WA.

11. Coleman, N., et al., *Distributed Policy Management and Comprehension with Classified Advertisements*. 2003, University of Wisconsin-Madison Computer Sciences Technical Report #1481.

12. Liu, C., et al. *Design and Evaluation of a Resource Selection Framework for Grid Applications*. in *Proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing*. 2002. Edinburgh, Scotland.

13. Liu, C. and I. Foster, *A Constraint Language Approach to Grid Resource Selection*. 2003, University of Chicago Department of Computer Science Technical Report (TR-2003-07).

14. Litzkow, M., M. Livny, and M. Mutka. *Condor -- A Hunter of Idle Workstations*. in *Proceedings of the 8th International Conference of Distributed Computing Systems*. 1988.

15. Oppenheimer, D., et al., *Scalable Wide-Area Resource Discovery*. 2004, UC Berkeley Technical Report UCB//CSD-04-1334.

16. Chien, A., et al., *The Virtual Grid Description Language: vgDL*. 2004, UCSD Technical Report CS2005-0817.

17. Kee, Y.-S., et al. *Efficient Resource Description and High Quality Selection for Virtual Grids*. in *Proceedings of the IEEE Conference on Cluster Computing and the Grid*. 2005.

18. Dinda, P. and D. Lu. *Nondeterministic Queries in a Relational Grid Information Service*. in *Proceedings of the Supercomputing Conference*. 2003.

19. Lu, D., P. Dinda, and J. Skicewicz. *Scoped and Approximate Queries in a Relational Grid Information Service*. in *Proceedings of the 4th International Workshop on Grid Computing*. 2003.

20. Oppenheimer, D., et al. *Design and Implementation Tradeoffs for Wide-Area Resource Discovery*. in *Proceeding of the 14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*. 2005.

21. Dongarra, J., et al., eds. *The Sourcebook of Parallel Computing*. 2002, Morgan Kaufmann Publishers: San Francisco.

22. *http://rocks.npaci.edu/Rocks/*.

23. Papadopoulos, P., M. Katz, and G. Bruno. *NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters*. in *In the Proceedings of the IEEE International Conference on Cluster Computing*. 2001.

24. Berman, F., G. Fox, and T. Hey, eds. *Grid Computing: Making the Global Infrastructure a Reality*. 2002, Wiley.

25. *OptIPuter project. http://www.optiputer.net/*.

26. *http://www.griphyn.org*.

27. *Globus Monitoring and Discovery System (MDS). http://www-unix.globus.org/toolkit/mds/*.

28. *Internet Scout Project, http://scout.cs.wisc.edu/scout*.

29. *The Virtual Grid Application Development Software Project, http://vgrads.rice.edu*.

30. *GRAM: http://www.globus.org/toolkit/docs/3.2/gram/ws/*.

31. Wu, M.-Y. and D.D. Gajski, *Hypertool: A Programming Aid for Message-Passing Systems.* IEEE Transactions on Parallel and Distributed Systems, 1990. **1**(3): p. 330-343.

32. Kwok, Y.-K. and I. Ahmad, *Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Mutiprocessors.* IEEE Transactions on Parallel and Distributed Systems, 1996. **7**: p. 506-521.

33. El-Rewini, H. and T.G. Lewis, *Scheduling Parallel Program Tasks onto Arbitrary Target Machines.* Journal of Parallel and Distributed Computing, 1990. **9**: p. 138-153.

34. Braun, T., et al., *A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems.* Journal of Parallel and Distributed Computing, 2001. **61**: p. 810-837.

35. *GridFTP: http://www.globus.org/grid_software/data/gridftp.php*.

36. Berman, F., et al., *The GrADS Project: Software Support for High-Level Grid Application Development.* International Journal of High Performance Computing Applications, 2001. **15**(4): p. 327-344.

37. Cooper, K., et al. *New Grid Scheduling and Rescheduling Methods in the GrADS Project*. in *Workshop for Next Generation Software*. 2004. Santa Fe, NM.

38. Ribler, R.L., H. Simitci, and D. Reed, *The Autopilot performance-directed adaptive control system.* Future Generation Computer Systems, 2001. **18**(1): p. 175-187.

39. Schopf, J., et al., *Monitoring and Discovery in a Web Services Framework: Functionality and Performance of the Globus Toolkit's MDS4*. 2005, Argonne National Laboratory.

40. *http://www.globus.org/toolkit/mds/*.

41. Foster, I. *Globus Toolkit Version 4: Software for Service-Oriented Systems*. in *IFIP International Conference on Network and Parallel Computing*. 2005: Springer-Verlag LNCS 3779.

42. Foster, I. and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of Supercomputer Applications, 1997. **11**(2): p. 115-128.

43. Massie, M.L., B. Chun, and D. Culler, *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience.* Parallel Computing, 2004. **30**(7).

44. Sacerdoti, F., et al. *Wide Area Cluster Monitoring with Ganglia*. in *In Proceedings of the IEEE Cluster 2003*. 2003.

45. *http://www.cs.wisc.edu/condor/hawkeye/*.

46. *http://www.openpbs.org/about.html*.

47. *http://www.platform.com/Products/Platform.LSF.Family/*.

48. http://glueschema.forge.cnaf.infn.it/.

49. Logothetis, D., et al., *Failure-Resilient Expectations for Federated Systems*. 2006, UCSD Technical Report CS2006-0865.

50. *PlanetLab: https://www.planet-lab.org/.*

51. Fu, Y., et al. *SHARP: An Architecture for Secure Resource Peering*. in *Proceeding of the Sumposium on Operating Systems Principles (SOSP)*. 2003.

52. Czajkowski, K., et al., *SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems.* Lecture Notes in Computer Science, 2002(2537): p. 153-183.

53. http://www.extreme.indiana.edu/swf-survey/.

54. Barish, B. and R. Weiss, *Ligo and detection of gravitational waves.* Physics Today, 1999. **52**(10).

55. Deelman, E., et al. *GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists*. in *Proceedings of the IEEE High Performance Distributed Computing*. 2002.

56. Hastings, S., et al. *Image Processing on the Grid: a Toolkit for Building Grid-enabled Image Processing Applications*. in *Proceedings of the International Symposium on Cluster Computing and the Grid*. 2003.

57. Berriman, G.B., et al. *Montage: a Grid Enabled Engine for Delivering Custom Science-Grade Image Mosaics on Demand*. in *Proceedings of the SPIE Conference on Astronomical Telescopes and Instrumentation*. 2004.

58. Foster, I. and C. Kesselman, eds. *Computational Grids: Blueprint for a New Computing Infrastructure*. 2nd ed. 2003, M Kaufman Publishers, Inc.

59. Graham, R.L., et al., *Optimization and approximation in deterministic sequencing and scheduling: A survey.* Ann. Discrete Math, 1979. **5**: p. 287-326.

60. Ali, S., et al., *Modeling Task Execution Time Behavior in Heterogeneous Computing Systems.* Special Tamkang University 50th Anniversay Issue, 2000. **3**(3): p. 195-207.

61. Singh, G., C. Kesselman, and E. Deelman, *Optimizing Grid-Based Workflow Execution*. 2005, University of Southern California 05-851 PDF.

62. Marin, G. and J. Mellor-Crummey. *Cross Architecture Performance Predictions for Scientific Applications Using Parameterized Models*. in *Proceedings of the joint ACM SIGMETRICS-Performance 2004 Conference on Measurement and Modeling of Computer Systems*. 2004.

63. Lu, D. and P. Dinda. *Synthesizing Realistic Computational Grids*. in *In the Proceedings of Supercomputing 2003*. 2003.

64. Kee, Y.-S., H. Casanova, and A. Chien. *Realistic Modeling and Synthesis of Resources for Computational Grids*. in *Proceedings of the ACM Conference on High Performance Networking and Computing*. 2004.

65. Waxman, B.M., *Routing of Multipoint Connections.* IEEE Journal of Selected Areas in Communications, 1988. **6**(9): p. 1617-1622.

66. Doar, M.B. *A Better Model for Generating Test Networks*. in *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM)*. 1996.

67. Faloutsos, M., P. Faloutsos, and C. Faloutsos. *On Power-Law Relationships of the Internet Topology*. in *Proceedings of the ACM SIGCOMM*. 1999.

68. Jin, C., Q. Chen, and S. Jamin, *Inet: Internet Topology Generator*. 2000, Technical Report CSE-TR443-00, Department of EECS, University of Michigan.

69. Medina, A., et al. *BRITE: An Approach to Universal Topology Generation*. in *Proceedings of MASCOTS '01*. 2001.

70. Buyya, R., D. Abramson, and J. Giddy. *An Economy Driven Resource Management Architecture for Global Computational Power Grids*. in *In the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. 2000.

71. *Xen Virtual Machine: http://www.cl.cam.ac.uk/research/srg/netos/xen/.*

72. Vahdat, A., et al. *Scalibility and Accuracy in a Large-Scale Network Emulator*. in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. 2002.

73. Kwok, Y.-K. and I. Ahmad, *Benchmarking and Comparison of the Task Graph Scheduling Algorithms.* Journal of Parallel and Distributed Computing, 1999. **59**(3): p. 381-422.

74. Deelman, E., et al. *Pegasus: Mapping Scientific Workflows onto the Grid*. in *Across Grids Conference 2004*. 2004. Nicosia, Cyprus.

75. Deelman, E., et al., *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems.* Submitted to Scientific Programming, 2005.

76. Ibarra, O.H. and C.E. Kim, *Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors.* Journal of the ACM, 1977. **24**(2): p. 280-289.

77. Jacob, J.C., et al. *The Montage Architecture for Grid-Enabled Science Processing of Large, Distributed Datasets*. in *Proceedings of the Earth Science Technology Conference (ESTC)*. 2004.

78. *Eagle Nebula Wiki: http://en.wikipedia.org/wiki/Eagle_Nebula.*

79. *Amazon Elastic Cloud: www.amazon.com/ec2.*

80. *EMAN, http://ncmi.bcm.tmc.edu/~stevel/EMAN/doc*.

81. *Southern California Earthquake Center. http://www.scec.org/*.

82. Shih, G.C. and E.A. Lee, *A Compile-Time Scheduling Heuristic for Interconection-Constrained Heterogeneous Processor Architectures.* IEEE Transactions on Parallel and Distributed Systems, 1993. **4**(2): p. 75-87.

83. Yang, T. and A. Gerasoulis, *DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors*. 1994, Rutgers Technical Report TRCS94-12: New Brunswick, NJ.