NETWORK AND CPU CO-ALLOCATION IN HIGH THROUGHPUT COMPUTING ENVIRONMENTS

By
James Basney

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

DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCES)

at the
UNIVERSITY OF WISCONSIN – MADISON
2001

Abstract

A High Throughput Computing (HTC) environment delivers large amounts of computing capacity to its users over long periods of time by pooling available computing resources on the network. The HTC environment strives to provide useful computing services to its customers while respecting the various resource usage policies set by the many different owners and administrators of the computing resources. Matching jobs with compatible computing resources according to the job's needs and the attributes and policies of the available resources requires a flexible scheduling mechanism. It also requires mechanisms to help jobs to be more agile, so they can successfully compute on the resources currently available to them. A checkpoint or migration facility enables long-running jobs to compute productively on non-dedicated resources. The work the job performs with each allocation is saved in a checkpoint, so the job's state can be transferred to a new execution site where it can continue the computation. A remote data access facility enables jobs to compute on resources that are not co-located with their data. Remote data access might involve transferring the job's data across a local area supercomputer network or a wide area network. These checkpoint and data transfers can generate significant network load.

The HTC environment must manage network resources carefully to use computational resources efficiently while honoring administrative policies. This dissertation explores the network requirements of batch jobs and presents mechanisms for managing network resources to implement administrative policies and improve job goodput. Goodput represents the job's forward progress and can differ from the job's allocated CPU time because of network overheads (when the job blocks on network I/O) and checkpoint rollback (when the job must "roll back" to a previous checkpoint). The primary contribution of this work is the definition and implementation of a network and CPU co-allocation framework for HTC environments. Making the network an allocated resource enables the system to implement administrative network policies and to improve job goodput via network admission control and scheduling.

Acknowledgements

It has been a great pleasure for me to do my research as a member of the Condor group at the University of Wisconsin. I thank my advisor, Miron Livny, for his support and guidance and for giving me the opportunity to work on the Condor system with such a talented group of people. I am indebted to all members of the Condor project for contributing to the system on which I based my work. I am particularly grateful to Rajesh Raman, Todd Tannenbaum, Derek Wright, Doug Thain, Jim Pruyne, and Mike Litzkow for their significant contributions to the Condor system and for their input on my research.

The Kangaroo system, described in Chapter 3, is joint work with Doug Thain and Se-Chang Son. The fine-grained bandwidth control, described in Chapter 6 is joint work with Se-Chang Son.

I thank the members of my oral exam committees: Miron Livny, Marvin Solomon, Paul Barford, Remzi Arpaci-Dusseau, Mukarram Attari, and Mary Vernon. I am grateful for their evaluation of my work. Their comments have significantly improved the quality of this dissertation.

I also thank Lorene Webber, Virginia Werner, Marie Johnson, and Cathy Richard for helping me navigate the University bureaucracy and the Computer Systems Lab staff for maintaining my computing environment during my work.

I am grateful to the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign for providing me an office from which I worked on this dissertation. Jeff Terstriep, Mike Pflugmacher, and Tom Roney have been gracious hosts.

Paolo Mazzanti's work with Condor at Italy's National Institute of Nuclear Physics provided inspiration for much of my research. I am grateful to him for that and also for hosting me in Bologna while I did some of this work.

I owe a debt of gratitude to Bob Geitz and Rich Salter, my advisors at Oberlin College, for getting me started in computer science on a solid foundation.

I thank my family for their support during my graduate career, particularly for understanding why someone would want to say in school for so long.

Most of all, I am grateful to my wife Sharon Lee for putting up with me throughout this process and giving me the daily encouragement and support I needed.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many users are unsatisfied with the computing capacity delivered by a single server, be it a desktop computer or supercomputer. The user's computational needs may simply be greater than the capacity of the server, or the server may be oversubscribed such that the user can obtain only a small fraction of its resources. At the same time, other servers may sit idle [40]. These users can benefit from some form of distributed computing, enabling them to harness additional computing capacity available on the network.

To distribute a computation manually, the user must choose the execution site, transfer any required data to that site, launch the computation, and collect the results when the computation completes. The user chooses the execution site from the set of servers where he or she has an account based on static server characteristics, including expected availability and performance, and current server load, obtained from a network information service or by probing the state of each server. If the servers do not share a common file service, the user must manually distribute the data for the computation to the execution site using a file transfer mechanism and manually retrieve the output when the computation completes.

Manually distributing jobs quickly becomes unwieldy as the number of jobs or potential execution sites grows. A distributed job scheduler frees the user from the burden of choosing the execution site and manually launching the computation. Users need not be aware of the state of the resources in the system or which resources are allocated to the job. The job scheduler locates resources compatible with each job and allocates resources to jobs according to a performance goal, such as minimizing weighted job response time or maximizing total system throughput. The scheduler can use load information gathered from the available execution sites to quickly choose the best execution site based on current conditions.

The scheduler manages the allocation of resources to implement administrative policies (such as user and group priorities) and to ensure that resources are not oversubscribed. For example, if a server's memory is oversubscribed, performance can degrade significantly due to virtual memory *thrashing*, where CPU utilization drops because jobs are blocked on paging activity. One solution to the oversubscription problem, called *space sharing*, is to allocate resources (or resource partitions) to jobs when they are scheduled. The user indicates the job's resource requirements when submitting the job to the system, or the system predicts the job's resource requirements based on previous experience with similar jobs. The job scheduler locates the required resources and allocates them to the job. If the job exceeds its resource allocation, the scheduler suspends it until its resource needs can be met with a larger allocation. While the job is suspended, its resources may be allocated to other jobs. The scheduler can also monitor resources for indications of oversubscription (for example, high paging rates and low CPU utilization) and incrementally decrease the load on the oversubscribed resource, by suspending or migrating jobs, until performance improves.

The main goal of a space sharing scheduler is to effectively allocate primary resources to jobs, such as threads of execution on a CPU, memory space, and disk space. The scheduler must consider many resource characteristics in determining which resources can potentially service a job, including relatively static characteristics such as CPU architecture, operating system version, and mounted file systems as well as dynamic performance and load metrics. Additionally, diverse and complex policies may be associated with the resources in the distributed system. For example, resources may be available to different job types at different times of the day, and each resource's policy may prioritize jobs and users differently, depending on who owns that particular resource. The scheduler must often address issues of resource fragmentation as well, packing jobs into the available memory and disk space at a server and packing parallel jobs into the available CPUs to minimize unallocated resources.

An important secondary scheduling goal is to ensure that allocated resources are in fact used efficiently. Efficient use of the primary resources can be a function of the performance of secondary resources, such as remote servers (for example, network file servers), as well as memory, disk, and network bandwidth. Unlike the allocated partitions of the primary resources, these secondary resources are shared with other jobs or users. Other processes running at the same execution site will compete with the job for memory and disk bandwidth, and processes at many execution sites may compete for bandwidth on a shared network or for service from a shared remote server. Allocation of these secondary resources can achieve similar benefits to allocation of the primary resources, namely, to police access to the resources according to administrative policies and to ensure that resources are not oversubscribed. Oversubscription of these secondary resources reduces the efficiency of the primary resource allocations because jobs spend more time waiting for bandwidth or remote service and thereby take longer to complete.

1.2 Remote Execution

Distributing jobs to servers on the network results in *remote execution*, where data and other resources required for the job's execution are not present locally on the execution site. In some systems, jobs are submitted from a front-end node, and the job's data must be transferred between the front-end node and the allocated compute node during execution. Long-term disk allocations at execution sites are often limited, so jobs with large datasets must transfer their data between a mass storage facility and allocated scratch disk space at the execution site during their run. When a shared network file system is used, the job's data must be transferred between the network file server and the execution node, using either direct remote procedure call style operations (as in NFS [72]) or via a local disk cache (as in AFS [30]). In computational grid environments [23, 63, 64], a job often must provide its own remote data access mechanisms [7, 24, 67, 68] to transfer data between the home site and the allocated remote site, because the job can make few assumptions about the computing environment provided at the remote site.

Data transfer can be characterized as a cost of remote execution. Jobs must pay this cost to obtain the benefit of the additional computing capacity available on the network. If the job is charged for network or file server usage, then data transfers can have a direct cost. If the job is charged for its allocated CPU time, any data transfers that cause the job to block have an associated CPU cost, because the job is charged for the CPU even though it is not using it while the transfer is being performed. Even if CPU and network resources are free, the data transfer can add a utility cost in terms of increased job run-time. Intelligently managing remote data access in a remote execution environment can reduce these costs, allowing the system to deliver greater computing throughput to its users.

1.3 Thesis Statement and Research Contribution

This dissertation explores the efficiency of compute jobs in distributed systems by focusing on the network costs of remote execution. As resources become more distributed and I/O requirements grow, data movement on the network presents significant challenges for the efficient use of compute resources.

The thesis of this work is that scheduling and allocating network transfers in a distributed job scheduling system can significantly improve delivered system throughput. We demonstrate this thesis with the following research contributions.

- We propose a *goodput* metric [3] to evaluate the execution efficiency of jobs in a remote execution environment.

- We present a profile of scheduling and network usage in the Condor distributed system [37], deployed in the Computer Sciences department at the University of Wisconsin-Madison, to understand the network requirements of jobs in the system and to motivate our network management mechanisms.

- We present a mechanism, called *execution domains* [4], for clustering compute services on the network to improve data locality.

- We present an implementation of CPU and network co-allocation in the Condor scheduling framework [2].

1.4 Organization of the Dissertation

The dissertation proceeds as follows. Chapter 2 presents an overview of the Condor High Throughput Computing (HTC) environment, in which this research was conducted. We develop our thesis in further detail in Chapter 3 by describing the network requirements of remote execution in a distributed system and presenting our *goodput* metric for evaluating the network efficiency of remote job execution. Chapter 4 presents a case study of execution efficiency in the Condor job scheduling system deployed at the University of Wisconsin. Then, in Chapter 5, we present a simple mechanism for improving data locality in job scheduling environments and show how we can apply this mechanism in the Condor resource management environment. Chapter 6 presents the primary contribution of this dissertation, a network and CPU co-allocation framework for HTC environments, and describes an implementation of the framework in the Condor resource management system. Chapter 7 concludes the dissertation with a summary of our contributions and a discussion of possible future work.

Chapter 2

High Throughput Computing with Condor

2.1 Introduction

In this chapter, we give an overview of the Condor High Throughput Computing environment [37], which served as the platform for the research presented in this dissertation. A High Throughput Computing (HTC) environment [42] strives to provide large amounts of processing capacity over long periods of time by exploiting the available computing resources on the network. The HTC system must meet the needs of resource owners, customers, and system administrators, since its success depends on the support and participation of each of these groups. Resource owners donate the use of their resources to the customers of the HTC environment. Before they are willing to do this, the owners must be satisfied that their rights will be respected and the policies they specify will be enforced. Customers will use the HTC environment to run their applications only if the benefit of additional processing capacity is not outweighed by the cost of learning the complexities of the HTC system. System administrators will install and maintain the system only if it provides a tangible benefit to its users which outweighs the cost of maintaining the system.

Resources on the network are often distributively owned, meaning that the control over powerful computing resources is distributed among many individuals and small groups. For example, individuals in an organization may each have "ownership" of a powerful desktop workstation. The willingness to share a resource with the HTC environment may vary for each resource owner. Some resources may be dedicated to HTC, while others are unavailable for HTC during certain hours or when the resource is otherwise in use, and still others which are available to only specific HTC customers and applications. Even when resources are available for HTC, the application may be allowed only limited access to the components of the resource and may be preempted at any time. Additionally, distributed ownership often results in decentralized maintenance, when resource owners maintain and configure each resource for a specific use, further increasing resource heterogeneity.

The Condor environment addresses these challenges with three primary mechanisms [41]: checkpointing, remote system calls, and classad matchmaking. Each of these mechanisms has important implications for scheduling and allocating network transfers. The checkpointing mechanism allows Condor to save a snapshot of a job's state so it can preempt that job and migrate it to a new execution site. These snapshots can be large, and transferring them between execution sites can generate significant network load. The remote system call mechanism allows Condor to present a friendly environment for jobs running on remote execution sites by forwarding system calls that can not be serviced at the execution site to the job's home node for processing, providing transparent access to remote data for Condor jobs and enabling monitoring and control of the job's behavior at the remote site. The classad matchmaking mechanism provides the powerful scheduling mechanisms required for harnessing heterogeneous, distributed computing resources. We will see in later chapters that the flexibility of the matchmaking framework enables the clustering and co-allocation mechanisms we have developed. We present additional details about each of these mechanisms in the following sections.

2.2 Checkpointing

A checkpoint is a snapshot of the current state of an application that can be used to restart the application from that state at a later time, potentially on a different machine. Checkpointing enables preempt-resume scheduling. If the scheduler decides to no longer allocate a CPU to an application, the scheduler can checkpoint the application and preempt it without losing the work the application has already accomplished. The scheduler can then resume the application at a later time when a CPU is available. Preempt-resume scheduling is essential in an HTC environment for implementing the resource allocation policies specified by resource owners and for allocating resources fairly to long-running jobs. The scheduler preempts running jobs to allocate resources to higher priority jobs and when resources are removed from availability. A checkpoint can be transferred over the network to implement process migration. Checkpointing also provides fault tolerance, allowing an application to restart from the most recent snapshot in the event of a service interruption.

Condor provides user-level, transparent checkpointing [36]. A job linked with the checkpoint library can be checkpointed at any time by sending it a checkpoint signal. The signal hander, installed by the checkpoint library, writes the job's state to a file or network socket. Checkpointing parameters can be set at run-time, allowing the checkpoint destination and compression options to be chosen immediately before performing each checkpoint. The checkpoint contains the entire memory state of the job, as well as additional information about open files, pending signals, and other process attributes, so jobs with large memory image sizes generate large checkpoints. Disk space for storing these large checkpoints may not be available locally at the execution site. Checkpoint servers can be deployed around the network to provide dedicated checkpoint storage space. Chapter 5 presents a mechanism for localizing checkpoint transfers to and from checkpoint servers.

2.3 Remote System Calls

Condor's remote system call mechanism, illustrated in Figure 1, exports the job's home environment to the remote execution site, so jobs need not be re-written to cope with the heterogeneity inherent in the HTC environment. Condor jobs can be linked with a remote system call library that interposes itself between the job and the operating system [31] at the system call interface. When the job performs a system call, the system call library can redirect the call to a server, called the *shadow*, running in the job's home environment.

Some system calls, such as memory allocation calls, are always performed locally and never

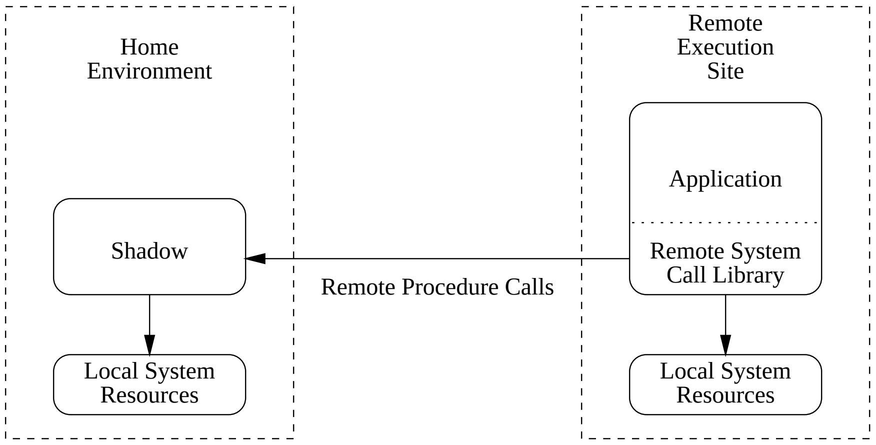

Figure 1: Remote System Calls

redirected to the shadow. Other calls, such as timezone or user identity lookups, are always redirected to the shadow, so it appears to the job that it is running in its home environment, instead of running in a guest account in a potentially different timezone.

The remote system call mechanism allows the shadow to control how the job accesses files at the remote execution site. When the job opens a file, the remote system call library makes a request to the shadow for instructions on how the file should be accessed. The shadow chooses if the library should access the file using remote procedure calls back to the shadow, local file access on the execution site, or a connection to a file server on the network, and the shadow provides a filename translation to the library when necessary. The shadow may choose local access for a file to improve performance if the file has been staged locally at the execution site or is available directly from the execution site using a distributed file system. The shadow can also choose to enable buffering or compression in the remote system call library for each file.

Trapping system calls in the remote system call library allows the shadow to monitor and control the job's behavior. The shadow sees many of the job's system calls directly because they are redirected to the shadow for execution. For calls that are not redirected, the library can provide callbacks to the shadow to keep the shadow informed of the job's behavior and to allow the shadow to control local system calls as well. The shadow can record statistics about the job's run to be presented to the user and can monitor the job's resource consumption to detect if the job's resource allocation should be modified. The checkpoint and remote system call libraries can be integrated to provide callbacks to allow the shadow to control the job's checkpointing as well. The clustering and co-allocation mechanisms presented in later chapters will use this functionality to implement resource management policies for running jobs in the shadow.

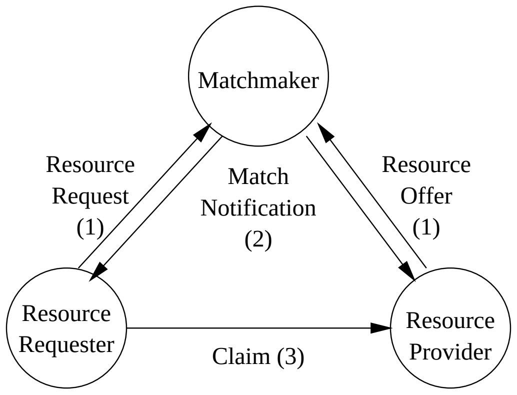

Figure 2: Matchmaking Protocol

2.4 Classad Matchmaking

Job scheduling in Condor is implemented with the classad matchmaking framework [56, 57]. The matchmaking architecture consists of resource providers, resource requesters, and a matchmaker. Resource allocation follows a three step protocol. Entities first advertise resource requests and offers in the classad language to the matchmaker. The matchmaker executes the matchmaking algorithm to locate compatible resource requests and offers and notifies the advertising entities when a match is found. The requester then contacts the providers directly to claim their resources. The requester and provider may perform additional negotiation in the claim step, and either party may decide not to complete the allocation. If the allocation succeeds, the entities actively maintain the claim relationship during the allocation's lifetime to monitor and update the allocation's status. Either entity may sever the relationship at any time, depending on the policies and the service guarantees negotiated between them. Claim monitoring can also include a keep-alive mechanism to detect if one of the entities has failed. This protocol is illustrated in Figure 2.

The classad language uses a semi-structured data model, so there is no fixed schema for the representation of resource requests and offers. Each resource request or offer contains a set of attribute definitions which describe the request or the offered resource. They each also contain a `Requirements` expression which specifies the compatibility between requests and offers and a `Rank` expression which indicates preferences. To locate a match between a request and an offer, the matchmaking algorithm evaluates the offer's `Requirements` expression in the context of the request and the request's `Requirements` expression in the context of the offer. If both `Requirements` expressions evaluate to `True`, the classads are successfully matched.

The Condor matchmaker executes the matchmaking algorithm periodically. The algorithm orders resource requests according to customer priorities and searches for the best matching resource offer (i.e., the offer which the request's `Rank` expression ranks the highest), for each resource request in turn. Resource requests represent a request from a customer agent to run a job, and resource offers represent an offer from a compute server to run a job. A successful match and claim therefore results in a job placement on the matched compute server. The matchmaker supports a preemption policy expression, configured by the Condor administrator. A job can be matched with a claimed compute server only if the preempting job's owner has higher priority than the preempted job's owner and both the compute server's local policy and the matchmaker's preemption policy expression allow the preemption. In this case, the running job is preempted during the claim protocol so the preempting job can start running in its place.

2.4.1 Gang-Matching

Until recently, matchmaking in Condor was strictly bilateral, as described above: matches occurred between a job and a compute server. However, some environments can benefit from co-allocation of multiple resources. For example, jobs in a distributed system may require a software license to run. Software license managers can limit the number of licensed applications running at any time and can control where software licenses may be used. The resources in the distributed system may have heterogeneous license configurations: some compute sites may have an unlimited use license, others may have a license limiting the number of running application instances, and others may have no license at all for a given application. Network-based licenses are not tied to specific servers but instead control the usage of an application across a network domain. Managing such an environment requires co-allocation of licenses and servers, to ensure that jobs run only when and where there is a license available for them.

Recent research [55, 58] has defined the *gang-matching* model, which extends the matchmaking framework to support co-allocation. The advertisement specification is extended to support a docking paradigm, to specify the relationships between entities in a match. Each advertisement defines an ordered list of labeled ports. Each port specifies a request for a bilateral match, so multilateral match requests are indicated by including multiple ports in the advertisement. The gang-matching algorithm searches for classads that can successfully "dock" at a request's ports such that the bilateral constraints specified for each port are satisfied.

A naive implementation of the gang-matching algorithm that enumerates all possible combinations in search of a successful gang-match quickly becomes untenable because of the combinatorics involved. Gang-matching research to-date has therefore focused on performance optimizations to make the gang-matching algorithm feasible in practice. Promising classad indexing and heuristic search strategies have been developed [55].

We developed the network and CPU co-allocation framework presented in Chapter 6 concurrently with the gang-matching research, and we have strived to make our work compatible with the gang-matching model. While the gang-matching research focused on the general co-allocation problem, we have explored issues specific to co-allocating network and CPU resources and evaluated network and CPU co-allocation strategies that can be implemented in the gang-matching framework. The remaining work required to integrate these two efforts is detailed in Section 7.1.

2.4.2 Flocking

As described above, resource requesters obtain matchmaking services by simply advertising their resource requests to a matchmaker, making it convenient for users to participate in the distributed system remotely. Users can configure a local job manager to submit resource requests to a "remote" matchmaker (in a different administrative domain, across a wide-area network), and assuming the user has sufficient priority and permission, the matchmaker will match the job's resource requests with resource offers in its domain. Using remote system calls, the user's jobs can transparently access the user's files while running at the remote compute sites.

A job manager that submits resource requests to multiple matchmakers is said to be *flocking*, i.e., grouping many "condors" together into a "flock" of resources that can be harnessed to run the user's jobs. Flocking extends the pool of resources available to the user. When there are insufficient resources available to satisfy the user's requests, the job manager sends requests to more matchmakers until the user's resource needs are satisfied or until all known matchmakers are consulted.

The flexibility provided by the flocking mechanism enables users to harness increasingly remote resources, making it more important to manage the efficiency of remote execution. Flocking jobs between administrative domains requires more data movement across longer network distances. Managing that data movement is the subject of this dissertation.

2.5 Summary

Checkpointing, remote system calls, and matchmaking are three fundamental services provided by the Condor system to meet the challenges of distributed ownership and resource heterogeneity. Each of these services has important implications for the network efficiency of remote execution. Checkpoint transfers and remote system calls can generate significant network load, and matchmaking makes it easier for users to harness resources distributed over longer network distances. In the following chapters, we will examine the network load in the Condor system, present mechanisms we have developed to improve remote execution efficiency in Condor, and describe how these mechanisms can be applicable in other contexts.

Chapter 3

Goodput

3.1 Introduction

The term *goodput* is used in the study of computer networks to describe the rate at which data is successfully received at the destination of a network stream [1, 59]. Goodput differs from the low-level throughput of the network stream due to lost or discarded data packets (i.e., “badput”) and protocol overheads.

$$goodput = throughput - badput - overheads \quad (1)$$

In the general sense, goodput is a measure of the service received at a higher system level, compared to the raw throughput measured at a lower level. It is a measure of the amount of useful work accomplished by the system and how efficiently system resources were used to accomplish that work.

The goodput versus throughput distinction can be applied to CPU allocations as well. Job scheduling systems often charge users for the elapsed wall clock time of their CPU allocation. However, users are not generally interested in the amount of wall clock CPU time they obtain. Instead, they are interested in the amount of work their jobs have accomplished. We define a goodput measure for CPU allocations in an attempt to understand the difference between the allocated CPU throughput and the amount of useful work accomplished by users’ jobs.

For CPU intensive jobs, we can define goodput as the accumulated CPU time of the job and compare CPU time to wall clock time. However, this definition can not be applied to other job classes. We can also define goodput in terms of application-specific metrics, like the number of simulation events processed or frames rendered. However, we prefer a measure in units of time so we can directly evaluate the efficiency of the system.

For the purposes of this study, goodput is equal to the time it would take to execute a job using local, dedicated compute resources. The badput and overheads introduced by the job scheduler are a function of the use of distributed or non-dedicated resources. This definition is somewhat subjective, since the distinction between local and distributed and between dedicated and non-dedicated resources is not always clear. The distribution of computing resources ranges from internal memory and I/O busses to crossbar supercomputer interconnections to high speed local area networks to wide area networks. Likewise, the spectrum of dedicated to non-dedicated computing resources ranges from systems where the job competes only with system processes to systems where the job is guaranteed some level of service (with some level of confidence—no system can make absolute guarantees) to systems where the job may be preempted at any time. Therefore, when we apply the goodput measure to a system, we also define the boundary between the benchmark environment (local, dedicated resources) and the system environment (remote or non-dedicated resources).

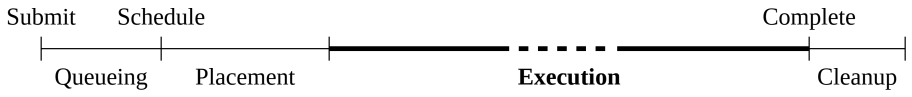

Figure 3: Job Timeline

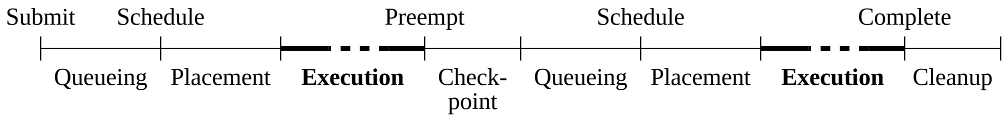

Figure 4: Job Timeline (Preemption)

3.2 Goodput of a Batch Job

We begin with a description of the factors that determine the goodput of batch jobs. A batch job is submitted to a queue where it waits until the scheduler can find resources available to run it. When the scheduler decides to run a queued job, it must first "install" the job on the allocated compute resource before the job can begin execution, during the job *placement* phase. In an early batch system, the placement phase would require loading the job from a card or tape reader. In current batch systems, the placement phase often requires loading the job's data from a front-end submission node to a compute node over a network. Once the placement is complete, the job begins its execution. When the job's execution completes, the job enters a *cleanup* phase, where the scheduler transfers the job's output to the front-end node (or to the printer in early batch systems) and deallocates the disk resources associated with the job on the compute node. The phases of batch job execution are illustrated in Figure 3. The job's goodput occurs in the execution phase (shown in bold in the figure). The queueing, placement, and cleanup phases are overheads of the job scheduling environment. If the job were instead run on local, dedicated resources, it would not need to wait in a queue for resources to become available or wait for its data to be transferred to and from a compute node.

The scheduler may terminate the job's resource allocation before the job completes (called *pre-empting* the job), when for example the job exceeds its resource allocation (by running too long or using too much memory) or if the system must be shutdown for maintenance. Preempt-resume schedulers also preempt jobs to give resources to higher priority jobs. Schedulers on non-dedicated resources will preempt jobs when the resource is reclaimed by its owner. When possible, the job's intermediate state is saved in a checkpoint when the job is preempted, allowing the job to continue execution from where it left off when it obtains a new allocation. Figure 4 shows an example of a job that is preempted during its run. A job's run may include many execution phases interrupted by preemptions. To continue execution from the checkpoint, the checkpoint data must be transferred to the new execution site during the placement of the job.

If the job is unable to checkpoint its state when preempted, the work it has accomplished at the execution site will be lost and must be redone (i.e., it is badput), analogous to a dropped network packet that must be retransmitted. Not all jobs on all systems can be checkpointed. Only some operating systems support process checkpointing. User-level checkpointing services [35, 36, 52] often

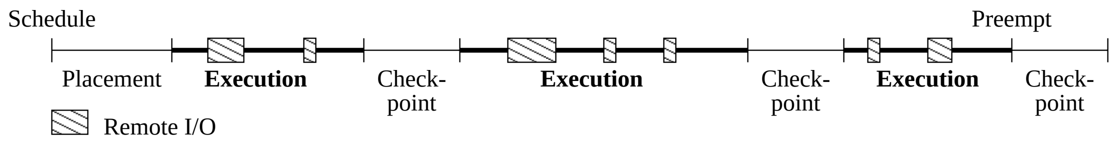

Figure 5: Job Timeline (Periodic Checkpoints, Remote I/O, and Preemption)

require the job to be re-linked with a checkpointing library (often not possible for commercial applications) and checkpoint only a subset of system state (for example, inter-process communication is often not fully supported). In some cases, jobs implement their own checkpoint/restart functionality to work around these restrictions.

Even if the job is able to checkpoint, it may not have the time or resources it needs to successfully complete a checkpoint. The batch system may preempt the job by immediately killing it without warning, or the job may have a limited amount of time to evacuate all resources at the execution site upon receiving a preemption notification. If the job can not transfer its checkpointed state over the network in that time, the checkpoint will be lost.

The job's execution can also be interrupted for remote data access and periodic checkpointing. In many cases, it is not possible to anticipate all of the data required by the job during the placement phase. Additionally, local disk resources on the execution site may be limited, so the job can not store all of its data locally during execution. In these cases, the job will need to access data from remote sources during its run using a remote I/O library [7, 24, 41] or a network file system such as NFS [72] or AFS [30]. The job may also interrupt its execution periodically to save its state in a checkpoint to guard against system failures. In particular, periodic checkpoints can guard against losing too much work if the job is unable to checkpoint when preempted. Figure 5 illustrates how a job's execution can be interrupted by remote I/O and periodic checkpointing.

3.3 Improving Batch Job Goodput

The goal of this work is to improve the goodput delivered to batch jobs. We want jobs to use allocated computational resources efficiently to improve job response time, improve overall system throughput, and enable users to get more computing done with smaller allocations. A batch job's goodput is determined by two factors: the amount of time the job waits on synchronous network operations (placement, cleanup, checkpoint transfers, and remote I/O) during its allocation and the amount of lost work due to failures, including a failure to checkpoint when preempted.

$$goodput = throughput - network\ wait\ time - lost\ work \tag{2}$$

Lost work is directly related to the efficiency of checkpointing in the system. Reducing the amount of time or resources required to perform a checkpoint can improve the success rate of preemption checkpoints. For example, if the system gives jobs a limit of one minute to checkpoint when they are preempted, then faster checkpointing will enable more checkpoints to complete before the deadline. Additionally, if we can make periodic checkpoints less expensive, then they can be performed more frequently, resulting in less work lost when a failure occurs.

We present two techniques for improving goodput. The first is to overlap I/O with computation whenever possible. The second is to schedule network I/O to improve the performance of large I/O operations. Scheduling network I/O is important for two reasons. First, it can increase the system's ability to overlap I/O with computation. Second, when it is not possible to overlap I/O with computation, scheduling can improve the performance of synchronous I/O. We begin with a discussion of two standard techniques for overlapping I/O with computation.

3.4 Overlapping I/O with Computation

Overlapping I/O with computation to improve performance is not a new idea [28]. Traditional approaches include *spooling* and *multiprogramming* [65]. Spooling interposes a higher performance intermediate storage device between the job and its lower performance data sources and targets. The spooler transfers the job's input data from the data source to the intermediate storage in advance of its execution. The job performs all I/O to the intermediate storage device, and the spooler transfers the job's output data from intermediate storage to its final destination. In this way, the job offloads blocking I/O operations to the spooler. In contrast, multiprogramming keeps the CPU busy by allocating the CPU to an idle job when the running job blocks on an I/O operation. The job still blocks on I/O, but it relinquishes the CPU when it can not use it to make forward progress. We consider spooling and multiprogramming in more detail below.

3.4.1 Spooling

The concept of spooling evolved from the operation of early batch processing systems. The punched cards for many jobs were assembled into a batch and the batch was read onto magnetic tape by an inexpensive peripheral computer. The operators transferred the tape to the main computer, which read the jobs from tape, executed them sequentially, and wrote their output to another tape. The operator then transferred the output tape to another peripheral computer which printed the tape's contents. Spooling systems (from Simultaneous Peripheral Operations On Line) were introduced to automate peripheral operations, so human operators did not need to transfer tapes between the peripheral and the main computers. The computer ran a reader and a writer program in addition to the compute job (using multiprogramming). The reader program would read input data for the next job(s) from punch cards to tape (or later, to disk) while a job was running, and the writer would move job output from tape (or disk) concurrent with the job's execution.

Modern operating systems implement a form of spooling in the file system buffer pool. When an application issues a read request to the operating system, the operating system reads a block from disk into a memory buffer and returns the requested data to the application. File blocks are cached in the buffer pool so that later requests to previously read disk blocks can be satisfied directly from memory. The application writes data to the memory buffer and continues processing, and the operating system commits the data to disk asynchronously. Distributed file systems, such as AFS, use local disk buffers to spool data from remote file servers.

3.4.1.1 Spooling Input

In general, spooling input requires some form of advance prediction or planning to ensure that the job's input data is available in the spooler's buffers when it is requested. Prediction can occur at two levels in the batch job environment. At the file access level, the job's future read requests are predicted based on past read requests. For example, read-ahead predicts that the job will access a file sequentially. When the job requests a file block, the I/O system also schedules reads for the next few blocks in the file. At the job scheduling level, the scheduler loads input data for a job at the compute site before launching the job. To do this, the scheduler must choose the compute site for the job in advance of the job's execution. In multi-server systems, this requires a prediction of future server availability, including when currently running jobs will complete. Priority inversion can result if the system predicts poorly, with high priority jobs waiting for servers that are not available when predicted while lower priority jobs run on servers that are available earlier than expected. Predicting when running jobs complete requires an estimate of the remaining run-time of each job. One "rule of thumb" for estimating job run-times based on a study of job lifetime distributions is that the median remaining lifetime of a process is equal to its current age [27, 33].

Enforcing run-time limits on jobs can limit the penalty for mispredicting run-times. Users submit their jobs to queues with configured run-time limits, choosing the appropriate queue based on their own estimates of their jobs' behavior. The scheduler can then use these run-time limits as worst-case run-time estimates for currently running jobs. A significant drawback to run-time limits, however, is that users often find it difficult to accurately predict their jobs' run-times, so they overestimate to avoid job preemptions [19], particularly for jobs that are not checkpoint-enabled.

In a dynamic resource environment, such as a cluster of non-dedicated workstations, the effectiveness of spooling input data can be limited because of the difficulty of predicting resource availability. In addition to uncertainty about job run-times, there are no guarantees that a compute resource will be available to run the next job when the previous job completes. The resource can potentially drop out of the pool at any time when it is reclaimed by its owner for another purpose. Likewise, compute resources can join the pool at any time, making it impossible to overlap the placement of the first job on a newly available compute resource with computation on that resource.

It can also be difficult for users to give information about their jobs' file access patterns in advance. The job's I/O requirements may depend on the inputs for the run and the results of complex calculations in the job. However, read-ahead strategies [12, 18, 45, 50] can still be useful in these cases. The job's first access to a file will be synchronous, but later sequential accesses to the same file can be serviced from speculatively pre-fetched data blocks in the spooling area.

3.4.1.2 Spooling Output

Spooling output can be much simpler than spooling input, in that it does not require any advance prediction or planning. One concern, however, is the fate of spooled data when the job terminates or is preempted. Ideally, the job should release its CPU allocation when it terminates, and any remaining spooled output data should be transferred from the execution site by the spooler as network resources permit. Just as input spooling requires a disk allocation at the execution site in advance of the CPU allocation, output spooling potentially requires the disk allocation to persist beyond the

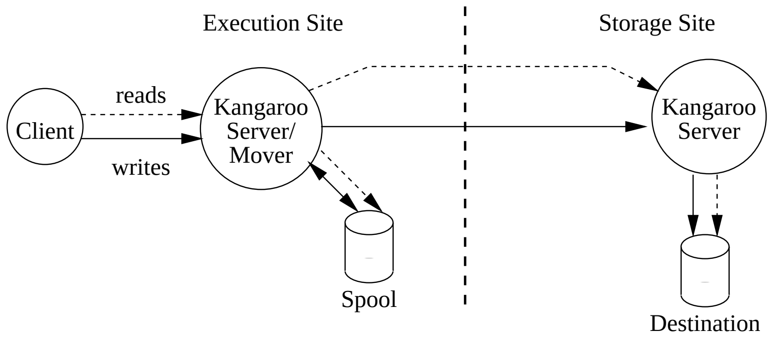

Figure 6: Kangaroo System Overview

end of the CPU allocation. If the job must quickly vacate all resources at the execution site when preempted, some of the job's spooled output may be lost. This is similar to a failed checkpoint: the job must rollback and reproduce the lost output.

Kangaroo [66] is an example of a simple user-level mechanism for spooling batch job output to overlap I/O with computation and hide network errors. As illustrated in Figure 6, clients connect to the Kangaroo server to issue block file read and write requests. The server satisfies read requests for local files by reading the requested file block and sending it in a reply to the client. Likewise, the server satisfies local write requests by writing the requested file block to disk. For writes to remote files, the server hands the requests over to the Kangaroo mover by writing each file block to a local spool directory, acknowledging the client's request, and notifying the mover that a new block is ready. The mover is responsible for reliably sending spooled file blocks over the network to the destination server. For reads of remote files, the server first checks the local spool directory to see if the request overlaps with any previously written but uncommitted data. If the read can not be satisfied from the local spool directory, the server forwards the read request to a remote server running on the remote host and forwards the server's reply to the client.

Kangaroo gets its name from the fact that data "hops" through intermediate servers and buffers on its way to the destination. In the scenario described above, data in Kangaroo hops from the application to the local disk buffer and then directly to a remote server where it is committed at the destination filesystem. The architecture allows for more advanced configurations that route data through intermediate servers, allowing Kangaroo to take advantage of variations in link capacity along the network path and to be more resilient to individual link failures.

3.4.1.3 Opportunities for Spooling

Spooling can be used to overlap a job's I/O with its own computation or to overlap one job's I/O with another job's computation. The job itself can implement read-ahead and write-behind buffering with a separate thread of execution for spooling. Parallel I/O libraries (such as Nexus [22]) can be used to add this functionality to jobs. When the job completes its execution, it must transfer any remaining buffered output from the execution site before relinquishing the allocated resource. In many cases, the remaining output will be minimal, because the spooler thread will have been transferring the

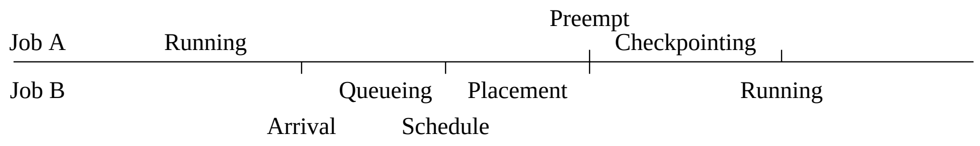

Figure 7: Spooling During Job Preemption

job's output as it was produced. However, the job may perform a large write at completion time to report the final results of the calculation. If the job's output rate is greater than the available network bandwidth, there can be a significant amount of buffered output remaining at the end of the job's run. Ideally we would like to overlap this synchronous I/O at the end of the job's run with the execution of another task or job.

The master-worker paradigm [5, 26, 34] provides a promising opportunity for overlap between a single user's tasks or jobs. In a master-worker (MW) application, the master partitions the goal, assigns a sub-goal to each worker, and receives results when workers complete their assigned tasks. The master can overlap network I/O with computation by sending data for the next work-step to a worker before that worker has completed its current work-step. When the worker completes a work-step, it uses a separate spooler thread to send the results back to the master. The compute thread(s) begin processing the next work-step immediately upon completing the previous work-step. The master can use application-specific knowledge to schedule work-steps effectively.

The job scheduler can overlap the I/O of one job with another without advance planning during preemption. When the scheduler preempts a lower priority job for a higher priority job, the scheduler can spool the input data for the higher priority job before evicting the lower priority job. When the higher priority job is ready to run, the scheduler starts it and evicts the lower priority job. The timeline for this case is illustrated in Figure 7, where higher priority job B preempts lower priority job A. This strategy is most effective when both jobs fit in memory at the execution site. Otherwise, when the new job starts, the jobs will compete for memory while the preempted job is checkpointing.

Previous work has shown that spooling techniques can be applied to reduce migration costs. Pre-copying [69] uses a copy-on-write mechanism to first save a checkpoint of the job's state while it continues running. Then, after the initial concurrent checkpoint is written the application is suspended and any memory pages modified since the concurrent checkpoint are transferred again. Copy on reference [15, 79] allows an application to begin execution on a destination workstation before all memory pages have been transferred. When the application references a page which has not yet been restored, the page fault handler first reads the page from the network and then allows the memory reference to proceed. Memory pages may be optimistically prefetched to reduce the latency of synchronous page transfers.

3.4.2 Multiprogramming

Multiprogramming relies on the ability to context switch between jobs at an execution site. Load balancing schedulers run multiple jobs per CPU, leveraging the multiprogramming services provided by the local operating system at each execution site. Load sharing schedulers, on the other hand, allocate one or more CPUs exclusively to one job and do not typically implement any form

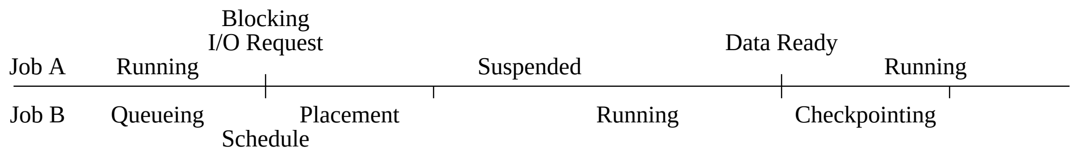

Figure 8: Multiprogramming Example

of multiprogramming. However, load sharing systems could also potentially benefit from multiprogramming for large I/O events. The challenge is that implementing multiprogramming at the load sharing layer has greater overhead, because starting and stopping jobs is more costly than local process context switches.

Figure 8 illustrates an example of multiprogramming in a load sharing system. Job A issues a read request on a file that currently resides on “distant” storage (for example, a tape archive or remote ftp server). Upon receiving the request, the remote I/O service determines that the request will block for a long time. For example, a query to the tape archive indicates that the file is currently not staged to disk or the remote ftp server is currently unreachable. Job A suspends and/or checkpoints itself and allows its CPU(s) to be allocated to job B until the file can be retrieved. When the file is ready, job A wakes up and preempts job B. Since the checkpoint of one job can be overlapped with the computation of the other, the primary cost of the context switch is the placement cost of job B. If this placement cost is significantly lower than the time the job is expected to block waiting to retrieve the distant file, then switching the jobs will improve system goodput. Multiprogramming at this level preserves the exclusive CPU allocations provided by the load sharing system to maximize performance rather than requiring that the job share its CPU with other jobs throughout its lifetime, as in load balancing systems. The job relinquishes the CPU only when it knows it will not be able to use it for a significant period of time.

3.4.3 Need for Network Scheduling

As described above, techniques for overlapping I/O with computation can significantly improve the goodput obtained by batch jobs. However, there are limitations to the effectiveness and applicability of spooling and multiprogramming techniques in real systems. Both techniques require additional buffering (in memory or on disk) at the execution site. When the buffer space is exhausted, remotely executing jobs must resort to synchronous remote I/O techniques. In dynamic environments, where resources frequently switch between available and unavailable states, prediction for input spooling is very challenging. If CPU availability can not be predicted, then the scheduler must perform synchronous placements when CPUs become available. Likewise, if resources may be reclaimed by their owners at will, jobs may have limited time for checkpointing.

For these reasons, network scheduling can play an important role in improving the effectiveness of techniques to overlap I/O with computation and can improve the performance of synchronous network I/O when the techniques can not be applied. For example, network reservations can help the scheduler ensure that data for the next job will be ready at the compute site before the previous job is expected to finish. The scheduler can also prioritize network traffic to improve goodput. For example, synchronous network I/O, such as blocking reads to a remote server, should take priority over asynchronous write-behind of spooled data, assuming sufficient buffer space at the compute site.

3.5 Summary

We have presented a *goodput* metric for measuring the efficiency of remotely executing batch jobs, reviewed standard techniques for overlapping I/O with computation, and described examples where these techniques can be applied in distributed job scheduling systems. I/O and computation overlap is not always possible due to the inability to predict future system state and to limited buffer space. In later chapters, we investigate scheduling techniques that do not rely on overlap to improve the efficiency of network I/O. In the next chapter, we provide further motivation for the goodput metric by presenting performance statistics gathered in a production Condor installation.

Chapter 4

Goodput Case Study

4.1 Introduction

In this chapter, we examine the factors that effect goodput in the Condor pool at the University of Wisconsin-Madison Computer Sciences department. We present statistics gathered over a two and a half year period. In that time, the size of the pool has grown from 300 to over 500 CPUs, in part due to the addition of 192 dedicated Intel Linux CPUs. The pool also includes approximately 100 non-dedicated lab workstations that are rebooted nightly and 200 non-dedicated desktop workstations for graduate students, professors, and departmental staff.

Significant network upgrades were performed during this period. At the start of the period, the machines were distributed between about 20 10 Mbps Ethernet subnets and 10 100 Mbps Ethernet subnets. The subnets were linked by a subset of the department's eight routers, which each provided approximately 30 Mbps throughput to a switched 155 Mbps ATM backbone. Since then, most machines were moved from 10 Mbps to 100 Mbps Ethernet networks, and the subnets were linked by a single backbone router capable of routing at link speeds.

4.2 Goodput Index

To evaluate the goodput delivered in the Condor pool, we implemented a "goodput index" by submitting a small number of representative jobs and monitoring their performance. The goodput index functions like an index in the stock market. Focusing on the performance of a small number of representative jobs gives an indication of the overall behavior of the system and allows comparisons of system behavior over time. In a distributed system like Condor, it can be difficult to gather and analyze statistics for all jobs in the system, particularly when jobs cross administrative domains using flocking. The data gathered for the index jobs can show how system changes (in policy configuration, job workload, and available resources) effect the goodput of different types of jobs.

We constructed the index by submitting 10 jobs with checkpoint sizes of 4, 24, 48, 96, and 180 MB between July 1998 and March 1999, to represent jobs that would run efficiently on the 32, 64, 128, and 256 MB workstations available in the Condor pool. Each job simply executed a busy-loop and did not perform any file I/O. The jobs' logs recorded how long each job ran at each execution site, when the job successfully checkpointed, and how much CPU time was saved in each checkpoint. From the logs, we computed the "badput" for the jobs as the difference between the job's run-time and saved CPU time. The badput has two components: work lost due to checkpoint rollbacks (i.e., when the job is unable to checkpoint when preempted, so work since the last checkpoint is lost) and low CPU utilization of checkpointed work due to remote execution overheads. The

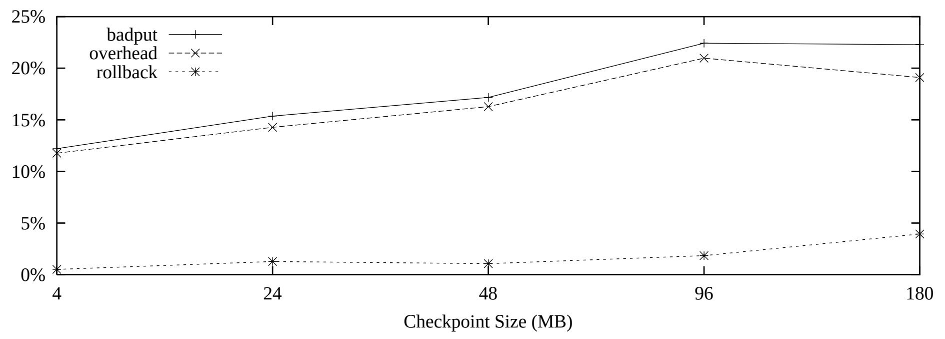

Figure 9: Goodput Index

total “badput” recorded for these jobs is plotted as a percentage of total run-time in Figure 9. The figure also plots the remote execution overhead (the percentage of the allocated CPU time that went unused) and the percentage of the job’s run-time that was lost due to checkpoint rollbacks.

The figure shows that all jobs used less than 90% of their allocated CPU time (i.e., the overhead for each job is over 10%). We believe that most of this overhead is caused by Condor’s suspend policy. When interactive activity is detected on a workstation, Condor suspends the job instead of immediately preempting it. If the interactive activity is short-lived, the job can be resumed, avoiding the overhead of preempting and re-scheduling the job. Jobs can be suspended frequently if there is intermittent interactive activity on a machine for long periods of time. Further evaluation of the suspend policy and its impact on system efficiency was beyond the scope of our study but would be worthwhile future work.

The figure shows an additional increase in badput for the jobs with larger checkpoints. Two factors account for this. First, the jobs with larger checkpoints spent more time saving and restoring their checkpointed state, accounting for an additional 8% of badput for the job with a 180 MB checkpoint compared with the 4 MB job. Second, we see an increase in work lost due to checkpoint rollbacks for the jobs with larger checkpoints. Rollbacks accounted for less than 1% of the 4 MB job’s badput, but the 180 MB job lost approximately 4% of its run-time to checkpoint rollbacks. During this time, we frequently saw throughputs of 3 Mbps or lower for individual checkpoint transfers to the checkpoint server, causing transfers of 180 MB checkpoints to take over 8 minutes. Condor was configured with a 10 minute preemption window, so jobs that took longer than 10 minutes to checkpoint were killed, accounting for 70% of the large job’s failed checkpoints.

Two additional causes of checkpoint rollbacks are apparent from the cumulative distributions shown in Figure 10. Over 85% of all rollbacks resulted in less than 10 minutes of lost work. This is explained by the fact that Condor was configured to checkpoint only those jobs that have run for at least 10 minutes. Jobs that were preempted within 10 minutes of startup were killed without a checkpoint because the cost of checkpointing was seen to outweigh the benefit of saving the small amount of work. Because only a small amount of work was lost in each case, those rollbacks accounted for under 20% of the lost work. A second cluster of checkpoint rollbacks appears at the 3 hour mark. Condor jobs performed periodic checkpoints every 3 hours. Unfortunately, the periodic checkpoint process was slightly error prone and would cause the job to abort and rollback under

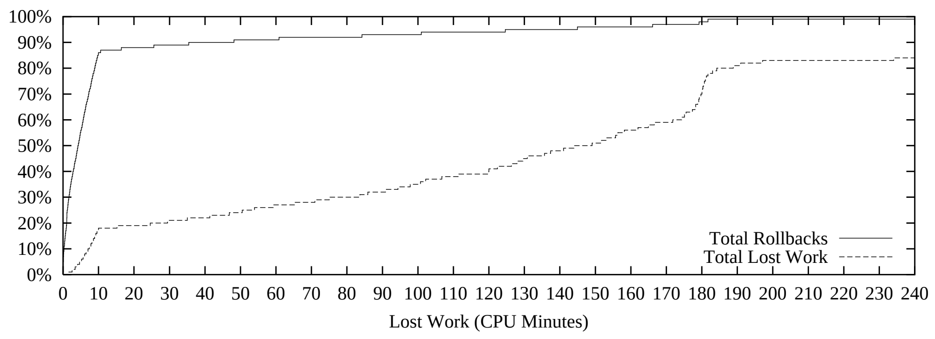

Figure 10: Cumulative Distribution of Checkpoint Rollbacks

some circumstances. Even though this error occurred very infrequently, it accounted for another 20% of the lost work because 3 hours of work was lost each time.

In summary, we saw approximately 12% badput related to checkpoint transfers for large jobs: 8% caused by the overhead of performing synchronous transfers of large checkpoints and 4% caused by checkpoint rollbacks. We have seen greater badput levels at sites that do not allow jobs to checkpoint when preempted (i.e., the job is immediately killed to avoid any interference with the workstation owner). For example, a set of jobs running in the Condor pool at the University of Bologna, where jobs were not allowed to checkpoint when preempted, lost over 25% of their run-time to checkpoint rollbacks. Increasing the frequency of periodic checkpoints can help increase goodput in these environments.

4.3 CPU Availability

We monitored keyboard activity for all non-dedicated hosts in the Condor pool between November 1999 and April 2001 to profile the capacity available to be harnessed by Condor jobs. Condor jobs only run on lab and desktop workstations when they are not in use by interactive users, according to keyboard activity and load average. We define an idle period to be a time interval between keystrokes of duration greater than 15 minutes. We logged all idle periods for 566 hosts during the monitoring experiment (241 lab workstations and 325 desktop workstations). Some hosts were online for only a part of the monitoring time.

Figure 11 presents cumulative distribution plots for the workstation idle times. Overall, the hosts were idle 75% of the time. This agrees with an earlier study that found more than 70% of workstation capacity in the department went unused by workstation owners [48]. As a group, the lab workstations are significantly less idle than the desktop workstations, and the lab workstations are never idle for more than 24 hours because of the daily scheduled reboots. We categorize each idle period as either short (less than one hour), medium (between one and three hours), or long (greater than 3 hours). 51% of the idle periods lasted one hour or less, accounting for 7% of total idle time. 25% of the idle periods lasted more than three hours, accounting for 83% of total idle time.

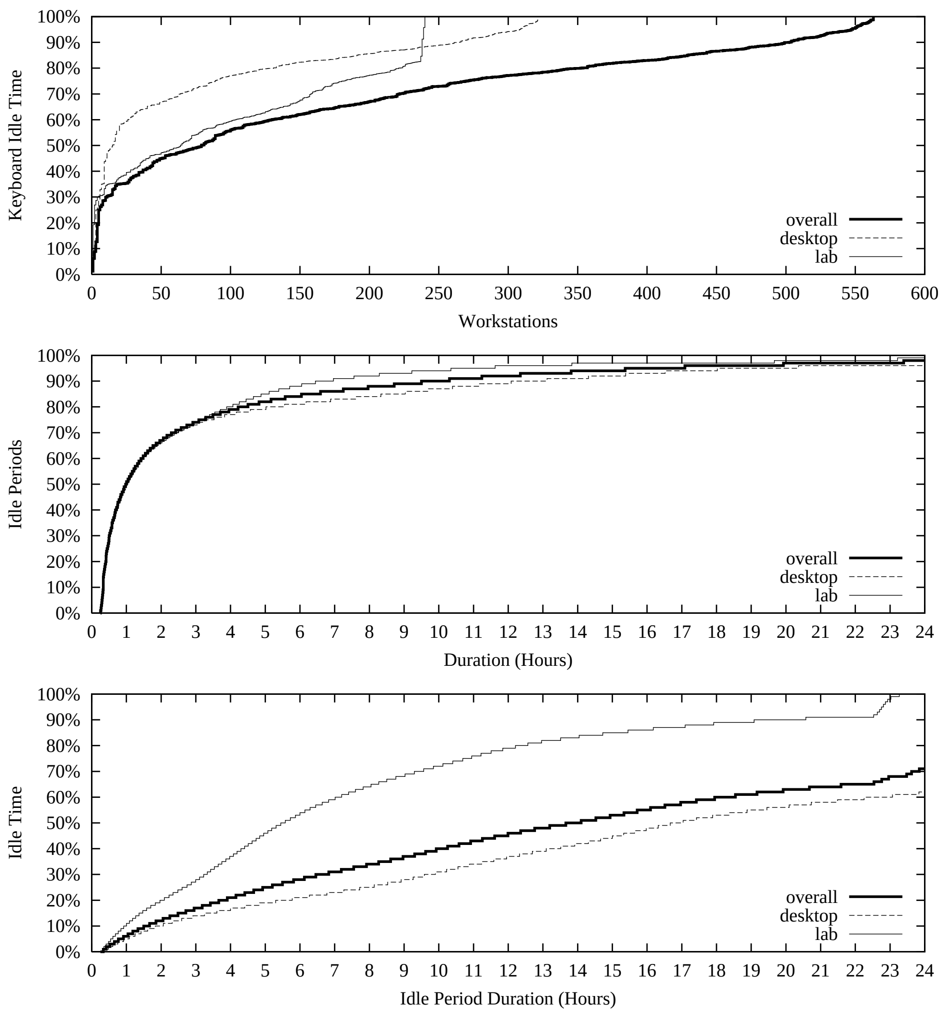

Figure 11: Distribution of Available Computing Capacity

The large number of short idle periods has important implications for the goodput in the system. The system can potentially expend a significant amount of network resources chasing the last 7% of capacity delivered from the short idle periods. If we can predict the future availability of CPU resources, we can avoid paying too high a cost for those short allocations. Previous work [47] explored simple predictors of future availability in the Condor system with promising results. Future availability was accurately predicted based on availability in the same hour on the previous day (accounting for weekday vs. weekend patterns) and on recent availability. Selecting the workstation that had been available the longest was shown to cause fewer preemptions than random selection.

The Network Weather Service [75] provides sophisticated performance forecasting using time-series analysis. In one study [76], the NWS made short term (10 seconds) and medium term (5 minutes) predictions of CPU performance with a reported typical mean absolute error of less than

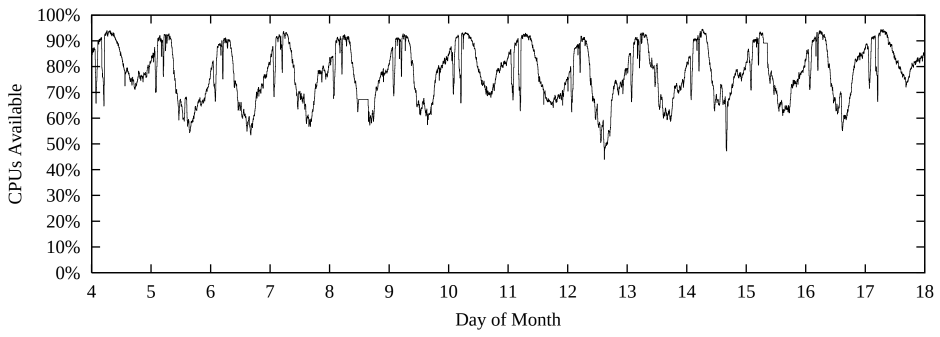

Figure 12: Available Condor CPUs (February 2001)

10%, where error was computed as the difference between the predicted CPU performance and the performance obtained by a test process.

We have added some simple support for future availability prediction into the current version of the Condor system. Condor's description of each machine now includes how long that machine has been available to run jobs, the length of the machine's last interval of availability, the percentage of time that machine has been available over its lifetime in the system, and Condor's own prediction of the machine's future availability based on past history and a configured level of confidence. If the machine's current period of availability is A seconds and the level of confidence is $L\%$, Condor predicts that the machine will be available for another P seconds, where L of the past available intervals greater than A were P seconds in duration or longer. If there were no previous intervals greater than A, Condor simply predicts $P = A * (2.0 - L)$. Jobs are free to use any combination of these availability statistics in choosing their execution site.

Figure 12 plots the percentage of available CPUs in the Condor pool for two weeks in February 2001. Daily cycles of CPU availability are clearly present. CPU usage increases steadily from about 8 AM to a daily peak at 4 PM, with another smaller increase later in the evening. Two sharp drops in the number of available CPUs occur each morning, first around 2 AM when the Windows lab workstations are rebooted and then again at 4 AM when the Unix lab workstations are rebooted.

4.4 Job Placements

Figure 13 plots the number of job placements and preemptions initiated by the Condor matchmaker between November 1998 and April 2001. The matchmaker preempts a job when it decides to run a higher priority job in its place. Job preemptions caused by resources becoming unavailable (for example, a workstation owner reclaiming the workstation) are initiated locally at the execution site without the matchmaker's involvement and so they are not included in these statistics. The upper plot illustrates that the number of job placements per day varies between 500 and 43000, with a median of 4279 job placements per day (i.e., approximately 3 per minute). Frequent job placements can generate significant load on the network. For example, 3 placements per minute for jobs that have 64 MB of data (on average) will generate an average of 25 Mbps of placement

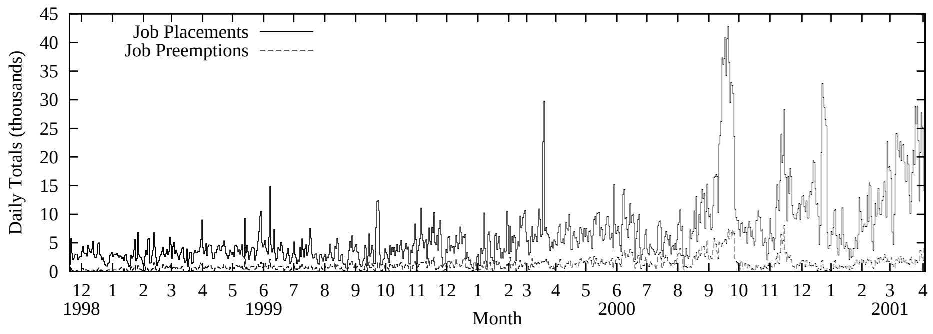

Figure 13: Job Placements and Preemptions

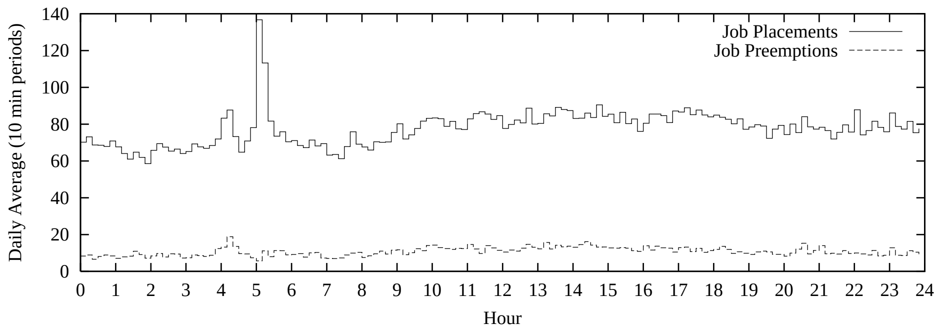

Figure 14: Job Placements and Preemptions by Time of Day

traffic. On average, about 18% of these scheduled job placements preempt a lower priority job. The frequency of job placements varies significantly from day-to-day according to the job workload. Large numbers of short-running jobs submitted to the system can generate significant spikes in scheduling activity, as seen, for example, in September 2000.

Figure 14 illustrates how the number of job placements and preemptions varied throughout the day in the first quarter of 2001. The number of job placements and preemptions increased during work hours, when workstation activity increases, causing CPUs to frequently switch between available and unavailable states. A spike in the number of job placements occurred after 5 AM, when the lab workstations completed their reboots.

4.5 Checkpointing

Condor job checkpoints include the job's entire memory image. We see checkpoint sizes range from 1 MB to over 512 MB. Checkpoint sizes have been increasing in the pool as the memory available on workstations in the department increases, as illustrated in the cumulative distribution plots in Figure 15. In 1998, over 97% of all checkpoints written were under 64 MB in size. That percentage decreased to 76% in 1999, 69% in 2000, and 51% in 2001. Interestingly, there was a

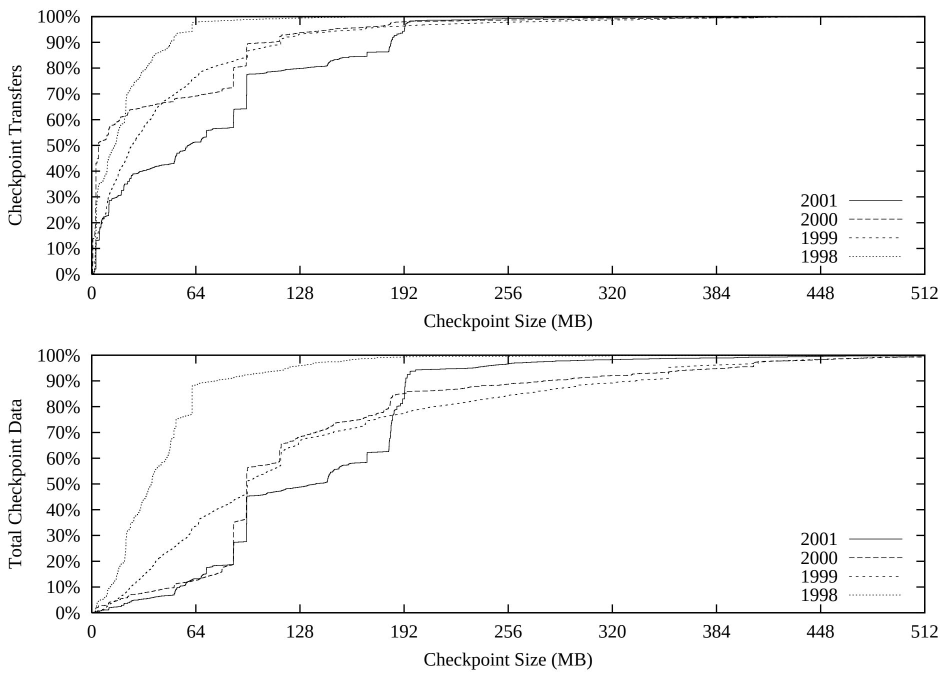

Figure 15: Cumulative Distributions of Checkpoint Sizes

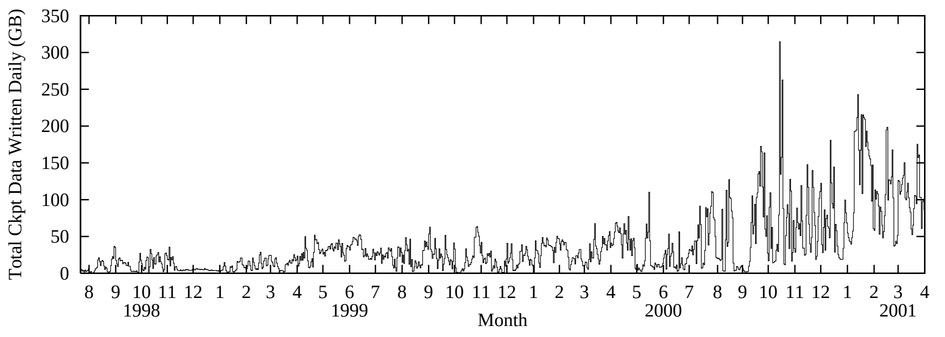

Figure 16: Total Daily Checkpoint Data Written

greater percentage of checkpoints over 192 MB in 1999 and 2000 than seen so far in 2001.

Figure 16 graphs the daily total amount of checkpoint data written by Condor jobs to the checkpoint servers, showing the general trend in increased checkpoint traffic, as checkpoint sizes and the number of CPUs in the pool increase. It is not uncommon to see over 50 GB of checkpoint writes in one day, for an average checkpoint write load of over 5 Mbps.

A Condor job will only read the same checkpoint more than once if there is a rollback, so comparing checkpoint write traffic to checkpoint read traffic can give insight into the goodput of the Condor pool. A job will fail to successfully read or write a checkpoint if its allocation is terminated

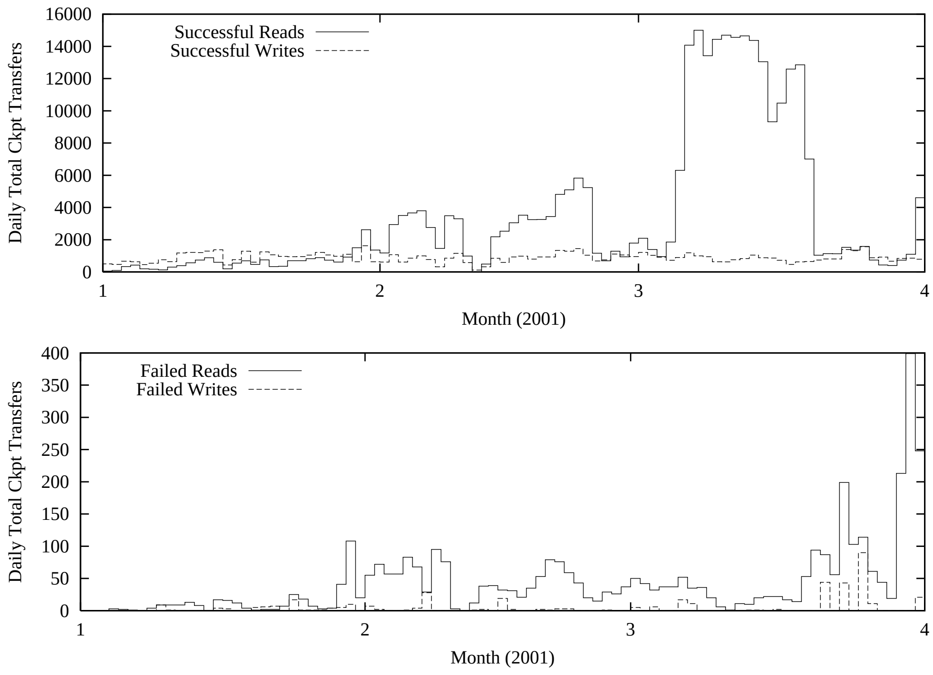

Figure 17: Daily Checkpoint Transfers

before the checkpoint is completely read or written or if there is a checkpointing error, so failed checkpoint writes are a sign of rollback and failed checkpoint reads are a sign that some allocations are shorter than the job placement time. Figure 17 shows the daily total number of checkpoint reads and writes to the main checkpoint server, both successful and unsuccessful, for the first three months of 2001. There were more checkpoint reads than writes in 35% of the days shown the upper graph, signalling a potentially significant amount of lost allocation time due to checkpoint rollbacks. An investigation of the Condor logs found two problems causing Condor jobs to abort after reading a checkpoint: missing data files and corrupt checkpoints. Condor kept attempting to restart the jobs each time they failed, causing the large number of additional checkpoint reads. These errors can generate significant badput in the pool, because CPU and network resources are allocated to jobs that keep failing shortly after startup.

The lower graph shows over 3000 failed checkpoint reads and 15000 failed checkpoint writes. A failed checkpoint read occurs when a job is preempted before it has finished reading its checkpoint. Since the job has not begun computing in this case, it simply aborts the checkpoint restore and vacates the execution site immediately. An investigation of the Condor logs shows two causes for the failed checkpoint writes. In many cases, the job was unable to establish a connection to the checkpoint server (after the job's resource manager initiated the transfer request), either because of a network error or because of high server load, resulting in a request timeout appearing in the logs. The remaining checkpoint writes failed because the job was killed by the workstation owner for taking too long to complete its checkpoint when preempted. As part of this study, we added a module

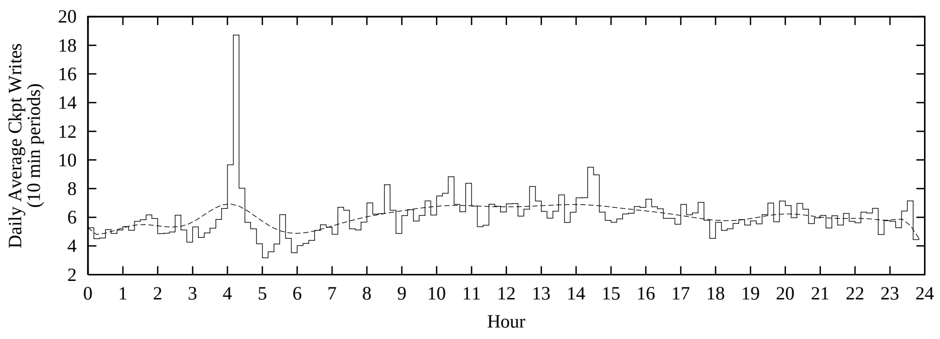

Figure 18: Checkpoint Transfers by Time of Day

to the CondorView tool to display checkpointing statistics for a pool, so users and administrators can use the statistics to detect and diagnose these types of problems.

Figure 18 shows the average number of checkpoints written throughout each day in the first quarter of 2001. The Bezier curve plotted with the data shows that checkpointing traffic increases by approximately 25% during business hours (8-17), when jobs are preempted more often by workstation owners. There is also an increase later in the evening. This curve is mirrored by the keyboard activity statistics presented above in Section 4.3. The large number of checkpoints after 4 AM resulting from the daily lab workstation reboots are clearly shown. Note that the 2 AM reboots don't appear in the graph because Condor is not yet able to checkpoint Windows jobs.

4.6 Summary

We have presented a profile of the Condor pool in the University of Wisconsin-Madison Computer Sciences department. We illustrated the difference between allocated throughput and obtained goodput with a simple set of "goodput index" jobs. Significant idle computing capacity is available in the network, but short idle periods provide diminishing returns because of the overheads of job setup and teardown. Changes in workstation availability and job workload can generate large scheduling events with high network load.

The profile illustrates that the efficiency of a distributed job scheduling system can vary dramatically based on available capacity, workload, and scheduling policies. When job placement and migration are inexpensive, the system can schedule aggressively to harness all available computing capacity. Jobs can run efficiently on workstations that will be available for only a short time and can migrate away immediately at the first sign of workstation owner activity. However, when network overheads increase, because of large checkpoints or a large number of CPUs relative to available network capacity, the high network load can significantly impact goodput, due to increased blocking on network I/O and an increase in checkpoint rollbacks. In the following chapters, we present two mechanisms we have developed to improve execution efficiency in these types of environments.

Chapter 5

Execution Domains

5.1 Introduction

In this chapter, we present a mechanism called *execution domains* [4] that we have developed to improve data locality in job scheduling environments by "clustering" compute nodes based on access to data resources. Execution domains ensure that jobs that require a given level of access to a data resource run at compute sites that can provide the needed access. If a job's input data is stored on a network file server, execution domains ensure that the job runs only on CPUs with reliable, high-speed access to that network file server. For jobs that produce large volumes of output, execution domains ensure that the job runs only on CPUs with access to sufficient storage capacity for the job's output. If the job produces large intermediate state (for checkpointing or out-of-core computation), execution domains ensure that, once the job begins its execution using a storage device, it migrates only among CPUs with high-performance access to that storage device. Since data resources may not be accessible from all CPUs in a wide-area computational grid for system administration and security reasons, execution domains are useful even for jobs that can tolerate low data rates, to ensure that the job only runs on CPUs with some type of access to the job's data. An execution domain may be empty if a dataset is not yet available. Jobs that require the dataset will not run until it is produced, so execution domains can also serve as a mechanism for controlling data dependencies between jobs.

An execution domain is a set of CPUs with a defined level of access to a physical or logical data resource. Levels of access may be defined according to performance, network distance, reliability, security, or other criteria. Execution domains may be defined at different access levels for a given data resource. However, a CPU is a member of at most one execution domain for each data resource. A data resource may be a *server* or *dataset*. A server is any data storage device, including network file servers and database servers. Servers occupy a fixed position on the network and can not be easily relocated. A dataset is a file or set of files stored on a server. Datasets may be dynamically replicated and migrated between servers on the network. Figure 19 illustrates an example execution domain configuration. Domains A, B, and C are each defined by their proximity to a set of data servers. As illustrated, a logical execution domain may include multiple physical servers. Domain D, which overlaps with the other domains, is defined by copies of a dataset staged on the local disks of four CPUs.

Domain managers are responsible for defining execution domain membership and maintaining the affinity between jobs and the execution domains of their data. We propose two types of domain managers: *domain migration agents* and *data staging agents*. The domain migration agent schedules the initial placement of job data files on file servers and ensures that jobs run within the execution

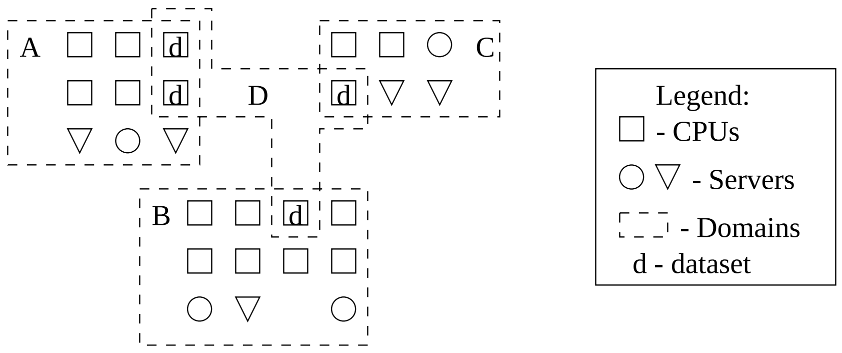

Figure 19: Example Execution Domain Configuration

domains of the file server(s) that store their files. The agent also monitors the demand for and availability of CPU and network resources in the server domains. If there is an insufficient number of CPUs in the execution domain of a file server, the domain migration agent can transfer the job's data files from the current file server to a server in a domain where more CPUs are available. The agent can then modify the job's domain requirements so it will run in the new execution domain. The data staging agent schedules the initial distribution of datasets and configures the execution domain membership for those datasets. The agent also monitors the demand for datasets by watching the job queues. If there are insufficient CPU and network resources in the execution domain of a dataset to meet current demand, the data staging agent expands the execution domain. The agent stages a copy of the dataset on additional storage devices and adds the CPUs in the domains of those storage devices to the domain of the dataset. Since domain managers are external to the job scheduler, they can be used to implement scheduling policies or algorithms that were not anticipated by the scheduler's developers. For example, the domain managers can use network load information to explicitly schedule data transfers for staging and migration over wide-area links.

5.2 Execution Domains in Condor

The Condor classad matchmaking framework [56, 57], presented in Section 2.4, enables easy implementation of execution domains with no changes to the Condor resource management system. Classad matchmaking gives us the ability to dynamically inject information into the system to achieve custom scheduling goals.

Domain managers can use the Condor APIs to modify the execution domain definitions and job requirements. To modify the domain definitions, the domain managers can insert, delete, or modify attributes in resource offers. When the domain staging agent stores a dataset on the local disk of a workstation, it inserts a new attribute for that dataset in the workstation's resource offer. When the domain migration agent moves a job's data files to a new file server, it modifies the `Requirements` and `Rank` expressions in the job's resource request so the job will execution in the domain of the new file server.

For example, the following resource offer describes a Sparc Solaris workstation with 256 MB of memory and a MIPS rating of 200:

```
OpSys = “Solaris2.6”;
Arch = “Sun4u”;
Memory = 256;
Mips = 200;
```

To include a workstation’s CPU in an execution domain, the domain manager inserts an attribute into the resource offer. For example, if this workstation has local access to the cs.wisc.edu AFS network filesystem, the manager inserts the following attribute:

```
AFSDomain = “cs.wisc.edu”;
```

Resource offers with different values defined for the `AFSDomain` attribute describe CPUs in different execution domains. The domain manager also inserts a Boolean attribute for each dataset to indicate that the CPU is a member of the execution domain of that dataset. For example:

```
HasDataSetXYZ97S3 = True;
```

The dataset may be staged differently for different CPUs. It may, for example, be located on the local disk of some CPUs and available to other CPUs via a network file server. If the jobs that require this dataset have different I/O characteristics, it is useful to define more restrictive execution domains. The domain manager can define an execution domain that includes only those CPUs with the dataset staged on the local disk as follows:

```
HasDataSetXYZ97S3Locally = True;
```

Jobs that need local-disk access speeds to this dataset should run only in this more restrictive execution domain. The Condor remote I/O library can be used to hide the different file access methods from the job. The library instruments the job’s I/O system calls and redirects them to the appropriate file access method. Support for many I/O access protocols is under development in the Condor remote I/O library, including FTP, HTTP, and GASS [7].

A job’s resource request indicates its requirements and preferences for an execution site. For example, the resource request below is compatible with the resource offer above. The request asks for a Sparc Solaris workstation with more than 80 MB of memory, with a preference for the CPU with the highest MIPS rating (i.e., CPUs are ranked in descending order by their Mips value):

```
Requirements = (other.OpSys == “Solaris2.6”) &&
               (other.Arch == “Sun4u”) && (other.Memory > 80);
Rank = Mips;
```

To indicate that the job stores its data files in the cs.wisc.edu AFS filesystem and so should run in the execution domain of that filesystem for best performance, the domain manager modifies the `Requirements` in the job’s resource request:

```
Requirements = (other.OpSys == “Solaris2.6”) &&
               (other.Arch == “Sun4u”) && (other.Memory > 80) &&
               (other.AFSDomain == “cs.wisc.edu”);
```

AFS [30] is an example of a global filesystem, where any file on an AFS server can be accessed from any AFS client, assuming the user has the necessary credentials, so if this job can tolerate higher-latency access to its AFS files, it could feasibly run on any CPU that serves as an AFS client. In this case, the job's resource request requires only that `AFSDomain` is defined, indicating that the CPU is an AFS client. The resource request indicates a preference for the cs.wisc.edu AFS domain, however, since a CPU in that domain will have the best access to the job's data files. This example uses the classad *isnt* operator to test if an attribute is defined. It also uses the Rank expression, where a value of True is ranked higher than a value of False.

```
Requirements = ... && (other.AFSDomain isnt Undefined);
Rank = (other.AFSDomain == "cs.wisc.edu");
```

To indicate that the job should run on a CPU in the execution domain of a dataset, the domain manager modifies the `Requirements` attribute of the resource request as follows:

```
Requirements = ... && other.HasDataSetXYZ97S3;
```

When the job begins its run, it uses the location attribute in the resource offer to find the dataset on the workstation. Jobs that require local disk access speeds to this dataset will require a CPU in the more restrictive execution domain:

```
Requirements = ... && other.HasDataSetXYZ97S3Locally;
```

Other jobs may not strictly require local disk access speeds, but will perform better at higher speeds. In this case, the domain manager specifies the job's preferences in the `Rank` expression of its resource request (where True is ranked higher than False):

```
Requirements = ... && (other.HasDataSetXYZ97S3 ||
                       other.HasDataSetXYZ97S3Locally);
Rank = other.HasDataSetXYZ97S3Locally;
```

5.3 Checkpoint Domains

We have also applied the execution domain mechanism to the management of checkpoints in Condor. In our experience, checkpoint transfers are often the main cause of network overhead for Condor jobs. As seen in Section 4.5, daily checkpoint traffic in our local Condor pool often exceeds 100 GB. The checkpoint of a job's state includes its entire memory state, so memory-intensive jobs can generate large checkpoints. When long-running jobs obtain short CPU allocations, they must store a checkpoint at the end of each allocation to save the work they have accomplished. Dedicated checkpoint servers, deployed across the network, provide storage space for these large job checkpoints.

To localize the transfer of checkpoints in the network, we define execution domains according to proximity to checkpoint servers. These *checkpoint domains* are defined by inserting a `CkptDomain` attribute into each CPU's resource offer. Jobs write their checkpoints to a checkpoint server in the current checkpoint domain and are restricted to migrate only to CPUs in the current checkpoint domain, to avoid transferring the checkpoint to a CPU beyond the domain. To implement this policy, the `Requirements` of the job's resource request must specify the checkpoint domain once the job has written its first checkpoint. For example:

```
CkptDomain = "ckpt.cs.wisc.edu";
Requirements = ... && (self.CkptDomain == other.CkptDomain);
```

A task may begin execution in any checkpoint domain, but once it performs its first checkpoint, it executes only on CPUs in the chosen checkpoint domain.

Since a task may wait a long time for an available workstation in its checkpoint domain, we support migration between checkpoint domains. As with other execution domains, a domain migration agent can transfer the checkpoint to a new checkpoint server and modify the *CkptDomain* attribute in the job's resource request. However, it is also possible for the domain migration agent to migrate the job without transferring the checkpoint between checkpoint servers by simply modifying the job's resource request. In this case, the job will transfer its checkpoint from the old checkpoint server directly to the CPU in the new checkpoint domain when it begins execution. For example, the domain manager can modify the resource request as follows so the job will run in either the ckpt.cs.wisc.edu domain or the ckpt.bo.infn.it domain:

```
CkptDomain = "ckpt.cs.wisc.edu";
Requirements = ... && ((self.CkptDomain == other.CkptDomain) ||
                       (other.CkptDomain == "ckpt.bo.infn.it"));
Rank = self.CkptDomain == other.CkptDomain;
```

The `Rank` expression specifies that the job should remain in the current checkpoint domain if a CPU is available there. Otherwise, if a CPU is available in the ckpt.bo.infn.it checkpoint domain, the job will transfer the checkpoint from the ckpt.cs.wisc.edu checkpoint server directly to that CPU to resume its execution. If the job is preempted again, it will send its checkpoint to the local checkpoint server in the ckpt.bo.infn.it domain and update its resource request to look for a new CPU in the new checkpoint domain:

```
CkptDomain = "ckpt.bo.infn.it";
Requirements = ... && (self.CkptDomain == other.CkptDomain);
```

It is possible for the migration agent to transfer a checkpoint only to find that CPUs are no longer available in the new checkpoint domain. By delaying the migration until the CPU is allocated to the job, the domain migration agent avoids performing potentially unnecessary checkpoint migrations. This savings represents a trade-off, because the job will need to transfer the checkpoint over a longer network distance at the start of the CPU allocation, increasing network wait time.

It is also possible to implement migration between checkpoint domains automatically (without intervention of a domain migration agent) by specifying more complex `Requirements` expressions. In the following example, the job is allowed to migrate to a new checkpoint domain if it has been waiting for an available CPU for over 24 hours.

```
LastCkptDomain = “ckpt.bo.infn.it”;
Requirements = . . . && ((self.LastCkptDomain == other.CkptDomain) ||
                         ((CurrentTime - self.StartIdleTime) > 24*60*60));
```

Alternatively, the job may be allowed to migrate between checkpoint domains only at night, when demand for capacity on the wide-area network is lower, as in the example below:

```
LastCkptDomain = “ckpt.bo.infn.it”;
Requirements = . . . && ((self.LastCkptDomain == other.CkptDomain) ||
                         (ClockHour < 7) || (ClockHour > 18));
```

We can also implement an even more permissive policy, which allows the checkpoint to migrate to a new domain at any time if there is insufficient CPU capacity in the current domain. We use the `Rank` expression to specify that we would prefer that the job remain in the current checkpoint domain, but it may migrate to any of the other domains specified in the `Requirements` expression when necessary.

```
Requirements = (other.CkptDomain == “ckpt.bo.infn.it”) ||
               (other.CkptDomain == “ckpt.cs.wisc.edu”) ||
               (other.CkptDomain == “ckpt.ncsa.uiuc.edu”);
Rank = (self.CkptDomain == other.CkptDomain);
```

Checkpoint domains differ from other execution domains due to the flexibility provided by checkpoint servers. Unlike file servers, which are often not under the administrative control of Condor administrators, checkpoint servers may be installed on any machines with available disk space. Since all nodes have access to all checkpoint servers through Condor APIs, checkpoint servers can be used as more general-purpose data staging areas to improve accessibility to job data, providing greater scheduling opportunities to the execution domain managers.

5.4 Related Work

Execution domains draw on the techniques of clustering and data staging to improve remote execution performance. Clustering is a well-established technique for improving performance and scalability in distributed systems.

Liu [38, 39] proposed a clustered load balancing model for job scheduling that leverages the natural clustering found in large distributed systems in terms of network performance and job workload. In this model, scheduling is performed independently in each cluster, enabling more accurate and dynamic scheduling, and jobs are dispatched globally only when local cluster resources are exhausted.

Ozden et al. [49] proposed a distributed clustering approach for designing load sharing systems. They show by simulation that the scalability of purely centralized load sharing algorithms is limited by the capacity of the central server, and the scalability of distributed load sharing algorithms is limited for non-homogeneous systems due to the increasing costs of distributed search. Distributed clustering takes a middle ground by performing centralized load sharing within each resource cluster and distributed load sharing between clusters, resulting in improved scalability.

The Utopia load sharing facility [73, 81] uses a cluster architecture for scalable distribution of resource load information. Load information can be exchanged between clusters according to a configured directed graph. The default job placement policy prefers to schedule jobs in the local cluster but jobs will run in remote clusters if a remote host is available with a significantly more attractive load index value.

Clustering techniques have also been used to improve locality of memory references and inter-process communication in large-scale non-uniform memory access (NUMA) multiprocessors.

Zhou and Brecht [10, 80] propose processor pool-based scheduling for large-scale NUMA multiprocessors. Processor pools are an operating system construct for scheduling parallel applications. The system is partitioned into a fixed set of equal sized pools, and the threads of a parallel job are scheduled to run within a single processor pool to improve memory locality.

The Hurricane operating system [70] uses a hierarchical clustering approach to improve performance and scalability in NUMA multiprocessors. Hurricane partitions hardware and software resources into clusters. Operating system resources are partitioned and replicated across the clusters to reduce resource contention within the operating system. Application requests to independent physical resources are managed by independent operating system resources. The scheduler performs fine-grained load balancing within clusters and course-grained job placement and migration between clusters, minimizing the number of clusters spanned by parallel jobs to improve communication locality. The scheduler directs application I/O to nearby disks and places application data close to where it will be accessed. Experiments with Hurricane found that clustering different resource classes independently can result in significant performance improvements.

The execution domains mechanism allows cluster definitions to be defined at any time and allows many independent cluster definitions to co-exist to support custom scheduling policies in the distributed system.

Data staging and replication are also well-known techniques for improving I/O performance, as illustrated by recent work in developing the Globus Data Grid [14, 71], an architecture for the management of storage resources and data distributed across computational grid environments. In the Globus Data Grid framework, a metadata service provides information about network connectivity and storage system details useful for choosing data storage sites, and a replica manager can create and delete copies of file instances (replicas) on the storage systems. A replica catalog provides information about where replicas are stored, and a replica selection service chooses the best available file replica based on user preferences and information about storage resource performance and policies. Condor classad matchmaking mechanisms have been used to implement a prototype Globus Data Grid replication selection service.

Replica selection services in the Globus Data Grid are independent of job placement services. A broker can combine replica selection with job placement to implement application-specific co-scheduling algorithms and policies. In contrast, execution domains are a simple mechanism using existing scheduling services provided by the Condor system to locate an execution site with the required access to a dataset.

5.5 Summary

Execution domains provide a flexible scheduling mechanism for ensuring that jobs have access to required data resources at the execution site. Unlike many existing clustering mechanisms, execution domains can be configured dynamically and can be defined for different classes of data resources on the network. We have shown how execution domains can be implemented in the Condor environment and illustrated how checkpoint domains are used to localize checkpoint transfers in Condor.

Chapter 6

Network and CPU Co-Allocation

6.1 Introduction

This chapter presents a framework for co-allocating network and CPU resources in batch scheduling systems. The goal of the framework is to avoid oversubscribing network resources. This goal is motivated by two concerns. First, batch jobs often share network resources with other best-effort network users. In many environments, it is important for the batch system to be a "good citizen" and not adversely impact interactive resource usage. Controlling (and monitoring) the system's network usage can help administrators manage available network resources more effectively. Second, goodput suffers when the network is oversubscribed. Transferring jobs' data to and from execution sites take longer, limiting the system's ability to overlap I/O with computation. Allocating network resources gives the system the ability to prioritize network streams when some network transfers are more time-critical than others.

Our network load control is not motivated by a need to improve low-level network performance or stability. We assume that underlying network resources are stable under heavy load. We evaluate our mechanisms on Ethernet [46] networks, which have been shown to be stable under heavy load [62]. We are instead motivated to control network load for the higher-level reasons stated above: to implement administrative policies and use available network resources more effectively to support efficient remote job execution.

To implement CPU and network co-allocation in the matchmaking framework, we define two types of resource providers: compute servers and network managers, as illustrated in Figure 20. Compute servers allocate CPU, memory, and disk resources at a compute site, and network managers allocate network resources. Compute servers advertise available computing capacity to the matchmaker, and network managers advertise available network capacity. Job managers make requests for compute and network resources. The matchmaker declares a match when CPU and network resources are available to satisfy a job's resource request. The job manager then contacts the compute server(s) and network manager(s) to claim the resources.

The matchmaker and network manager work together to allocate network resources. The matchmaker implements network admission control at the granularity of job placements and preemptions, according to the capacity advertised as available by the network manager. The network manager supports fine-grained network scheduling with advance reservations in the claiming protocol.

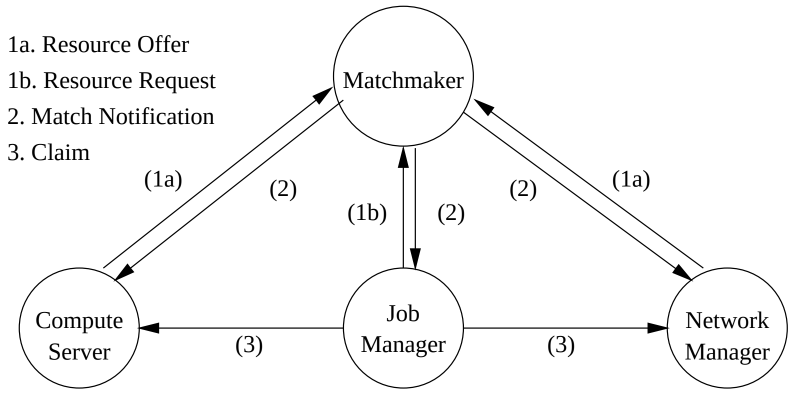

Figure 20: Network and CPU Gang-Matching Protocol

6.2 Admission Control

Admission control in the matchmaker restricts matches to only those allocations that can be implemented with available network capacity within the matchmaker's current scheduling horizon. The matchmaker builds a network map from the information provided by the network manager that includes the available capacity of each network resource. For each job placement request, the matchmaker determines if sufficient network capacity is available to perform the placement of the job at each candidate compute site. When it finds a match, the matchmaker subtracts the network capacity allocated in that match from its network map.

Admission control is particularly beneficial during large scheduling events, when there is a large change in the job workload or in resource availability. The following scenarios are examples of large scheduling events.

A high priority user submits a large cluster of jobs. This often occurs in high throughput computing environments, where users submit a large batch of jobs that explore a parameter space or perform discrete work-steps in a large computation. It also occurs in high performance environments, where large parallel jobs require an allocation of a large cluster of CPUs. Lower priority jobs are preempted as necessary so that compute resources can be allocated to the new, high priority jobs, and the input data for the new jobs is transferred to the execution sites before they begin execution.

A large cluster of jobs completes. This is also a frequent occurrence in high throughput computing environments, when the last work-step is processed in a master-worker application and all of the workers exit. Similarly, in high performance environments, the completion of a large parallel job can trigger significant scheduling activity. The output from the completed jobs is transferred from the execution site(s) to its final destination, and queued jobs are scheduled on the newly available compute resources.

A large number of CPUs join or leave the pool. In a cluster of workstations environment, large clusters of CPUs may join or leave the pool of available resources as a result of external usage. For example, all workstations in a classroom or laboratory may become unavailable to batch jobs as students login at the start of class and become available again at the end of class. Additionally, large system maintenance events, such as scheduled system upgrades, network or power outages, or upgrades to the batch scheduling environment itself can cause these large events. When CPUs become available, the scheduler attempts to start jobs on those CPUs as quickly as possible. The input data for the scheduled jobs is transferred to the execution sites before the jobs begin computing. When CPUs become unavailable, the result is either a large number of checkpoint events or a large amount of work lost by jobs unable to checkpoint. In either case, the preempted jobs return to the queue and may immediately preempt a large number of lower-priority jobs running at other compute sites.

Taken individually, job placements may not impose a significant load on the network. However, when these large scheduling events occur, the large number of job placements can dramatically oversubscribe network resources. Some mechanism of throttling the scheduler is required to manage these events. Simple controls, such as configured limits on the number of simultaneous job placements, can be inadequate in large, dynamic, heterogeneous systems. Network admission control provides a mechanism for controlling network load directly.

The following example illustrates the benefits of admission control in the matchmaker during large scheduling events. Our experimental setup contains 16 dual-processor machines and a checkpoint server connected by a switched, private, Fast Ethernet. 32 SimpleScalar [11] simulation jobs are waiting in the queue with checkpoints stored on the checkpoint server when the 32 CPUs become available. These jobs are taken from examples found in the University of Wisconsin-Madison Computer Sciences Condor pool. 16 of the jobs are running the SPEC95 mgrid benchmark and have 92 MB checkpoint files, and the other 16 jobs are running the SPEC95 compress benchmark and have 278 MB checkpoint files. When the jobs are scheduled with admission control disabled, all 32 jobs are scheduled immediately and begin transferring their checkpoints from the checkpoint server simultaneously. The mgrid jobs complete their checkpoint transfers first and begin running in approximately 4 minutes, and the compress jobs complete their checkpoint transfers and begin running in about 8 minutes. When the jobs are scheduled with admission control enabled, only a small number of transfers are scheduled simultaneously, so the first jobs begin running within 32 seconds. The admission controlled schedule does not fully utilize the network capacity due to the scheduling granularity of the Condor matchmaker, so the final 3 jobs begin running later in the controlled case than they did in the uncontrolled case. However, the benefit of starting most of the jobs earlier results in an overall increase in goodput. The start times for the jobs are plotted in Figure 21. The area under each curve is the total goodput obtained by the jobs with and without admission control enabled. Since most of the jobs start running earlier in the admission controlled case, the area under the admission controlled curve is greater until beyond the 8 minute mark where the lines cross because the admission controlled case does not fully utilize the network. The overall area in the admission controlled case is greater, however, resulting in over 70 CPU minutes of additional delivered goodput.

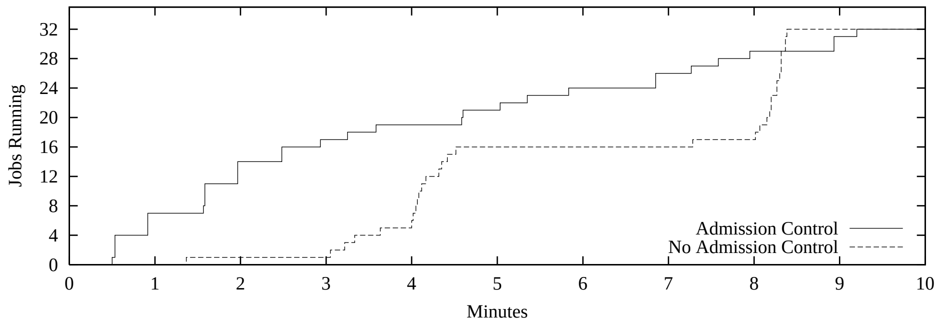

Figure 21: Allocating Network Capacity

```
  // request network capacity for job placement
  // returns 2 if capacity is available to this user,
  //         1 if user’s fair-share is already allocated
  //            but capacity is otherwise available,
  //         0 if the network is already over capacity, and
  //        -1 on error
int RequestPlacement(ClassAd request, ClassAd offer,
                     ClassAd preemptedAllocation);

  // commit last placement request (and charge for usage)
  // returns 0 on success and -1 on error
int CommitLastPlacementRequest();
```

Figure 22: Admission Control Module Interface

6.2.1 Implementation

The *admission control module* implements network allocation and accounting in the matchmaker. The interface to this module is shown in Figure 22. The `RequestPlacement` method computes the network capacity required to satisfy the job manager’s request with the compute server’s offer, including the capacity required to preempt current allocations when necessary, from the provided ClassAd attributes. The `CommitLastPlacementRequest` method commits the last placement request and charges the requester for the usage. The commit interface allows the matchmaker to test the feasibility of multiple matches before committing an allocation when searching for the best match according to the requester’s preferences.

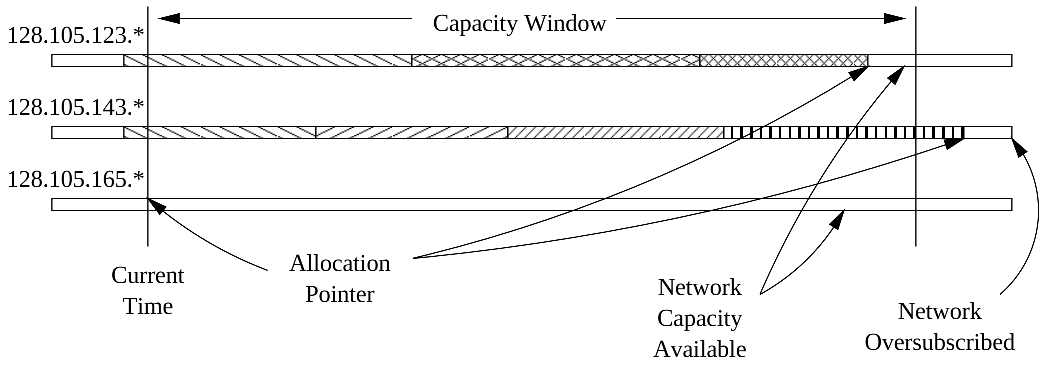

Figure 23: Allocating Network Capacity

6.2.1.1 Capacity Allocations

The network manager associates a capacity with each network resource (for example, 10 Mbps) and defines a time window in which the matchmaker may allocate network capacity (for example, 30 seconds). The window defines the matchmaker's scheduling horizon by restricting the matchmaker to performing only those allocations that can be supported by the network in that window. Defining the window equal to the matchmaker's scheduling interval has the natural result that the matchmaker postpones scheduling decisions until the interval in which they can be implemented, thereby enabling the matchmaker to effectively incorporate new information into its scheduling decisions at the start of each new scheduling period by not committing network resources far into the future in previous scheduling periods.

The admission control module determines if network capacity is available in the current allocation window using the following mechanism. The network map contains an "allocation pointer" for each network resource that tracks previous allocations. Each allocation moves the allocation pointer forward by the allocated capacity in units of time. For example, satisfying a request to transfer 100 MB over a 10 Mbps link would move the pointer forward $100\ MB * \frac{8\ Mb}{1\ MB} * \frac{1\ second}{10\ Mb} = 80\ seconds$. The pointer never falls behind the current time: an unallocated network resource has a pointer equal to the current time. Figure 23 illustrates the interaction between the allocation pointer and the capacity window. Capacity is allocated until the pointer passes beyond the current capacity window. After that point, no additional capacity is allocated until sufficient time passes for the window to shift forward to again include the pointer inside its boundaries.

We allow the pointer to pass beyond the end of the window for the final allocation to avoid fragmentation problems. Otherwise, the end of the window would frequently remain empty when remaining large requests would not fit in the available space. This also allows the system to support requests larger than the capacity window. A maximum request size can be configured as a sanity check to ensure that a single large request does not starve other requests for a long period of time.

```
  // returns 0 on success, -1 on failure
int LoadRoutingTable(const char filename[]);

  // sets current to point to the next network hop
  //   in the route between src and dest
  // returns 1 if current is an intermediate hop
  //         0 if current is the final hop (current == dest)
  //        -1 on error
int NextHop(unsigned int &current, unsigned int src,
            unsigned int dest);
```

Figure 24: Router Module Interface

6.2.1.2 Routing

To calculate the network capacity required for a flow, the admission control module must determine the network route of the flow between the endpoints. The *router module* loads a routing table from a configuration file and calculates the routes of network flows according to the configuration on behalf of other modules. The format of the route configuration file is described in Appendix A.3. The router module interface is shown in Figure 24.

6.2.2 Goodput Allocations

In some cases, minor deviations from a strict priority-based allocation of CPU and network resources can significantly improve the total goodput delivered by the system, especially when user priorities vary significantly. In particular, a heavy network user with high priority can potentially monopolize network resources to such an extent that a large number of CPUs are left idle because of insufficient network capacity to place jobs on them.

For example, consider a 128 CPU cluster served by an NFS server with 200 Mbps I/O bandwidth, where local users have strict priority over guests. A local user's jobs perform transformations on 20 MB image files. Each job reads an image file and produces 5 new images of the same size, requiring 60 CPU seconds to compute each new image from the input, so each job performs 120 MB of NFS I/O every 5 minutes (3.2 Mbps). The scheduler can successfully overlap I/O with computation for at most 62 of these jobs. If it runs additional jobs, the file server's bandwidth will be oversubscribed and the CPUs will be underutilized. Meanwhile, a guest user has a large number of compute-intensive jobs in the queue. Running the guest's jobs on 66 of the CPUs can double overall system goodput with minimal impact on the high priority jobs. However, with strict priority-based network allocation, the guest user's jobs will never start because the user will not be allocated any network capacity. We need to give the scheduler some discretion to deviate from strict priority-based allocation when it can significantly improve goodput.

As a second example, consider the same 128 CPU cluster served by a checkpoint server with

200 Mbps I/O bandwidth. The job scheduler was disabled on the cluster for a day while researchers benchmarked a distributed database system. A local user has a set of jobs in the queue, each with 256 MB checkpoint files, and the same guest user from the previous example has a set of compute-intensive jobs in the queue. If the job scheduler follows strict priority-based allocation of network resources when it is restarted, it will schedule the local user's jobs on the 128 CPUs. It will take approximately 22 minutes to load the 128 256 MB checkpoints from the checkpoint server to the compute nodes. If the scheduler performed those transfers simultaneously, the 128 CPUs would sit idle for those 22 minutes, for a loss of over 46 hours of potential computing capacity. With admission control, the scheduler can do much better by performing the transfers sequentially, so the first job begins computing in approximately 10 seconds, the second job in 20 seconds, and so on, limiting the lost CPU time to 23 CPU hours. However, if the scheduler can deviate from strict priority-based allocation, it can allocate some of those 23 CPU hours to the guest's jobs without significantly increasing the startup delay for the local user's jobs.

These examples illustrate that it is useful to give the matchmaker discretion to deviate from strict priority-based network allocation. We therefore allow the matchmaker to use some network capacity for the purposes of improving overall system goodput when the network is oversubscribed. The matchmaker uses the capacity to run jobs with low network requirements ("backfill jobs") on CPUs that would go idle if strict priority-based network allocation were enforced. We have implemented this discretionary mechanism in the matchmaker with two configuration parameters. The first parameter controls which jobs qualify as backfill jobs according to a network usage ceiling. The second parameter controls how much discretionary network capacity is available to the matchmaker. Limiting the amount of discretionary capacity controls how much the matchmaker can deviate from strict priority-based allocation, so high-priority users with heavy network requirements will not be starved.

6.2.3 Fair Allocation

Network and CPU resources are allocated fairly in the matchmaker according to user priorities as follows. Each user's fair-share of network and CPU resources is based on the user's *base priority* and recent usage of that resource. The base priority defines the relationship between the users. For example, users with base priority of 1.0 should receive twice as many resources as users with base priority of 2.0 and three times as many resources as users with base priority of 3.0. The allocation algorithm gives requesters their fair-share of CPU resources unless they are using more than their fair-share of network resources. Those using more than their fair-share of network resources must wait until other user's requests have been satisfied. The algorithm has two phases. Each phase allocates network and CPU capacity to requesters according to their CPU fair-share. At each iteration, the algorithm attempts to allocate an additional CPU to the most deserving requester among the users with outstanding requests (i.e., the user with the greatest difference between CPU fair-share and current allocation). If a request can not be satisfied because of insufficient CPU or network resources, it is discarded (until the next run of the algorithm). As requests are discarded, the number of requesters decreases.

In the first phase of the algorithm, requesters are limited to their fair-share of each network resource. The fair-share is calculated according to the base priorities of the outstanding requesters

```
Run analysis summary.  Of 32 resource offers,
    0 do not satisfy the request’s constraints
    0 resource offer constraints are not satisfied by this request
    7 are serving equal or higher priority customers
    0 do not prefer this job
    0 cannot preempt because PREEMPTION_REQUIREMENTS are false
   25 are available to service your request
Last successful match: Sat Mar 31 12:07:47 2001
Last failed match: Sat Mar 31 12:27:43 2001
Reason for last match failure: insufficient bandwidth
```

Figure 25: Scheduler Diagnostics Example

and their past network usage. As the number of requesters decreases, the relative share of each remaining requester increases. Heavy network users are penalized by this mechanism, allowing users with lighter network requirements to obtain allocations sooner. It is possible for all remaining requests to require more than the requester’s fair-share of network resources. In that case, the algorithm proceeds to the second phase, where the network fair-share limitation is removed.

The first phase of the algorithm includes both a test for available network capacity and a test for fair-share allocation. The fair-share test can potentially ensure that the network is not overallocated on its own, making the capacity test superfluous. However, in some cases the capacity limits and fair-share limits may not be directly comparable. For example, fair-share may be calculated over a longer interval than that used by the network capacity controls, to allow short-term overallocation to avoid fragmentation problems (as discussed in Section 6.2.1.1).

The network allocation required to satisfy a request depends on the execution site. Execution sites are located on different network segments and may require different data access methods. For example, if the job’s data is replicated, the nearest copy of the data will depend on the execution site. Both endpoints of data transfers can be located on different networks depending on the execution site chosen. For this reason, the algorithm must proceed to consider other matching CPUs when network capacity is unavailable to match a request with a previous CPU.

6.2.4 Diagnostics

One of the important lessons we learned from deploying network allocation in the Condor environment is the importance of providing feedback to help users answer the question “Why isn’t my job running?”. Condor users have been trained to expect a fair-share of CPU resources. When their jobs are being restricted by bandwidth limitations, they receive less than their CPU fair-share and want to know why lower priority users are getting more CPU resources than they are. We modified Condor’s implementation of the matchmaking protocol to return a description of the reason for each failed match. Users can view the reasons for each job by issuing a query to the job manager. Figure 25 shows output from an example query. The first section of the output lists the standard Condor diagnostics for the CPU allocation: how many CPUs do not satisfy the job’s request, how

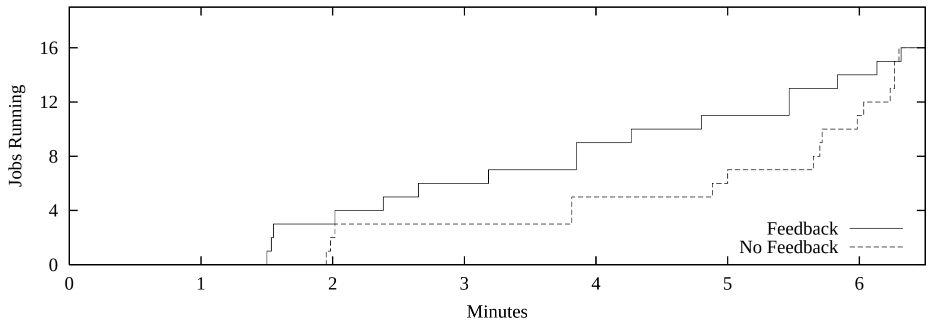

Figure 26: Job Starts With and Without Feedback

many CPUs are not willing to run the job, how many are serving higher priority customers, etc. The timestamps indicate when the matchmaker last attempted to find matching resources for the job's request. The reasons for match failure now reported include insufficient bandwidth, network share exceeded, insufficient CPU priority, and no matching resources found, so in the above example, the user can see that the job hasn't started yet, even though there are available CPUs, because of insufficient network bandwidth.

6.3 Network Manager

We have seen in the previous section that the matchmaker implements network admission control at the granularity of job placements. Once the matchmaker completes a match with available network resources, the job manager contacts the network manager to receive additional network scheduling services, including requests for additional network capacity, future reservations, and bandwidth control of active network streams.

The job manager also keeps the network manager informed of its jobs' network usage whenever possible. This helps the network manager determine how much network capacity is available to be allocated to other jobs, either in the network manager itself or in the matchmaker, as illustrated by the following example. 16 SimpleScalar compress jobs are waiting in the queue, in the same experimental setup used previously, when 16 CPUs become available. The network manager is configured to expect 160 Mbps available throughput from the checkpoint server, even though the server is on 100 Mbps Ethernet. Without feedback, the matchmaker continues to allocate bandwidth at 160 Mbps, starting new checkpoint transfers before the previous transfers have completed. When feedback is enabled, the network manager detects that the checkpoint transfers are taking longer than expected and backs off its allocations to the actual available bandwidth. As seen in Figure 26, the jobs start running earlier when feedback is enabled, delivering over sixteen additional CPU minutes to the jobs (i.e., the difference in area under the two curves).

The network manager can be effective even when its knowledge of available network capacity is limited. Simple static control policies configured by an administrator can avoid costly scheduling

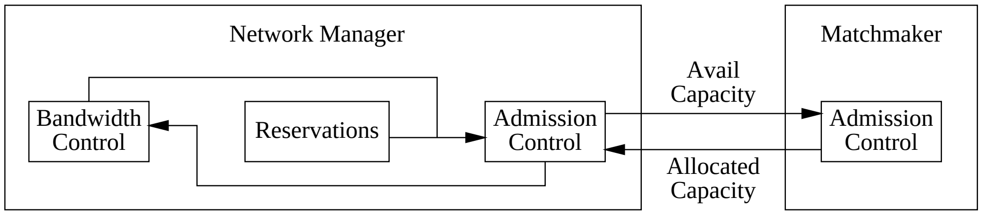

Figure 27: Network Allocation Architecture

decisions in the matchmaker, and simple feedback from the job managers concerning the completion time of job placement transfers can help the network manager back-off network allocations when less network capacity than expected is available. However, the network manager can take advantage of additional information when it is available. For example, an external network weather service [75] can provide dynamic predictions of available network capacity to the network manager, and interposition agents [31], such as Bypass [67, 68] or Condor's remote system call mechanism [41], can provide additional information about job behavior, including network usage.

Given some knowledge about available network capacity (however limited), the network manager must decide how that capacity should be distributed. Network allocation responsibilities are divided between the matchmaker and multiple schedulers in the network manager, as illustrated in Figure 27. Configured limits ensure that each admission control module does not monopolize network resources. First, advance reservations are restricted to a configured percentage of current available capacity (for example, 40%). Advance reservations provide only a limited guarantee because of competing network traffic external to the system. Therefore, this limit also serves to make advance reservations more conservative, so there is a greater probability that the reserved capacity will in fact be available when the time comes. Second, bandwidth controlled allocations are guaranteed a minimum percentage of available capacity (for example, 10%). Capacity is available to the admission control modules to allocate if it is not reserved by the reservation module or allocated from the bandwidth control module's guaranteed capacity. With the above example limits, the admission control module will be able to allocate at least 50% of available network capacity. The bandwidth control module can then allocate any remaining unallocated capacity to active streams.

6.3.1 Reservations

The network manager's *reservation module* implements a slot scheduler [20, 74] to allocate advance reservations of network resources. The reservation interface supports both capacity and bandwidth requests. A capacity request is a request to transfer a fixed amount of data between a source and a destination, and a bandwidth request is a request to transfer data over a fixed time interval (for example, during a job's run). Capacity reservations are used for bulk data transfers while bandwidth reservations are used for ongoing network streams, such as remote I/O streams or inter-task communication. Requests include time and rate constraints and scheduling preferences. The slot scheduler searches the allocation schedule for each network resource in the route between the source and destination of the requested flow for the best reservation (according to the indicated preferences) that

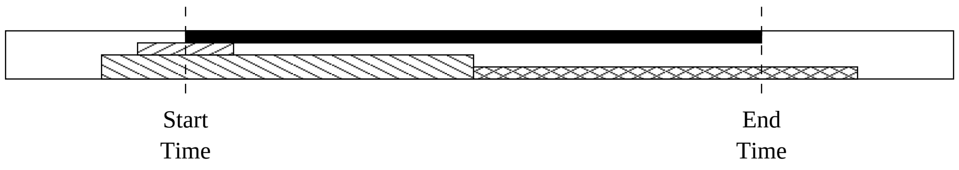

Figure 28: Slot Scheduling Example: First Fit

satisfies the request's constraints. The resulting reservation is defined by a start time, end time, and a rate. Clients that require allocations that vary in rate must either make multiple reservations or must use the bandwidth control module, described below in Section 6.3.2.

The slot scheduler supports the following search strategies for implementing reservation preferences.

- **First Fit** finds the reservation with the earliest start time and **Last Fit** finds the reservation with the latest end time, within the specified time constraints. These search strategies are acceptable for reservations that are not time critical. For example, the request may be for a "background" data transfer that need not be completed quickly, so long as it completes before the specified end time.

- **Earliest Completion** finds the reservation with the earliest end time and **Latest Start** finds the reservation with the latest start time, within the specified time constraints. These search strategies are used for time critical transfers that should be completed as quickly as possible after an event or should be completed as late as possible before a deadline, for example, when scheduling checkpoint transfers before a scheduled eviction deadline, as discussed below in Section 6.3.1.1. Starting the checkpoint as late as possible maximizes the job's compute time before it must stop and perform the checkpoint.

- **Shortest Duration** finds the (earliest or latest) reservation with the shortest duration (i.e., highest rate). This strategy is useful for blocking transfers with flexible time constraints, to minimize the blocking transfer time. For example, if checkpoint transfers are a blocking operation, scheduling periodic checkpoints using shortest duration reservations can minimize the checkpoint time.

Figure 28 illustrates an example first fit reservation (the solid black bar). 75% of network capacity is reserved by two previous requests at the start time, and the current request can be satisfied with the remaining 25% of network capacity inside the time boundaries. Assuming this reservation also meets the request's minimum rate requirement, it is an acceptable allocation.

It is clear, however, that additional capacity is available to improve the performance of this transfer. Figure 29 illustrates the results of the same request with a preference for the earliest transfer completion time. The start of the transfer is delayed until the end of one of the other reservations so 50% of network capacity can be allocated to it. The doubling of the speed of the transfer more than compensates for the later start time, yielding an earlier completion time.

Figure 30 shows the reservation that would be found in the same example as above with the shortest duration strategy. The start of the transfer is delayed still further, and the resulting rate

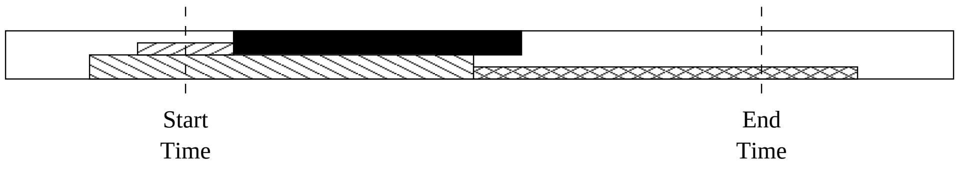

Figure 29: Slot Scheduling Example: Earliest Completion

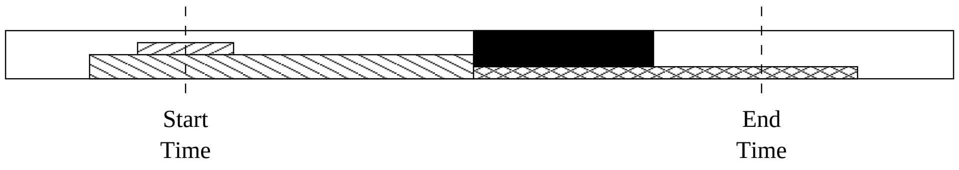

Figure 30: Slot Scheduling Example: Shortest Duration

increase is not sufficient to result in an earlier completion time than in the previous figure.

For the sake of completeness, we include a “greedy” reservation in Figure 31. This type of reservation is not supported by the slot scheduler, due to the fixed rate requirement. Greedy bandwidth allocations are supported in the bandwidth control module, as described below in Section 6.3.2.

The slot scheduler is implemented with a list of reservation markers (a begin and end marker for each reservation) sorted by time for each network resource. Each marker records a change in the reserved bandwidth. Begin markers record an increase in the reserved bandwidth (i.e., a positive delta) and end markers record a decrease in the reserved bandwidth (i.e., a negative delta). Inserting n reservations is therefore an $O(n^2)$ operation. Figure 32 plots the worst-case performance of the slot scheduler on a 200 MHz Pentium Pro workstation for inserting up to 2000 capacity reservations, where the scheduler must search to the end of the list to satisfy each reservation. For multi-hop reservations, the slot scheduler searches the reservation list for each hop, so increasing the number of hops increases the search time by a constant. The scheduler inserts 1000 three hop reservations in about one second. The performance of this simple scheduler is acceptable for moderate workloads. We could use tree data structures to improve the scalability of the slot scheduler if needed [60].

6.3.1.1 Scheduled Shutdown Events

The initial motivation for developing the slot scheduler was to implement scheduled shutdown events in the Condor environment. As described above, when CPUs leave the pool, the jobs running

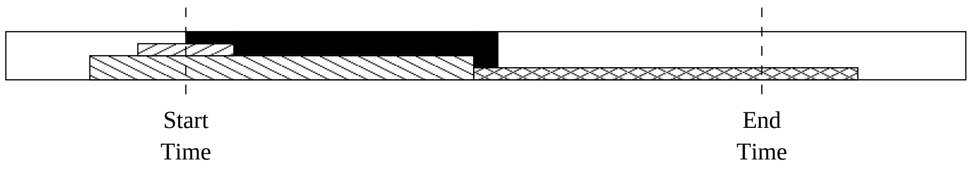

Figure 31: Slot Scheduling Example: Greedy Reservation

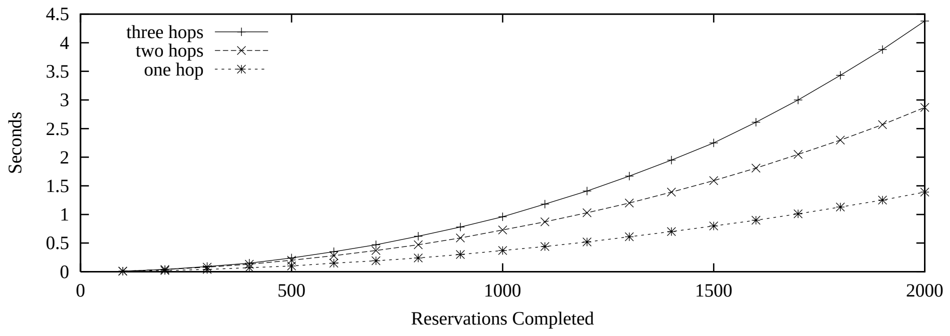

Figure 32: Slot Scheduler Performance

on those CPUs must be evacuated from the execution sites. For checkpointable jobs, this requires transferring the jobs' checkpoints to stable storage before the evacuation. The two causes of large shutdown events in the University of Wisconsin-Madison Computer Sciences department Condor pool are scheduled system reboots and Condor system maintenance. Lab workstations are rebooted each night to cleanup any orphaned user processes resulting from that day's activity, and workstation reboots are frequently scheduled to install system patches. The Condor system is shutdown for software upgrades or scheduled server maintenance. Condor has the ability to restart itself on each machine when a new version of the software is released. However, these restarts can generate a large number of simultaneous checkpoint transfers because running jobs must be evicted before each compute server can restart. Scheduling these checkpoint transfers ensures that all jobs are checkpointed before a shutdown deadline. Scheduling can also improve aggregate delivered goodput by scheduling fewer simultaneous checkpoints, so while the first few jobs are checkpointing, other jobs can continue computing until their scheduled checkpoint time.

A shutdown event scheduler called the *eventd* schedules the checkpoints for these large shutdown events. Each scheduled shutdown event is specified in a configuration file with a start time, duration, and constraint. The eventd schedules job preemptions for these events so there will be no jobs running on execution sites that match the constraint at the event start time. As the event approaches, the eventd reserves network capacity to transfer checkpoints for all running jobs. First, it computes the duration of each checkpoint transfer based on the checkpoint size, the route between the execution site and the checkpoint server, and the capacity of each network resource in the route. Then, for each transfer, from shortest to longest, it makes a *latest start* reservation request. Scheduling the longer transfers earlier keeps the largest number of jobs running as long as possible up to the shutdown event. Once the initial reservations are established, the eventd monitors the state of the pool until the time for the first transfer arrives. During this time, it cancels reservations for jobs that complete and makes new reservations for new jobs that start running. When the time for the first checkpoint arrives, the eventd configures all execution sites that match the constraint to no longer start checkpointable jobs because network capacity will not be available to checkpoint their work. Without this step, jobs might be scheduled again at the execution sites after the eventd checkpoints them. The eventd then initiates the job checkpoints according to the scheduled reservations. When the start of the event arrives, the eventd preempts all non-checkpointable jobs and configures all

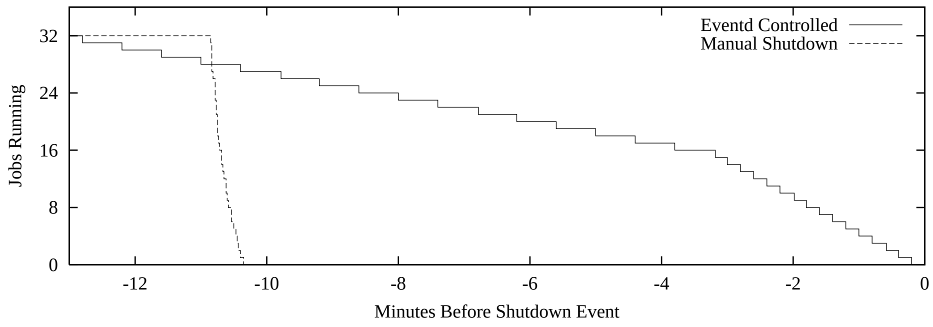

Figure 33: Shutdown Before Deadline

execution sites that match the constraint to not run jobs until the end of the shutdown event.

The following experiment demonstrates the eventd's effectiveness. 32 SimpleScalar jobs are running, and 32 CPUs are reserved for a timing experiment starting at time 0. To manually evict the jobs before the experiment, the administrator estimates how long it will take for all jobs to checkpoint and sends shutdown commands to all of the jobs in advance of the deadline. In our experiment, it took about 11 minutes for all jobs to checkpoint when manually evicted. Figure 33 shows the manual shutdown case assuming the administrator made a perfect estimate of the shutdown time. All jobs stop running approximately 11 minutes before the deadline to write their checkpoints. The results under an eventd controlled shutdown are also shown in the figure. The eventd schedules the checkpoints in advance of the deadline using *latest start* reservation requests. All checkpoint transfers share a bottleneck at the checkpoint server, so the reservations do not overlap. Since the eventd sorts the reservation requests from shortest to longest, the mgrid jobs, with their 92 MB checkpoints, get the reservations closest to the deadline and the 278 MB checkpoints for the compress jobs are performed first. The eventd leaves slack in the schedule in case there is competing network traffic or some estimation error in the transfer times, so the network is not fully utilized by the eventd's schedule. Comparing the area under the two curves in Figure 33 shows that the eventd's schedule results in over 3 additional CPU hours delivered to the jobs. In addition to the increase in goodput, the eventd evicts the jobs automatically before the deadline, so the administrator need not manually estimate the time required to shutdown the jobs before the deadline.

Scheduling the checkpoints sequentially would be less useful if the jobs pre-copied their checkpoints while they continued execution. However, scheduling the checkpoints before the shutdown event would be equally beneficial, so the jobs can checkpoint successfully and avoid a rollback. Capacity for two checkpoint transfers per job would be required. First the job would write its full checkpoint while it continues running. When it completes the first checkpoint transfer, it would then perform a blocking transfer of any modified state. Scheduling the checkpoints close to the event would continue to be the most efficient approach, as it would minimize the run-time after the start of the first checkpoint transfer, thereby minimizing the amount of modified state that would need to be transferred later. Therefore, the larger checkpoint transfers would still be initiated before the smaller checkpoint transfers. The schedule for the first transfers would need to leave some slack for the second transfers to complete before the deadline. The most conservative approach would

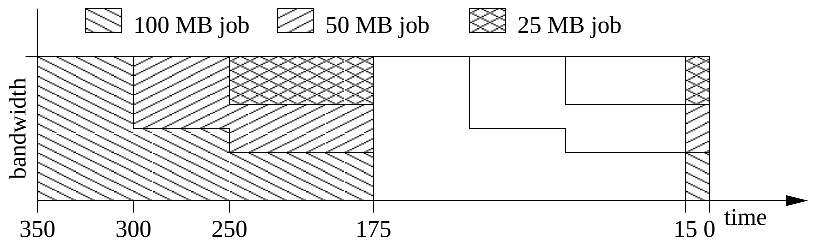

Figure 34: Shutdown Event With Pre-Copying

be to assume the worst case: that all data would be modified and need to be re-checkpointed. The jobs can then transfer the modified data at the end of the second checkpoint reservation, according to how much data must actually be sent.

As a simple example, consider three jobs to checkpoint over Ethernet before a deadline, with 100 MB, 50 MB, and 25 MB checkpoints. The total transfer time for the 175 MB of data would be approximately 175 seconds (at 1 MB/s), so the eventd would schedule the first round of checkpoints to start 350 seconds before the deadline, leaving 175 seconds of slack for the final checkpoints. The 100 MB transfer would be scheduled to begin 350 seconds before the deadline, the 50 MB transfer 300 seconds before, and the 25 MB transfer 250 seconds before, as illustrated in Figure 34. Each job has a second reservation of equal shape for the second checkpoint. However, each job only modifies 5 MB of its state after the checkpoint, so they each delay their final checkpoint transfers until 15 seconds before the deadline.

In practice, we need not be so conservative when reserving bandwidth for the second checkpoint. The size of the second checkpoint is a function of the job's working set and memory reference rate, and we expect it to be small. We further expect that the size of the second checkpoint could be accurately predicted based on past job behavior. Finally, the cost of failing to perform the second checkpoint is very low, because the job can always rollback to its first checkpoint without losing much work.

6.3.2 Bandwidth Control

The network manager also supports fine-grained bandwidth control for long-lived network streams. Requesters register a bandwidth request for each active stream and provide frequent feedback to the network manager regarding their network usage and needs. The network manager allocates available bandwidth to the registered streams according to a max-min fair share algorithm, modified from [43]. This supports the notion of a *nice* stream, tolerant of short-term bandwidth fluctuations, that the network manager can control to improve the performance of time-critical network flows. Examples include spooled output, network backups, or bulk transfers with deadlines far in the future. The network manager sends a message to a requester whenever it needs to adjust the requesters sending rate, either because of a change in available bandwidth (due to higher priority flows) or a change in the bandwidth requests.

6.4 Related Work

The Globus Architecture for Reservation and Allocation (GARA) [6, 25], developed concurrently with our work, defines a general-purpose framework for co-allocation of different types of resources, including CPU and network. The architecture provides APIs for discovery, reservation, allocation, and monitoring of generic resource objects. Co-reservation agents use these APIs to search for sets of resources that satisfy their applications' requirements and then reserve the resources. Clients can register callbacks for event notification on resource objects, for example when packet losses are detected. GARA uses a resource-neutral slot manager to implement reservation services when the underlying resource manager for a given resource does not support reservations.

GARA provides mechanisms for performing co-allocation and relies on external agents to use the mechanisms intelligently. In contrast, our co-allocation mechanism leverages the monitoring and control provided by Condor's remote system call and checkpointing mechanisms to transparently manage applications' network requirements. We believe that our approaches for network and CPU co-allocation in high throughput computing environments can be used in the future design of co-allocation agents in the GARA framework.

We have developed and evaluated our network allocation and reservation mechanisms using best-effort network services. The IETF integrated services [61, 77] standards provide end-to-end quality of service (QOS) to flows on IP networks. QOS reservations are established using RSVP [9, 78]. Scalability, security, and policy control for these services are active areas of study [44] and deployment is in the experimental stage. The differentiated services [8] framework addresses scalability issues present in the integrated services standards by using no per-flow state in routers. Instead, bandwidth brokers [29, 60] allocate and police bandwidth at the network edges, mapping packets into a small number of configured per-hop service behaviors in the network. In networks where bandwidth brokers are deployed, our network manager can serve as an interface between the job scheduler and the broker, exporting the bandwidth broker's QOS service to the batch system.

Network and CPU co-allocation is a form of load control that avoids network *thrashing*, where jobs spend a significant amount of their time blocked on network transfers because the network is oversubscribed. Our co-allocation techniques and results are similar to previous techniques for avoiding virtual memory thrashing. LT/RT (Loading Task / Running Task) control in the WS-CLOCK algorithm [13] limits the number of tasks being concurrently paged in to reduce contention for paging bandwidth, just as network and CPU co-allocation limits the number of job placements. Carr and Hennessy observe that contention for paging bandwidth causes task loads to take longer, with the potential for a cascade effect as more jobs complete their time slices, resulting in virtual memory thrashing. LT/RT control is shown to have a number of benefits, including more effective memory utilization, since fewer memory pages are committed to loading tasks, and improved processor utilization by balancing the number of loading and running tasks. We obtain similar results with network and CPU co-allocation, improving network and CPU utilization by balancing the number of loading and running jobs. Carr and Hennessy also observe that delaying the activation of additional tasks allows the system to predict which tasks will fit in memory more accurately because the memory needs of previously loaded tasks can be taken into account when deciding to load later tasks, just as delaying job placements until network capacity is available allows the scheduler to take the current state of the distribution system into account when making future placement decisions.

Carr and Hennessy propose two optimizations which also have analogies to network and CPU co-allocation. First, when tasks reach the end of their time slice, they should not be deactivated until their memory can be allocated to a loading task. This is analogous to delaying the preemption of a lower-priority job to run a higher-priority job until sufficient network capacity is available to perform the preemption efficiently. Second, process page reads for running tasks should have priority over reads for loading tasks, so running tasks continue executing efficiently. Our network manager similarly prioritizes network allocations for running jobs over network allocations to loading jobs.

6.5 Summary

We have presented a framework for co-allocating network and CPU capacity in job scheduling systems, based on the matchmaking framework used in the Condor High Throughput Computing environment. The framework co-allocates network and CPU resources for job placements and supports advance reservations and bandwidth control for active streams. We illustrated the effectiveness of the framework by implementing it in the Condor system and presenting the results of controlled experiments.

Chapter 7

Conclusion

7.1 Future Work

The research presented in this dissertation has focused on specific problems that occur in high throughput computing environments and mechanisms to solve those problems. There are more problems to solve and many ways in which this work can be extended. We discuss some possible areas of future work below.

7.1.1 Overlapping I/O with Computation

As we have seen, the Condor system currently does little to overlap I/O with computation. Progress to date includes the Kangaroo system and buffering in the remote I/O library. We have discussed some additional possibilities for I/O and CPU overlap in high throughput environments in this dissertation. In particular, there are opportunities to overlap checkpoint transfers with computation, when preempting one job to run another and by performing checkpoints asynchronously. Kangaroo servers can potentially be local spoolers for job checkpoints, so checkpoints can be performed at local disk speeds and spooled to central servers as bandwidth permits [51].

7.1.2 Compression

One approach for efficient checkpointing reduces the amount of checkpoint data to be transferred. Techniques for efficiently compressing checkpoints include saving only those memory pages modified since the last checkpoint [54]. Memory exclusion allows the application to specify ranges of memory which need not be saved across a checkpoint [52, 53]. The performance benefits of compression vary according to the relative speed of the compression algorithm compared to available I/O bandwidth. Compression can be particularly effective when many processors are checkpointing across a shared network, since the checkpoints can be compressed in parallel to better utilize the shared I/O bandwidth. An adaptive approach could choose the appropriate compression mechanism for a job at run-time, according to the available network capacity.

7.1.3 Goodput Scheduling

The goodput model gave us some intuition about network scheduling strategies in high throughput computing environments. Network wait time and checkpoint rollbacks are two of many factors that impact the performance of batch jobs. For example, we saw that Condor's suspend policy, where jobs are suspended at the first sign of workstation owner activity instead of being immediately preempted, can have a significant impact. It would be interesting to extend the goodput model to better understand the tradeoffs of the suspend policy so it can be applied more effectively. Additional evaluation of mechanisms to choose execution sites based on performance predictions would also be useful.

The expected goodput delivered by a given allocation depends on attributes of the job and the allocated resources. First, an estimate of the job's CPU, memory, and I/O requirements is needed. These requirements can then be compared with each compute site's available network capacity, available memory, CPU speed, (estimated) duration of allocation, and preemption policy. The goodput expected for a given job can vary for each compute site when the resource pool is heterogeneous. The expected goodput will be zero for some sites—for example, when there is a mismatch between the job's executable and the processor architecture at the compute site. The expected goodput will be low for sites with insufficient memory for the job because of expected virtual memory thrashing.

To choose the allocation that enables the job to accomplish the most work in the least amount of time, we want to maximize $\frac{goodput}{duration}$. The time required for job placement can be computed from the size and location of the job's input data and the available network capacity to the compute site. Likewise, the job cleanup time can be computed from estimates of the job's output and available network capacity. The allocation's maximum duration may be set by queueing system policy or may be estimated based on previous availability of the compute resource. If the allocation is preemptible, the duration estimate must also consider the preemption probability. If the job is expected to complete before the end of the allocation, then the expected duration can be calculated based on the job's expected run-time instead. A rollback factor must account for the possibility that some or all of the job's work will be lost due to preemption or failure at the compute site. The amount of work lost depends on whether the job is checkpointable, and if so, the frequency of periodic checkpointing and the probability that the job will not be able to checkpoint when preempted. Finally, the goodput estimate must include an estimate of the job's execution speed at the remote site, possibly calculated from previous job history and CPU benchmarks at the site. If the job's I/O can not be overlapped completely with computation during its run, the estimate must be decreased appropriately.

In practice, estimating many of these factors with sufficient precision may be impossible. However, we can draw several conclusions from the model that suggest simpler policies that can be applied in practice. All jobs prefer allocations with longer expected duration, better network performance to I/O servers, and faster processors. However, the trade-offs between these three factors can differ between jobs.

- Jobs that are not checkpointable have a strong requirement for an allocation with high probability of duration longer than the job's expected run-time.
- Alternatively, jobs that can checkpoint and migrate can tolerate shorter allocations, particularly when good network performance reduces migration costs.
- I/O-intensive jobs will place high value on the network capacity available throughout the allocation, and may be less concerned with small changes in CPU speed.
- In contrast, jobs with small I/O and migration requirements will place greatest value on CPU speed, since they can migrate to the best compute site with little overhead.

Given a basic understanding of the type of the job (ratio of computing to I/O, checkpoint size, etc.), users can choose simple metrics to request allocations more suited to the needs of their jobs.

7.1.4 Migrating to Improve Goodput

The benefits of job migration and the policies under which jobs should be migrated have been extensively studied (see, for example, [16, 17, 27, 32, 33]). We briefly discuss potential applications of the goodput framework to the job migration question here. The benefit of migrating to a site that promises better performance must be compared to the expected cost of migration. From the goodput perspective, the cost is the time the job spends transferring its state over the network to the new site and the benefit is the resulting improvement in goodput.

Migration can improve goodput when the new execution site has a faster CPU, more memory or network capacity, or a longer expected duration. The most important factor depends on the requirements of the job. A CPU-bound job can improve goodput by migrating to a site with a faster CPU. Likewise, a network I/O-bound job can improve goodput by migrating to a site with better network I/O performance. If the current allocation has a high probability of rollback (for example, because jobs are killed by the scheduler without warning when the resource is claimed by its owner), the job can improve expected goodput by migrating to a compute site with better service guarantees. To determine when it is advantageous to migrate, we can compare the expected remaining goodput of the current allocation with the expected goodput of a potential new allocation.

7.1.5 Crossing Administrative Domains

Our primary experience with these network allocation mechanisms has been in a single administrative domain. Many of the overheads of remote execution are exacerbated when crossing administrative domains, for the simple reason that network distances increase.

As described in Section 2.4.2, job managers in Condor support a simple mechanism called *flocking* for cross-domain job execution. When flocking, the job manager advertises resource requests to matchmakers in multiple administrative domains and runs jobs in the domain where available resources are found. Interfaces between Globus [21] and Condor have also been developed to allow Condor jobs to harness computational grid resources using Globus mechanisms and to make Condor resources available to Globus jobs. Condor's remote system call facilities are particularly useful for cross-domain job execution because they emulate the job's home environment at the remote compute site.

We have found checkpoint domains to be very effective at managing checkpoint traffic for cross-domain job execution. Jobs store their checkpoints on the checkpoint server local to the domain where they are running. Execution domain requirements are also effective in cross-domain execution for restricting the execution sites for a job according to the availability of data resources. However, execution domain preferences are less effective because they are evaluated locally in each Condor pool. Each matchmaker will find the most preferred execution site in its domain for the job, but the job manager must compare the results from multiple matchmakers to determine which domain provides the best resource.

To allocate or reserve network capacity across multiple administrative domains, the job manager must contact one or more network managers in each administrative domain. Mechanisms for co-allocating resources from multiple network managers would be useful in this case. One approach would be to interface Condor's network managers with the Globus Architecture for Reservation and Allocation described in Section 6.4.

7.1.6 Integration with Gang-Matching

As discussed in Section 2.4.1, we have strived to make our work compatible with the recently developed gang-matching model for resource co-allocation. We see two areas in which the gang-matching model must be extended to facilitate network and CPU co-allocation.

First, some support for partial allocation of resources is required, since many slices of network capacity are allocated in one matchmaking cycle. The gang-matching framework as currently formulated assumes that resources are completely consumed when matched, so shared resources must be partitioned into individual resource offers. However, it is not feasible to partition network resources in advance. Classad replacement, where resources offers can be partially allocated automatically in the matchmaking process, is cited as future work in [55] and should address this issue.

Second, the gang-matching framework does not currently support gangs of dynamic size. However, requiring fixed-sized gangs makes it difficult to allocate network capacity on intermediate network devices for a network flow. The endpoints for network capacity requests may be determined at match time, according to the chosen execution site, so intermediate network hops can not be precomputed by the requester. Likewise, pre-computing all n^2 routes for n endpoints in the system is not feasible—the routes should be computed at match time. Therefore, some mechanism for dynamically sized gangs is required.

7.2 Summary

This dissertation has presented the case for allocating network resources in batch job environments. We presented a *goodput* metric that compares a job's performance in the batch environment to its ideal performance using local, dedicated resources and suggested scheduling strategies based on this metric for different classes of batch jobs, according to the jobs' network requirements. We showed that batch jobs generate significant network load in the Condor pool in the University of Wisconsin-Madison Computer Sciences department and profiled the pool's capacity. In particular, we confirmed previous results that over 70% of workstation capacity goes unused by workstation owners and illustrated that harnessing the last 10% of idle capacity provided by short idle periods can account for over 50% of the job placement overheads in the system. We introduced *execution domains*, a mechanism to improve data locality in HTC environments by clustering compute nodes based on their access to network resources. Finally, we presented an implementation of network and CPU co-allocation in the matchmaking framework and demonstrated its ability to improve goodput by avoiding the oversubscription of network resources.

Bibliography

[1] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, 1997.

[2] J. Basney and M. Livny. Managing network resources in Condor. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, August 2000.

[3] J. Basney and M. Livny. Improving goodput by co-scheduling CPU and network capacity. *International Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[4] J. Basney, M. Livny, and P. Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. To appear in Computer Physics Communications, 2001.

[5] J. Basney, R. Raman, and M. Livny. High throughput monte carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. A distributed resource management architecture that supports advance reservations and co-allocation. In *International Workshop on Quality of Service*, 1999.

[7] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. IETF RFC 2475 (Informational), December 1998.

[9] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP) – version 1 functional specification. IETF RFC 2205 (Standards Track), September 1997.

[10] T. Brecht. An experimental evaluation of processor pool-based scheduling for shared-memory numa multiprocessors. In *Proceedings of the IPPS 1997 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 139–165. Springer Verlag, April 1997.

[11] D. Burger and T. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[12] P. Cao, E. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1995.

[13] R. Carr and J. Hennessy. WSCLOCK — a simple and effective algorithm for virtual memory management. In *Proceedings of the 8th Symposium on Operating System Principles*, volume 15 of *Operating Systems Review*, December 1981.

[14] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. To appear in Journal of Network and Computer Applications, 2001.

[15] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[16] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, pages 662–675, May 1986.

[17] D. Eager, E. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 662–675, May 1988.

[18] R. Feiertag and E. Organisk. The Multics input/output system. In *Proceedings of the 3rd Symposium on Operating System Principles*, pages 35–41, 1971.

[19] D. Feitelson and M. Jette. Improved utilization and responsiveness with gang scheduling. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[20] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. *Lecture Notes in Computer Science*, 1018, 1995.

[21] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[22] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[23] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 2001.

[24] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote i/o: Fast access to distant storage. In *Proceedings of the Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 14–25, 1997.

[25] I. Foster, V. Sander, and A. Roy. A quality of service architecture that combines resource reservation and application adaptation. In *Proceedings of the Eighth International Workshop on Quality of Service*, pages 181–188, June 2000.

[26] J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm. Technical Report ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, 2000.

[27] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), August 1997.

[28] H. Hellerman and H. Smith, Jr. Throughput analysis of some idealized input, output, and compute overlap configurations. *ACM Computing Surveys*, 2(2), June 1970.

[29] G. Hoo, W. Johnston, I. Foster, and A. Roy. QoS as middleware: Bandwidth reservation system design. In *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing*, pages 345–346, 1999.

[30] J. Howard. An overview of the Andrew file system. In *Proceedings of the Winter 1988 USENIX Conference*, pages 23–26, February 1988.

[31] M. Jones. Interposition agents: Transparently interposing user code at the system interface. *14th ACM Symposium on Operating Principles*, 27(1), December 1993.

[32] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *8th International Conference on Distributed Computing*, pages 123–130, June 1988.

[33] W. Leland and T. Ott. Load-balancing heuristics and process behavior. In *ACM SIGMETRICS*, volume 14, pages 54–69, 1986.

[34] J. Linderoth, S. Kulkarni, J. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 43–50, August 2000.

[35] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Conference Proceedings of the Usenix Winter 1992 Technical Conference*, pages 283–290, January 1992.

[36] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, Computer Science Department, University of Wisconsin-Madison, April 1997.

[37] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[38] J. Liu. A multilevel load balancing algorithm in a distributed system. In *Proceedings of the 19th ACM Annual Computer Science Conference*, page 670, March 1991.

[39] J. Liu. A model for job scheduling in a distributed computer network. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing (Vol. II): Technological Challenges of the 1990s*, pages 818–824, 1992.

[40] M. Livny. *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems*. PhD thesis, Scientific Council of the Weizmann Institute of Science, August 1983.

[41] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP Journal*, 11(1):36–40, June 1997.

[42] M. Livny and R. Raman. High-throughput resource management. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 13. Morgan Kaufmann Publishers, Inc., 1998.

[43] Q. Ma, P. Steenkiste, and H. Zhang. Routing high-bandwidth traffic in max-min fair share networks. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 206–217, August 1996.

[44] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O'Dell, A. Romanow, A. Weinrib, and L. Zhang. Resource reservation protocol (RSVP) version 1 applicability statement: Some guidelines on deployment. IETF RFC 2208 (Informational), September 1997.

[45] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for Unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[46] R. Metcalfe and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[47] M. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[48] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, July 1991.

[49] B. Ozden, A. Goldberg, and A. Silberschatz. Scalable and non-intrusive load-sharing in owner-based distributed systems. In *5th IEEE Symposium on Parallel and Distributed Processing*, pages 690–699, December 1993.

[50] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM symposium on Operating systems principles*, pages 79–95, 1995.

[51] J. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, pages 76–85, October 1996.

[52] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Usenix Winter 1995 Tech. Conf.*, pages 213–223, January 1995.

[53] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software – Practice and Experience*, 29(2):125–142, 1999.

[54] J. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.

[55] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin-Madison, 2001.

[56] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high-throughput computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[57] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2:129–138, 1999.

[58] R. Raman, M. Livny, and M. Solomon. Resource management through multilateral matchmaking. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, August 2000.

[59] A. Romanow and S. Floyd. The dynamics of TCP traffic over ATM networks. In *Proceedings of the SIGCOMM Conference*, pages 79–88, 1994.

[60] O. Schelén, A. Nilsson, J. Norrgård, and S. Pink. Performance of QoS agents for provisioning network resources. In *Proceedings of IFIP Seventh International Workshop on Quality of Service*, June 1999.

[61] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. IETF RFC 2212 (Standards Track), September 1997.

[62] J. Shoch and J. Hupp. Measured performance of an ethernet local network. *Communications of the ACM*, 23(12):711–721, December 1980.

[63] L. Smarr and C. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[64] R. Stevens, P. Woodward, T. DeFanti, and C. Catlett. From the i-way to the national technology grid. *Communications of the ACM*, 40(11):31–60, November 1997.

[65] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 1992.

[66] D. Thain, J. Basney, S. Son, and M. Livny. The Kangaroo approach to data movement on the grid. August 2001.

[67] D. Thain and M. Livny. Bypass: A tool for building split execution systems. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, August 2000.

[68] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, Spring 2001.

[69] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-system. In *10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[70] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9:105–134, 1995.

[71] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the Globus Data Grid. In *Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID 2001)*, pages 106–113. IEEE Computer Society Press, May 2001.

[72] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the SUN network file system. In *Proceedings of the Winter Usenix Conference*, 1985.

[73] J. Wang, S. Zhou, K. Ahmed, and W. Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, April 1993.

[74] L. C. Wolf, L. Delgrossi, R. Steinmetz, and S. Schaller. Issues of reserving resources in advance. *Lecture Notes in Computer Science*, 1018, 1995.

[75] R. Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, August 1997.

[76] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared Unix systems. Technical Report CS98-602, Computer Science Department, University of California at San Diego, October 1998.

[77] J. Wroclawski. Specification of the controlled-load network element service. IETF RFC 2211 (Standards Track), September 1997.

[78] J. Wroclawski. The use of RSVP with IETF integrated services. IETF RFC 2210 (Standards Track), September 1997.

[79] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 13–24, November 1987.

[80] S. Zhou and T. Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 133–142, May 1991.

[81] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load-sharing facility for large heterogeneous distributed computing systems. *Software – Practice and Experience*, 23(2):1305–1336, December 1993.

Appendix A

Network Management Library

A.1 Introduction

The network management library contains the following classes:

- Router: enumerates the network hops between two addresses
- NetworkCapacity: tracks the capacity of network segments
- NetworkCapacityAllocator: allocates aggregate network capacity
- NetworkBandwidthAllocator: allocates bandwidth (FCFS)
- NetworkShareAllocator: allocates bandwidth (max-min fair-share)
- NetworkReservations: allocates future bandwidth
- NetworkUsage: tracks usage per subnet over an interval
- NetworkUsageAllocator: fair-share network allocation interface
- NetworkManager: ties all the classes together for use in Condor

The NetworkAllocator classes all use the Router class. In each scheduling interval, reservations for the current window are transferred from the ReservationAllocator to the BandwidthAllocator. The BandwidthAllocator in turn tells the NetworkCapacityAllocator how much it can allocate. The NetworkUsageAllocator controls fair-share allocation.

A.2 Allocation Algorithm

Network capacity is allocated in the NetworkManager as follows. First, the administrator sets the following parameters:

- NETWORK_ROUTING_INFO: The path to the network routing table configuration file (described below).
- NETWORK_CAPACITY_INFO: The path to the network capacity configuration file (described below).

- NETWORK_HORIZON: What is the bandwidth allocation granularity (the size of the allocation window in seconds)? This parameter should usually be equal to the matchmaker's scheduling granularity set by NEGOTIATOR_INTERVAL.

- NETWORK_USAGE_HORIZON: Over what horizon (in seconds) do we calculate per-user fair-share network allocations (3600 by default)?

- NETWORK_CAPACITY_ALLOCATION_LIMIT: What is the maximum network capacity (in seconds) allowed in a single allocation (900 by default)?

- NETWORK_RESERVATION_LIMIT: What percentage of expected available future bandwidth may be reserved (50% by default)?

- NETWORK_BANDWIDTH_CONTROL_LOWER_LIMIT: The minimum percentage of bandwidth to be allocated to active rate controlled connections, if any, to avoid starvation (10% by default).

- MAX_GOODPUT_NETWORK_CAPACITY_PER_JOB: What is the maximum percentage (between 0.0 and 1.0) of network capacity for job placement that a qualified goodput transfer may request (0.0 by default)? Jobs that require less network capacity than this limit get a priority boost when bandwidth is oversubscribed to start running on idle CPUs. This allows Condor to keep CPUs busy even when the network is a bottleneck for higher priority jobs.

- NETWORK_CAPACITY_RESERVED_FOR_GOODPUT: What percentage of capacity (between 0.0 and 1.0) do we reserve for qualified goodput transfers when needed (0.0 by default)? This controls how much of a priority boost jobs with low network requirements receive when bandwidth is oversubscribed.

Condor allocates CPUs to jobs according to CPU fair-share, controlling job placements so the network will not be oversubscribed. If a job placement requires capacity on a network that is already allocated to its horizon, Condor will try to find a different CPU on which to place the job for which network capacity is available. If no such CPU can be found, the job must wait until capacity becomes available in the network horizon.

Condor makes the following guarantee to jobs waiting for network capacity. If the user has not already received his or her fair-share in the current "usage horizon", network capacity will not be allocated to any other users with lower CPU priority before it is allocated to this user. In other words, so long as the user has not exceeded his or her fair-share of network resources, no other users will move ahead of this user in the job queue because of Condor's network scheduling.

There is one caveat to this guarantee, however. The administrator may reserve some percentage of network capacity for overall system "goodput". Qualified jobs may use this reserved capacity for their placements, potentially moving ahead of the waiting user, if those jobs will use CPUs that would otherwise have remained idle. The impact of these jobs is limited by the percentage of capacity reserved. Reserved goodput capacity not used by goodput jobs is returned to the general-purpose pool, so the actual increase in waiting time will often be less.

A.3 Routing

The format of the NETWORK_ROUTING_INFO file is:

```
IP-ADDR SUBNET-MASK
--> NEXT-HOP IP-ADDR SUBNET-MASK
```

where IP-ADDR, SUBNET-MASK, and NEXT-HOP are all given in the standard numbers-and-dots notation. The first line defines a network resource and the "`-->`" lines that follow define hops from that network resource to other network resources. For a given route, the source network is the network resource for which:

```
source-ip-addr & SUBNET-MASK == IP-ADDR & SUBNET-MASK
```

with the largest SUBNET-MASK, and likewise the destination network is the network resource for which:

```
destination-ip-addr & SUBNET-MASK == IP-ADDR & SUBNET-MASK
```

with the largest SUBNET-MASK. For the hop definitions, the NEXT-HOP field specifies the IP-ADDR of the network resource that is the next hop for destination addresses for which:

```
destination-ip-addr & SUBNET-MASK == IP-ADDR & SUBNET-MASK
```

with the largest SUBNET-MASK. The routing algorithm starts at the source network and follows the next-hop configuration until it reaches the destination network. The current implementation requires that:

```
IP-ADDR == IP-ADDR & SUBNET-MASK
```

(i.e., the masked-out IP-ADDR fields must be 0). Since the routing algorithm searches all resources/hops for the largest matching SUBNET-MASK, the order in which the resources and hops are specified is not important.

The simplest configuration is:

```
        0.0.0.0 0.0.0.0
```

This configuration defines a single network segment connecting all endpoints. The SUBNET-MASK of 0.0.0.0 will match any IP address. Any bandwidth limits defined for the 0.0.0.0 network will be applied to all transfers between endpoints. Bandwidth limits can also be set for specific endpoint addresses using this configuration.

The example in Figure 35 describes a network with 2 subnets, connected to each other and to the internet. The "internet" is defined with a SUBNET-MASK of 0.0.0.0, so it will match any IP address. However, addresses on either of the two subnets will correctly match the subnets with the larger 255.255.255.0 mask.

Depending on how you intend to use it, the routing table can be very detailed or may describe a very idealized representation of your network. The routing table is used to allocate capacity

```
0.0.0.0 0.0.0.0                               # internet
--> 128.105.101.0 128.105.101.0 255.255.255.0 # --> 101
--> 128.105.102.0 128.105.102.0 255.255.255.0 # --> 102
128.105.101.0 255.255.255.0                   # 101
--> 128.105.102.0 128.105.102.0 255.255.255.0 # --> 102
--> 0.0.0.0 0.0.0.0 0.0.0.0                   # --> inet
128.105.102.0 255.255.255.0                   # 102
--> 128.105.101.0 128.105.101.0 255.255.255.0 # --> 102
--> 0.0.0.0 0.0.0.0 0.0.0.0                   # --> inet
```

Figure 35: Example Routing Table

```
128.105.101.3 --> 128.105.101.0 --> 128.105.101.5
128.105.101.3 --> 128.105.101.0 --> 128.105.102.0
  --> 128.105.102.5
128.105.101.3 --> 128.105.101.0 --> 0.0.0.0
  --> 216.115.108.245
```

Figure 36: Example Routes from Figure 35

on shared network resources, so it must include a definition for all resources defined in the NETWORK_CAPACITY_INFO file. There is no need to include endpoints in the table, however. The route always starts with the source address and ends with the destination address of the flow. For example, the table in Figure 35 will yield the routes shown in Figure 36.

If the router connecting the 2 subnets and the internet is a bottleneck, we can explicitly include it in the routing table so we can allocate its capacity for flows that traverse it as in Figure 37. We just chose one of the router's interfaces to identify it. Since we do not expect the router itself to be an endpoint, we just needed to choose an address and mask that would not match actual endpoints in our system. The internet has next-hop definitions for both subnets through the router, the router has next-hop definitions to the subnets and the internet, and the subnets have one next-hop defined to the router (i.e., all routes to endpoints that are not on the subnets must traverse the router). This modified routing table will yield the routes shown in Figure 38.

These routes will usually be different from what we would see from traceroute because our routes include network segments in addition to routers and endpoints so we can allocate capacity on all network resources. Our routing table defines a "virtual" network, abstracting away details we are not interested in and using virtual addresses for network resources that in reality have no assigned IP address (network segments, Ethernet bridges) or have multiple IP addresses (routers).

```
0.0.0.0 0.0.0.0                               # internet
--> 128.105.101.2 128.105.101.0 255.255.255.0 # --> router
--> 128.105.101.2 128.105.102.0 255.255.255.0 # --> router
128.105.101.2 255.255.255.255                 # router
--> 128.105.101.0 128.105.101.0 255.255.255.0 # --> 102
--> 128.105.102.0 128.105.102.0 255.255.255.0 # --> 102
--> 0.0.0.0 0.0.0.0 0.0.0.0                   # --> inet
128.105.101.0 255.255.255.0                   # 101
--> 128.105.101.2 0.0.0.0 0.0.0.0             # --> router
128.105.102.0 255.255.255.0                   # 102
--> 128.105.101.2 0.0.0.0 0.0.0.0             # --> router
```

Figure 37: Another Example Routing Table

```
128.105.101.3 --> 128.105.101.0 --> 128.105.101.5
128.105.101.3 --> 128.105.101.0 --> 128.105.101.2
  --> 128.105.102.0 --> 128.105.102.5
128.105.101.3 --> 128.105.101.0 --> 128.105.101.2
  --> 0.0.0.0 --> 216.115.108.245
```

Figure 38: Example Routes from Figure 37

A.4 Network Capacity

The format of the NETWORK_CAPACITY_INFO file is:

```
IP-ADDR CAPACITY
```

where IP-ADDR indicates an endpoint IP address or a network resource from the NETWORK_ROUTING_INFO file in the standard numbers-and-dots notation and CAPACITY is a floating-point number indicating the network capacity (in Mbps) of the resource. For example:

```
      128.105.101.0 40.0
      128.105.65.3 5.0
```

defines a 40 Mbps limit on the 128.105.101.0 subnet and a 5 Mbps limit for the host 128.105.65.3.