8-2008

Scalable, Data- intensive Network Computation

Huadong Liu
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Huadong Liu entitled "Scalable, Data- intensive Network Computation." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Micah Beck, Major Professor

We have read this dissertation and recommend its acceptance:

Jian Huang, Gregory Peterson, James Plank

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a dissertation written by Huadong Liu entitled "Scalable, Data-intensive Network Computation". I have examined the final paper copy of this dissertation for form and content and I recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Micah Beck, Major Professor

We have read this dissertation
and recommend its acceptance:

Jian Huang

Gregory Peterson

James Plank

Accepted for the Council:

Carolyn R. Hodges,
Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

SCALABLE, DATA-INTENSIVE NETWORK COMPUTATION

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Huadong Liu
August 2008

Acknowledgments

Firstly, I would like to thank my advisor Dr. Micah Beck for introducing me to the area of Logistical Networking and for encouraging me to work in this area. His generous support and commitment is indispensable to the completion of this work. Throughout my studies he encouraged me to develop independent thinking and research skills. I would also like to express my gratitude to my other exceptional doctoral committee members, Dr. Jian Huang, Dr. Gregory Peterson and Dr. James Plank for their time and input to this dissertation. I would further like to thank Dr. Terry Moore for the many technical discussions with regard to Logistical Networking.

Being a part of the Logistical Computing and Internetworking (LoCI) Laboratory, I owe many thanks to both LoCI staff and students. I would like to give special thanks to Scott Atchley and Yong Zheng, who gave me tremendous help throughout my work. I enjoyed working with Luis Sarmenta, Jean-Patrick Gelas, Ying Ding and Chris Sellers. I would like to thank them for the many conversations that have inspired my work. Many administrative staff in the Computer Science Department have provided me great help in countless occasions. I would like to give thanks to Dorsey Bottoms, Donna Bodenheimer, Markus Iturriaga Woelfel and other lab staff.

I have been fortunate to meet many friends outside of my study. This dissertation could not have been written without their encouragement and support. In particular, I would like to express my appreciation to Jeffrey Barnett and his wife Margaret Greco-Barnett. They have constantly helped out with living in Knoxville and have grown my interest in Tennessee football. The friendship with Jun Xu, Shaotao Liu, Yifan Tang, Yuanyuan Li, Xingyan Li, Fang Tang, Siyuan Liu, Chang Cheng and many others have been the source of the greatest pleasures.

I'd like to give my heartfelt thanks to my parents and my brother, who encouraged me to pursue a Ph.D. degree and supported me consistently. I owe a special thanks to my uncle who has always encouraged me to get higher education. I am greatly indebted to my devoted wife Yun Zhang and my son Erick Liu. They form the backbone and origin of my happiness. Yun deserves unique recognition for being incredibly understanding, supportive and most of all, patient during the course of this dissertation. Without her encouragement and backing, I would not have been able to finish this work. Her many intangible yet highly significant contributions are sincerely appreciated.

Lastly, I appreciate the financial support from DOE and NSF that funded the research discussed in this dissertation.

Abstract

To enable groups of collaborating researchers at different locations to effectively share large datasets and investigate their spontaneous hypotheses on the fly, we are interested in developing a distributed system that can be easily leveraged by a variety of data intensive applications. The system is composed of (i) a number of best effort logistical depots to enable large-scale data sharing and in-network data processing, (ii) a set of end-to-end tools to effectively aggregate, manage and schedule a large number of network computations with attendant data movements, and (iii) a Distributed Hash Table (DHT) on top of the generic depot services for scalable data management.

The logistical depot is extended by following the end-to-end principles and is modeled with a closed queuing network model. Its performance characteristics are studied by solving the steady state distributions of the model using local balance equations. The modeling results confirm that the wide area network is the performance bottleneck and running concurrent jobs can increase resource utilization and system throughput.

As a novel contribution, techniques to effectively support resource demanding data-intensive applications using the fine-grained depot services are developed. These techniques include instruction level scheduling of operations, dynamic co-scheduling of computation and replication, and adaptive workload control. Experiments in volume visualization have proved the effectiveness of these techniques. Due to the unique characteristic of data-intensive applications and our co-scheduling algorithm, a DHT is implemented on top of the basic storage and computation services. It demonstrates the potential of the Logistical Networking infrastructure to serve as a service creation platform.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in computing technology have enabled a wide range of scientific applications such as galaxy search in astronomy, CMS data analysis in high energy physics [Innocente, 2003] and protein sequence matching in biology (`http://www.ncbi.nlm.nih.gov/blast`). The use of computational simulations and scientific instruments generates datasets ranging from hundreds of gigabytes to terabytes. In today's increasingly collaborative research environment, these datasets are often distributed over the wide area network to be frequently shared among and analyzed by groups of geographically separated scientists. As the size of datasets and the number of distributed users have increased over time, the challenge of creating a scalable distributed infrastructure that can provide the fundamental bandwidth, storage, and computation services to data-intensive applications has proved daunting.

An example of such applications is the visualization of simulation datasets produced by the Terascale Supernova Initiative (TSI) project (`http://www.phy.ornl.gov/tsi/`), which is a collaborative effort at Oak Ridge National Laboratory with several universities across the United States. As the size of datasets (from the order of terabytes to petabytes) and the number of distributed researchers (originally 42 researchers from 11 institutions, expanding to 121 researchers from 24 institutions worldwide) have increased over time, it has become impractical to replicate the interesting datasets to sites where sufficient computing and storage resources are provisioned, due to long transport latency and the overhead of maintaining data consistency.

To meet the challenge, there has been considerable interest in creating public-resource computing projects to solve these problems. Projects such as SETI@Home and Folding@Home have demonstrated that it is technically feasible to scale scientific applications to thousands of heterogeneous, independent computers distributed over the wide area network. Various grid systems also have the goal of providing scalable application performance over large-scale shared resources connected by high speed networks. The overall trend is that groups of geographically separated scientists are increasingly using distributed resources across organizations to conduct collaborative research. As a computing system grows into heterogeneous environments, both the intermediate nodes that provide the service and the endpoints that consume the service need to be scalable so that the system can be widely deployed with increasing number of users and types of applications.

This dissertation complements other activities in data-intensive computing [Baru et al., 1998,Chervenak et al., 2001,Foster and Kesselman, 1997,Litzkow et al., 1998,Shoshani et al., 1998, Verma, 2002] by providing an integrated computation and storage infrastructure that can be leveraged by a wide range of data-intensive applications. The system that we create

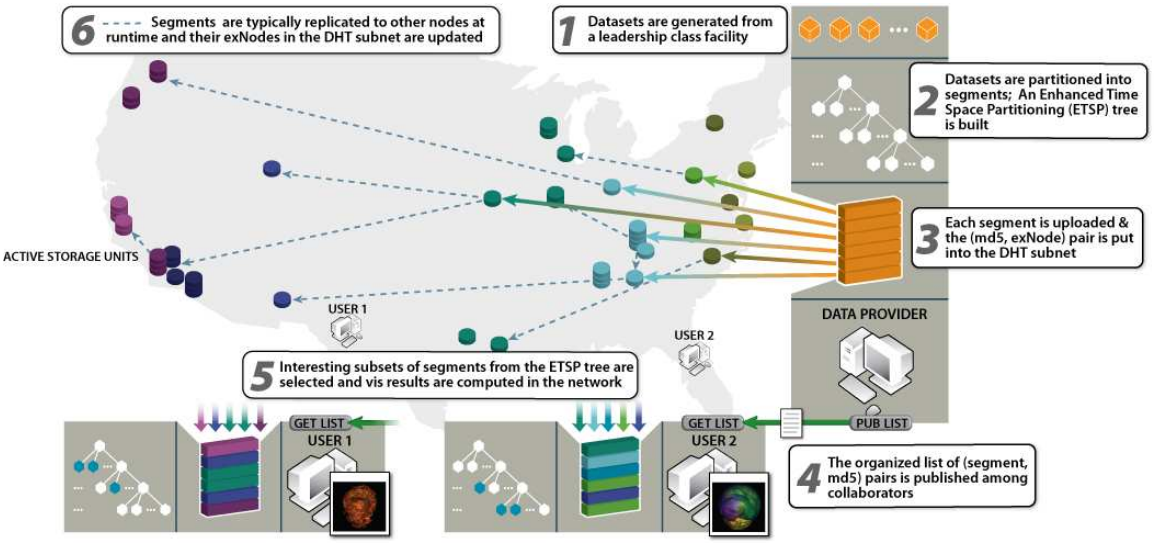


Figure 1.1: An illustration of the system using visualization of as an example

is composed of (i) a number of best effort, integrated storage and computation servers (logistical depots) built on Logistical Networking (LN) technologies to enable large-scale data sharing and in-network data processing, (ii) a set of utilities to effectively aggregate, manage and schedule a large number of network computations with attendant data movements, and (iii) a Distributed Hash Table (DHT) on top of LN storage and computation services for scalable data management.

Figure 1.1 shows the system in operation using remote visualization of the TSI dataset as an example. After a simulation dataset has been generated (1), preprocessed (2), uploaded and replicated (3) in segments, the list of data pointers is published (4) among collaborators. Researchers can then independently create and start parallel visualizations on a subset of the data (5). New replicas are made at runtime and cached in the network when necessary to improve performance (6).

The term "scalable" is defined as "capable of being easily expanded or upgraded on demand". It is a widely used term in research papers and software descriptions. However, its precise meaning is seldom defined but rather left to the readers' imagination [Bondi, 2000, Duboc et al., 2006, Hill, 1990]. In our system, two aspects of scalability are desired: *architectural scalability* and *deployment scalability* [Beck and Moore, 2004]. Architectural scalability is the ability of a system to expand in a chosen dimension (e.g. the number of applications or the number of heterogeneous hosting nodes) without major modifications to the architecture. Deployment scalability is the ability of a system to perform gracefully (i.e. preserving or improving valued properties) as the size of the system deployment increases. Our system achieves architectural scalability by adhering to the end-to-end arguments or principles [Saltzer et al., 1984] in the implementation of the logistical depot. Meanwhile, the system implements necessary scheduling techniques as well as a DHT replica management substrate to maintain deployment scalability.

The development involves three major steps: (i) to implement a best effort computation service at intermediate nodes that is integrated with storage, (ii) to implement necessary mechanisms and algorithms at endpoints so that fine-grained network computations can be

effectively used in high level data-intensive applications, and (iii) a DHT implementation on top of LN storage and processing primitives for scalable replica management.

1.1 Motivation

The end-to-end argument [Saltzer et al., 1984] is a set of rational principles for organizing placement of functions among modules of a distributed computer system. It suggests that functions at intermediate nodes be simple and weak, and functions with strong guarantees be applied at the endpoints by building higher software layers. Previous research in LN has shown the use of a best-effort network storage service, the Internet Backplane Protocol (IBP), for scalable data sharing by applying the end-to-end principles [Beck et al., 2002]. In [Beck et al., 2003], the Network Function Unit (NFU) is proposed as an orthogonal extension to IBP for scalable network computation, applying the same paradigm.

Following this lead, we first implements and extends the architectural guidelines presented in the NFU paper. Several data-intensive applications are then developed to run on the system though collaborative efforts. By dealing with these applications, limitations of the system are identified. New tools and mechanisms (e.g. dynamic scheduling, programming libraries and data management) are developed to improve scalability and usability of the system. We attempt to validate the following two hypotheses:

1. An end-to-end approach to scalable programmable networking is feasible by creating a best effort computation service at intermediate nodes and constructing necessary software layers at endpoints.

2. The system we built on top of LN storage and computation services is useful to a variety of data-intensive applications.

The first hypothesis has been theoretically evaluated by making an analogy between the NFU data transformation service and the Internet Protocol (IP) data forwarding service in the original NFU paper. We further prove its validity by developing necessary mechanisms and tools to build a working system that meets the architectural guidelines and to test its limits. The second hypothesis, however, is an assertion of users' judgment about the system we built. Its validity will be evaluated by demonstrating that the system is capable of providing the kind of generality (e.g. supporting scalable data management and various parallel programming models) that many data-intensive applications require. To date, applications that run on the system include distributed volume visualization and data mining for bioinformatics. We will have a brief introduction of them.

1.2 Applications

Typically, researchers have three primary concerns when choosing a system to run scientific data-intensive applications. First, the system should offer both storage and computing resources without the requirement of stringent authentication. Without this characteristic, it is extremely hard for any individual group to come up with a testbed of any practical significance. Second, the system must be sufficiently robust, and hence redundancy for fault tolerance is required as a native ingredient of the infrastructure. It is rather unrealistic for application programmers to focus on the application and fault-tolerant distributed computing at the same time. Third, the system must have some basic constructs that

```
<viewing_parameters>
    <lighting num_lights> ... </lighting>
    <viewport_transform> ... </viewport_transform>
    <data_specific> ... </data_specific>
    ...
</viewing_parameters>
<raycasting>
    <scale> ... </scale>
    <corlor_scheme> ... </corlor_scheme>
    <interval_range> ... </interval_range>
</raycasting>
```

Figure 1.2: A fragment of the visSpec file.

can compose slices of storage and computing resources to provide a higher level abstraction of resources (e.g. unlimited storage and computation) and to deliver a certain level of performance. Without these features, the complexity involved in achieving functionality, scalability, robustness and performance could quickly become overwhelming to the application. Fortunately, our system has them all.

1.2.1 Distributed Visualization

Distributed visualization is the driver application. Visualization is a research tool which computational scientists use for qualitative exploration, hypothesis verification, and result presentation. Visualization users routinely deal with datasets that require large-scale parallel computing in order to be analyzed and rendered. The output size of simulation or data captures already frequently goes beyond tens of gigabytes. Unfortunately, computing resources of that scale are not always available to support spontaneous research needs in an efficient way. In addition, driven by increasing user needs to collaborate across geographical distances, visualization must also serve as an effective means to share concrete data as well as abstract ideas over the Internet.

As a prototype, common volume rendering operations, in particular software ray-casting and isosurface extraction from the visualization cookbook library, have been wrapped into the `vcblib` library and deployed on our system. A dataset independent `visSpec` file (shown in Figure 1.2) containing all needed visualization parameters is defined to set up a visualization task. Once the dataset is uploaded, with the `vcblib` library and the `visSpec` file, parallel rendering operations are spawned off by a viewer program that calls the scheduler described in Section 5 to orchestrate a test run. Besides doing well-received demonstrations at SuperComputing conferences, significant results on dynamic sharing of large-scale visualization have been reported in [Huang et al., 2007].

1.2.2 Data Mining for Bioinformatics

In the past, IBP has been used to distribute biological sequence databases over the Internet to avoid warehousing and maintaining the complete databases locally. In that work, an exNode (reference to the database chunks) is obtained from a centralized directory server upon a user request. The user can then proceeds to download data from IBP depots using the exNode, to run local FASTA or BLAST queries on these chunks, and to merge

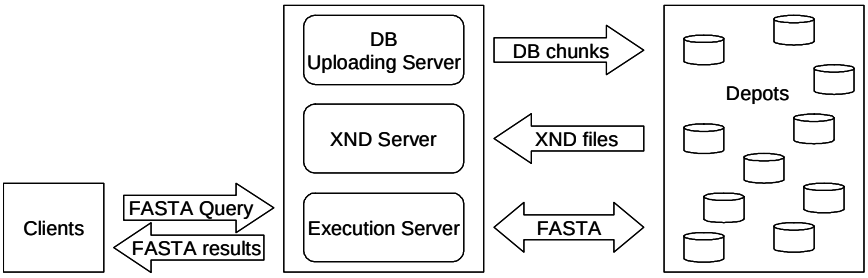


Figure 1.3: An overview of NFU-FASTA

intermediate results. FASTA and BLAST are the most widely used sequence analysis tools. BLAST runs faster than FASTA, but is less sensitive for DNA sequences analysis. As an important optimization, the processing of a FASTA query is moved to logistical depots where the database chunks are stored. Figure 1.3 illustrates the NFU-FASTA implementation [Baker et al., 2007] from Baylor University.

Another application in Bioinformatics is to find correlated gene sets from a correlation coefficient matrix that is produced from large DNA microarray data. The usual approach is to download the entire correlation coefficient matrices to a site where clusters or shared memory machines are provisioned, filter the data with a user defined threshold, convert them to smaller unweighted graphs, and run the maximum or maximal clique enumeration algorithm to find cliques in graphs, which represent a set of genes that are correlated. This process is optimized by pipelining the filtering, converting and enumeration computations to logistical depots that are close to where the matrices are produced.

1.3 Contributions

The system we built goes beyond the basic idea of using active storage for data-intensive computing, i.e. moving computation close to where data is stored. The main objective is to develop a shared infrastructure with integrated storage and computation resources that can be effectively used by data-intensive applications to achieve functionality, scalability, robustness and performance. The contributions of this work can be summarized as follows:

A data-intensive distributed computing system The system has been used for convenient sharing and processing of "network resident" data in large-scale volume visualization, video transcoding and protein sequence matching. More importantly, it demonstrates that a best effort computation service can be placed at intermediate nodes by confining its consumption of local resources. Higher layer software components need to be constructed at endpoints to aggregate and schedule a large number of primitive data movements and computations in the network;

End-to-end techniques for performance and stability Two instruction scheduling techniques – loop unrolling and speculation are combined to hide wide area instruction issue latency in the presence of control dependency. The co-scheduling of computation and replication scheme for task farming applications implements mechanisms for performance, fault-tolerance and distributed workload control;

A closed queuing network model for logistical depots Although the modeling methods and mathematics are not brand new, the model is an ideal vehicle to understand the performance characteristics of logistical depots and to evaluate approaches to scale up systems that build on top of logistical depots;

A new approach to creating network services By creating a working DHT service on top of the primitive storage and processing resources exposed by logistical depots, we demonstrate the potential of the LN infrastructure to serve as a service creation platform.

1.4 Document Organization

The remainder of this dissertation documents the details of each component in our system and the above contributions. Chapter 2 presents a survey of the literature to provide a context for the discussion. We present a design and implementation of the NFU to support data-intensive applications in Chapter 3. A closed queuing network model is developed in Chapter 4 to study the performance characteristics of logistical depots. The modeling results justify the use of scheduling techniques described in Chapter 5. The experimental results with distributed merge and distributed visualization demonstrate the effectiveness of the scheduling techniques. Chapter 6 explores an approach to create a DHT network based on primitive LN services, followed by a discussion of the relationship between fault-tolerance and scalability. Finally, we conclude in Chapter 7.

Chapter 2

Literature Review

It is not advisable to constantly move a large amount of data in data-intensive applications, as this is always an error-prone and time, resource consuming process. Hence, it would be ideal if a system infrastructure can provide co-located resources for both storage and processing. While there are a variety of system infrastructures for data-intensive scientific computing, ours is built by extending the LN infrastructure because of its Internet style bottom up design and its integrated view of resource management in wide area distributed systems. While providing an alternative solution for data-intensive computing, the development of our system presents novel challenges, due to the best effort nature of LN services. Below we describe the building blocks of LN, followed by a review of other infrastructure alternatives and major techniques used in our system to meet the challenges.

2.1 Logistical Networking

Logistical Networking is an overlay implementation of a new architectural approach to highly generic, interoperable computer networking. The fundamental idea behind LN is that the end-to-end principle that has guided the development of the Internet also applies to other types of shared information infrastructure, specifically to infrastructure that includes substantial storage and processing resources. Because it incorporates such resources, which in conventional network applications are provisioned at hosts or endpoints, the scope of LN is significantly broader than that of traditional network protocols. It encompasses application areas previously addressed by active networking [Alexander et al., 1998, Tennenhouse and Wetherall, 1996], overlay networks [Andersen et al., 2002, Fall, 2003], storage [Rhea et al., 2001] or content distribution networks [Johnson et al., 2000]. The scope of LN overlaps areas of wide area distributed systems that are typically thought to lie outside the domain of networking, such as distributed visualization and data mining, database management, and even Grid computing.

The central tenet of LN is that the end-to-end principle, which has served as a guide to the design of shared infrastructure for the transfer of data between endpoints, can be applied to the design of shared infrastructure for storage and processing services. The motivation for generalizing the idea of shared infrastructure in this way is the same as one expressed by the authors of the end-to-end arguments: application autonomy [Reed et al., 1998]. However, LN questions the view that applications can best achieve the desired freedom to change as needed by sharing only data transfer services and implementing all other resource management at endpoints.

Despite the fact that LN shares many of the goals of and has achieved a reasonable degree of early acceptance by the networking community [Beck et al., 2002, Beck et al., 2003], it faces three significant objections that impede further engagement by a broader community. First, some have objected that a network comprised of LN's "fat" intermediate nodes, provisioned with substantial storage and processing resources, would run foul of the end-to-end principle, and so could not possibly scale. The current deployment of IBP as a global storage service should have cleared up the objection.

Second, the weak semantics of the basic LN storage and processing services and the use of untrusted intermediate nodes to provide them would render the resulting service unusable to real applications. However, the diligent construction of end-to-end protocols and tools has made it possible to develop services with semantics strong enough to meet, and in some cases exceed, the requirements for performance and reliability of the application communities that have adopted LN, including content distribution, remote visualization, video transcoding, and data mining. Techniques and tools developed in the dissertation is part of this effort.

Finally, it has been objected that LN's steadfast adherence to mechanisms that take account of the end-to-end principle, requiring that detailed control over the functioning of intermediate nodes be managed by endpoints, cannot possibly provide adequate performance in applications where the sequence of operations to be performed is dynamically determined during the execution of an application protocol. In Section 5.1 we address this objection by showing how the techniques used to obtain performance from processors in architectures with high instruction issue latency can be applied to IBP operations executing at intermediate nodes.

2.2 Grid Computing

There has been a tremendous growth in the research of Grid Computing since the early 90's. Grid Computing aims to provide unified, coordinated access to computing resources across organizations. For example, Globus [Foster and Kesselman, 1997], Condor [Thain et al., 2002] and Legion [Grimshaw and Wulf, 1996] all allow users to access resources across administrative domains. However, hierarchically managed resources in a grid system are shared in a highly controlled manner, among selected communities, with authentication, access control and even resource accounting as primary design issues. In contrast, storage and processing resources in LN depots are designed to be shared on the model of IP datagram service, i.e. in a relatively unbrokered manner, open to the entire community. The differences between various grid systems and ours are:

- There is no need to obtain accounts or make advanced reservations in order to use our system. With a system operated in an interactive manner as opposed to a batch mode of operation, scientists can easily explore and share spontaneous research ideas.
- Grid systems involve resource sharing between organizations that are mutually accountable and trustworthy. In contrast, our users are not accountable and not trustworthy because of their unknown identities. Moreover, LN depots are assumed to be best effort and unreliable. As a result, LN depots need to be protected from potentially malicious applications while applications need to establish a certain level of confidence in the outputs produced by LN depots.

- In Grid systems, hierarchical resource brokers schedule applications' access to distributed resources, i.e. resource management is centralized. In contrast, resources management in our system is decentralized in the sense that every endpoint in the system is essentially a resource scheduler that is responsible for increasing or decreasing resource usage dynamically and to maintain stability of the system.

2.3 Volunteer Computing

Volunteer computing projects [Anderson, 2004,Larson et al., 2003,Sarmenta, 2001] in high-energy physics, molecular biology and astrophysics have been using public computing resources for years. These projects aim to aggregate a large number of geographically distributed, administratively independent computers for solving large-scale scientific problems. Volunteer computers typically download, run an application specific client program on their local resources, and periodically contact the central task server to send back results and to receive more tasks to process [Anderson et al., 2005]. Our system differs from volunteer computing systems in a number of ways:

- Volunteer computing systems mainly focus on computation-intensive applications. For example, a 350KB work unit in the SETI@home project can keep a typical desktop computer busy for one day [Anderson et al., 2002]. In contrast, our system is built on top of logistical depots to enable large-scale data sharing and in-network data processing. Thus, it is more appropriate for data-intensive applications.

- Most volunteer computing systems do not have a model for communication between volunteer computers, i.e. an embarrassingly parallel programming model is assumed. In our system, common programming models (e.g. Message Passing Interface (MPI) and Bulk Synchronous Parallel (BSP) [Sujithan, 1996]) can be supported using NFU operations with embedded IBP data movement calls.

- Volunteer computers have to explicitly trust the application specific code executed on local resources. In contrast, logistical depots provide a generic, secure execution interface by running external code in a lightweight sandbox, protecting themselves from potentially malicious applications from the network. Running legacy applications is also supported by implementing a library that maps file system operations to memory operations.

- Individual volunteer computers are usually dedicated to a single task server (i.e. a single application) while logistical depots are shared by a group of concurrent users running different applications. Consequently, task farming in volunteer computing systems is straightforward while the task scheduling in ours has to carefully consider resource contention to maintain stability of the system.

2.4 Programmable Networking and Active Storage

Using programmable resources at intermediate nodes has been explored to enable dynamic service creation. The research on active networking [Alexander et al., 1998, Tennenhouse and Wetherall, 1996] attempts to create advanced network services by placing a program in a packet and then have the program executed at intermediate nodes that the packet traverses.

The work on active content distribution and Open Pluggable Edge Services [AkamaiINC, 2002, Huang et al., 2004] impose an application proxy between a client and a server on the Internet to reduce traffic and latency. All these works encapsulate process execution from endpoints with the assumption that the transformation of state performs correctly at intermediate nodes, which unfortunately runs against the end-to-end arguments that the semantics of active features need to be constrained [Reed et al., 1998]. In contrast, NFU puts limits on network computations and exposes the control over them to endpoints. This is similar to Ephemeral State Processing (ESP) [Wen et al., 2002] except that ESP's extremely fine-grained bounds on resources (e.g. time-bounded associated memory) makes it inappropriate for data-intensive applications.

Motivated by advances in ASIC technology, especially the increasing processor speed and decreasing memory cost, running data processing code inside storage devices is advocated [Acharya et al., 1998, Riedel et al., 2001]. However, this is not likely to be practical very soon because the disk interface has to be general and safe. To date, there is no such generally useful and secure disk interfaces available in the market. As an alternative, adding computational extensions to existing network file systems has been proposed. In [Anastasiadis et al., 2005, Srinivasan and Singh, 2002, Wickremesinghe et al., 2002], applications use the traditional file access interfaces and computation occurs as a side effect of data access (i.e. implicitly) with bounded per-record processing and bounded internal state. This approach involves changing the existing I/O semantics and is usually application specific. In contrast, NFU exposes an interface for explicit invocation of computations. Other works (e.g. the Scriptable PRC [Sivathanu et al., 2002]) take advantage of the mobile code technology to implement new functionalities using scripts or executables at the file servers which do not have control over the computations in general.

2.5 Scheduling and Network Congestion Control

Instruction scheduling in computer architectures (e.g. list scheduling [Fisher et al., 1981], trace scheduling [Fisher, 1981], and software pipelining [Lam, 1988]) aims to minimize the overall program execution time on the functional units by reducing pipeline stalls and exploiting available instruction-level parallelism. Some of these techniques have been studied in the wide area setting. For example, pipelining is a default transfer method since HTTP/1.1 to improve performance and the Distributed Pipelining Framework is proposed in [Chatterjee, 1996] to guarantee the end-to-end execution deadline over a set of distributed resources. As a complement, we combine techniques such as loop unrolling and speculation to hide network latency caused by data or control dependencies.

Job scheduling in distributed systems has been extensively studied [Berman et al., 2003, Berman et al., 1996, Heymann et al., 2000a, Page et al., 2004, Shao et al., 1998]. What distinguishes our work is that the scheduling of runtime data movements is considered an important part of job scheduling in data-intensive applications. The scheduler extends the work on downloading replicated data blocks in the wide area [Allen and Wolski, 2003, Plank et al., 2003] to on-demand data movement during the process of job scheduling. Other works have also realized the importance of data movement in the scheduling of data-intensive jobs. For example, the Integrated Replication and Scheduling Strategy is proposed in [Chakrabarti et al., 2004] to iteratively improve application performance; algorithms that compute an optimal placement of replicas prior to job execution are described in [Desprez and Vernois, 2005]; and several scheduling and replication strategies in a two-level scheduling

framework are evaluated in [Ranganathan and Foster, 2003]. Most of these schedulers assume a static resource performance. The scheduler in our system is able to adapt to the underlying resource performance dynamically.

Network congestion control at endpoints is special type of job scheduling in the sense that the endpoints determine when and where to send a packet. TCP as the most widely used transport layer protocol plays an integral role in determining overall network performance. Amazingly, TCP has changed very little since its initial design [Jacobson, 1988]. A few tweaks have been added [Brakmo and Peterson, 1995, Semke et al., 1998], but the most part remains fairly stable. The fundamental design choice in TCP is that the endpoints are responsible for controlling the rate of data flow. Endpoints in our system mimic the behavior of TCP but to control the computation workload on an intermediate node, hoping to get the best performance out of the non-dedicated intermediate nodes but not to overload them (i.e. to use shared resources friendly).

2.6 Data Management

Data management is a challenging problem in high performance computing environments, and the difficulty of managing scientific datasets increases as they scale to terabytes and petabytes. Besides providing interfaces to read and write data, a data management system should also provide mechanisms to manage *metadata*, i.e. information about data stored on heterogeneous storage systems, such as replica locations.

In a grid setting, hierarchal catalogs are commonly used for data management [Chervenak et al., 2001, McCance, 2003], while DHT is widely employed for data location in storage peer-to-peer systems [Cai et al., 2004, Rhea et al., 2005]. In LN, a directory service, LoDN [Beck et al., 2004], has been deployed for automated replica management. In our system, a proprietary DHT is implemented on top of the generic storage and computation services exposed by LN depots to keep track of replica locations. The DHT implementation not only demonstrates the feasibility of the LN infrastructure to serve as a service creation platform but also improve data availability by coupling the management of metadata with data itself.

2.7 Queuing Network Based Performance Modeling

The discovery that certain queuing networks have tractable product form solutions [Jackson, 1963, Gordon and Newell, 1967, Whittle, 1967] had a profound influence on computer performance modeling. In such systems the stationary distribution of the network is composed of a product of the distributions of each queue analyzed in isolation (subject to a normalization constant). The important BCMP paper [Baskett et al., 1975] established that a useful class of queuing networks satisfying partial balance also satisfied the product form. This had a significant influence on computer performance modeling and set a direction for further works. After that, queuing networks have been extensively used to model computer and communications networks. Computer systems being modeled includes IBM mainframes running the VM [Bard, 1978] and MVS [Buzen, 1978] operating systems. Modeling of communication networks using product form networks includes [Henderson and Taylor, 1989] and [Dijk, 1991], just to name a few.

Modeling of file and storage systems includes [Ramakrishnan, 1986, Ramakrishnan and Emer, 1989, Disz et al., 1997, Hac, 1992]. We are interested in the performance characteristics

of logistical depots because they are the most used resource in our system. We want to make sure that the scheduling techniques developed at the endpoints can effectively increase resource utilization at intermediate nodes. The modeling process is quite similar to those in previous works except for the system components modeled, model selection and parameterization methods. For example, [Ramakrishnan, 1986] explicitly modeled client behaviors in a file system. We didn't model that because endpoints in a system are usually dedicated for a single user or application at a time.

Chapter 3

Extensions to the Network Function Unit

To provide a scalable network computation service in the wide area, the end-to-end principles require that semantics of the service be simple and weak. If the semantics are too complex, it will fail the requirement that services be generic. Likewise, if a service makes strong guarantees, it will not scale like the Internet. By taking advantage of the fact that we already have IBP to manage primitive network storage, the NFU is implemented as an extension to IBP for data transformation. Before getting into the details of NFU, we will have a brief overview of IBP.

3.1 The Internet Backplane Protocol

IBP implements a generic network storage service that scales globally. IBP storage is managed by servers called "depots", on which clients perform remote storage operations. The depot is designed for simplicity and robustness by using a stateless protocol like NFS. The term "stateless" means that the depot does not need to maintain any information in its main memory. All essential information about allocations is kept on disk. Thus, nothing is lost during a crash unless the disk itself is corrupted. As an optimization, information of allocations can be cached in main memory to improve performance, but the depot does not depend on this information to function correctly.

The clients view a depot's storage resources as a collection of byte arrays and initially gain access to them by making allocations. If the allocating is successful, the depot returns three cryptographically secure URLs, called capabilities: one for reading, one for writing, and one for management. IBP client calls fall into three different groups as shown in Table 3.1. Key characteristics of IBP storage are:

Table 3.1: IBP Client API

Depot Management	`IBP_status`			
Storage Management	`IBP_allocate`	`IBP_manage`		
Data Transfer	`IBP_store`	`IBP_load`	`IBP_copy`	`IBP_mcopy`

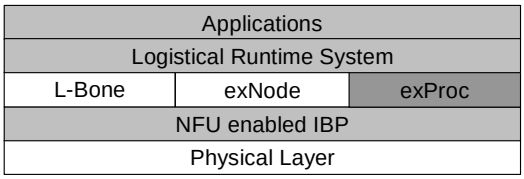


Figure 3.1: NFU enabled LN stack

1. Allocations of IBP storage are limited in size and duration. An allocation request can be refused in response to over allocation and storage resource can be revoked when the lease expires;

2. Semantics of IBP storage are weaker than conventional storage services. To encourage sharing of idle disk resource, IBP supports "soft" storage which can be revoked at any time before the lease expires. Moreover, IBP storage can be transiently unavailable or even permanently lost.

Because of the best-effort nature, IBP does not directly implement strong storage services such as files. Instead, these services must be built on top of IBP using techniques such as replicating and striping over multiple depots, much as TCP builds on IP's unreliable datagram delivery to provide reliable transport. The LoCI Lab has built a network storage stack as shown in Figure 3.1 to facilitate the use of IBP storage. The L-Bone, that is, the Logistical Backbone maintains a directory of IBP depots and metadata about these depots. Users can query the L-Bone for depots that meet certain requirements. The exNode is an XML description of IBP allocations. It aggregates IBP storage to provide file services over the network. The Logistical Runtime System (LoRS) is an API and associated set of software tools that use the exNode, L-Bone and IBP to implement e2e services such as encryption and checksum. exProc is a brand new component to aggregate NFU computations, and all the other shaded components are extended to incorporate NFU computation.

3.2 NFU Operations

NFU is an abstract service based on managing the underlying computational capabilities of the depot as "operations" [Beck et al., 2003]. It is considered as the transformation of bytes stored in IBP storage. Similar to IBP allocations, NFU operations are by default limited in size and duration, which means that the size of byte arrays that any computation can transform has a bound, and the duration of execution also has a bound. Semantics of NFU operations are weaker than typical process creation or procedure call on a local processor, as is necessary to model computations accessed across the network. Because of the weak semantics of NFU operations, abstractions with strong properties, such as unbounded size and unbounded duration, must be constructed at a higher layer by aggregating primitive NFU operations. NFU is implemented as an add-on module since IBP version 1.4. In the rest of this dissertation, IBP operations refer to both the legacy storage operations (e.g. `IBP_allocate` and `IBP_copy`) and the new processing operation `IBP_nfu_op`. The term depot and intermediate node are interchangeable in the context of IBP storage and computation services. Similarly, the term endpoint, client, and user are are interchangeable as well in such context.

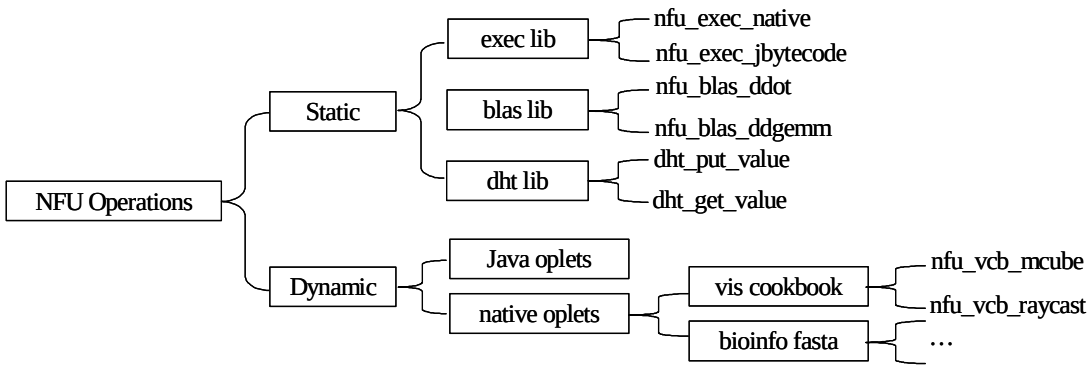


Figure 3.2: A taxonomy of NFU operations

NFU operations are usually grouped as libraries so that they can be managed hierarchically. Libraries of NFU operations are either static or dynamic as shown in Figure 3.2. Static operations are built-in modules that are highly standard and useful to many applications in general. For instance, the Basic Linear Algebra Subroutines (BLAS) operations are wrapped as a static library. Static libraries are deployed by being verified, compiled and linked as part of the depot, and so require no further deployment action to be usable by that depot's clients.

In contrast, dynamic operations are implemented by code that is executed or interpreted by a particularly general static operation. The code that defines a dynamic operation is stored in an IBP allocation and passed to the appropriate static operation as an argument. Because of the dynamic nature of the operation, the code is not known to the depot before it is invoked, and may be delivered from an arbitrary client across the network. Thus the code that defines a dynamic operation is a kind of mobile code [Czajkowski and von Eicken, 1998, Garfinkel, 2003, Hulaas and Binder, 2004, Jain and Sekar, 2000], which we refer to as an "oplet". A extensible static library that loads and executes oplets from an IBP allocation, acting on arguments stored in other IBP allocations, defines an execution environment, referred to as the `exec` library. Currently, there are two static operations to load and run oplets stored as machine dependent native code (native oplets) and oplets stored as machine independent Java bytecode (Java oplets). Figure 3.3 illustrates how static and dynamic operations are invoked using the `md5` hashing as an example.

Note that static and dynamic operations have varied performance and scalability implications. For example, static operations are highly optimized with the local depot during installation while dynamic operations can be invoked without being installed by making use of a previously installed static operation that executes or interprets an oplet. Creating dynamic operations avoids installation and the accompanying issues of authentication and trust. It pushes issues like providing NFU wrappers, cross-compiling and even adjusting byte orders to endpoints while keep the service at intermediate nodes application independent, an approach that is advocated by the end-to-end arguments. Thus, the use of static operations for data transformation should be limited. However, static operations provide an option for data-intensive computing systems where performance is the primary design concern. Due to the variety of applications, it is impractical to install all computations as static operations. We use oplets to dynamically define operations needed by applications. Mobile programs are appealing because they support efficient use of network resources and

```
// allocation of the crypto lib
paras[0]: {IBP_REF_RD, cap1->readCap, len_of_nfu_crypto_lib};
// name of the md5 operation
paras[1]: {IBP_VAL_IN, "nfu_crypto_md5", strlen("nfu_crypto_md5")};
// allocation of the data block to be hashed
paras[2]: {IBP_REF_RD, cap2->readCap, len_of_data};
// 128-bit buffer to receive the hash result
paras[3]: {IBP_VAL_OUT, buffer_md5, 16 };
// use static op nfu_crypto_md5 (assume it is installed) to calculate the md5 of data in cap2
if (IBP_OK != IBP_nfu_op(ln_depot, NFU_CRYPTO_MD5, paras+2, timeout)) {...}
// use oplet nfu_crypto_md5 in cap1 to calculate the md5 of data in cap2
if (IBP_OK != IBP_nfu_op(ln_depot, NFU_EXEC_NATIVE, paras, timeout)) {...}
```

Figure 3.3: An example of calling static and dynamic NFU operations

make depots extensible. This approach provides generality but requires that execution of oplets be weakened to ensure security and stability of the depot.

3.2.1 Running Native Oplets

The end-to-end arguments require that the semantics of any scalable service be weakened in order to scale. As a result, NFU operations are limited in size and duration by default. For static operations, the execution space complexity $\mathcal{S}$ and time complexity $\mathcal{T}$ are known when they are installed, thus these limits can be easily enforced by checking the input parameters. However, the behavior of dynamic operations is unknown untill they are invoked. The mobile code approach provides generality but requires that oplet execution be monitored and constrained at runtime.

We uses the dynamic linking loader API (`dlopen`, `dlsym`, and `dlclose`) to load and run native oplets in a system call interposition based sandbox (as illustrated in Figure 3.4) under the assumption that a native oplet can do little harm if its access to the underlying operating system resources is appropriately monitored and restricted. `ptrace` is chosen for system call interposition because it is provisioned on almost all flavors of UNIX without requiring the root privilege. As described in [Wagner, 1999], two main limitations of `ptrace` are coarse-grained all-or-nothing tracing and no system call cancellation mechanism on some OS implementations, for example Linux. We will show that these limitations are not a problem in sandboxing the execution of oplets.

NFU operations transform data blocks that have been mapped to memory buffers in user space. Coarse-grained all-or-nothing tracing in the execution of native oplet is not a problem because the only explicit system calls a native oplet can use are allocating temporary memory during computation and inquiring system time for various purposes (e.g. seeding random numbers). Native oplets are not allowed to invoke system calls to perform other functions such as process control, file or network access. Any system call other than memory allocation or system time inquiry simply results in termination of a native oplet execution. As a result, performance impact due to system call interposition is minimal if an oplet does not allocate memory or inquiry system time frequently.

A sandboxed native oplet cannot access an unbounded amount of memory, nor can it run indefinitely. After initialization, an oplet's subsequent requests for memory are intercepted and recorded. If an oplet asks for more memory than the depot can afford in the current

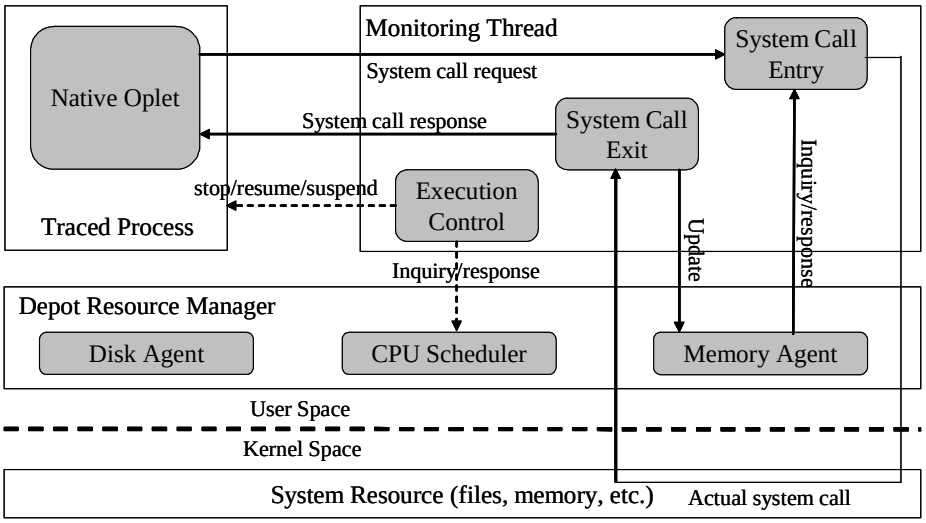


Figure 3.4: The execution environment of native oplets

state, `brk` or its equivalent will fail even if the underlying OS could fulfill the request (possibly at the risk of thrashing). The second limitation of `ptrace` is no interface to abort system calls. This is circumvented by replacing the oplet supplied system call argument with a value, for example -1, that will cause the system call to fail at the system call entry and then modifying the return value to indicate an error inside the depot that no more memory can be allocated at the system call exit. The `ptrace` based system call interposition framework also provides an easy way to suspend or terminate execution of an oplet by sending signals to the the traced NFU process in which the native oplet is loaded to execution.

When an oplet is terminated before completion, all states are lost except the changes made to IBP allocations during the execution. Endpoints can use this feature to determine the extent of execution by examining the content of IBP allocations, which provides a weak form of checkpointing at the user level as opposed to checkpointing all process states at the system level. System level checkpointing is valuable primarily for long-running computations on closely-coupled clusters where the cost of migration is low. However, NFU operations are intended to be used in aggregate, and to be spread across nodes in the wide area. Thus, the set of possible resources to use is large, but any individual node is not reliable, and communication may be expensive. When writing NFU operations, it is not expected to enforce an upper bound on their resource utilization that matches the limits set by depots, because resource utilization is difficult to quantify and control in non-trivial code. To use NFU operations, endpoints can code operations to enable smaller units of work and switch to smaller units when they are having trouble with larger units. Endpoints can also resume execution from the last "checkpoint" retrieved from IBP allocations.

3.2.2 Running Java Oplets

Java has been increasingly used for cycle sharing over the Internet because of its portability across execution environment and built-in sandboxing technology. Instead of constructing and destructing a JVM for every Java oplet, a single JVM is created at depot startup using

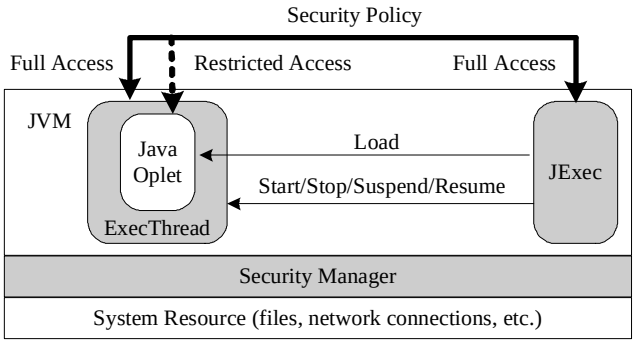


Figure 3.5: The execution environment of Java oplets

the Java Native Interface. This JVM is shared later by all Java oplets. The execution environment for Java oplets is shown in Figure 3.5.

When a request comes for running a Java oplet, a new JExec object is initialized. The default Java class loader caches previously loaded objects. This causes new versions of an oplet with the same name never get loaded because the class loader assumes that they have been loaded. Jexec fixes this by creating a customized class loader so that only the intended oplet is loaded into JVM as long as different versions of oplets are stored in different IBP allocations. Java oplets are executed in a newly forked Java thread, which is under complete control of JExec, i.e. the execution of an oplet can be suspended, resumed or stopped. JExec is also responsible for reading data from input IBP allocations, passing them to newly constructed Java objects, invoking the intended oplet method to perform computation on Java objects, updating output IBP allocations and sending results back to the endpoint.

The Java security manager is used to restrict a Java oplet's access to system resources, such as files and network connections. The security policy is configured in a way that Java oplets have the minimum access permissions while the controlling objects such as JExec have full access. Unfortunately, the Java security manager does not provide a way to enforce any kind of limit on memory allocation or thread creation. JVM Tool Interface (JVMTI) provides a way to inspect the state and to control the execution of code running in the JVM. Although JVMTI can be used to control Java oplet's consumption of CPU and memory by transforming the Java bytecode, it is not the emphasis of this dissertation. Java is not a mainstream programming language for data-intensive computing because of the overhead to interpret and monitor bytecode execution. `nfu_exec_jbytecode` is created to demonstrate that the `exec` library can be easily extended to include other execution environments.

3.3 The exProc: A Data Structure for Aggregating Network Computations

Because of the weak semantics of NFU operations, computation abstractions with strong properties must be built at a higher layer by aggregating a large number of primitive NFU operations. The fundamental tool of such aggregation is the decomposition of high level algorithms and data they operate on to a set of operations and dependencies between them. An XML encoded data structure called the exProc is defined to represent outputs of such decomposition. Similar to the exNode, which describes the mapping from logical files to IBP

```
<exproc xmlns="http://loci.cs.utk.edu/nfu" name="raycast" version="0.1">
   #use the exec lib to dynamically load an operation
   <import url="localhost:///export/execlib.xml" name="execlib"/>
   #use the visualization cook book library
   <import url="localhost:///export/vcblib.xml" name="vcblib"/>
   #operations defined in this section will be executed in parallel
   <parallel name="rcast-parallel" spot-check-rate="0.1">
      <for counter="i" from="0" to="100" step="1">
         #use the raycast function in vcblib
         <operation name="raycast-${i}" type="vcblib:raycast">
            <parameter index="0" iotype="IBP_REF_RD">#input data specified by an exnode
               <value datatype=xnd">jet-${i}.xnd</value>
            </parameter>
            <parameter index=1" iotype="IBP_VAL_IN"> #visualization parameters in XML
               <value datatype=xml">vis_spec.xml</value>
            </parameter>
            <parameter index=2" iotype="IBP_VAL_OUT">#output dumped to a local file
               <foutput>output/rcast-jet-${i}</foutput>
            </parameter>
         </operation>
   ...
```

Figure 3.6: An example exProc used in volume visualization

storage allocations, exProc aggregates NFU operations to implement a full-fledged network computation. Since the decomposition is not an easy task when data cannot be easily partitioned and complex data or control dependencies are involved, the primary design goal of the exProc is to allow users to concentrate on the decomposition of algorithm and data without dealing with the complexity of running operations over the network. The `lors_compute` tool from LoRS will take care of the runtime scheduling and management of network computations.

Figure 3.6 shows an example exProc. The `spot-check-rate` attribute is explained in Section 6.2. exProc borrows ideas from works in Grid workflow specification languages [Alt et al., 2005,Fahringer et al., 2005,Jagatheesan, 2004] that aim to shield the complexity of the underlying grid infrastructure. exProc allows users to define a graph of operations connected by control and data flow links in an intuitive way. A basic set of constructs is provided to simplify the specification of control, including sequential, parallel, loop and conditional sections. For example, the `while` loop construct allows an operation to be repeated with different arguments dynamically from the result of its predecessor until the stop condition is satisfied. Data dependency between operations is represented by input-output relationships that are automatically enforced in the workflow execution.

3.4 Supporting Legacy Applications

In the "pure" architectural model as described in [Beck et al., 2003], NFU operations are restricted to typeless bytes in RAM that are either local memory allocations or memory mappings of local disk allocations. Typecasting has to be performed and any data movement between depots must be explicitly directed by endpoints using IBP data movement calls

```
int main(int argc, char **argv) {                                                    main
    if (argc < 3) {...}
    fd = open(argv[idx], flags);
    read(buf, fd, size);
    lseek(fd, offset, where);
int nfu_func(int npara, nfu_para *paras) {
    if (npara < 3) {...}
    //associate fd with paras[idx-1];paras[idx-1].offset=0;
    fd = nfu_fs_open(idx, flags);
    //memcpy(buf,paras[idx-1].data+paras[idx-1].offset, size);paras[idx-1].offset+=size;   10
    nfu_fs_read(buf, fd, size);
    //switch (where) {case SEEK_SET: paras[idx-1].offset=offset; ...}
    nfu_fs_lseek(fd, offset, where);
```

Figure 3.7: Function substitutions in a legacy program

(e.g. `IBP_copy`). The need for type conversion in NFU operations is not a problem because it has to be handled by nearly all programs running across platforms, but there is still a gap between NFU operations and legacy applications.

File system calls File system is usually the only data abstraction layer in legacy applications (e.g. video transcoding and text mining) and programmers use standard file operations (e.g. `open`, `read` and `write`) to access application data stored in files.

Inter-process communication With the exception of embarrassingly parallel problems, many legacy applications require some form of communication among distributed parallel programs for various reasons.

To automate the migration from legacy programs to NFU operations, the `nfucc` tool is implemented to compile source code of a legacy program to a static or dynamic NFU operation without the knowledge of special data structures, header files or linkages. The `nfucc` tool runs on top of `gcc`. It constructs a list of necessary header files, performs a number of function substitutions for standard file operations and links with the `nfu_file` library. Similar to the `libxio` library, which implements the standard UN*X I/O alike interfaces to access files in the LN represented by an exNode, the `nfu_file` library is used to access data blocks in memory buffers instead of files.

Figure 3.7 shows the kind of function substitutions. The standard `main` function is replaced with the programming interface of NFU operations, where `nfu_func` is a user defined symbol to be used when loading a legacy program with the dynamic linking loader API. All file operations are replaced with their equivalents in the `nfu_file` library. For instance, the standard `open` function gets replaced with `nfu_fs_open`, which sets up an association between a memory buffer and a file descriptor. Subsequent read and write to the file descriptor will be applied to the memory buffer that has been mapped from an IBP allocation before `nfu_func` is loaded to run.

In LN, any communication between stand-alone depots must be explicitly controlled by endpoints using IBP data movement calls, which means that computation and communication (if there is any) in a single program unit must be invoked as NFU operations and IBP

data movement calls separately. Imagine we are doing text pattern search in a web log. Due to data partition, web log segments might not be perfectly separated at the end of a text line, i.e. the first and the last lines in a text block might be broken. Using pure data movement and NFU calls, the search process can be implemented using three apart calls: (i) finding the start of the last line `ll` if it is fragmented, (ii) copying `ll` to the depot where the next text segment is stored, and (iii) doing text search on the last line from the previous segment and the local segment, not including the last line. However, this is not the usual way to write a parallel program because programmers have to first identify computation and communication portions in their programs, and then apply the techniques presented in Section 5.1 to improve the performance when running a sequence of calls.

One way to avoid communication is to split some of the computation workload to endpoints. In our example, the first and last lines can be skipped and returned together with the partial search results to the endpoint where they are concatenated and searched to generate the final results. Although searching a thousand lines of text at an endpoint is not a big deal, doing a thousand "ghost area" matrix multiplication may overwhelm the endpoint.

Since communication is inevitable for problems that are not embarrassingly parallel and application developers are used to well-established programming models and libraries, a tradeoff is made to allow inter-depot communication in NFU operations through IBP data movement calls: `IBP_load`, `IBP_store` and `IBP_copy`. With this relaxation, common parallel programming models can be implemented and used in NFU operations. For instance, the logistical MPI implementation has been reported.

To enable this relaxation, the system call interposition framework presented in Section 3.2.1 is extended by allowing system calls (e.g. create, read from and write to a socket) made in IBP data movement calls. To ensure that no other systems calls can go through the sandbox, the call stack of an oplet is examined at every system call trap. Using the current and previous `ebp` base pointers obtained from the `PTRACE_PEEKDATA` calls, stack frames are traversed and the calling addresses are extracted. The backtrace addresses are then compared with the call stack graphs that have been generated by profiling the locally linked IBP data movement calls. If the addresses of the call sequence match the Deterministic Finite Automata of an IBP data movement call stack, the trapped system call is allowed. Otherwise, the running oplet is terminated. If communication among parallel computations is not frequent, (e.g., only one out of thousands of text lines is exchanged in the web log search example), the overhead of trapping system calls and checking against call graphs should be negligible.

Note that allowing IBP data movement calls in NFU operations does not necessarily imply that the LN architecture is impractical. Smart task analysis and scheduling tools may make this relaxation unnecessary. For instance, tools can transform a MPI program into a graph of NFU operations and IBP data movement calls, and then schedule their execution in the LN infrastructure from endpoints. This would be a good research topic, but it is not part of the dissertation.

Chapter 4

Modeling the Logistical Depot

Previous research in LN has shown the use of IBP for scalable data sharing and NFU for scalable network computation, by applying the end-to-end principles. In addition to its capability of being able to provide storage and processing services in a heterogeneous environment, performance of the logistical depot is critical because it is the most heavily used resource in the system. The ability to effectively support a reasonable workload (a number of distributed users with a number of data-intensive applications) will encourage its use and the migration of data-intensive applications from their traditional execution environments to LN.

Understanding the performance characteristics of logistical depots and evaluating approaches to scale them up are useful for the design of effective scheduling schemes to improve both application performance and depot resource utilization. We desire to develop a methodology to investigate the performance impacts of running storage and processing services simultaneously within the logistical depot. Assuming the hardware and software configurations of the logistical depot are fixed, our primary goal is to identify the performance effects of different mixes of data movement and data processing workloads, and to evaluate approaches to improve the performance of the logistical depot, primarily utilization and throughput.

Two commonly used evaluation techniques for performance characterization are measurement and modeling. As an initial step, measurements performed on an idle system can provide service time distributions of various components in a system. Further more, one can use a representative workload generated by actual or emulated users to understand the behavior of a system. Measurement yields the greatest fidelity with respect to the actual hardware and software configurations. However, measurements are often made without a structural model as guidance. Thus, they cannot be easily analyzed and conclusions drawn are difficult to generalize to other systems or even to the same system with small configuration changes.

Modeling includes simulation, statistical analysis and analytical methods. Modeling usually gives results in the form of response time, system throughput and device utilization under various workloads and configurations. Among them, the analytical method is the most popular because it can economically generate insights into a system. However, constructing analytical models often involves simplifying assumptions for the sake of mathematical tractability. As a result, they are usually gross approximations of real systems. The successes of these models depend heavily on how closely they can predict the performance of actual systems.

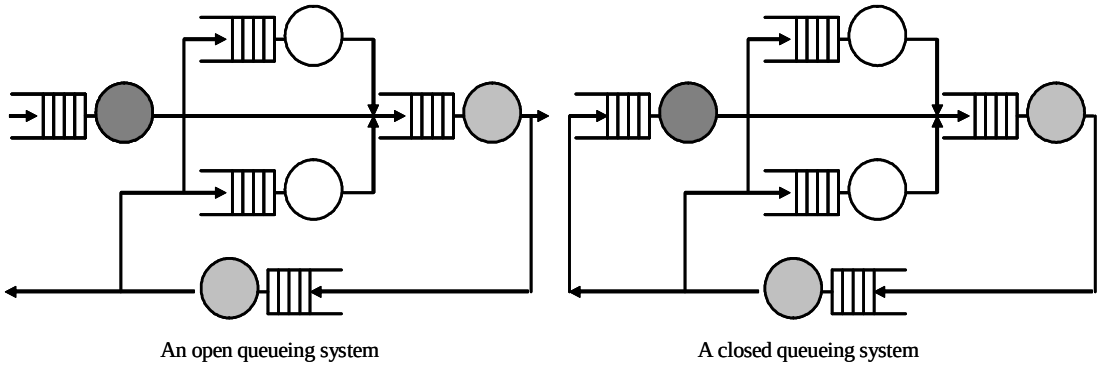


Figure 4.1: Two types of queuing networks.

Table 4.1: Three classes of jobs that the logistical depot is designed to serve

Management	**class 0**	Reserve and maintain storage allocations
Data Movement	**class 1**	Download from, upload to and copy between allocations
Data Processing	**class 2**	Transform bytes in storage allocations

This chapter attempts to combine the two techniques, measurement and analytical modeling, for the performance analysis of logistical depots. In particular, a closed queuing network model is chosen to represent the logistical depot. Initial measurements are made on a logistical depot to parameterize the model, and the results obtained from the analytical model are analyzed for the design of effective scheduling techniques at endpoints.

4.1 Model Development

In computer system modeling, a queuing network is a network of arbitrarily connected queues of jobs. Each queue of jobs is served by one or more servers (devices). A job that finishes its service in one queue may leave the network or join another queue. Two distinct types of queuing networks have been previously investigated are open systems and closed systems as illustrated in Figure 4.1. Jobs in an open system enter the network from exterior sources, and leave the network to external sinks. The number of jobs in the system is a random variable since jobs enter the network, are serviced at various servers, and exit. In contrast, jobs in a closed system do not enter or leave the network, but pass through various servers. The number of jobs in the system is a constant. It has been shown in [Gordon and Newell, 1967] that an open queuing system with exponential servers is equivalent to a closed system with one more stage of service.

CPU, disk and network channels are the resources where contention occurs in a logistical depot, and therefore require detailed modeling. Table 4.1 lists three types of jobs in a logistical depot. Job processing starts with evaluating a request. After the class of the job is identified, a process is dispatched to execute the corresponding service routine. The steps involved in each class of jobs are explained below. We will concentrate on the most important steps. The connection setup and exchange of protocol messages are not discussed here and they are not considered in this modeling exercise.

Storage management routines (e.g. `IBP_allocate` and `IBP_manage`) only cause a small amount of CPU, disk and network activity, therefore, they are excluded from the model.

The service routine for data movement jobs (`IBP_load`, `IBP_store` and `IBP_copy`) is modeled by the following two loops:

- `CPU processing 1` → `read from disk` → `CPU processing 1` → `write to network`;
- `CPU processing 1` → `read from network` → `CPU processing 1` → `write to disk`.

The CPU service components associated with the disk and network activities are modeled as a single service time `CPU processing 1`, which includes IBP protocol processing and OS interrupt handling. `CPU processing 1` ends when the job blocks on a disk or network I/O. Both loops seem inefficient because the disk and network activities are not overlapped. However, when there are multiple outstanding data movement jobs, disk and network activities belonging to different jobs tend to overlap. Similarly, the service routine for data processing jobs is modeled as a single loop:

- `CPU processing 2` → `read from|write to disk`

Similar to `CPU processing 1`, `CPU processing 2` involves protocol handling and OS related processing, and it stops when the job blocks on a disk I/O. More importantly, `CPU processing 2` transforms bytes in memory buffers. To separate data processing and data movement jobs, we require that users leave the transformed bytes in storage allocations when the transformation is done. If any user needs the output at different locations, data movement requests have to be explicitly invoked to direct the date transfer. As a result, no network activities are modeled for data processing jobs.

We assume that disk read and write activities are triggered by page faults in both data movement jobs (class one) and data processing jobs (class two). When the data movement routine or the data processing routine references a byte that is not in main memory, a disk read or write request will be issued by the OS to load the missing page.* By using the same mechanism to trigger disk activities instead of using a simpler `read` → `write` loop for class one jobs, we attempt to ensure that both classes of jobs have the same disk service time distribution, which is necessary for the queuing network model to have consistent independence balance equations for a solution.

A queuing network model attempts to represent in detail the processing of an individual job. From the above description, it seems to be a convenient representation of class one and class two jobs and it can provide us with a clear understanding of the logistical depot performance under various mixes of workloads. The closed queuing network model is chosen here in which multiple clients request storage or computation services. At any time, a client may have more than one outstanding request to a particular logistical depot. However, the clients are not explicitly modeled because they normally use non-shared resources and they primarily interact with the network module to send or receive data. We assume that the logistical depot being modeled operates under heavy demand conditions, i.e. there are sufficient number of jobs waiting for services, so that once a job leaves the system another job enters immediately and joins the end of a queue. Thus, the closed queuing network model is an appropriate representation of the logistical depot.

Our model is essentially a multi-class queuing network because we have two distinct classes of jobs. In multi-class queuing networks, the choice of scheduling discipline is important because jobs are distinguishable. There are a large number of scheduling disciplines

*In Linux and many other operating system implementations, more than one pages will be loaded because of prefetching.

that can be represented in a multi-class queuing network. For our purposes, however, the following three disciplines are sufficient. There are referred as type 1, type 2 and type 3 servers in the BCMP [†] paper [Baskett et al., 1975].

Type 1 The service discipline is first-come-first-served (FCFS). All jobs have the same exponential service time distribution. Type 1 servers are appropriate models of disks because preemptive scheduling is not efficient for disks.

Type 2 The service discipline is processor sharing (PS) (i.e. each job receives $1/n$ of the service when there are n jobs at the server). Each class of jobs may have a distinct exponential service time distribution. Type 2 servers are appropriate for CPUs because round robin (RR) scheduling approaches PS.

Type 3 Each class of jobs may have a distinct exponential service time distribution. The service rate depends on the number of jobs at the server (i.e. the number of virtual servers is greater than or equal to the maximum number of jobs in queue). Type 3 servers are appropriate for routing delays in a network.

In order to calculate the steady state probabilities and performance statistics such as utilization and throughput, a number of assumptions are made in the modeling process for analytical tractability.

1. The CPU processing of class one and class two jobs are modeled as a single type 2 server. This is a reasonable assumption for data processing jobs because jobs from distributed clients may belong to different applications, and therefore require differing amounts of processing time. However, the exponential assumption is pessimistic for data movement jobs because the interrupt service routine for a page fault or a network I/O has much smaller variance in the service time.

2. The processing of disk read or write caused by a page fault is modeled as a type 1 server. The assumption of an exponential distribution for page fault service times gains some credence from the realization that the durations of disk access, transfer latency and posting delays are in general independent of one another. If there are multiple disks, the model assumes that there is an equal probability that the requests are routed to any one of the disks.

3. The network module of a logistical depot contains a finite number of outstanding `send|receive` buffers, therefore the network interface is explicitly modeled as multiple type 3 servers. Each server represents a group of connections to clients that have similar network conditions and processing speeds. This is a reasonable approximation and the details are discussed in Section 4.3.

Note that our goal is not to accurately predict the capacity of a logistical depot, but to provide an understanding of the overall performance characteristics, and to identify the effects of different mixes of computation and data movement workloads. Therefore, the accuracy of the results obtained from the queuing network model with the assumptions listed above will be considered sufficient.

[†]The acronym is composed of a concatenation of the first letters of the last names of the authors.

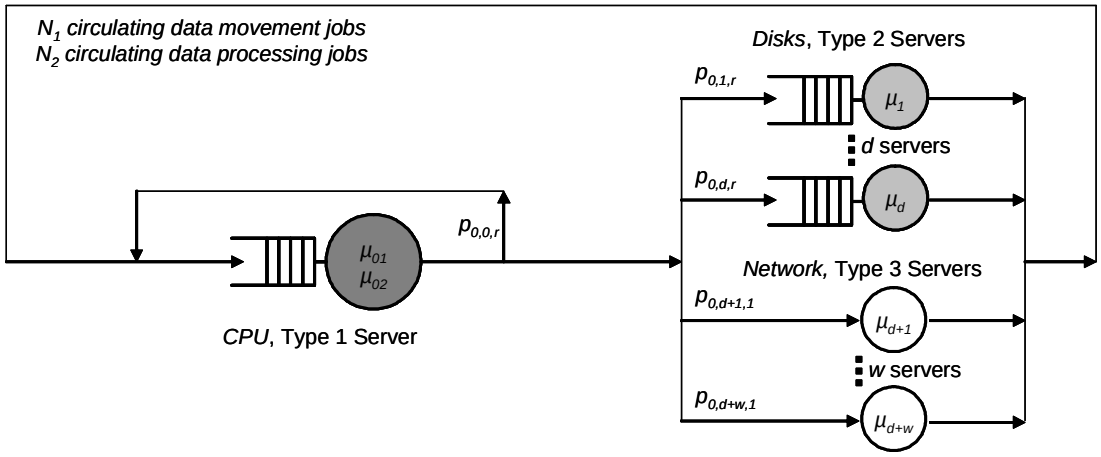


Figure 4.2: A closed queuing network model of logistical depots

The final model is represented schematically in Figure 4.2. The system has $r = 2$ classes of circulating jobs and $d + w + 1$ servers, with server 0 representing the CPU, server $1, ..., d$ representing the disks and server $d + 1, ..., d + w$ modeling the network delays. A class r job leaving the CPU proceeds to the j^{th} disk or network delay server with probability $p_{0,j,r}$, and jobs leaving the peripheral servers proceed directly to the CPU with probability one. Unlike the general multi-class queuing network, jobs in Figure 4.2 do not change their classes during service in the network. Other parameters of the model are:

- N_1 and N_2 are the number of class one (data movement) and class two (data processing) jobs respectively. $N = N_1 + N_2$ is the total number of jobs in the system;
- μ_{01} and μ_{02} are the service rate of class one and class two jobs at the CPU;
- $\mu_1, ..., \mu_d$ are the pagefault service rates of disks and $\mu_{d+1}, ..., \mu_{d+w}$ are the service rates of network delay servers;
- $p_{0,1,r}, ..., p_{0,d,r}$ are the transition probabilities of class r $(r = 1, 2)$ jobs from the CPU to the disks and $p_{0,d+1,1}, ..., p_{0,d+w,1}$ are the transition probabilities of class one jobs from the CPU to network delay servers;
- $p_{0,0,r}$ is the probability of one class r job terminates its execution and a new job of the same class enters the network. The probability of a job making exactly n requests for CPU processing is $(1 - p_{0,0,r})^{n-1} p_{0,0,r}$. Therefore, the expected number of CPU requests per job is

$$\sum_{n=1}^{\infty} n(1 - p_{0,0,r})^{n-1} p_{0,0,r} = \frac{p_{0,0,r}}{{p_{0,0,r}}^2} = \frac{1}{p_{0,0,r}} \tag{4.1}$$

As mentioned earlier, we don't consider changing the hardware or software configurations for the performance improvements of a logistical depot. For example, if the disks were to be a bottleneck under a particular workload, an enhancement could be caching files to avoid disk access. Instead of making this modification, we desire to balance the CPU and disk activities by adjusting the mix of class one and class two jobs injected by distributed users.

4.2 Steady State Analysis of the Model

In our model, two classes of jobs travel through the network according to certain transition probabilities without changing their classes, i.e. a job of class r that completes service at server i requires service at server j with a probability $p_{i,j,r}$. The resulting transition matrix can be considered as defining a Markov chain whose states are labeled by the pairs (i, r). Let n_{i_r} be the number of class r jobs at server i in state S of the model. S is represented by a vector $(x_0, x_1, ..., x_n)$ where $n+1$ is the number of servers in the system and x_i represents the prevailing conditions at server i. Note that $n = d + w$ in the logistical depot model where d is the number of disk servers and w is the number of network delay servers. The interpretation of x_i depends on the type of server i.

- For type 1 servers, $x_i = [x_{i_1}, x_{i_2}, ..., x_{i_{n_i}}]$, where n_i is the number of jobs at server i and x_{i_j} $(1 \leq j \leq n_i)$ is the class of the job that is at the j^{th} position in FCFS order. For any network containing one or more type 1 servers, the expression for a state S of the network is long to write. Consequently, writing expressions for the balance equations to find the steady state probabilities is a tedious task. As an alternative, we define an aggregate state as the number of jobs of each class in a type 1 server, i.e. $x_i = [n_{i_1}, n_{i_2}]$ where n_{i_r} is the number of class r jobs in server i. This form is preferred because they are more concise and lead to computationally more efficient methods of calculating the normalization constant for closed networks.

- For type 2 and type 3 servers, $x_i = [n_{i_1}, n_{i_2}]$ where n_{i_r} is the number of class r jobs in server i. Note that $n_{i_2} = 0, i = d + 1, ..., d + m$ because only class one jobs enter the network delay servers in the model. x_i has the desired form because positions of jobs in the queues of type 2 and type 3 servers are not part of the state representation.

Steady state probabilities of queuing networks can be obtained by solving global balance equations or local balance equations. The global balance equations equate the rate at which a process enters a state to the rate at which the process leaves that state. The number of states and the complexity of the global balance equations increase with the number of classes and jobs. The local balance equations equate the rate at which the process enters a state due to the movement of a job into a given queue to the rate at which the process leaves that state due to the movement of a job out of that queue, i.e. local balance is concerned a queue isolated from the rest of the network. Local balance is a sufficient but not necessary condition for global balance.

Suppose a state S of the system be $(x_0, ..., x_n)$ where $\sum_{i=0}^{n} n_{i_1} = N_1$ and $\sum_{i=0}^{n} n_{i_2} = N_2$, and the steady state probability of being in state S is $P(S)$. The system can transit out of this state into $(x_0, ..., [n_{i_1} - 1, n_{i_2}], ..., [n_{j_1} + 1, n_{j_2}], ..., x_n)$ if $p_{i,j,1} > 0$, and the system can transit into this state from $(x_0, ..., [n_{i_1} + 1, n_{i_2}], ..., [n_{j_1} - 1, n_{j_2}], ..., x_n)$ if $p_{i,j,1} > 0$. Let $R(S_i)$ be the rate of flow leaving state S_i and $R(S_i \rightarrow S_j)$ be the rate of flow from state S_i to state S_j. A solution for the steady state probabilities must satisfy the global balance equations

$$\forall S_i, \sum_{\forall S_j} P(S_j) R(S_j \rightarrow S_i) = P(S_i) R(S_i). \tag{4.2}$$

In contrast, local balance equations deal with a single queue in the network. Easily observed, the rate at which the system leaves a state S due to the movement of a class r job

Table 4.2: Server type dependent net service rates

Server type	Type 1	Type 2	Type 3
$g(\mu_{ir})$	$\mu_i n_{i_r}/(n_{i_1}+n_{i_2})$	$\mu_{ir} n_{i_r}/(n_{i_1}+n_{i_2})$	$\mu_{ir} n_{i_r}$

out of queue i is $P(S)g(\mu_{ir})$. Function g depends on the type of server i as listed in Table 4.2. Considering all the possible movements of class r jobs from queue j in state S' into queue i in state S, the rate at which the system enters S due to the movement of the job into queue i is $\sum_{p_{j,i,r}>0} P(S')g(\mu_{jr})p_{j,i,r}$. By equating the departure rate and the arrival rate, the local balance equations can be written as

$$P(S)g(\mu_{ir}) = \sum_{p_{j,i,r}>0} P(S')g(\mu_{jr})p_{j,i,r}. \tag{4.3}$$

Suppose a valid state S of the network model shown in Figure 4.2 be $(x_0, ..., x_n)$, the local balance equations are listed below with the left side being the rate of a class r job leaving a particular queue in state S and the right side being the rate of a class r job entering that queue. Clearly, the global balance equation can be obtained by equating the sum of the left sides to the sum of the right sides.

$$\begin{aligned}
P(S)(\frac{n_{0_1}}{n_{0_1}+n_{0_2}})\mu_{01} &= P(S)(\frac{n_{0_1}}{n_{0_1}+n_{0_2}})\mu_{01}p_{0,0,1} \\
&\quad + \sum_{i=1}^{d} P([n_{0_1}-1, n_{0_2}], ..., [n_{i_1}+1, n_{i_2}], ..., x_n)(\frac{n_{i_1}+1}{n_{i_1}+n_{i_2}+1})\mu_i \\
&\quad + \sum_{i=d+1}^{d+w} P([n_{0_1}-1, n_{0_2}], ..., [n_{i_1}+1, 0], ..., x_n)(n_{i_1}+1)\mu_i \\
P(S)(\frac{n_{0_2}}{n_{0_1}+n_{0_2}})\mu_{02} &= P(S)(\frac{n_{0_2}}{n_{0_1}+n_{0_2}})\mu_{02}p_{0,0,2} \\
&\quad + \sum_{i=1}^{d} P([n_{0_1}, n_{0_2}-1], ..., [n_{i_1}, n_{i_2}+1], ..., x_n)(\frac{n_{i_2}+1}{n_{i_1}+n_{i_2}+1})\mu_i \\
P(S)(\frac{n_{i_1}}{n_{i_1}+n_{i_2}})\mu_i &= P([n_{0_1}+1, n_{0_2}], ..., [n_{i_1}-1, n_{i_2}], ..., x_n)(\frac{n_{0_1}+1}{n_{0_1}+n_{0_2}+1})\mu_{01}p_{0,i,1}, \forall_{i=1}^{d} \\
P(S)(\frac{n_{i_2}}{n_{i_1}+n_{i_2}})\mu_i &= P([n_{0_1}, n_{0_2}+1], ..., [n_{i_1}, n_{i_2}-1], ..., x_n)(\frac{n_{0_2}+1}{n_{0_1}+n_{0_2}+1})\mu_{01}p_{0,i,2}, \forall_{i=1}^{d} \\
P(S)n_{i_1}\mu_i &= P([n_{0_1}+1, n_{0_2}], ..., [n_{i_1}-1, 0], ..., x_n)(\frac{n_{0_1}+1}{n_{0_1}+n_{0_2}+1})\mu_{01}p_{0,i,1}, \forall_{i=d+1}^{d+w}
\end{aligned} \tag{4.4}$$

The important BCMP paper [Baskett et al., 1975] established that queuing networks satisfied the local balance equations also have a product form solution. The steady-state probabilities are given by

$$P(x_0, x_1, ..., x_n) = \frac{1}{G} f_0(x_0)f_1(x_1)...f_n(x_n), \tag{4.5}$$

where G is a normalizing constant to make the steady state probabilities sum to one and each f_i is a function that depends on the type of server i. Let e_{ir} be the probability that a class r job passes through server i on its passage from the source to the destination, $f_i(x_i)$ can be represented as

- For type 1 servers, $f_i(x_i) = n_i!\ \prod_{r=1}^{2}\{(1/n_{ir}!){e_{ir}}^{n_{ir}}\}(1/\mu_i)^{n_i}$, where $n_i = n_{i_1} + n_{i_2}$ and μ_i is the service rate for both class 1 and class 2 jobs;
- For type 2 servers, $f_i(x_i) = n_i!\ \prod_{r=1}^{2}\{(1/n_{ir}!)(e_{ir}/\mu_{ir})^{n_{ir}}\}$, where $n_i = n_{i_1} + n_{i_2}$ and μ_{ir} is the service rate for class r jobs at server i;
- For type 3 servers, $f_i(x_i) = \prod_{r=1}^{2}\{(1/n_{ir}!)(e_{ir}/\mu_{ir})^{n_{ir}}\}$.

Since a class 1 (data movement) job could be with any one of the CPU, disks or network delay servers during its lifetime, therefore,

$$e_{01} + \sum_{k=1}^{d} e_{k1} + \sum_{k=d+1}^{d+w} e_{k1} = 1 \tag{4.6}$$

However, a class 2 (data processing) job does not cause any network activity and it can only be queued in the CPU or disk servers, thus

$$e_{02} + \sum_{k=1}^{d} e_{k2} = 1 \tag{4.7}$$

Clearly, the probability that a class r job passes through server i is the sum of the probabilities of that job passes through all server k that can transit to server i with a transition probability p_{kir}, that is

$$e_{ir} = \sum_{k=0}^{d+w} e_{kr} \times p_{kir} \tag{4.8}$$

In Figure 4.2, jobs leaving the peripheral servers proceed directly to the CPU server with probability one and jobs do not transit between the disk servers and the network delay servers without passing through the CPU server, i.e. $p_{k,0,r} = 1$ and $p_{k,i,r} = 0$ for $k, i = 1, ..., d + w$. Thus, equation 4.8 is equivalent to the following:

$$\begin{aligned}
e_{01} &= e_{01}p_{0,0,1} + \sum_{k=1}^{d+w} e_{k,0,1} \\
e_{02} &= e_{02}p_{0,0,2} + \sum_{k=1}^{d} e_{k,0,2} \\
e_{k1} &= e_{01}p_{0,k,1}, k = 1, ..., d + w \\
e_{k2} &= e_{02}p_{0,k,2}, k = 1, ..., d
\end{aligned} \tag{4.9}$$

The equations 4.6, 4.7 and 4.9 can be easily solved to obtain e_{kr}, which is the probability that a class r job passes through server i during its circulation in the system.

$$
\begin{aligned}
e_{01} &= 1/(1 + \sum_{k=1}^{d+m} p_{0,k,1}) = 1/(2 - p_{0,0,1}) \\
e_{02} &= 1/(1 + \sum_{k=1}^{d} p_{0,k,2}) = 1/(2 - p_{0,0,2}) \\
e_{k1} &= p_{0,k,1}/(2 - p_{0,0,1}), k = 1, ..., d + w \\
e_{k2} &= p_{0,k,2}/(2 - p_{0,0,2}), k = 1, ..., d
\end{aligned}
\tag{4.10}
$$

$P(x_0, x_1, ..., x_n) = f_0(x_0)f_1(x_1)...f_n(x_n)/G$ with the above e_{ir} satisfies the local balance equations 4.4, therefore, it also satisfies the global balance equation. The normalization constant G can be calculated using the fact that the sum of all valid $P(x_0, x_1, ..., x_n)$ over a restricted state space will be equal to one. Since every state $S = (x_0, x_1, ..., x_n)$ in which $\sum_{i=0}^{n} n_{i_1} = N_1$ and $\sum_{i=0}^{n} n_{i_2} = N_2$ represent a possible state of the system, it follows that

$$
G = \sum_{\forall \sum_{i=0}^{n} n_{i_1} = N_1, \sum_{i=0}^{n} n_{i_2} = N_2} f_0(x_0)f_1(x_1)...f_n(x_n). \tag{4.11}
$$

Computational efficient algorithms [Buzen, 1973, Reiser, 1977] have been published to calculate the normalization constant G. However, it is not the focus of this dissertation. A recursive implementation is employed to calculate G for simplicity.

4.3 Parameterization

A closed queuing network model logistical depots has been established and solved in Section 4.1 and 4.2. To understand its implications on a production logistical depot, several stand-alone measurements have been performed to parameterize a system configured as a logistical depot. The system runs Linux-2.6.20 and its configuration is shown in Table 4.3.

Statistics of different operations performed in a typical data movement job or data processing job need to be obtained through measurements. If a measurement cannot be obtained, an estimate for the corresponding parameter is used, which is intended merely to provide an idea of the range. Two design goals of the experiments are

Functionality Service time distributions of the CPU, disk and network delay servers and transition probabilities between various components in the model should be measured with acceptable accuracy;

Simplicity The operating system kernel seems to be the most appropriate place to obtain the service time of a disk or network I/O. The instrumentation code planted into the kernel should be easy to manage, and it should not impose too much overhead.

Table 4.3: Configuration of a logistical depot

CPU	Memory	Disk	Network Interface Card
2.2GHz AMD64	2GB	66MHz SATA	Gigabit Ethernet PCI Express

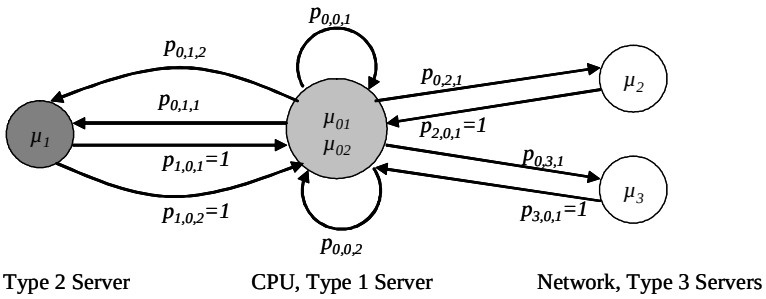


Figure 4.3: The model of a logistical depot (`dram.cs.utk.edu`) for parameterization.

Table 4.4: Examples of class two CPU service times (ms per request)

plain text search	64×64 matrix multiplication	raycasting for volume visualization
0.8	2.9	13.6

We instrument the Linux kernel scheduler (`kernel/sched.c`) with code to log process scheduling events. Instead of using the heavy weight kernel `printk`, the log entries are written in a memory buffer during the experiment and then dumped to a file after the experiment finishes. In this way, the logging overhead is kept minimal and the system behavior is not affected by `syslogd` that outputs log messages. A set of stand-alone experiments are designed, each generating one of the breakdowns in a typical class one or class two job, for example disk and network I/O requests. By designing the kernel patch and experiments appropriately, we have been able to separately quantify the parameters of the queuing network model shown in Figure 4.3. We are fully aware of the fact that these numbers obtained in the following subsections are approximates of their true counterparts. Since we didn't run other services other than the experiment process, the measurements are considered accurate enough.

4.3.1 Service Time of CPU Requests

Service times of a class one ($1/\mu_{01}$) or class two ($1/\mu_{02}$) CPU request can be calculated by $t_1 - t_2$ where t_2 is the time that a process is scheduled to take over the CPU and t_1 is the time the process gives up the CPU voluntarily because the service has been completed. During $t_1 - t_2$, the process may be stopped because a time slice has expired. These data points need to be identified and purged. As mentioned earlier, the exponential assumption is weak for class one jobs as shown in Figure 4.4.

Since different applications may have distinct CPU service time distributions, a general distribution for all possible class two jobs cannot be easily obtained. Instead, we obtain the mean service times of three typical class two applications in Table 4.4. The raycasting is performed with image size 800 and step size 0.1.

While there may be considerable dependence among the statistical properties of jobs from a particular user, these jobs begin their service intervals at widely spaced times, mixed with jobs from many other users, therefore making the assumed exponential service time distribution of class two jobs a reasonable approximation. The mean service time of class one jobs in Figure 4.4 is about 0.067 ms/request. Compared with those of class two jobs,

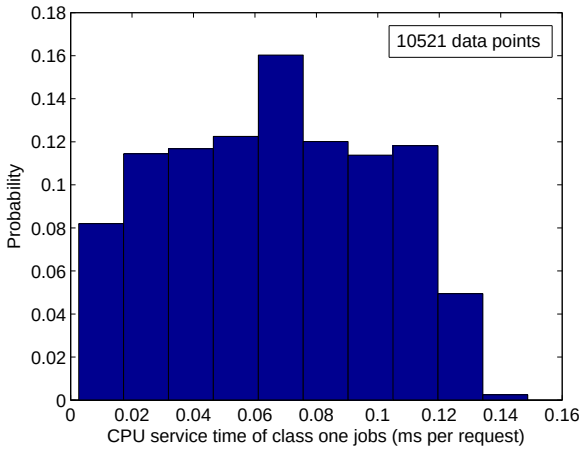


Figure 4.4: Histogram of CPU service times for class one (data movement) jobs

it has a much smaller utilization of the CPU, thus the exponential assumption should not have a significant impact to the accuracy of the model.

4.3.2 Service Time of Disk I/O

Service times of page faults (μ_1) can be calculated as $t'_1 - t'_2$, where t'_2 is the time a process gives up CPU voluntarily because of a blocking disk I/O request and t'_1 is the time that process is ready to be scheduled because the I/O has been completed. The distribution for service times of page faults is also assumed to be exponential. Figure 4.5 shows that the distribution is not skewed and approximating it by an exponential distribution with 0.6 ms/request as the mean will not have a significant effect [Price, 1976].

In order to measure the time spent on every page fault, the experiment program keeps generating page faults memory mapped to four different files on disk alternately, emulating four simultaneous class one jobs. We believe the distribution tends to be "more exponential" when there are more independent users accessing the disk, and we assume read from the disk and write to the disk because of a page fault have the same service time distribution. Note that the number measured is the service time to read or write several system pages (16 pages, totaling 64KB) instead of a single page due to the anticipatory prefaulting in the Linux page fault handler.

4.3.3 Service Time of Network I/O

Service times of a network I/O (i.e. $1/\mu_2$ or $1/\mu_3$) can be calculated as $t'_1 - t'_2$ as well. We assume the service time of sending and receiving have the same distribution. Figure 4.6 shows the network service time distributions of sending data from `dram.cs.utk.edu` (connected to a 100Mbps Ethernet switch) to two hosts in the wide area. Similar to disk I/O, approximating them by exponential distributions will not affect performance.

The distribution of mean network I/O service times of 45 wide area links originating from `dram.cs.utk.edu` is shown in Figure 4.7. The plot approximates the combination of two normal distributions with means 61.4 ms/requests and 136.7 ms/request. This motivates the use of two separate delay servers in the model, each representing a group of hosts whose mean service time fits in one of the normal distributions.

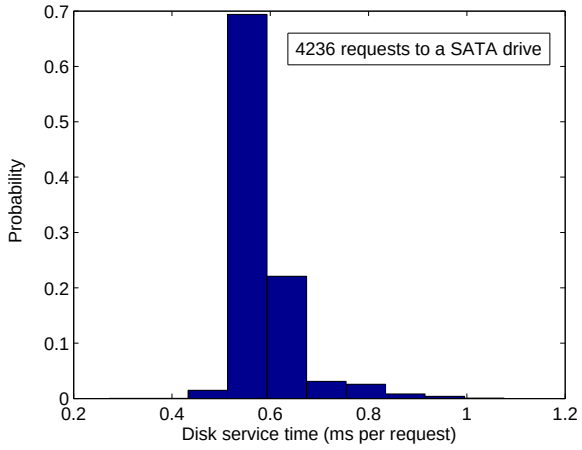


Figure 4.5: Histogram of page fault service times

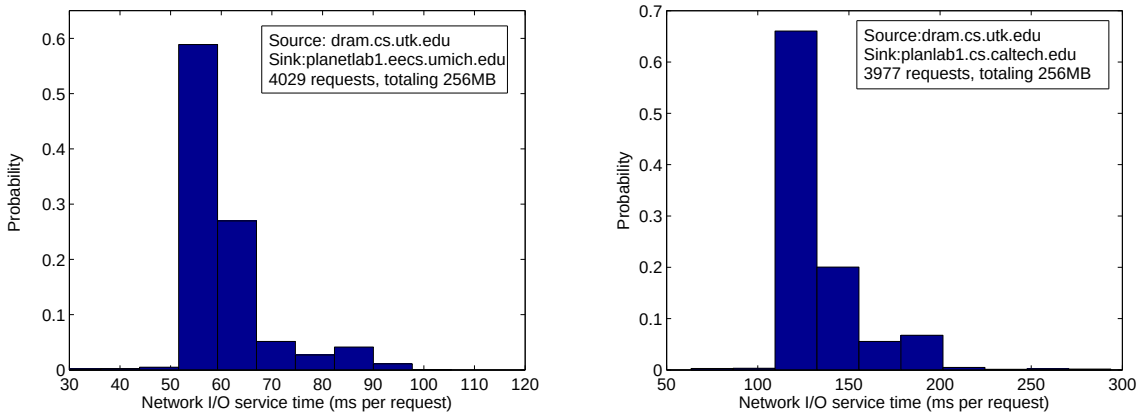


Figure 4.6: Network service time Histograms of two wide area links

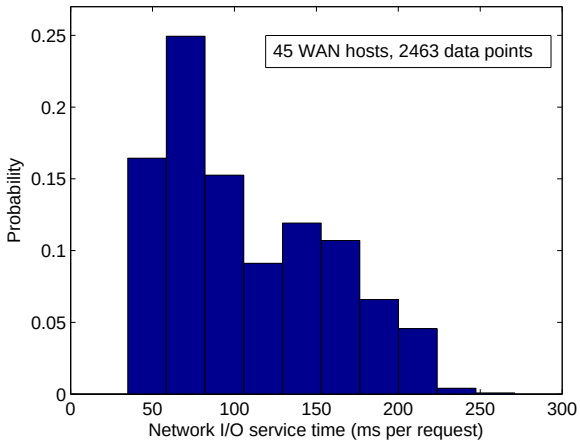


Figure 4.7: Histogram of mean network I/O service times of 45 wide area links.

4.3.4 Transition Probabilities

Recall that $p_{0,0,1}$ and $p_{0,0,2}$ determine the expected number of iterations that a class one or class two job circulates in the system. For example, $p_{0,0,1} = 0.01$ means that a class one job repeats the `cpu`→`disk`→`cpu`→`network` cycle 100 times before exiting the system. It is clear that smaller $p_{0,0,r}(r = 1, 2)$ makes a longer job execution.

Class two jobs make use the CPU server and the disk server. Thus, the transition probability from the CPU server to the disk server $p_{0,1,2} = 1 - p_{0,0,2}$. Class one jobs could depart from the CPU server to both the disk server and the network delay servers, thus $1 - p_{0,0,1} = p_{0,1,1} + p_{0,2,1} + p_{0,3,1}$. We observe that the average number of page faults and network I/O requests in a class one job are roughly equal, therefore $p_{0,1,1} = p_{0,2,1} + p_{0,3,1}$. For example, sending 256MB data across the wide area causes on average 4110 page faults and 3956 network I/O requests. By measuring the area covered by the two normal distributions in Figure 4.4(b), $p_{0,2,1}/p_{0,3,1} = 1.3$. It follows that

$$\begin{aligned} p_{0,1,1} &= (1 - p_{0,0,1})/2; \\ p_{0,2,1} &= 0.57(1 - p_{0,0,1})/2; \\ p_{0,3,1} &= 0.43(1 - p_{0,0,1})/2. \end{aligned} \tag{4.12}$$

4.4 Model Results

To compare the performance of various system configurations, model results are presented and interpreted. We are primarily interested in the utilization of system resources. At the same time, we consider throughput of the system from a user's perspective. These results provide us with a quantitative understanding of the behavior of logistical depots under different workloads of data movement and data processing jobs.

4.4.1 Device Utilization

Utilization of a device is equal to the portion of time that the device is not idle. Let a_i denote the steady state probability that the $i^{th}, i = 0, ..., d+w$ device is not idle. Since the i^{th} device is active if and only if $n_{i_1} + n_{i_2} > 0$, it follows that

$$a_i = \sum_{\forall_{n_{i_1}+n_{i_2}>0}} P(x_0, x_1, ..., x_n) = \sum_{\forall_{n_{i_1}+n_{i_2}>0}} \frac{1}{G} f_0(x_0) f_1(x_1) ... f_n(x_n) \tag{4.13}$$

The utilization curves for the CPU and disk with various configurations are shown in Appendix Figure 7.1, 7.2 and 7.3. Indents in the curves are caused by rounding the number of jobs to the nearest integer. In general, disk utilization increases with the number of jobs in the system. The same trend holds for CPU utilization except when $\mu_{02} = 0.4ms$. In that case, disk is the system bottleneck since $\mu_{02} < \mu_1$. Other observations from these plots are

- "Heaviness" of class two jobs affects resource utilization significantly. For example, with $1/\mu_{02} = 2.0ms$, the system has nearly 100% CPU utilization and 72% disk utilization. In contrast, with $1/\mu_{02} = 1.2ms$, the lines with green triangle in Figure 7.1, the system can improve the disk utilization to 86% without sacrificing the CPU utilization. On a system with fixed hardware configuration, it is desirable to choose class two jobs with appropriate mean service time to make a balanced system.

- The length of jobs has very limited impacts to system utilization. Usually a longer job is preferred because it has less execution overhead compared with dividing it into a bunch of shorter jobs. However, in a distributed system, a longer job means more unpredictability in timeouts and losing more work when a fault occurs. Choosing the right length of jobs requires to consider fault rate in a system and depot resource allocation policy at the same time.

- More heavy weight class two jobs in a system, more jobs are waiting for the CPU, making a low disk utilization. In a unbalanced system ($1/\mu_{02} = 2.0ms$ and $N_1 = N_2$, the lines with solid sky blue rectangle in Figure 7.3), system utilization can be improved by choosing an optimal job mix, for example 70% of class one jobs and 30% of class two jobs, the lines with empty red rectangle in Figure 7.3.

4.4.2 Throughput

Suppose a logistical depot is observed for time T. Let the amount of time that the CPU is busy during this time interval for class r jobs is T_{0r}. Since the CPU is active for a class r job if and only if $n_{0_r} > 0$, it follows that

$$T_{0r} = \sum_{\forall_{n_{0_r}>0}} \{P(x_0, x_1, ..., x_n) T \frac{n_{0r}}{(n_{0_1} + n_{0_2})}\} = \sum_{\forall_{n_{0_r}>0}} \{\frac{1}{G} f_0(x_0) f_1(x_1) ... f_n(x_n) T \frac{n_{0r}}{(n_{0_1} + n_{0_2})}\} \tag{4.14}$$

Equation 4.1 shows that the expected number of CPU requests per job is $1/p_{0,0,r}$, thus the expected amount of CPU processing time per class r job is $1/\mu_{0r} p_{0,0,r}$. It follows that the expected number of class r jobs processed during the time interval is $T_{0r}/(1/\mu_{0r} p_{0,0,r})$. Hence the average number of class r jobs processed per unit time t_r is

$$t_r = \sum_{\forall_{n_{0_r}>0}} \{\frac{1}{G} f_0(x_0) f_1(x_1) ... f_n(x_n) \frac{n_{0r}}{(n_{0_1} + n_{0_2})}\} \mu_{0r} p_{0,0,r} \tag{4.15}$$

The throughput curves for class one and class two jobs with various configurations are shown in Appendix Figure 7.4, 7.5 and 7.6. In general, throughput of class one jobs increases with the number of class one jobs in the system. This is because the bottleneck for class one jobs is the network delay servers whose net service rates increase with the number of customers. As a result, the same trend does not hold for class two jobs, whose bottleneck depends on their mean service time. Other observations from these plots are

- Due to PS at the CPU and FCFS at the disk, the throughput of class two jobs decreases and that of class one jobs increases with a larger class two mean service time. Its implications are: for logistical depots, it is important to choose class two jobs with the right "weight" for the best system throughput; for users, choosing a large mean service time does not gain any benefit.
- The length of class two jobs does not impact the throughput of class one jobs. This is a desirable property because different classes of jobs are isolated. By using a shorter job, one can improve job response time without losing utilization of remote resources.
- The percent of class two jobs in a system has a much larger influence on the throughput than class one jobs. On a heavily loaded system, sacrificing a few class two jobs to get more class one jobs might be an option for better system throughput.

As would be expected, network is still the performance bottleneck of wide area data-intensive computing systems due to limited bandwidth and long latency. We have seen that the network delay servers in our mode have a large mean service time. One way to improve bandwidth is to run multiple jobs on a set of logistical depots. The latency issue can be lightened if data can keep flowing in a network connection. In both cases, an intelligent endpoint scheduler is needed to manage job execution in the network so that an adequate application performance can be achieved.

Chapter 5

Scalable Scheduling of Network Computations

Data-intensive applications typically use large amounts of CPU cycles and extremely large datasets to solve scientific problems. However, the design intent of IBP and NFU is to maximize the generality of services that can be provided by intermediate nodes. The idea of generality includes the ability to support a widely varied set of high level applications and the ability to subsume a highly heterogeneous collection of low level technologies, for example, general-purpose computation on FPGA and graphics hardware. LN approaches generality by exposing a model of services that is very fine-grained. IBP allocations are limited in size and duration. Similarly, NFU imposes limits on CPU cycles and storage needed by a computation. As a result, fragmentation of both the processing algorithm and the data it transforms must be applied, either at a single node or across many nodes with attendant data movement for fault tolerance and possible parallelism. It is the responsibility of endpoints to orchestrate a large amount of fine-grained computations distributed in the network. The adaptive scheduling techniques developed can scale down to a single intermediate node to hide network latency and can scale up to a network of heterogeneous nodes for dynamic load balancing and workload control.

5.1 Hiding Network Latency in the Wide Area

Fragmentation of computation and data introduces detailed control over the functioning of intermediate nodes. It is often objected that LN cannot possibly provide adequate performance in applications where the sequence of operations to be performed is dynamically determined and issued during the execution of an application protocol, due to high synchronous communication latencies between endpoints and intermediate nodes.

Instruction caching and scheduling have been successfully used in computer architectures to hide instruction issue latency on a single processor [Patterson and Hennessy, 1998]: *instruction caching* fetches operations from the high speed cache instead of from the low speed main memory and *instruction scheduling* populates multiple operations to different stages of a processor pipeline by reducing control and data hazards. Considering the intermediate node as a instruction execution unit and the endpoint as an instruction issue unit, both techniques can be borrowed to reduce the instruction issue latency in the wide area.

Prior to HTTP/1.1 [Fielding et al., 1997], a separate TCP connection was established for each URL in a web page. The use of embedded images and other objects often require a client to make multiple requests to the same web server. In HTTP/1.1, persistent connections are the default behavior of any HTTP connection. Its advantages include:

- CPU and memory resources are saved in intermediate nodes (routers) and endpoints (clients, servers, proxies, gateways, or caches);
- Network congestion is alleviated by reducing the number of TCP open/close packets, and by giving TCP sufficient time to determine the congestion state of a network;
- Latency on subsequent requests is reduced since there will be no time spent to setup a TCP connection.

Similarly, an asynchronous interface of the IBP protocol is developed to enable operations to be issued and responses to be received separately. With that interface, we can pipeline IBP and NFU operations on a single persistent TCP connection: later operations can be issued before earlier operations have completed, up to a maximum pipelining depth that is controlled by the local policies of a depot. Requests in the pipelining are received by the depot in-order. However, response from the depot can be out-of-order to maximize performance and minimize the use of buffer resources at the depot, using per-operation tags to keep service requests correctly associated with their responses.

In computer architectures, a hazard is a potential problem that can happen in a pipelined processor supporting out-of-order execution of instructions. It refers the potential danger when a CPU tries to simultaneously execute multiple instructions with certain data dependence or control dependence. Hazards prevent next instruction from executing during its designated clock cycle, thus reduce the performance of pipelining. There are three fundamental types of hazard:

- Structural hazards occur when a piece of hardware is needed by more than one instruction pipelines at the same time;
- Data hazards occur when operands of an instruction depend on the result of a prior instruction that is still in the pipeline. Typical data hazards are: read after write (RAW), write after read (WAR) and write after write (WAW);
- Control hazards (also known as branch hazards) occur when the processor is told to branch, but the information needed to make a conditional branch is not available yet.

Any IBP or NFU operation in the pipelining has three stages: sending the request, performing the operation, and receiving the response. We don't consider structural hazards because depots are shared among operations using per-operation tags by default. Data hazards between adjacent operations are detected dynamically and enforced by the depot, using read and write locks. In this section, instruction caching and scheduling techniques in the context of wide area computation are developed and evaluated to deal with control hazards. This is the first work that combines instruction scheduling techniques such as loop unrolling and speculation to hide instruction issue latency in the wide area.

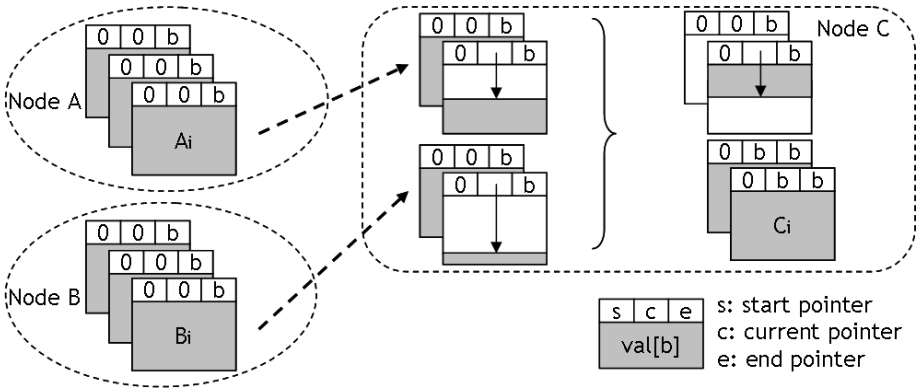


Figure 5.1: Global flow of data in the distributed merge example

5.1.1 A Distributed Merge Example

In order to demonstrate the power of instruction scheduling and instruction caching techniques to increase the performance of network applications implemented using IBP and NFU, we present an example of how these approaches can be used to perform a merge operation on ordered data streams distributed in the wide area. Consider a scenario in which two large streams of linearly ordered data records originate from different locations, A and B, within the wide area network, and a single ordered collection, consisting of the merged contents of both input streams, has to be generated and stored at a third location, C. As shown in Figure 5.1, the merging of the streams is a multistage process that requires the movement of data from both A and B to C, as well as the application of a record-level comparison at C in order to generate the results.

To simplify the example, we assume that each data block consists of a sequence of data records, plus an additional control record that is used to hold the state of the merge process itself. If we choose not to combine the representation of data and control state into a single block, we will need extra input arguments for the merge operation without changing the algorithm. Each merge block M_i is a fixed-size vector of b records, $M_i.val[b]$, plus three pointer values, $M_i.start$, $M_i.current$ and $M_i.end$. These pointers are initialized to 0, 0 and b respectively.

A simple non-pipelined merge operation operates on three merge blocks A_i, B_j and C_k, the first two representing the current blocks in the two streams being merged, and the last representing the partially generated next block (possibly empty) in the merged result. If $A_i.start \leq A_i.current < A_i.end$, then block A_i has data available to be used in this merge operation. The same check is applied on B_j, and analogous check ensures that space is available in C_k for the result. At the end of any merge call, one of the input blocks A_i or B_j is exhausted, or the output block C_k is full.

Using this simple merge operation, the merge process can be expressed as a sequence of merge calls, each of which exhausts an input block or fills an output block, or both. In response to each call, the endpoint issues the next merge call, replacing an empty block with the next full block in the input stream or replacing a full output block with an empty one. The movement of data from nodes A and B to C is also directed by the endpoint in response to the results of each merge call, with an appropriate level of read-ahead to keep data flowing.

The obvious problem with this simple scheme is that the endpoint must intervene between merge calls in order to issue data transfer and merge operations, i.e. the next merge call cannot be issued until the previous one has finished. Thus, this highly data-dependent sequence of operations has a collateral sequence of control dependencies that are being resolved at the endpoint, imposing latency between two merge operations, which is roughly the network round trip time (RTT) between the endpoint and the merge node.

5.1.2 Instruction Level Scheduling of IBP Operations

In computer architectures, significant effort has been devoted to the research of instruction scheduling techniques and algorithms to exploit more Instruction Level Parallelism (ILP) by overcoming structural, data and control hazards. We will show that control dependency in the distributed merge example described above can be resolved by a combination of instruction scheduling techniques used in processor pipelining: *loop unrolling* and *speculative issuing of instructions*.

The first step is to expand the inputs and outputs of the merge operation, allowing multiple blocks in each input stream and in the output stream to be represented in a single operation. The most basic form of expansion represents two input blocks from each stream, and four blocks in the output stream in order to balance inputs with outputs. The purpose of this expansion is not to allow multiple blocks to be processed by a single operation; each operation still terminates as soon as an input block is exhausted or an output block is filled. Instead, its purpose is to enable enough ambiguity in the location of data accessed by the merge operations to transform control dependence into data dependence.

When a pipelined instruction $merge(A_0, A_1, B_0, B_1, C_0, C_1, C_2, C_3)$ arrives at the depot, $A_0.current$ is compared with $A_0.start$ and $A_0.end$. If $A_0.start \leq A_0.current < A_0.end$, A_0 is used as one of the inputs in this merge operation. If not, pointers of A_1 are similarly checked, and if data is available A_1 is used. Otherwise, the merge operation is interpreted as a no-op. The same checking is performed on the streams from B and C to select the other merge input and the merge output respectively. The pointers of the merge blocks operated on are updated by the merge operation accordingly during the computation.

Because each stream is represented by multiple input blocks, and multiple output blocks are also available, successive operations can proceed without the intervention of the endpoint. With the addition of more input and output blocks, it becomes possible for one merge operation to compute the pointer values to be used in the next merge operation at the same time as its result being returned to the endpoint. Thus, as long as neither input stream exhausts all of its available blocks and the output blocks are not full, the same merge operation can be restarted multiple times without endpoint's intervention. Once an input stream has exhausted all of its available input blocks or all the output blocks are full, the merge process terminates.

Using the merge operation in this form to implement a pipeline of depth d, the endpoint issues d identical merge operations, on the assumption that the data will be evenly distributed to allow several operations to be executed without exhausting all of the input blocks available on either stream. Each time a block is emptied or filled, the result returned to the endpoint specifies which has occurred, and the next operation issued reflects the result of the previous operation. To see how this works, an example execution is listed as follows (the pipelined operations at representative steps are shown in Table 5.1, with active blocks marked with bars and exhausted or filled blocks marked with hats):

Table 5.1: Pipelined NFU operations at representative steps

Step	Operations in the pipeline
2	$merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(\bar{A'_0}, A'_1, \bar{B'_0}, B'_1, \bar{C_0}, C_1, C_2, C_3)$
4	$merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(\bar{A'_0}, A'_1, \bar{B'_0}, B'_1, \hat{C_0}, \bar{C_1}, C_2, C_3)$
7	$merge(A'_0, A'_1, B'_0, B'_1, C_1, C_2, C_3, C_4)$ $merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$ $merge(\hat{A'_0}, \bar{A'_1}, \bar{B'_0}, B'_1, \hat{C_0}, \bar{C_1}, C_2, C_3)$
10	$merge(A'_1, A'_2, B'_0, B'_1, C_1, C_2, C_3, C_4)$ $merge(A'_0, A'_1, B'_0, B'_1, C_1, C_2, C_3, C_4)$ $merge(\hat{A'_0}, \bar{A'_1}, \hat{B'_0}, \bar{B'_1}, \hat{C_0}, \hat{C_1}, \bar{C_2}, C_3)$
11	$merge(A'_1, A'_2, B'_1, B'_2, C_2, C_3, C_4, C_5)$ $merge(A'_1, A'_2, B'_0, B'_1, C_1, C_2, C_3, C_4)$ $merge(A'_0, A'_1, B'_0, B'_1, C_1, C_2, C_3, C_4)$ $merge(\hat{A'_0}, \bar{A'_1}, \hat{B'_0}, \bar{B'_1}, \hat{C_0}, \hat{C_1}, \bar{C_2}, C_3)$

1. Choosing a data read-ahead depth of n, the endpoint issues $2n$ `IBP_copy` operations to move $A_0 - A_{n-1}$, and $B_0 - B_{n-1}$ to temporary buffers at C, which are referred to as $A'_0 - A'_{n-1}$ and $B'_0 - B'_{n-1}$ respectively.

2. Choosing an instruction pipeline depth of 4, the endpoint waits until the contents of $A'_0 - A'_1$ and $B'_0 - B'_1$ are available, and then issues four identical operations: $merge(A'_0, A'_1, B'_0, B'_1, C_0, C_1, C_2, C_3)$.

3. If the output block C_0 becomes full, the first merge operation will terminate and a result indicating this fact will be sent to the endpoint.

4. The second merge operation will proceed without any intervene from the endpoint, generating merged data to C_1.

5. When the endpoint receives the result of the first merge operation, it will issue a fifth merge operation: $merge(A'_0, A'_1, B'_0, B'_1, C_1, C_2, C_3, C_4)$.

6. If the temporary input buffer A'_0 is exhausted next, the second merge operation will terminate and the result indicating this status will be sent to the endpoint.

7. The third merge operation will proceed, drawing data from A'_1.

8. When the endpoint receives the result of the second merge operation, it will wait until the contents of A'_2 are available and then issue an `IBP_copy` to transfer A_n to A'_n as well as issuing a sixth merge operation: $merge(A'_1, A'_2, B'_0, B'_1, C_1, C_2, C_3, C_4)$.

9. If the temporary input buffer B'_0 is exhausted and the output block C_1 becomes full, the third merge operation will terminate and result indicating this fact will be sent back to the endpoint.

10. The fourth merge operation will proceed automatically, drawing data from A'_1 and B'_1, and generating data to C_2.

11. When the endpoint receives the result of the third merge operation, it will wait until the contents of B'_2 are available and then issue a seventh merge operation: $merge(A'_1, A'_2, B'_1, B'_2, C_2, C_3, C_4, C_5)$ and an `IBP_copy` to transfer B_n to B'_n.

A set of experiments were run using depots located at the Starlight co-location facility in Chicago and the Abilene Network Operations Center in Indianapolis as nodes A and B, and a depot located on the Knoxville campus of the University of Tennessee as node C. The depth of data pre-fetching on the input streams, i.e. `IBP_copy` pipelining depth between the client and nodes A and B, varied from 2 to 8, and the pipelining depth of merge operations between the client and node C varied between 1 to 6. Experiments were then run with three different client locations, from

1. Being co-resident with the depot that implements the merge operation on node C.

2. Running on a host in the same local area network as node C.

3. Running on a host in the wide area network at the San Diego Supercomputing Center.

Note that case 1 approximates the degree of coupling between issue and execution that is present in active routers. The record used in all experiments was a single four-byte integer. Experiments were performed with 256KB and 512KB input/output blocks respectively. Thus the sizes of a block in records were 64K and 128K. The experimental results of 512KB blocks are plotted in Appendix Figure 7.7.a to 7.7.d and those of 256KB blocks are plotted in Appendix Figure 7.7.e to 7.7.h. Each experiment was run 9 times, with the result reported being the average. The results are summarized as follows:

- As would be expected, the performance of the merge operation increased with increasing block size and depth of read-ahead. Parallelism in the implementation of read-ahead may have led to the exploitation of more simultaneous TCP streams as the depth of read-ahead increased.

- In all scenarios, the performance of the sequential cases (merge operation pipeline depth 1) was lower when the client was separated from node C by the wide area network than in the other two cases, generally by a factor of 3-4.

- In all cases, a pipeline depth of 6 was sufficient to yield merge performance with the client separated from node C by the wide area network equivalent to the other two cases, and in some cases this was achieved with a pipeline depth of 4.

- One unexpected result was that in some cases, the performance was slightly better when the client was located on another machine in the same local area network as node C (case 2) than when the client was co-resident on node C (case 1). We believe that this is due to paging and other contention for resources between the depot and the client when they were running on the same node.

The results above show that the use of instruction pipelining can effectively overcome latencies caused by the separation of endpoint and intermediate node in the wide area. However, our experience with pipelining has shown two things:

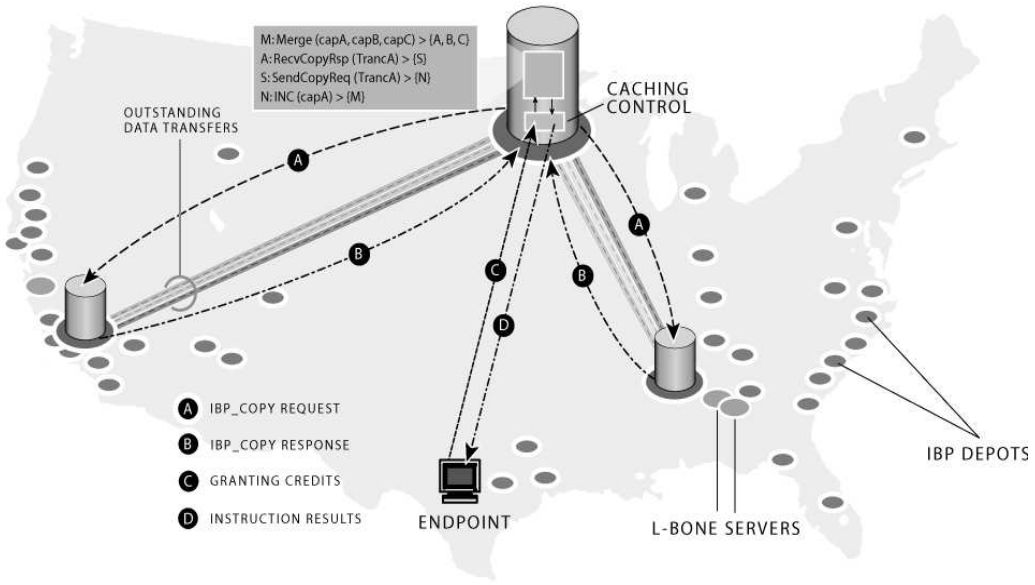


Figure 5.2: An illustration of caching using the distributed merge as an example

- The complexity and specificity of NFU operations required to support pipelining are increased when techniques such as loop unrolling described above are used to hide deep pipelining. This approach breaks down when the dynamic resolution of control dependencies is not predictable.
- When implementing highly repetitive processes such as stream processing, pipelining requires that IBP operations that are identical or that differ only in the identity of buffers being processed be continually resent from the client.

5.1.3 Caching IBP Operations

The idea behind caching is to delegate branching decisions close to the execution unit without going through the wide area network. Following this lead while not violating the LN architectural principles, the semantics of the IBP protocol are extended with a stored instruction model. With the caching mechanism, control over the execution of fine-grained operations can be delegated to the depot explicitly and reverted to the endpoint as needed. The degree of autonomy granted to the depot at any time is under the complete control of the endpoint, and can be modified dynamically according to the nature of the computation, and the changing conditions in the network and the execution environment. Figure 5.2 illustrates how caching works.

In Figure 5.2, a labeled instruction in the cache is defined to be `op` > {`successors`}, consisting of a data movement or computation operation `op`, and a specifier, `successors`, of a set of possible successor instructions within the cache. When allowed to issue instructions autonomously, the depot can issue another instruction from within the set specified by `successors`, the choice being determined by a return value generated from the executing `op`. A new IBP operation `IBP_istore(icap, instr)` is defined which loads an IBP instruction `instr` with a particular label `l` into an instruction cache specified by a capability `icap`.

Introducing greater autonomy into the issue of IBP operations requires that proper consideration be given to application of the end-to-end principle in this context. It would be

a simple matter to allow the client to download a program to the depot cache, and then initiate the execution of a process in the style of an active networking router or other dynamically extensible encapsulated network service. However, such a strategy would generate encapsulated process state at the depot that is inaccessible to the end system. Instead, we choose to adopt an approach which allows the client to grant autonomy to the end system in a limited way that is subject to continual monitoring by the endpoint and requires renewal in order to continue.

With caching, each connection between an endpoint and a depot is considered to have a session state consisting of a number of instruction issue credits. The endpoint can increase or decrease the number of credits by an integer n by issuing a special operation `IBP_credit(n)`. The meaning of the issue credits is that if the endpoint issues a special operation `IBP_start(l)`, then the instruction with the specified label `l` will be executed, and the issue credits will be decreased by one. The successor instruction will then be issued, and autonomous issue will continue until all the issue credits are consumed. As each instruction completes, its result is returned to the endpoint, just as if the instruction had been issued by itself. At that point where all issue credits have been depleted, issue stops. This allows a task consisting of a fixed number of n instructions to be initiated by setting the issue credits to n and then issuing an `IBP_start`.

While the issue credits are nonzero, the client may grant further autonomy to the depot by means of additional `IBP_credit` operations, which can be performed during the execution of such a task, prolonging the execution and deepening its pipeline indefinitely. At the extreme, a special call sets the issue credit value to infinity granting the depot full autonomy to execute processes to completion until an instruction is executed which has no successors. In order to lessen the burden of supervision on the endpoint, further options can be added to allow the endpoint to suppress return values. Such option reducing the direct role of the endpoint in execution at the depot would be exercised at the discretion of the endpoint, with the option always being available to return to a highly supervised or even synchronous mode of operation. Thus, autonomy is granted by the endpoint to the depot in a graduated way, when the endpoint feels that it is necessary and advisable in view of factors such as congested or faulty behavior of the wide area network.

The caching mechanism allows the endpoint to obtain performance without sacrificing control when it is needed. Note that issue credits granted by the client allow but do not obligate the depot to continue issuing instructions from the cache. The depot may, in order to preserve its own resources, choose not to issue instructions at the earliest opportunity, and can even reject operations issued directly by the endpoint, allow the pipeline to drain and close the endpoint's connection. However, accepting a given amount issue credits is considered an agreement by the depot to make a best-effort attempt to complete that number of operations without endpoint's intervention.

The cached version of the distributed merge example is shown in Figure 5.3. With a stored instruction model at the depot, the endpoint is able to delegate the resolution of control dependencies between operations to the depot, and to overcome the problem of re-sending a stream of identical operations when implementing a highly repetitive process. The INC operation is used to increase the index of the next active capability for processing in the capability array. Arrays of IBP capabilities are stored in other IBP allocations and index variables are used to choose the specific capabilities to operate on in each iteration. Each array of capabilities has two integer indices associated with it, one to indicate the length of the array, and the other representing the next active capability for processing. Figure 5.4 shows an example state of the capability array.

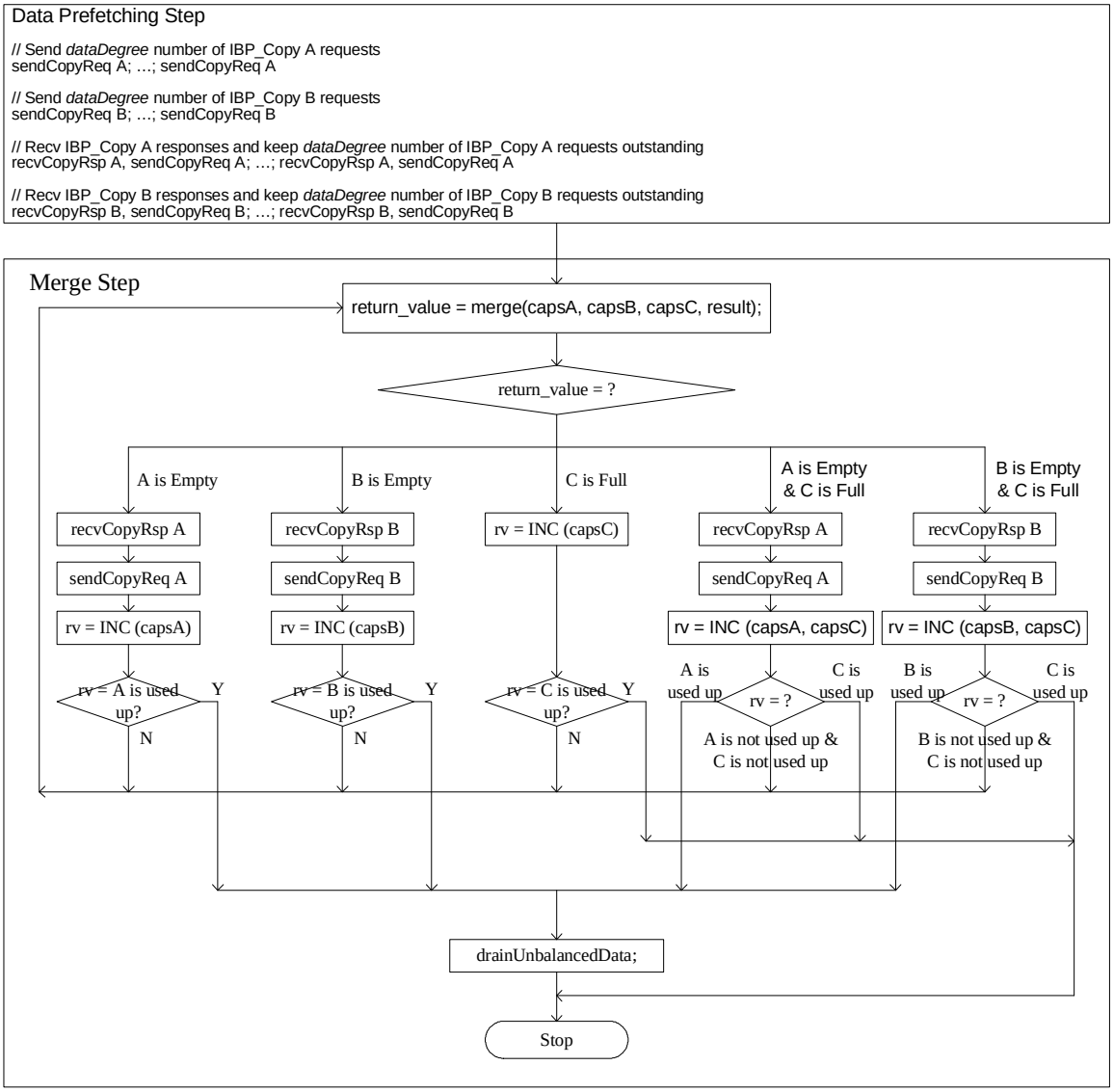


Figure 5.3: Flowchart of the cached distributed merge example

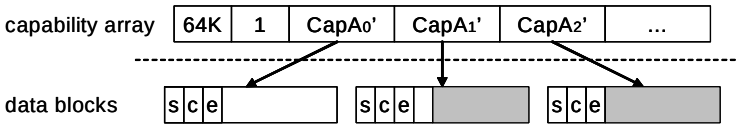


Figure 5.4: An example state of the capability array for data blocks from node A.

In addition to processing arrays of capabilities representing stored data, the need arises to handle multiple capabilities in a uniform manner when data transfers implemented as `IBP_copy` operations are pipelined. In this case, it is necessary to be able to initiate asynchronous send operations and to later block on receiving their responses. The additional state required to implement this synchronization is implemented as a transaction control block that has one entry for each outstanding data transfer. The control block is stored in an allocation that is read and written by the `sendCopy` and `recvCopy` operation.

Core of the merge is comprised of a small set of instructions which executes in a highly repetitive fashion, using a single NFU operation and attendant data movement calls. In an environment where data transfer is reliable, it is easy to imagine the execution of this process being very efficient, given a sufficient budget of credits. However, when execution faults are possible and data transmission errors were to occur, an error might be reported back to the client, requiring corrective action that is not part of the core merge loop. Both cases require the endpoint to be able to take control of the merge process.

A sequence of experiments was run to simulate the effect of pipelined execution of the distributed merge using the instruction cache and issue credits. After issuing some initial data movement operations to prime the data pipeline, the endpoint would issue a number of credits allowing the depot to issue that number of instructions from the cache. Then, as the program executes, credits are issued to allow the continued execution, maintaining approximately the same number of outstanding credits.

Appendix Figure 7.8 shows the performance of the cached version of the merge example running with 512KB and 256KB block size respectively. Each data point represents an averaging of the results of 9 runs. The line marked with "infinite credits" represents the performance when more than the number of credits required for the entire merge process to complete is issued initially. For example, in the case of block size 512KB with data prefetching depth 8, 1920 credits are issued initially. Because several instructions must be issued corresponding to each pipelined operation, due to the use of asynchronous data movement, credits and pipeline depth are proportional but not equal. Careful readers may have noticed that almost in all cases, performance of the pipelined merge with depth 6 is slightly better than the cached merge with infinite credits. This is because all execution status of cached instructions (sending `IBP_copy` request, receiving `IBP_copy` response, increasing capability index, and merging) are returned to the remote client. When responses except that of the merge instruction were suppressed, throughput of cached merge with infinite credits outperformed that of pipelined merge as expected.

The endpoint grants autonomy to the depot in a limited way that is subject to continual monitoring by the endpoint and requires renewal in order to continue. Thus, the caching mechanism is a special form of pipelining in that the endpoint continuously grants credits instead of issuing specific operations to control program execution at intermediate nodes. The experience with caching IBP operations has shown two things:

1. The caching model does not have a notion of process states. Thus, it is awkward to deal with temporary program variables in a graph of cached operations. For instance, an array index has to be stored in an IBP allocation, use another operation to change its value, and be passed as an argument to operations that use the array.

2. The depot needs to maintain "session" states (e.g. the number of available credits) between the execution of operations in the instruction cache, complicating the functionality at intermediate nodes. It appears to contradict the end-to-end arguments that a function should be placed at intermediate only if it is needed by all endpoints.

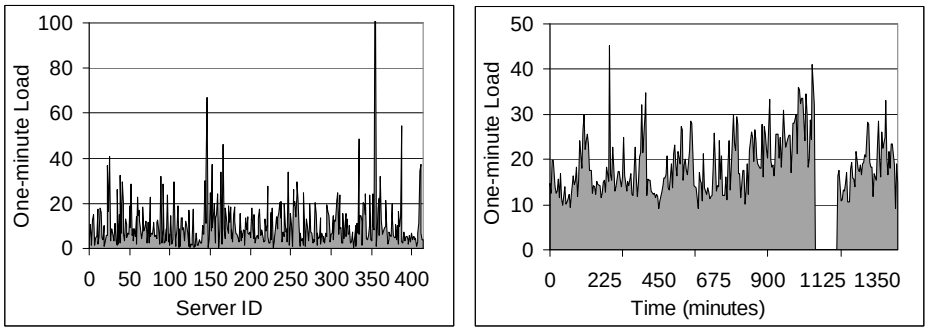


Figure 5.5: One-minute load of PlanetLab nodes.

For these reasons, it is desirable to employ instruction scheduling techniques at endpoints to overcome control dependency and implement complex program logics, while keeping the depot as simple as possible.

5.2 Co-scheduling of Network Computation and Replication

As described in previous chapters, computation tasks and the datasets they work on are partitioned in order to run on the LN infrastructure. To improve throughput and reliability in the wide area, the partitions are often replicated among k different depots. The value of k is typically small due to limited storage capacity, transport latency and maintenance overhead. Thus, the same computation task can be performed at k depots and many computation tasks can happen at the same time. The programming paradigm of these parallel implementations is usually master-worker, also known as task-farming [Buyya, 1999]. In the master-worker paradigm, the master (endpoint) is responsible for distributing tasks among a farm of workers (depots) and collecting partial results. Each worker simply waits for a task from the master, computes the partial result and sends it back. Considering that most computations work on local data, tasks are only assigned to workers that have the corresponding partitions. The master has to explicitly initiate any data movement between workers before task assignment.

The LN infrastructure is a best-effort, un-orchestrated environment where computation and storage resources are not reserved beforehand, but, rather, co-scheduled at runtime. On such a shared distributed infrastructure, traditional parallel algorithms would need to be adapted with some special methods of scheduling. To get the best performance out of non-dedicated depots, dynamic management for computation and data replication has to be tightly coupled. Computation management involves the assignment of parallel tasks, while data replication management deals with data movement between selected depots. Both scheduling of computation and scheduling of replication aim at maximizing depot utilization and minimizing application execution time. While scheduling of computation improves server utilization by distributing tasks intelligently to optimize load balancing among depots, scheduling of replication moves partitions around so that work assigned to each depot is proportional to its performance.

There are several technical hurdles to achieving the above goal. For instance, in a distributed system composed of heterogeneous servers shared by a community without explicit orchestration, it would be hard to have all the involved servers produce a guaranteed

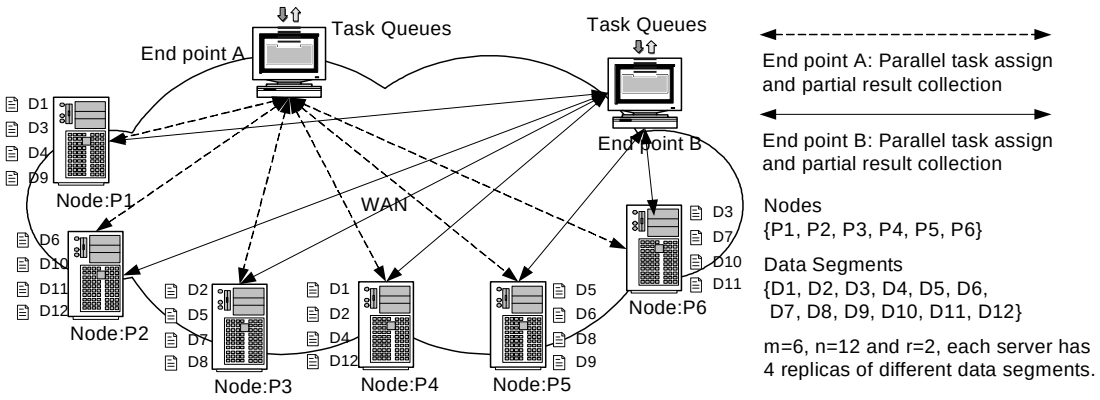


Figure 5.6: A typical structure of task parallel applications on replicated datasets

level of performance. Figure 5.5 shows a snapshot of the one-minute load of 415 PlanetLab nodes starting from 15:50 on Nov.16, 2005 on the left and the one-minute load of `pl1.cs.duke.edu` in 24 hours on the same day on the right. Load of the duke node is sampled every five minutes. The node was unavailable during 18:10 to 19:55, which happens frequently in a large distributed system. Although PlanetLab nodes are server-class machines, they are shared among a large community. They are even virtualized as "slices" to enable large-scale sharing. Loads on these nodes differ dramatically and vary over time.

Many well-known middleware systems have been developed over the past few years to implement task-farming applications. However, few have examined co-scheduling of computation and replication for operating on replicated data in the wide area. In this section, we describe an integrated scheduling algorithm that considers both computation and storage aspects simultaneously in a distributed system. It adaptively measures server performance in terms of computation power and data transfer rate. This information is used to dynamically assign tasks to intermediate nodes and direct data movements among them to achieve the best server utilization, minimizing application execution time. In addition, our co-scheduling algorithm is novel in runtime data movement schemes that use the deadline based partial download from multiple sources. User provided knowledge of the application such as computation complexity also contributes to an effective scheduling.

5.2.1 Problem Definition

In a distributed environment where shared resources cannot be brought under the control of a single global scheduler, the application must be scheduled by the endpoint or by some middleware agent. For the latter case, the middleware agent itself can be viewed as an endpoint. Figure 5.6 shows a typical structure of task parallel applications on datasets that are partitioned and replicated at distributed nodes. We assume that every node is capable of handling both computation and data movement requests. Each endpoint accesses and analyzes datasets independently without knowing activities of other endpoints.

Before the discussion of various job scheduling algorithms, we define the scheduling problem on wide area replicated datasets. Suppose we have:

A collection of computational nodes P_1,P_2,...,P_m where m is the number of nodes. P_i is described by b_i and c_i. Bandwidth b_i represents the bandwidth between P_i and the endpoint. Computational power c_i defines how fast a partition can be processed for an application. For convenience, both b_i and c_i are measured in megabytes per second. If a

node has multiple processors, c_i is the aggregate computational power of all processors that can contribute to the computation. When several endpoints contend for resources, b_i and c_i are a fraction of physical resources that is delivered to the user.

A large dataset that is partitioned into d_1,d_2,..., d_n. n is the number of partitions and s_j is the size of d_j. We define $\delta_i^j = 1$ if d_j is on P_i, $\delta_i^j = 0$ otherwise. Data is distributed with k-way replication ($k \geq 1$), i.e. each partition is replicated on k out of m randomly selected servers. Formally, $\sum_{i=1}^{m} \delta_i^j = k$ for each partition d_j. On average, each node has $n \times k/m$ partitions.

An application (e.g. parallel rendering) that is able to make use of the entire collection of partitions in parallel. Thus, we have a set of independent computational tasks T_1,T_2,...,T_n. We assume that d_j is the only partition required by task T_j. We further assume that execution time of T_j is proportional to $f(s_j)$ and the output size of T_j is $g(s_j)$. $f(x)$ is known as the complexity function and $g(x)$ is often constant or linear, for example, in linear algebra operations. $f(x)$ and $g(x)$ are application specific and usually required for an effective application level scheduling. If $\delta_i^j = 1$ and T_j is assigned to P_i, the time required to complete T_j can be formulated as $f(s_j)/c_i + g(s_j)/b_i$. $f(s_j)/c_i$ is the time required for computation and $g(s_j)/b_i$ is the time spent on communication. Since the required partition already resides on the target server when a task is assigned, we assume that communication time is solely the time to receive the output. Although many effective techniques such as pipelining can be employed to overlap computation and communication between successive tasks, we assume they are not overlapped in our model.

A set of data movement tasks M_{ij} that makes a fresh copy of d_j on P_i. To exploit the fact that there are multiple replicas of d_j, data is downloaded from multiple sources distributed in the wide area network. Thus, the time required to perform M_{ij} is approximately $s_j / \sum_{r=1}^{m} (b_{ir} \times \delta_r^j)$, where b_{ir} represents the bandwidth between P_i and P_r. $\sum_{r=1}^{m} (b_{ir} \times \delta_r^j)$ is the aggregate bandwidth to P_i from all sources that have d_j.

To mitigate resource contention on shared servers with heavy load, we assign at most one computational task to a node at a time. Data movement tasks can co-exist with a computational task because a good mix of CPU-bound and I/O-bound processes can actually improve system throughput. However, due to process scheduling, too many concurrent data movement tasks can slow down computational tasks, especially in a non-dedicated system. Thus, the number of active data movement tasks at each node is also set to be one. The number of simultaneous downloads could be k because we have k replicas.

Suppose each node P_i runs for time t_i and all nodes start at the same time, then the execution time of an application would be $\max_{i=1}^{m} t_i$, which is the time required for the last node to finish its assigned tasks. For a given application, the shortest execution time occurs when all nodes can be kept doing useful work and they all finish roughly at the same time.

Given that the dataset is replicated throughout a wide area network, does there exist a scheduling of computation and data movement tasks such that the execution time of an application over the entire partitions is minimal?

This is the scheduling problem of replicated datasets we will explore in this section. Formally, let $\sigma_i^j = 1$ denote that task T_j is assigned to server P_i. A schedule is a set of $\sigma_i^j \in \{0, 1\}, i \in [1...m]$ and $j \in [1...n]$, such that $\forall j, \sum_{i=1}^{m} \sigma_i^j \geq 1$. $\forall j, \sum_{i=1}^{m} \sigma_i^j \geq 1$ mandates that each task T_j must be assigned to at least one server. If $\delta_i^j = 1$ and $\sigma_i^j = 1$, T_j can be immediately assigned to P_i as long as there is no other active task on P_i. However,

Algorithm 1: Basic work-queue scheduling over replicated datasets

```
1 while not IsEmpty(Q) do
2     foreach available node P_i do
3         DeQueue (T_f, Q), T_f is the task that has been finished by P_i;
4         T_j = GrabTask (P_i, Q), d_j is on P_i;
5         AssignTask (T_j, P_i);
```

if $\delta_i^j = 0$ and $\sigma_i^j = 1$, a copy of d_j must be moved to P_i before T_j can be assigned to P_i. Note that, $\delta_i^j = 1$ does not necessarily imply $\sigma_i^j = 1$ because there are r distributed replicas to choose from and a fresh replica can be made at runtime when necessary. The best schedule satisfies that $\max_{i=1}^{m} \sum_{j=1}^{n} \sigma_i^j \times (f(s_j)/c_i + g(s_j)/b_i)$ is minimal assuming that no fault happens after the schedule is made. For the intermediate node that finishes last, time spent on explicit data movement completely overlaps with computation, thus it is not included in the formula.

5.2.2 Co-scheduling of Computation and Replication

Shared datasets are typically replicated and accessed by geographically distributed users with competing goals. As a result, resource performance varies over time and is hard to predict. Experience with distributed applications indicates that adaptability is fundamental to achieving application performance in dynamic environments [Berman et al., 2003]. It is imperative to employ heuristics and dynamic load balancing to obtain a good approximation solution to the optimization version of the scheduling problem, while addressing fault-tolerance at the same time. The conventional work-queue scheduling of parallel tasks is presented as a basis. After that, the co-scheduling algorithm is discussed in two steps: adaptive scheduling of computation and dynamic scheduling of replication.

Work-queue scheduling

Work-queue scheduling [Hagerup, 1997] is a variation of the master-worker model. In contrast to static scheduling techniques in which tasks are allocated before the application is started, work-queue scheduling attempts to deal with variability in resource performance and individual task workload by deferring task assignment. In work-queue scheduling, tasks are not distributed to workers until they have finished a previously assigned task. In this way, fast workers tend to deliver more tasks than slow workers over time.

Algorithm 1 illustrates a scheduling of parallel tasks over replicated datasets using work-queue scheduling. To avoid data movement, tasks are only assigned to compute nodes that have the required partitions, i.e. task T_j is assigned to P_i only if $\delta_i^j = 1$.

In `GrabTask`, an unassigned task T_j is picked sequentially from the work queue. When there is no more unassigned task that a fast node can do, it will try to help slow nodes on already assigned tasks if it holds the required data blocks for these unfinished tasks. The algorithm is straightforward and theoretical work has proved that work-queue scheduling yields a good approximate solution to the optimization version of the scheduling problems [Hochbaum, 1997].

Even though it is very adaptive, the above algorithm ignores the fact that distributed servers have very diverse performance, which has two potential consequences. First, each

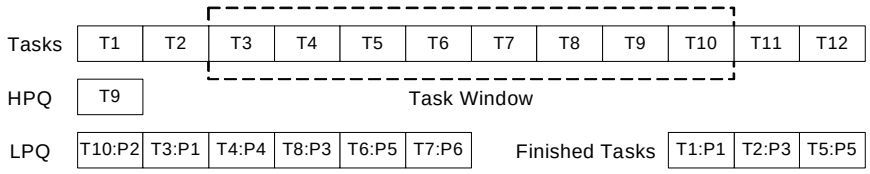


Figure 5.7: A snapshot of the scheduling state

task is performed by one compute node unless that node fails, the scheduler times out or other compute nodes that have the corresponding data partition have no more tasks to work on. If one compute node lags, the overall application cannot progress if the application (e.g. streaming, interactive visualization) needs ordered partial results. Second, each available compute node always sequentially picks an unfinished task that it can do, which in some cases might be performed by a faster compute node. In this case, slow compute nodes "steal" work from fast nodes. When all candidate tasks on fast nodes get depleted, they have to stop while slow nodes still need to finish the tasks for which they hold the required data partitions exclusively. Thus, there is a need for more sophisticated scheduling techniques that can perform adaptive resource selection and on-demand data movement.

Adaptive scheduling of computation

We employ a heuristic-based method that makes use of historical performance about participating compute nodes in the network. The approach depends on discovering fast compute nodes on the fly, assigning as many tasks to them as possible and avoiding being stalled by slow or faulty nodes. Three generic mechanisms were devised for this purpose: (i) a dynamically ranked pool of compute nodes, (ii) a two level priority queue of tasks and (iii) a competition avoidant task assignment scheme. This framework is very generic and can be applied to other distributed computing applications in general.

Each compute node P_i is ranked by its estimated time t_u^i to process a task of unit size u (e.g. 10MBytes). This measurement roughly reflects performance of the node delivered to an application. The less time a node needs to process the unit task, the higher rank this node has. Recall that $t_u^i = f(u)/c_i + g(u)/b_i$. Rather than a simple average, c_i is calculated from $c_i' + \rho \times (\tau - c_i')$, where c_i' is the previous value of c_i and τ is the most recent value. Similarly, $b_i = b_i' + \rho \times (\beta - b_i')$, where b_i' is the previous value of b_i and β is the most recent value of b_i. The parameter $0 \leq \rho \leq 1$ determines the influence of previous values, with the influence of outdated values tending towards zero over time. This causes the client scheduler to continuously adapt to the constantly changing resource performance.

When a compute node finishes a task, it returns the computation time t_c and the output. The client scheduler records the time t_s when it starts to receive the output and the time t_r when it finishes. With t_c, t_s and t_r, τ and β are formulated as $f(s_j)/t_c$ and $g(s_j)/(t_r - t_s)$ respectively. Note that both t_r and t_s are obtained from the local time service at the scheduler. Although a more accurate β can be obtained by using the time when the compute node starts to send back the output, it requires time on both the client scheduler and compute nodes to be closely synchronized, which is not very practical in a large distributed system.

A two-level priority queue maintains unfinished tasks. The higher priority queue (HPQ) contains tasks that are ready to be assigned and the lower priority queue (LPQ) contains tasks that have been assigned to one or more servers but not finished. If a task is assigned

to an idle compute node, it is moved from HPQ to LPQ. Initially, only the first w tasks $T_1, T_2, ..., T_w$ are placed in HPQ and task T_x ($x > w$) can not be added until task T_{x-w} has been completed, where w is the size of the task window (TW). w controls how far out of order tasks can be finished. For example, if $w = 1$, all tasks will be completed in order. In contrast, if $w = n$, every task is allowed to be finished out of order. In most cases, w is greater than m so that every compute node can contribute to the computation. Figure 5.7 shows a snapshot of tasks in the two-level priority queue on the dataset as illustrated in Figure 5.6. The task window cannot move forward at this moment because node P_1 is still working on task T_3, which is at the head of TW.

Each task T_j in HPQ is keyed by $\min_{i=1}^{m} t_u^i \times \delta_i^j$, which is the minimum unit task process time of all compute nodes currently having partition d_j. This priority ranks new tasks by their likelihood to be finished by a fast node in terms of computational power and available bandwidth. Assume a task T_j in LPQ has been assigned to P_i. T_j is keyed by its estimated waiting time, which is the estimated execution time $E_j = f(s_j)/c_i + g(s_j)/b_i$ minus the time that has elapsed since start. E_j is static during task execution because b_i and c_i will not be updated until the task is completed. This priority ranks assigned tasks by its likelihood to finish soon. The client scheduler can dynamically sleep the minimum estimated waiting time to avoid busy waiting. Note that Tasks in both HPQ and LPQ are sorted by their keys in decreasing order.

When the parallel computation starts, the client scheduler sequentially assigns each available compute node the first task in HPQ that it is able to perform, moving the task to LPQ. When P_i completes task T_j, b_i and c_i are updated, and T_j is removed from LPQ. If T_j is the first task in the task window, TW is moved forward, adding one or more tasks to HPQ. Since b_i and c_i are adjusted, HPQ is resorted by the latest t_u^i as well. In case of a failure, the task in LPQ is promoted back to HPQ so that other nodes can take it over. Then, there are three possible scenarios:

1. Both HPQ and LPQ are empty. This case signifies the completion of scheduling.

2. There are unassigned tasks in HPQ. In this case, every available compute node will be directly assigned the first task in HPQ that it can handle. This is the slowest task among all unassigned tasks that the compute node can work on. In this way, slow nodes do not compete for tasks with fast nodes so that fast nodes can be assigned as many tasks as possible.

3. There are unfinished tasks in LPQ. In this case, we would like unfinished tasks to be computed by additional compute nodes (up to $k - 1$, k is the number of replicas for each partition), which work in parallel with the node that was originally assigned for the task. These nodes compete to finish the same task. Again, the first task in LPQ is assigned to an available compute node that holds the required data partition. This is the slowest task among all unfinished tasks that the compute node can help. If any of the duplicated tasks is completed, others are aborted immediately.

Dynamic scheduling of replication

So far the adaptive scheduling algorithm makes use of performance history to allocate tasks so that slow compute nodes do not compete with fast ones for tasks. Fast nodes can further help slow nodes by repeating tasks on replicas. However, data placement in the scheduling is still static, i.e. there is no active data movement in the process of computing. There is

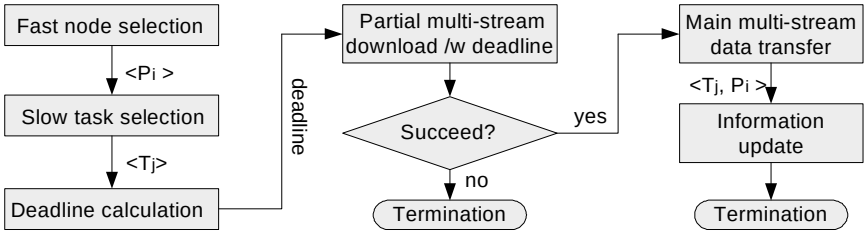


Figure 5.8: Dynamic scheduling of replication

the possibility that some partitions only reside on a set of slow compute nodes. In that case, fast compute nodes cannot help slow compute nodes because they do not have the required partitions to proceed.

One natural thought is to move partitions to fast nodes before they become idle. In order to make sure that time spent on data movement does not exceed the profit that is gained from migrating the task, bandwidth information between computes needs to be acquired. However, this needs non-trivial setup and management of bandwidth estimation or prediction tools [Wolski et al., 1999,Dovrolis et al., 2004]. Also, the information obtained is not always up to date. Instead of using existing tools to insert extra test traffic into the network and query for available bandwidth, a partial download scheme with deadline is used for data movement between compute nodes.

Ideally, the shortest execution time of an application occurs when all compute nodes finish roughly at the same time. As parallel computation proceeds, the total amount of work unassigned is $U = \sum(f(s_j)+g(s_j))$ for all T_j in HPQ. The scheduler actively monitors tasks in HPQ that each compute node can perform. The maximum amount of unassigned work a node P_i can contribute is $W_i = \sum(f(s_j)+g(s_j))$ for all T_j in HPQ and $\delta_i^j = 1$. P_i's share of the unassigned work is calculated as $U_i = U \times (1/t_u^i)/\sum_{j=1}^{m}(1/t_u^j)$, where $(1/t_u^i)/\sum_{j=1}^{m}(1/t_u^j)$ is P_i's proportion of the unassigned work, according to its observed performance. Since the speeds of data processing and data transmission for each compute node are different, both W_i and U_i are rough estimations.

Once $W_i < U_i$, i.e., the total number of work P_i can do in HPQ is less than its proportion of all unassigned work according to its performance, the scheduler tries to initiate a data movement task M_{ij}, moving data blocks from all nodes that have d_j to P_i. Since partitions are replicated and W_i increases with k, there is the possibility that no data movement is necessary at all ($W_i \geq U_i$). The scheduler starts from the first task in HPQ, which has the least likelihood to be finished by a fast node. To avoid always moving partitions out of the same set of slow nodes, the task T_j should satisfy that sum of W_i of all nodes that have d_j is above their aggregate share (the sum of all corresponding P_i). If this condition cannot be satisfied, the scheduler will skip it and try the next task in HPQ, i.e. a task is moved only when all nodes that can perform this task have sufficient work to remain busy.

Before sending and receiving bits over the network, the maximum data transfer time allowed is calculated as the deadline. For example, suppose T_8 has been picked to be migrated to P_1. Also assume that T_8 can also be performed by P_2 and P_3. d_8 is moved only if $\min(F_2, F_3) > (T_m + T_c)$, where T_m is the time for data movement, T_c is the time to compute T_8 on P_1, F_2 and F_3 are the time required to complete all remaining tasks, including T_8, on P_2 and P_3 respectively. The deadline of M_{18} is set to be $\min(F_2, F_3) - T_c$. After the deadline is calculated, the data movement task M_{ij} starts. The scheduler does not try to transfer the complete partition from the beginning. Instead, it tries a small

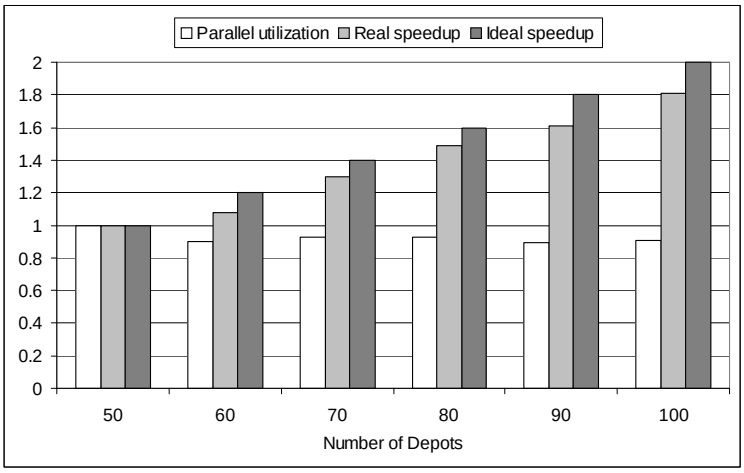


Figure 5.9: Parallel speedup and parallel utilization measured up to 100 depots using 50 randomly selected depots as the benchmark

fraction p of the partition and see if it can be finished in p of the deadline. p is configured at runtime so that dynamics such as TCP slow start can be avoided. If the fractional transfer completes in p of the deadline, the scheduler proceeds to move the rest of the data partition; Otherwise, M_{ij} is aborted. Since the partition is replicated on k nodes, the destination node takes advantage of downloading data from multiple sources by using the progressive driven redundancy algorithm [Plank et al., 2003]. When M_{ij} is done, key of T_j in HPQ is updated because a fast node can now work on it. Figure 5.8 illustrates the whole process.

5.2.3 Performance Evaluation

The design goal of the co-scheduling of computation and replication algorithm is to get the best performance out of shared intermediate nodes in the wide area network. The algorithm needs to perform gracefully (i.e. preserving or improving performance) as the number of nodes in the system increases. To this end, experiments were run on a 30 time-step subset (75GB) of the TSI dataset using 10 depots from the National Logistical Networking Testbed (NLNT) and 90 depots from the PlanetLab project. The TSI dataset is of $864 \times 864 \times 864$ spatial resolution. Each time step is partitioned into 64 blocks and uploaded with 3-replication.

The wall clock time for test runs of volume visualization on the TSI dataset using a collection of 50 to 100 depots was recorded. For each configuration, 10 tests were run to obtain the average. On 100 depots, it takes about 470 seconds to complete software raycasting of 30 time steps TSI data with 800×800 image resolution and 0.5 step size. Knowing that it is not a rigorous comparison, but only to provide context, the same volume rendering takes 218 minutes on a dedicated 2.2GHz P4 CPU with 512KB cache. The performance achieved with 100 shared, distributed heterogeneous processors roughly equals that of a dedicated 32-node cluster, assuming 90% parallel utilization on the cluster.

In the literature of parallel visualization, researchers commonly use parallel speedup and parallel utilization to evaluate the efficiency of a parallel algorithm. Albeit they do not directly apply to heterogeneous systems, the timing results with 50 depots was chosen as a reference, R. In Figure 5.9, we plot the ideal parallel speedup as calculated as $m/50$, the

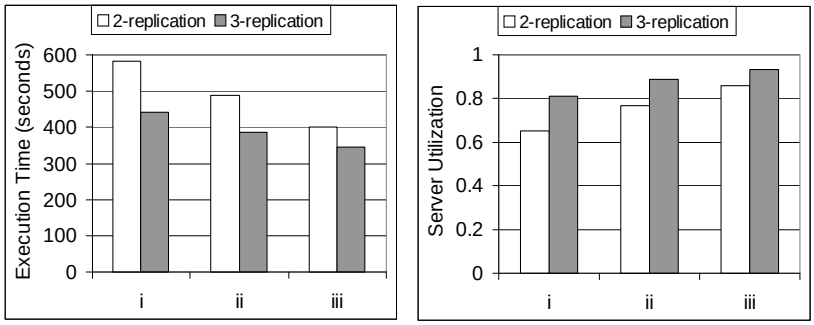


Figure 5.10: Volume rendering with $w = 800$.

actual parallel speedup as T/R and parallel utilization as $(T/R)/(m/50)$, where T is the running time using m depots. The results show that the co-scheduling algorithm scales well with an increasing number of depots.

To investigate the actual performance of the co-scheduling algorithm, its wall clock execution time and server utilization were compares with those of the basic work-queue scheduling and the adaptive scheduling of computation but without replication. Server utilization measures the efficiency of n servers allocated for an application. It is defined as the ratio of the time that n servers spent on doing useful work to the time those servers would be able to do useful work [Heymann et al., 2000b].

80 depots were randomly selected from the above 100 nodes to run isosurface extraction and volume rendering on a time-varying dataset simulating a Jet shockwave with 100 time steps. The spatial resolution of each time step in the Jet dataset is $256 \times 256 \times 256$. Every time step is partitioned into 8 partitions with spatial resolution $128 \times 128 \times 128$ of 8.4MB in storage. There are in total 800 partitions, covering 100 time steps. Total size of the entire dataset is 6.7GB. These partitions are uploaded and augmented with k copies evenly on all depots. For example, using $k = 2$, per-depot storage is roughly $800 \times 2 \times 8.4/80 = 168$ MBytes. Note that volume rendering does a high quality image reconstruction, which consumes more CPU cycles than isosurface extraction for the Jet dataset.

In Figure 5.10 and Figure 5.11, execution time and server utilization for volume rendering and isosurface extraction were compared with $k = 2$ and $k = 3$ respectively. In both figures, (i) is the basic work-queue scheduling, (ii) is the adaptive scheduling of computation and (iii) is the co-scheduling of computation and replication. To maximize the differences, the maximum window size $w = 800$ is used. With each particular combination, 8 tests were run and only the average is reported. Since conditions might change between one execution and the next due to resource contention, we run instances of the three scheduling algorithms back-to-back, hoping that all three executions would enjoy similar conditions on average.

In general, increasing the number of replicas, k, increases storage overhead on each depot and consumes more network bandwidth when copying partitions between depots during the data staging phase. Both isosurface extraction and volume rendering have shorter execution time and higher server utilization with 3-replication than with 2-replication for all the three scheduling algorithms. With a larger k, both fast and slow depots have more candidate partitions to work on, thus fast depots have more chances to help slow depots. For the heavyweight volume rendering with $k = 2$ and $w = 800$, on average, the co-scheduling algorithm reduces execution time by 31% and increases server utilization by 32% at the cost of moving 56 partitions from the slow depots to fast depots, compared with the basic

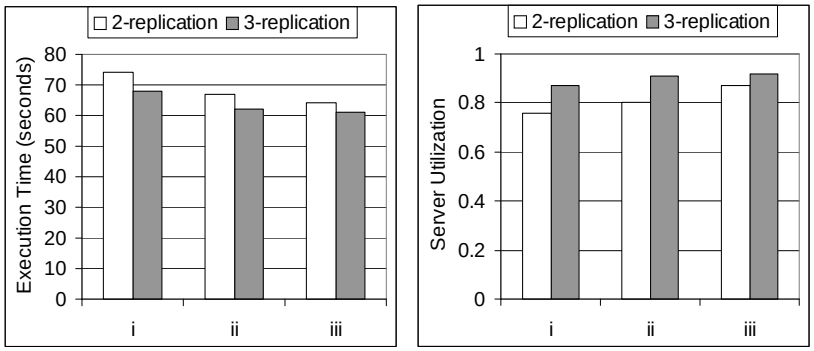


Figure 5.11: Isosurface extraction with w=800.

work-queue scheduling. For the lightweight isosurface extraction, in most cases, the cost of moving a partition out of k slow depots exceeds the profit gained from transferring the task to a fast depot. We only see a slight improvement of execution time and server utilization for the co-scheduling algorithm over the adaptive scheduling of computation with $k = 2$ and $w = 800$ because of the overhead of vainly trying the deadline based data movement.

Size of the task window w also has a similar effect to execution time as k does. The execution time of volume rendering with different w for $k = 2$ is shown in Figure 5.12. By increasing w, the amount of duplicated tasks is reduced and the work completed by each depot gets more proportional to its performance. For instance, suppose we have 4 tasks of unit size. They are replicated with $k = 2$ on server P_1 and P_2. P_1 can finish a task in 30 seconds and P_2 can finish a task in 10 seconds. Initially, task T_1 was assigned to P_1 and task T_2 was assigned to P_2. If w is set to 2, then P_2 has to help P_1 after it finishes T_2. Thus, the number of duplicated tasks is 2 and total execution time is 40 seconds. In contrast, with $w = 4$, P_2 can proceed to work on T_3 and T_4 without helping P_1. The number of duplicated tasks would be 0 and total execution time is 30 seconds. Although larger w offers better performance, it needs to be decreased in case of severe resource contention as a processor "back-off" strategy to enable resource sharing. Also, w needs be limited if the application requires ordered partial results.

To better illustrate the dynamics of load balancing between the three scheduling algorithms, the number of active depots during a typical execution using $k = 2$ and $w = 800$ is shown in Figure 5.13. Server utilization is calculated as the area covered by the curve divided by the area of the bounding rectangle. Initially, every depot works on one of its $800 \times 2/80 = 20$ partitions. As tasks on a particular depot are completed, the choice of the next task for this depot becomes constrained. For volume rendering, in the basic work-queue scheduling, when tasks on the faster depots are eventually depleted, the slower depots still need to finish the tasks for which they hold the corresponding partitions. This explains why the basic work-queue scheduling has the least server utilization. Adaptive scheduling of computations improves server utilization by optimizing the task assignment process so that the fast depots can be assigned as many tasks as possible. With co-scheduling of computation and replication, fast depots are kept busy by moving extra tasks to them from slow depots.

Careful readers may have noticed that for isosurface extraction, server utilizations with the three scheduling algorithms are better than their counterparts in volume rendering and they do not have too much difference. This has to do with process scheduling on depots that

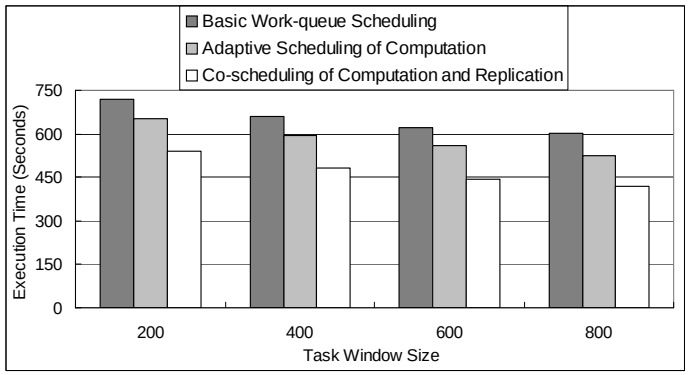


Figure 5.12: Volume rendering with different w

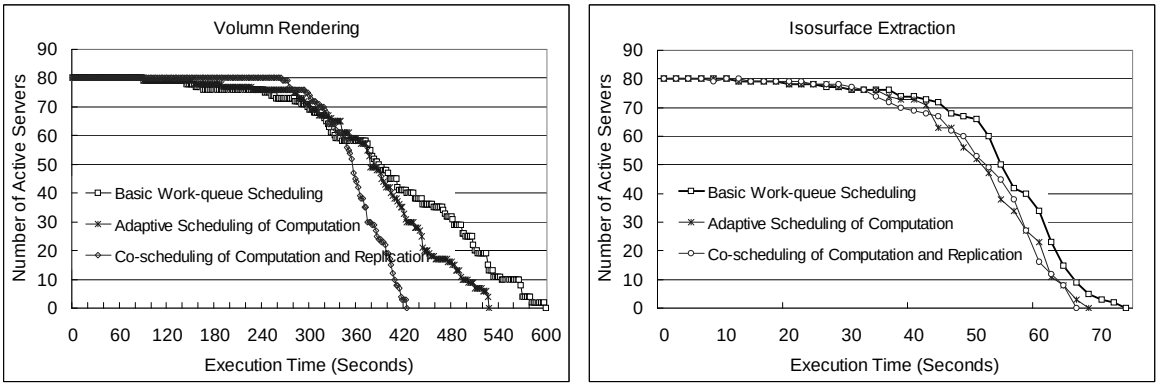


Figure 5.13: The number of active depots over the time span of a typical run.

have a high average load, i.e. a large number of active processes are waiting in the ready queue for execution. Most operating systems schedule process execution by priority. Linux (installed on all PlanetLab servers and more than 50% of NLNT servers) process scheduler keeps track of process execution and adjusts their priorities dynamically. Processes are assigned the highest priority initially. They are penalized by decreasing their priority for running a certain amount of time. Correspondingly, their priority is increased if they have been denied the use of CPU for a certain amount of time. Remember that the process doing isosurface extraction needs less CPU cycles. Thus, it is more likely to have a higher average priority than process doing volume rendering. As a result, depots that have high load tend to look faster when running lightweight computations than running heavyweight computations. When all depots perform similarly fast, the system tends to have higher server utilization.

Having studied the relative performance between the three scheduling algorithms, we are further interested in knowing how close is the execution time obtained from the co-scheduling algorithm to the optimal execution time. In order to calculate the optimal execution time, we need to find out the optimal schedule first. However, it is very difficult to figure out the optimal assignment of tasks, even if we know the performance of all depots. Since each task must be assigned to one of the k depots that have the required replica, there are k^{800} possible schedules in total. When data movement is considered, the scheduling problem is much more complex.

Fortunately, tasks in the test roughly have the same size, thus introducing similar workload. To obtain an estimation of the optimal execution time, the time taken for each depot to complete a task is logged and then the average task processing time $\bar{t_i}$ for depot P_i is computed when all tasks are finished. Ideally, the optimal execution time occurs when all depots stop at the same time. The "super optimal" execution time is calculate as $800 \times (1/\bar{t_x})/(\sum_{i=1}^{80} 1/\bar{t_i}) \times \bar{t_x}$ where $800 \times (1/\bar{t_x})/(\sum_{i=1}^{80} 1/\bar{t_i})$ is the number of tasks assigned to depot P_x according to its performance. It does not matter which depot's average task completion time is chosen for the calculation because all depots finish at the same time. We call it "super optimal" because $800 \times (1/\bar{t_x})/(\sum_{i=1}^{80} 1/\bar{t_i})$ is usually a fractional number, which is not true in real task assignment. Thus, the "close optimal" execution time is also calculated by rounding the number of tasks that each depot is assigned. The execution time is formulated as $\max_{x=1}^{80} \lceil (800 \times (1/\bar{t_x})/(\sum_{i=1}^{80} 1/\bar{t_i})) \rceil \times \bar{t_x}$. Execution time of the optimal scheduling should be somewhere between the "super optimal" and "close optimal". Using the co-scheduling algorithm, the average execution time of volume rendering with $k = 2$ and $w = 800$ is 1.07 times of the "close optimal" value and 1.16 times of the "super optimal" value. That would be considered very close to the optimal execution time.

5.3 End-to-end Workload Control

By using the co-scheduling algorithm described in Section 5.2, a single endpoint can achieve load balance and make good utilization of available resources on distributed and heterogeneous intermediate nodes. In Section 5.1, instruction pipelining is introduced to address high instruction issue latency in the wide area for even better resource utilization. In that technique, multiple instructions populate different stages, or modules, of the hardware of intermediate nodes. There are two issues when combining these two techniques together:

1. Proximities between the endpoint and compute nodes are different. Not all tasks have the same amount of computation workload, nor do they incur the same amount of

network communication. In addition, compute nodes may differ in their capability to handle multiple requests. Resources available on a shared intermediate node vary over time in a distributed computing environment. Hence, a static, universal pipelining depth would be too restrictive to be used at the endpoint.

2. When multiple endpoints contend for resources, especially with deep pipelines, the resulting system workload is hard to predict. Aggressive endpoints can cause situations similar to network congestion. Throughput of the overall system, as well as on each individual node, could significantly decrease due to the overhead of extensive task switching and even thrashing. When that happens, the system spends a disproportionate amount of time just accessing the shared resource, while not contributing to the advancement of any task.

To address these issues, it would be better for an endpoint to dynamically decide proper node-specific pipelining depth in stead of a static one in order to get the best performance out of non-dedicated and possibly heterogeneous compute nodes. The endpoint scheduler initially issues one task to an intermediate node following the task assignment process described in Section 5.2. When that task is successfully completed or terminated due to resource contention at the intermediate node, the endpoint determines to increase or decrease the next pipelining depth d to use on the fly. The dynamic nature of d enables the endpoint to keep a network of compute nodes busy without incurring bottlenecks related to network latency. At the same time, it avoids using too large a value of d to ensure an individual intermediate node not get overloaded.

5.3.1 Dynamic Pipelining

By treating each intermediate node as if it were a special router that also transforms incoming data, the overall system can be viewed as a data-intensive network. This analogy provides a bridge to apply the concept of flow control, first studied by the networking community. Few would argue that one of TCP's strengths lies in its congestion control mechanism [Jacobson, 1988]. The design of TCP was heavily influenced by the end-to-end argument [Saltzer et al., 1984] in its method of handing congestion and network overload. The premise of the argument and fundamental to TCP's design is that the endpoints are responsible for controlling the rate of date flow. In this model, there are no explicit signaling mechanisms in the network which instruct the endpoint when to speed up or when to slow down. TCP at each endpoint is responsible for answering these questions from implicit knowledge it obtains from the network.

Similarly, the dynamic pipelining scheme infers load conditions at intermediate nodes according to their recent response times to assigned tasks. When an endpoint orchestrates a parallel run, it treats nodes in the network offering different levels of performance. Most importantly, the scheduler dynamically increases or decreases the pipelining depth used by each intermediate node over time to make sure that they do not get overloaded.

However, the challenge here is to define when a depot should be considered as "being overloaded". In TCP congestion control, congestion is indicated by a timeout or the reception of duplicate ACK messages. But in the best-effort Logistical network, due to the high temporal and intermediate node-specific variability, it is difficult to find a timeout value that reliably indicates an overloaded system. Instead, for each intermediate node, the endpoint scheduler starts with $d = 0$ for each node in the network, i.e., no more requests are sent until the response of the previous request has been received. When a task is finished

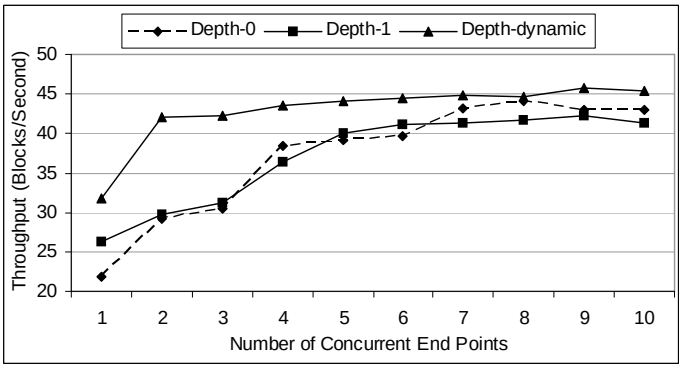


Figure 5.14: System throughput with different number of endpoints and pipelining depth.

successfully, the node's current processing throughput τ is calculated. Supposing the previous processing throughput being τ', pipelining depth d of this particular node is updated by the following rules:

$$d = \begin{cases} d-1, & \text{if } \tau < (1-\rho) \times \tau'; \\ \min(d+1, d\_max), & \text{if } \tau > (1+\rho) \times \tau'; \\ d, & \text{otherwise.} \end{cases}$$

ρ is a tolerance of acceptable variations in the measured performance, e.g. $\rho = 10\%$. Any changes in measured throughput lesser than 10% will not cause d and τ' to be updated. ρ needs to be carefully selected. A large ρ makes the scheduler less adaptive to performance changes due to pipelining depth change. In contrast, a small ρ makes the scheduler sensitive to even performance fluctuation within a reasonable range.

When τ drops by ρ, d is decreased by one. Note that d is allowed to be negative, which means that the scheduler cannot make any new request to the corresponding node. Instead, the scheduler waits for a delay of $1/(\tau' \times 2^{d+1})$. $1/\tau'$ is the time to complete one task by that compute node. $d = -1$ sets the delay to $1/\tau'$ and $d = -2$ causes the delay to be $2/\tau'$. When a previous assigned task is successfully returned to the scheduler and the observed performance causes τ to increase by ρ, d is incremented by one, until d reaches d_max. If the updated value of d is positive, the scheduler issues new tasks to fill up the pipeline. Initially, since there are no outstanding requests, τ' is set to a negative value, thus the scheduler is guaranteed to update every node's τ' with the first processing throughput and starts the pipeline with two concurrent requests (i.e. $d = 1$). In summary, the endpoint scheduler gradually "starts" or "stops" a remote node, according to how much work that node has been able to provide. Slow, overloaded nodes, or nodes with policies limiting usage by the endpoints can be discovered on the fly and treated differently. The process iterates over time until the queue of unfinished tasks is depleted.

Using volume rendering on the same testbed as in Section 5.2.3, system throughput with up to ten concurrent endpoints using different pipelining depth is plotted in Figure 5.14. The ten endpoints asynchronously start their tasks at random instants within a ten seconds span of time. The overall system throughput increases as more endpoints enter the system and peaks when serving ten concurrent endpoints. Note that, we assume each endpoint has only one connection to each depot, hence avoiding the possibility of one single endpoint overloading the entire system. Due this design choice, when there is only one endpoint

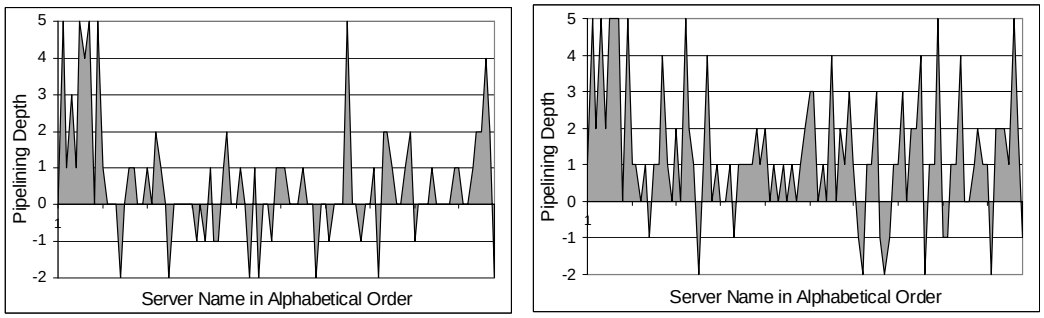


Figure 5.15: A snapshot of pipelining depth of all depots in the testbed

in the system, the peak system-wide throughput is lower than that of multiple concurrent endpoints. Pipelining allows more overlap between CPU and I/O operations, and resulting better system throughput.

Tests with pipelining depth 1 and depth 0 behave similarly except in the single endpoint case. When there is only one endpoint in the system, pipelining does increase system performance since network communication and computation can overlap. The extent of overlapping depends on the nature of the volume rendering operation. Tests with dynamic pipelining depth performed best among the three. With a dynamic pipelining depth, the system reaches its capacity quickly with two concurrent endpoints, delivering more computing power to each parallel visualization application at the endpoint.

To further illustrate the dynamics of task assignment, variations in the pipelining depths of all compute nodes in the system is plotted in Figure 5.15 on the left, when there are four concurrent endpoints. This diagram shows the snapshot of a random time from the perspective of one of the four endpoints, i.e. the other three endpoints appear as background users. Processors with negative pipelining depth have reached their full capacity and the the point is performing back-off on those nodes. For comparison, on the right graph of Figure 5.15, a similar snapshot of node-specific pipelining depth is provided when there is only one endpoint in the system. Less negative pipelining depth means that a lot more active compute nodes can be used for the volume rendering. In all cases, $d\_max = 5$.

5.3.2 Application-level Back-off

With dynamic pipelining, the endpoint scheduler is able to get the best performance out of non-dedicated intermediate nodes without overloading the system. However, as the number of concurrent endpoints in the system increases, the proportion of physical resources that is delivered to each endpoint decreases. For some time-critical applications (e.g. video transcoding, on-demand visualization), system response time is more important than video or image quality. In these situations, quality must be sacrificed in order to obtain timely delivery of results. To meet the user-specified time limit, the endpoint can dynamically measure the system throughput and evaluates whether it is necessary to perform back-off at the application level, i.e. replacing several tasks that would generate high quality results with one task that generates results of lower quality.

We will use an on-demand visualization example to illustrate how the application level back-off can be integrated with the co-scheduling framework presented in Section 5.2. To ensure the highest affordable image quality, data partitions are prioritized according to an importance metric. This importance metric, measured for each data partition, is based on

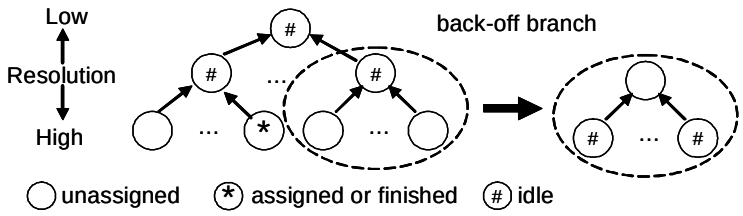


Figure 5.16: An example of back-off selection.

the visual contribution of a partition to the final image. The importance metric becomes the primary key of the two priority queues and the original performance based key is downgraded as the secondary key. In this way, more important data partitions are to be rendered earlier and in higher resolution. Less important partitions are rendered later, close to the deadline. This step is very generic and can be applied to other applications to integrate with the scheduling framework in general.

The importance of a data partition depends on a number of factors that can be application dependent, value dependent or view dependent. The importance metric is a relative term, not on absolute scale, solely for sorting purposes. Specifically, the rules are:

1. A more transparent block is less important;
2. A block with higher variance in voxel values is more important;
3. A block closer to the eye is more important;

Periodically, the scheduler compares the user-specified deadline with the estimated time to finish all the required tasks under the current system throughput, calculated as a summation of all compute nodes' throughput. If the deadline cannot be met, the scheduler marks less important tasks are marked to switch to a lower resolution. The application level back-off is supported by a hierarchical data structure, essentially an octree. The root of the tree represents the lowest resolution of the data and the leave nodes of the tree correspond to the full resolution of the data. In order to ensure that tasks with high importance are rendered with the highest possible resolution, back-off tasks are selected from the tail of the HPQ. To ensure back-off efficiency, as shown in Figure 5.16, only branches of the multi-resolution tree in which all leaf tasks have not been assigned for rendering are selected. Tasks marked with # will not be rendered. Since the data blocks are replicated on several depots, if the scheduler has to choose from several back-off branches that have the same importance, it selects the branch that can be done faster. For example, suppose $T_x, ..., T_y$ can be reduced to a lower resolution task T_m and $T'_x, ..., T'_y$ can be reduced to a lower resolution task T_n. If T_m can finish earlier, we choose Tm to replace $T_x, ..., T_y$. Note that the marking process is dynamic, i.e. if the system throughput improves, tasks are de-marked; and if the system throughput decreases, more tasks are marked.

User-specified rendering time requirement decides the number of blocks at each resolution level to be rendered. Taking the rendering of visible blocks of the TSI dataset at the 31st time step as an example, the actual number of blocks rendered at each solution level with different deadline is plotted in Figure 5.17 on the left. Note that there is only one endpoint using the system when the data is collected. As shown in the figure, a longer deadline allows more blocks with high resolution (e.g. level 0) to be rendered. Conversely, a shorter deadline forces the scheduler to select lower resolution blocks (e.g. level 2). Number

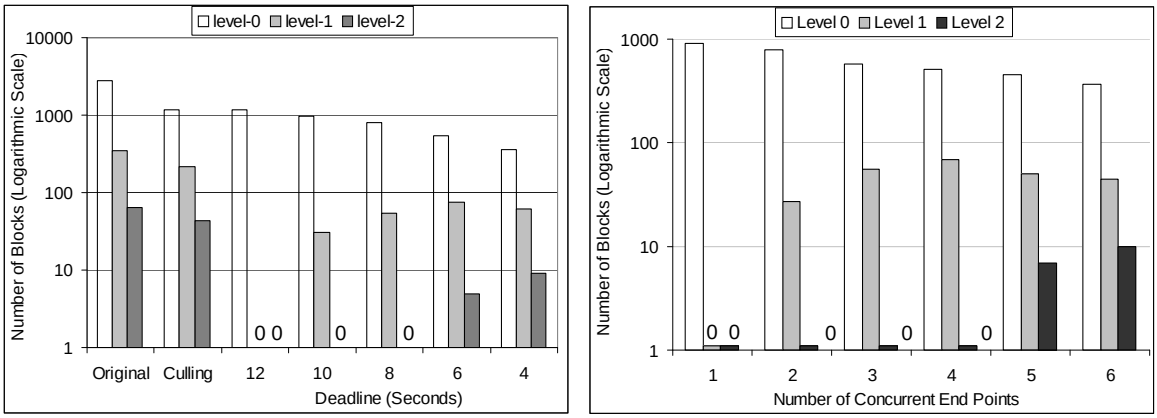


Figure 5.17: Logarithmic plot of the number of blocks rendered at different resolution levels.

of blocks at different resolution levels when the data was partitioned and after the data was pre-processed are shown in the graph as well.

Figure 5.17 on the right shows the number of blocks rendered at different resolution levels when there are up to five concurrent endpoints in the background, running large-scale visualization without any deadlines. The endpoint monitored requested a volume rendering of the first 10 time steps of the TSI dataset. When there were no background endpoints, this rendering job at the highest quality could almost always finish within 55 seconds (on average around 41 42 seconds in most cases) on 100 shared depots. To observe application-level back-off, 55 seconds was used as the deadline, and a different number of background endpoints where added. As shown, as the number of endpoints increased, it became less likely that an intended job could be finished within the user specified deadline. Fewer concurrent endpoints allowed more data blocks with full resolution (e.g. level 0) to be rendered. Conversely, more users would force the scheduler to back-off and select lower resolution blocks (e.g. level 2) in order to meet the deadline of 55 seconds.

Chapter 6

Building a Scalable System

Scalability, as a highly significant property of distributed systems, is generally difficult to define [Hill, 1990] and in any particular case it is necessary to define the specific requirements for scalability on those dimensions which are deemed important. Typically a distributed system is said to be a scalable when if it is suitably efficient and practical when applied to large situations, for example, a large input data set or large number of participating nodes. If the design fails when the quantity increases then it does not scale.

With an implementation of intermediate nodes that is able to provide a best-effort processing service, and techniques to aggregate and schedule fine-grained computations over the network, it is time to build a scalable system. The storage service in our system is provided by IBP. As a generic, best-effort network storage service, IBP itself has been proved to scale globally. However, the management of data is solely the responsibility of upper layer applications. LoRS allows users to store files or parts of files into exNodes, add replicas to existing exNodes, remove replicas from existing exNodes, extend an exNode's duration and retrieve all or part of the logical data stored within an exNode. But LoRS does not has the ability to exploit replication and data reuse to efficiently schedule data-intensive applications, especially when the dataset is large. In this section, we will describe a DHT service built on top of basic LN storage and processing services to manage data replicas in distributed depots.

Due to fragmentation of processing and data in LN, endpoints have to spread a large amount of fine-grained computations onto depots in the wide area network. There is no assumption that those depots are reliable and trustworthy. When the number of nodes participating in LN increases, fault tolerance becomes another issue. In the case of storage, LoRS offers replication and end-to-end correctness in the form of checksum and encryption. We will have a discussion of fault tolerance and scalability in the context of NFU computation.

6.1 Scalable Data Management

Distributed resources in a shared system like ours cannot make absolute service guarantees. The resource contribution delivered by a given depot is a function of its inherent characteristics and its workload. In this respect, our system has two basic types of resources:

1. Those that are well provisioned and lightly burdened, and therefore tend to be reliable and deliver good performance;

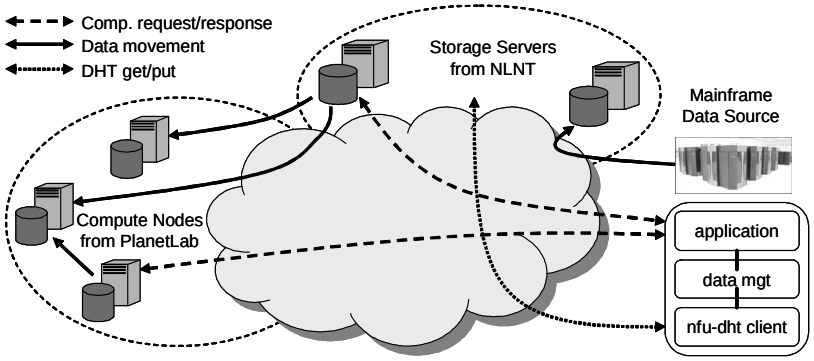


Figure 6.1: System architecture

2. Those that are modestly provisioned and relatively heavily used, and therefore tend to be slower and less reliable.

The NLNT nodes are good examples of the first type of resource. They are dedicated depots and have a substantial amount of storage and processing power. The PlanetLab nodes and other miscellaneous nodes are of the second type. They are shared among a vast community and provide rather limited per-slice storage and processing. Because of the unique characteristic of data-intensive applications, i.e. extremely large datasets shared by distributed users, it is crucial to carefully consider data management in the system.

1. While each user accesses only a subset of the entire dataset, there could be considerable overlaps among the subsets accessed by all users. In addition, distributed users may make runtime replications on-demand. These replicas need to be managed globally so that subsequent jobs started by different users with overlapped data partitions can make use of them.

2. When the size of datasets scales to terabytes and petabytes, the size of metadata tends to be large due to fine-grained allocations and the use of replicas for performance and fault-tolerance. It is problematic to propagate metadata among collaborators or to update it at a central location.

Our system uses a DHT to manage data replicas in distributed depots. Similar to the Translation Lookaside Buffer in classic computer architecture which converts a virtual memory address to physical memory address, the DHT implementation converts a logical name (i.e. which data partition) to a set of memory addresses (i.e. IBP capabilities pointing to cached replicas). Mappings between partition name and capabilities are maintained in the DHT network, avoiding single point failures.

NLNT depots and PlanetLab depots play different roles in the system due to their disparate characteristics as illustrated in Figure 6.1. NLNT depots are mainly used as storage nodes to provide persistent long-term access to the original datasets. They can also participate in any computations on the condition that they do not become overloaded. In addition, NLNT depots form a DHT network for replica management. By contrast, PlanetLab depots are used as compute nodes. The co-scheduling algorithm described in Section 5.2 directs data partitions to be streamed from NLNT depots to PlanetLab depots. After a computation is done, the data partition is cached on the PlanetLab depot and is addressable through the DHT. Subsequent computations would then only need to move the partitions

from NLNT depots that are not cached on PlanetLab depots. Due to abrupt increases in workload or loss of network connection, PlanetLab nodes can fail sporadically, similar to volunteer nodes do in various volunteer computing systems. The system essentially matches the *distributed memory multiprocessor* architecture. Every depot is mapped to a parallel node, with NFU acting as the processor, IBP as the processor's main memory, and storage space provided by NLNT depots as external disks. Before elaborating on the DHT service, we will first describe how the system works in detail.

6.1.1 The System in Operation

A dataset generated on a mainframe or a cluster is partitioned and the hash of each partition is calculated. The terms "partition" and "block" are used interchangeably. As a one time operation, data partitions are uploaded into NLNT depots with a small replication ratio k for relatively long term storage. The resulting data capabilities are stored in the DHT network comprised of NLNT nodes. Clients get data capabilities by querying any of the DHT nodes, and then use the returned capabilities to direct data movement or to assign computation tasks.

When a partition is available on a depot, a $\langle key, value \rangle$ pair is inserted into the DHT network using the interfaces described in Section 6.1.2, where key is the hash of the partition and $value$ is the set of IBP capabilities for the partition. The owner of the dataset may choose not to put the write capabilities and the manage capabilities into $value$, making allocations read only. The time-to-live (TTL) value of the $\langle key, value \rangle$ pair is set to be the duration of the IBP allocation. The entire list of $\langle key, name \rangle$ pairs of all partitions of a dataset is then published among interested groups, for example, via the web. The field $name$ describes the partition. For example, `tsi09.29.bin.ub` indicates that the key field is the hash of partition 29 in time step 09 of a simulation data called "TSI" in unsigned byte binary format.

To use the system, a client needs to obtain a $\langle key, name \rangle$ list of the dataset of interest. For the best performance, the client also needs to obtain a list of PlanetLab nodes that may participate in the computation. The list is either a static file or is dynamically exported by a resource discovery service. To invoke a computation on a particular partition, the client queries the DHT network using the hash key and chooses one of the returned read capabilities as a pointer to a replica to work on.

Initially, DHT queries return only capabilities of allocations on NLNT depots, since all PlanetLab nodes are blank. To balance workload between NLNT depots and PlanetLab nodes, the client starts to move some of the partitions from NLNT depots to PlanetLab nodes when doing computations on NLNT depots. Nodes are chosen randomly or by some performance metric from the published list. When a new replica is made, the resulting $\langle key, value \rangle$ pair is inserted into the DHT network with duration of the allocation as TTL. Computation on this partition can now be assigned to the PlanetLab node that the replica is made on. Later on, when other clients come in and do computation on the same data partition, DHT queries will also return capabilities of replicas made during previous executions on PlanetLab nodes in addition to the original capabilities. IBP capabilities of the partition and the corresponding $\langle key, value \rangle$ pairs in the DHT network are refreshed (when the manage capability is available) to keep hot data on compute nodes.

The effects of distributed data caching are shown in Figure 6.2. We ran seven consecutive visualization jobs back-to-back and plotted the computation and data movement

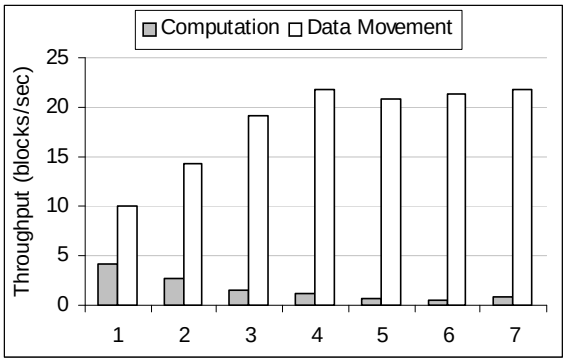


Figure 6.2: Computation and data movement throughput of 7 sequential visualization jobs

throughput. At any particular time, there is only one job in the system. Each job runs ray-casting from a constant view angle on the same 10 time steps of the TSI dataset. However, the view angles used by the seven jobs are 10 degrees apart. On average each parallel job involves about 918 data blocks. In total, the seven parallel runs processed 1019 unique data blocks. Because of the co-scheduling algorithm, the set of depots involved changes over time for each run, although there are rather significant overlaps from one time to another. Thus, constant cache hits are not guaranteed. However, as more jobs get completed, subsequent jobs enjoy increasing cache hit rates. Fewer data blocks are streamed from NLNT depots to compute nodes. As a result of these factors, over time computational throughput steadily increases while the overhead of data movement decreases.

Note that data blocks are not evenly distributed among PlanetLab nodes, instead they are incrementally replicated and cached. Fast nodes with good network connections and large storage space tend to cache more blocks than less well-provisioned nodes. During the entire test, 772 blocks were moved to PlanetLab nodes, while the other 1019-772=247 blocks were completed on the NLNT nodes. Without caching, up to $918 \times 7 = 6426$ blocks would need to be moved across the wide area network, which is a huge overhead in the visualization application.

6.1.2 DHT on top of Logistical Networking Services

When designing a DHT service for replica management, there are several choices. One way is to use a public DHT service. For instance, the OpenDHT [Rhea et al., 2005] has been used in several projects to manage replica at distributed locations. This approach involves the least development and management. However, it decreases data availability because metadata and data are separated into two networks. Suppose the availability of the collection of IBP depots is p_1 and the availability of the OpenDHT network is p_2, then the availability of the system is reduced to $p_1 \times p_2$. If our system had depended on OpenDHT, data stored in IBP depots would not be accessible in November 2004. During that month the OpenDHT service was not available due to PlanetLab V3 rollout.

PlanetLab was designed as both an experimental testbed and a platform for network service creation and deployment. It has been quite successful as a research testbed, providing "realistic" network behavior and client workloads. However, as a service creation platform, it is less successful [Cappos and Hartman, 2005]. PlanetLab is designed to provide only best-effort service to each slice. If several long running slices share a node, scheduling delay can

Table 6.1: Important functions in the `nfu-dht` library

`dht_get_node`	Takes a key and returns k nodes whose IDs are the closest
`dht_put_node`	Adds $\langle ID, node \rangle$ to the routing table, removes and returns a list of $\langle$key, value$\rangle$ pairs that are closer to the input ID than to the local ID
`dht_get_value`	Gets the stored value if a $\langle key, value \rangle$ pair matches the input key
`dht_put_value`	Tells a node to store a $\langle key, value \rangle$ pair to the capability table

Table 6.2: Key functions in the `kad` library

`kad_lookup`	Locates the k closest nodes to a given key in the DHT network
`kad_get`	Retrieves the stored value associated with a key in the DHT network
`kad_put`	Stores a $\langle key, value \rangle$ pair to the DHT network if it is not there; otherwise, refresh the pair
`kad_join`	Put a node in the DHT network by updating its own and its neighbors' routing table

crowd out any service that needs predictable response time [Anderson and Roscoe, 2006]. In contrast, IP routers offer a predictable store-and-forward service that can be modeled by an application of the queueing theory. As a result, IP routers are more scalable. Similar to IP's bounded packet transport service, the LN infrastructure provides limited storage and processing services. Together with IP connectivity, it may provide an alternate platform for creating network services.

By choosing a proprietary DHT implementation on top of the basic LN services, our system not only couples the management of metadata with data itself to improve data availability, but also demonstrates the feasibility of the LN infrastructure to serve as a service creation platform. The implementation is based on Kademlia DHT [Maymounkov and Mazières, 2002]. Kademlia takes the basic approach of many peer-to-peer systems: each node has a 128-bit node ID and keys are also 128-bit identifiers. Compared to other peer-to-peer systems [Ben Y. Zhao and Joseph, 2001,Morris et al., 2001,Rowstron and Druschel, 2001], Kademlia has a symmetric, unidirectional topology by using the XOR metric to measure distance between points in the key space. The choose of Kademlia is because of its provable performance and relatively easy development. The DHT implementation has two separated components as shown in Table 6.1 and Table 6.2: the depot side `nfu-dht` library and the client side `kad` library. For simplicity, hashes of host names are used as node IDs.

The `dht_get/put_node` calls are used to query/update the DHT routing table. Note that `dht_put_node` is a privileged operation that can only be performed by the system administrator who decides whether a node should be in the DHT network. `dht_put_node` also removes and returns a list of $\langle key, value \rangle$ pairs that are closer to the input ID. These pairs are migrated to the new DHT node. `dht_get/put_value` calls are used to retrieve/publish values associated with a key from/to the capability table. The `dht_put_value` call can be rejected if the node knows another node in the DHT network that is closer to the $\langle key, value \rangle$ pair. Both the routing table and the capability table are stored in IBP allocations and are used as implicit arguments to `nfu-dht` calls.

The `kad` library implements a simplified Kademlia protocol at the endpoints, making NFU calls to the passive `nfu-dht` library. The most important procedure in the `kad` library is `kad_lookup`. Initially, it makes a `dht_get_node` call to a well-known DHT node. Then it recursively makes `dht_get_node` calls to the k closest nodes it has learned about from

previous calls. The lookup terminates when `kad_lookup` has queried and gotten responses from the k closest nodes it has seen. With `kad_lookup`, procedures to publish values to and retrieve values from the DHT network can be easily composed. For example, `kad_get/put` is a combination of `kad_lookup` and `dht_get/put_value`. `kad_join` is needed by the system administrator to setup a new DHT node N. `kad_join` shares the same procedure with `kad_lookup`, but to lookup the new node N's ID. Additional steps it takes are: calling `dht_put_node` to add the IDs it has learned to N's routing table, inserting N's ID to other nodes' routing table, and calling `dht_put_value` to put $\langle key, value \rangle$ pairs that were returned by `dht_put_node` to N's capability table.

When a partition is made available on a depot, a $\langle key, value \rangle$ pair is inserted into the DHT network using `kad_put`, where *key* is the hash of the partition and *value* is the set of IBP capabilities and the TTL value of the replica. The `dht_get_value` will expunge expired IBP capabilities if the TTL value indicates that the data is stale. However, the TTL value can be refreshed when needed. Then the entire list of $\langle key, name \rangle$ pairs of all partitions of a dataset is published. The *key* field bridges a partition logical name (the *name* field) and the capabilities of the partition (the *value* field). To access a particular partition, an endpoint needs to obtain the $\langle name, key \rangle$ pair of the partition, query the DHT network using the *key* with `kad_put` and chooses one of the returned read capabilities as an argument to IBP or NFU calls.

6.1.3 Protecting the DHT Network

Security considerations for peer-to-peer DHTs have been studied in the literature [Castro et al., 2002,Fu et al., 2002,Sit and Morris, 2002]. One class of attacks originates from the fact that the nodes involved in the functioning of a DHT cannot be fully trusted. For example, malicious DHT nodes can return incorrect data to an application or forward lookups to an incorrect node or a node that does not exist. Our system is shielded from this kind of attacks by choosing the NLNT nodes that are within our administrative domain to build a DHT network for replica management. The `nfu-dht` library is installed on every NLNT node as a static NFU library for optimized performance. It will do what it is supposed to do, never returning a tampered node or value. In addition, `dht_put_node` is made a privileged operation, protecting routing tables stored on DHT node.

The DHT network is essentially a trustworthy storage network that is shared by many mutually untrusted users with competing goals. Storage allocation policies and algorithms are presented in [Rhea et al., 2005] to ensure fair allocation and prevent starvation so that aggressive users are not favored. The per-user fair allocation algorithm does not fit in here because users with different roles tend to insert different number of $\langle key, value \rangle$ pairs into the DHT network. For example, the user uploading and publishing a dataset will have a large number of `kad_put` calls, while the user running an experiment on the dataset may only have a couple of `kad_put` calls to take care of runtime replicas. Thus, per-user fairness is not enforced in our system. Instead, we put a limit on the number of replicas (i.e. the number of values associated with a key) that a data partition can have to provide a kind of "per-block" fairness in a weak form because the strict "per-block" fairness cannot be achieved without the global information of replicas.

The system also considers the threat that malicious users could insert invalid values into the DHT network. Those values not only waste system resources (e.g. storage spaces) at DHT nodes, but also cause performance issues at applications that use them. To be protected from this attack, `dht_put_value` is extended with a checking module. Every time

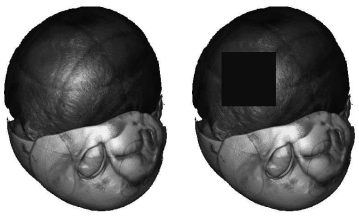


Figure 6.3: An example of computation errors

a user puts a value including at least an IBP read capability, the checking module invokes a NFU operation to validate the value. For example, if the key (hash of the data partition) is calculated using MD5, a MD5 operation is called on the data pointed to by the read capability. The additional check not only ensures that the values in the DHT network are valid, but also guarantees that the new replica and the original data partition are consistent. The extra overhead on the DHT nodes caused by the check procedure should be minimal because the number of `dht_put_value` calls is far less than the number of `dht_put_value` calls through which the cached data blocks are reused.

6.2 Fault Tolerance and Scalability

The difficulty of overcoming faults is one of the great challenges of networking and distributed programming of all kinds. The end-to-end principle requires that intermediate nodes be not only generic but also best-effort, which means that any operation can fail and there is no assumption that intermediate nodes are trustworthy. Traditionally, fault tolerance requires a program to continue operation, possibly at a reduced level of performance (i.e. graceful degradation), rather than failing completely, when some part of the program fails. Faults of this category are usually referred to as program faults. Program faults can be easily detected by error codes and can be gracefully handled. However, in many distributed applications, even if execution is not 100% correct, for example, a malicious intermediate node returns a faked result or a hardware fails transiently, the program can still appear to execute perfectly from the user's perspective.

Fail-stop program faults in LN can be handled through redundant state management. The important point is that the entire state of a NFU operation is exposed to the endpoint through IBP. This means that any form of migration and replication of state can be implemented by the endpoint using an appropriate sequence of IBP data movement operations, properly synchronized with the execution of an instruction pipeline. Even with instruction caching, the credit system provides sufficient control over the execution on the depot to ensure that a very tight loop such as the merge example be interrupted and state management operations inserted by the endpoint.

When preparing for a demo at Super Computing 2005 using depots distributed in the North America, one of the depots always returned incorrect computation results, which resulted to a black area in the composed image as shown in Figure 6.3. The chronic issue here is whether the remote depot can be trusted.

Security and correctness can be ensured to a high level of confidence in data storage and transfer services through the use of end-to-end checksums and encryption. However, for general computing tasks, the application of techniques analogous to checksums and encryption is still a challenging field of research. Mechanisms of NFU present an asymmetry

of trust between depots and endpoints, i.e. the depot does not need to trust the endpoint at all while the endpoint has to rely on a depot to do the work. Trust asymmetry is a core, albeit rarely discussed, problem in scalable computing [Dinda, 2004]. Techniques for protecting a server's operating system from a client's process are well understood and widely deployed. For example, the sandboxing techniques have been used to protect the depot from potentially malicious or faulty oplets. However, there is currently no way to protect the endpoint's computation from the depot. The depot can perform incorrect computations or deliberately produce a faked output at its will.

According to the end-to-end principles, the fewer assumptions that are made about trust placed in depots, the more scalable the depot infrastructure will be. As the system scales to more and more intermediate nodes, the endpoint has less and less trust over depots on which the service module runs because the number of intermediate nodes under the control of the same organization is very limited. The use of un-trusted depots improves scalability, but it requires that the work they do be checked by the endpoint to establish confidence of the results computed. The issue of un-trusted depots can be further divided into three categories: 1) whether there are malicious depots; 2) whether an operation on a depot simply is not producing correct results, for instance, due to a mistakenly assumed endian order; and 3) whether transient faults (e.g. memory bit error) or recoverable errors (e.g. disk read retry) happen on certain depots.

The last two issues are problematic for general scientific applications. As multiprocessor systems and distributed systems become more complex, there is a high likelihood that at any given time, some part of the system will exhibit faulty behavior. For the first one, several plausible techniques can be applied in the context of NFU operations to establish confidence of results computed by un-trusted depots.

Probabilistic or Deterministic Checking

In theoretical computer science, correctness of an algorithm is asserted when it is said that the algorithm is correct with respect to a specification. Functional correctness refers to the input-output behavior of the algorithm (i.e., for each input it produces the desired output). In the case of data transmission, if redundancy is added in the form of checksums, then the receiver can make a probabilistic check whether the data is the same as that was sent. The equivalent of checksums for computation is for an operation to return enough information for it to be verified. For example, yes or no $\mathcal{NP}$-complete problems are difficult to solve exactly but very easy to check if any solution is returned. Thus, redundancy in operation results provides a deterministic check on correctness against malicious depots. However, these applications specific checks place a burden on application programmers.

Authenticated Computation

Given that arbitrary operations cannot be effectively checked by adding redundancy to results, some level of trustworthiness of depots must be assumed. Computation can be pushed onto already authenticated depots. However, authentication is a usually heavy-weight process and a trusted but faulty depot might cause even more damage. Authentication only helps in assigning blame for incorrect results, it does nothing to detect or correct them.

Redundant Computation

Redundant computation has been done in some peer-to-peer computing systems by replicating data, performing computation redundantly and verifying results using techniques like majority voting and spot-checking [Sarmenta, 2002]. In majority voting, each computation is performed several times at different depots and the result wins the highest votes is accepted. In spot-checking, each computation does not be repeated several times. Instead, the client randomly assigns depots a "spotter" computation whose result is already known or can be computed afterwards. Once a depot is caught with bad results, all previous results received from that depot are invalidated.

The spot-checking with blacklisting technique is embedded in our scheduler to reduce the cost of checking while keep a high probability of catching a malicious depot. Suppose p is the rate that a malicious depot submits an invalid result, q is the spot-checking rate and n is the total number of values inserted by the depot. Then the redundant work due to spot-checking is $n/1-q$ and the probability of the catching that depot is $1-(1-pq)^n$ where $(1-pq)^n$ is the probability of the depot surviving through n turns.

Program Slicing

Program slicing [Weiser, 1984] is one of the program analysis techniques that is used to determine the subprogram that is relevant to the computation of specific variables. For example, when the programmer clicks on a variable in a statement, all program statements that potentially affect the variable will be highlighted. This technique can be effectively applied to oplets to increase checking efficiency. When writing oplets, the programmer can identify a number of critical regions in the program. When running oplets, all inputs and outputs to these regions are recorded in an allocation or sent back as part of the results. The endpoint can then choose to redo any of the critical regions and compare the outputs. This technique is a variation of checking and can be fully automated. It reduces the cost and improves the efficiency compared to redundant computation because only part of the real computation is replayed and validated. Since there are potentially many candidate regions to choose for verification, a malicious depot needs to work on all the regions in order to do something bad without being caught.

The techniques discussed so far allow endpoints to work with less trustworthy depots. However, they do not protect algorithms and data against a malicious depot. Ideally, a trustworthy computing system should not only produce the right result, but also be incapable to read or write the operation's inputs, outputs and algorithm. This is the research filed of encrypted computation, for example encrypted string search [Song et al., 2000]. To date, no pursuit of a general solution has been made to arbitrary computation. One goal of scalable computing is to use a large number of nodes that are controlled by other people or organizations for parallel speedups. Unfortunately, authentication of remote depots is not always an option; redundant computation reduce efficiency of the system; checking and even light-weight program slicing techniques put a burden on the client to redo some of the computation which is opposite to the goal.

Heterogeneous network computation should be best-effort, like IP and IBP, in order to be scalable. However, scalability is achieved at the cost of aggregating and scheduling smaller computations as well as building confidence of the results. One might easily reject our system because of the non-negligible cost involved. However, we argue that it's an

instance of a more general problem that all petascale computing systems will encounter. Firstly, scientific problems have to be mapped onto a large number of processors in order to be parallelized, for instance, by fitting into the MPI parallel programming model. Secondly, all computation is arguably best-effort because the result is whatever the best the machine can do if it is not checked. As petascale computing systems grow to thousands of nodes, the usual assumption that systems are fully reliable tends to break [da Lu and Reed, 2004]. In situations where machines have a chance to make an error, something has to be done to establish confidence in the outputs of large-scale scientific applications.

Chapter 7

Conclusions

Logistical Networking focuses on the co-scheduling of storage, computation, and data transmission in computer systems. The result combines data persistence, data transformation and data transfer in a uniform model that can be applied to many difficult problems in wide area computation. In this dissertation, we have shown how to build a distributed computing system for data-intensive applications on top of LN technologies. The resulting system embraces scalability as a defining characteristic because it is designed for scalability using the end-to-end paradigm.

As a novel contribution, end-to-end scheduling techniques including wide area pipelining, co-scheduling of computation and replication, and dynamic workload control are developed to effectively support resource demanding data-intensive applications using the best-effort storage and processing services provided by non-dedicated intermediate nodes. With these techniques, even without specially provisioned resources, our system can already support collaborative data-intensive applications.

In addition, a DHT service on top of LN services has been created to demonstrate the possibility and potential of the LN infrastructure as a service creation platform. In this dissertation, we haven't evaluated the performance of the DHT lookup procedure which decides how fast the data partitions can be pinpointed in the wide area network. The pipelining technique definitely can be applied in this scenario to improve the lookup performance. Detailed latency tests under different conditions may shed some light on the development of service creation platforms.

End-to-end scheduling techniques are the key to achieve performance and robustness of any shared information infrastructure. However, as the system (e.g. the number of concurrent users and the number of diverse applications) grows, it is no longer practical to rely on all endpoints to use end-to-end scheduling techniques for best-effort services. Similarly, it is no longer possible to reply on all application developers to incorporate end-to-end scheduling techniques in their applications. The infrastructure itself must participate in controlling its resource utilization.

Clearly there is more work still to be done in developing and investigating approaches at logistical depots to promote the use of end-to-end scheduling. Fortunately, we have developed a closed queueing network model of the logistical depot to understand its performance characteristics. Modeling results have shown that the system utilization and throughput can be greatly improved by controlling the mix of workloads. The model provides us an ideal vehicle to study workload admission control policies to provide an incentive in support of end-to-end scheduling techniques discussed in this dissertation.

Bibliography

Bibliography

[Acharya et al., 1998] Acharya, A., Uysal, M., and Saltz, J. (1998). Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91, San Jose, CA.

[AkamaiINC, 2002] AkamaiINC (2002). Turbo-charging dynamic web sites with akamai edgesuite. Technical report. White paper.

[Alexander et al., 1998] Alexander, D., Arbaugh, W., Hicks, M., Kakkar, P., Keromytis, A., Moore, J., Gunder, C., Nettles, S., and Smith, J. (1998). The switchware active network architecture. *IEEE Network*.

[Allen and Wolski, 2003] Allen, M. S. and Wolski, R. (2003). The livny and plank-beck problems: Studies in data movement on the computational grid. In *SC'03: Proceedings of the ACM/IEEE conference on Supercomputing*.

[Alt et al., 2005] Alt, M., Hoheisel, A., Pohl, H.-W., and Gorlatch, S. (2005). A grid workflow language using high-level petri nets. In *PPAM'05: Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*.

[Anastasiadis et al., 2005] Anastasiadis, S. V., Wickremesinghe, R., and Chase, J. S. (2005). Lerna: An active storage framework for flexible data access and management. In *HPDC'14, Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*.

[Andersen et al., 2002] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R. (2002). Resilient overlay networks. *SIGCOMM Comput. Commun. Rev.*, 32(1):66–66.

[Anderson, 2004] Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *GRID'04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Los Alamitos, CA.

[Anderson et al., 2002] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

[Anderson et al., 2005] Anderson, D. P., Korpela, E., and Walton, R. (2005). High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the 1st International Conference on e-Science and Grid Computing*, pages 196–203.

[Anderson and Roscoe, 2006] Anderson, T. and Roscoe, T. (2006). Learning from planetlab. In *Proceedings of Usenix WORLDS*.

[Baker et al., 2007] Baker, E. J., Lin, G. N., Liu, H., and Kosuri, R. (2007). Nfu-enabled fasta: moving bioinformatics applications onto wide area networks. *Source Code for Biology and Medicine 2007*, 2(8).

[Bard, 1978] Bard, Y. (1978). An analytic model of the vm/370 system. *IBM Journal of Research and Development*, 22(5):498–508.

[Baru et al., 1998] Baru, C., Moore, R., Rajasekar, A., and Wan, M. (1998). The SDSC storage resource broker. In *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 5, Toronto, Ontario, Canada.

[Baskett et al., 1975] Baskett, F., Chandy, K. M., Muntz, R. R., and Palacios, F. G. (1975). Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260.

[Beck et al., 2004] Beck, M., Gelas, J.-P., Parr, D., Plank, J. S., and Soltesz, S. (2004). LoDN: Logistical Distribution Network. In *WACE'04: Proceedings of the Workshop on Advanced Collaborative Environments*, Nice, France.

[Beck and Moore, 2004] Beck, M. and Moore, T. (2004). Locating interoperability in the network stack. Technical report, Department of Computer Science, University of Tennessee, Knoxville, TN.

[Beck et al., 2002] Beck, M., Moore, T., and Plank, J. S. (2002). An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM '02*, Pittsburgh, PA.

[Beck et al., 2003] Beck, M., Moore, T., and Plank, J. S. (2003). An end-to-end approach to globally scalable programmable networking. In *FDNA-03: Workshop on Future Directions in Network Architecture*, Karlsruhe, Germany.

[Ben Y. Zhao and Joseph, 2001] Ben Y. Zhao, J. D. K. and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley.

[Berman et al., 2003] Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., and Zagorodnov, D. (2003). Adaptive computing on the grid using apples. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382.

[Berman et al., 1996] Berman, F. D., Wolski, R., Figueira, S., Schopf, J., and Shao, G. (1996). Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Pittsburgh, PA.

[Bondi, 2000] Bondi, A. (2000). Characteristics of scalability and their impact on performance. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, Ottawa, Canada.

[Brakmo and Peterson, 1995] Brakmo, L. S. and Peterson, L. L. (1995). Tcp vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8).

[Buyya, 1999] Buyya, R. (1999). *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Buzen, 1973] Buzen, J. P. (1973). Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531.

[Buzen, 1978] Buzen, J. P. (1978). A queueing network model of mvs. *ACM Computing Surveys*, 10(3):319–331.

[Cai et al., 2004] Cai, M., Chervenak, A., and Frank, M. (2004). A peer-to-peer replica location service based on a distributed hash table. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*.

[Cappos and Hartman, 2005] Cappos, J. and Hartman, J. (2005). Why it is hard to build a long-running service on planetlab. In *Proceedings of Usenix WORLDS*.

[Castro et al., 2002] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. (2002). Secure routing for structured peer-to-peer overlay networks. In *OSDI'02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 299–314.

[Chakrabarti et al., 2004] Chakrabarti, A., Dheepak, R. A., and Sengupta, S. (2004). Integration of scheduling and replication in data grids. In *HiPC'04: Proceedings of the 11th International Conference on High Performance Computing*, Bangalore, India.

[Chatterjee, 1996] Chatterjee, S. (1996). *Distributed pipeline scheduling: a framework for design of large-scale, distributed, heterogeneous real-time systems*. PhD dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA. Adviser-Jay Strosnider.

[Chervenak et al., 2001] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., and Tuecke, S. (2001). The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200.

[Czajkowski and von Eicken, 1998] Czajkowski, G. and von Eicken, T. (1998). Jres: A resource accounting interface for java. Technical Report TR98-1679, Cornell University, Ithaca, NY.

[da Lu and Reed, 2004] da Lu, C. and Reed, D. A. (2004). Assessing fault sensitivity in mpi applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 37, Washington, DC. IEEE Computer Society.

[Desprez and Vernois, 2005] Desprez, F. and Vernois, A. (2005). Simultaneous scheduling of replication and computation for data-intensive applications on the grid. Technical Report RR2005-01, Lyon, France.

[Dijk, 1991] Dijk, N. V. (1991). Product forms for random access schemes. *Comput Networks ISDN System*, 22:303–317.

[Dinda, 2004] Dinda, P. A. (2004). Addressing the trust asymmetry problem in grid computing with encrypted computation. In *LCR '04: Proceedings of the 7th workshop on*

Workshop on languages, compilers, and run-time support for scalable systems, pages 1–7, New York, NY. ACM.

[Disz et al., 1997] Disz, T., Olson, R., Olson, R., and Stevens, R. (1997). Performance model of the argonne voyager multimedia server. In *ASAP '97: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 316, Washington, DC. IEEE Computer Society.

[Dovrolis et al., 2004] Dovrolis, C., Ramanathan, P., and Moore, D. (2004). Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking*, 12(6):963–977.

[Duboc et al., 2006] Duboc, L., Rosenblum, D. S., and Wicks, T. (2006). A framework for modelling and analysis of software systems scalability. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 949–952, Shanghai, China.

[Fahringer et al., 2005] Fahringer, T., Qin, J., and Hainzer, S. (2005). Specification of grid workflow applications with agwl: an abstract grid workflow language. In *CCGRID'05: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, pages 676–685.

[Fall, 2003] Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY. ACM.

[Fielding et al., 1997] Fielding, R., Gettys, J., Mogul, J., Frysyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1997). RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. Technical report, IETF.

[Fisher, 1981] Fisher, J. A. (1981). Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490.

[Fisher et al., 1981] Fisher, J. A., Landskov, D., and Shriver, B. D. (1981). Microcode compaction: looking backward and looking forward. In *Proceedings of the 1981 National Computer Conference*, pages 95–102.

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

[Fu et al., 2002] Fu, K., Kaashoek, M. F., and Mazières, D. (2002). Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24.

[Garfinkel, 2003] Garfinkel, T. (2003). Traps and pitfalls: Practical problems in in system call interposition based security tools. In *NDSS'03: In Proceedings of the 10th Network and Distributed System Security Symposium.*

[Gordon and Newell, 1967] Gordon, W. J. and Newell, G. F. (1967). Closed queuemg systems with exponential servers. *Operations Research*, 15:254–265.

[Grimshaw and Wulf, 1996] Grimshaw, A. S. and Wulf, W. A. (1996). Legion-a view from 50,000 feet. In *HPDC'96, Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 89.

[Hac, 1992] Hac, A. (1992). Modeling distributed file systems. *ACM SIGMETRICS Performance Evaluation Review*, 19(4):22–27.

[Hagerup, 1997] Hagerup, T. (1997). Allocating independent tasks to parallel processors: an experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185–197.

[Henderson and Taylor, 1989] Henderson, W. and Taylor, P. (1989). Alternative routing networks and interruptions. *ITC*, 12:1352–1358.

[Heymann et al., 2000a] Heymann, E., Senar, M. A., Luque, E., and Livny, M. (2000a). Adaptive scheduling for master-worker applications on the computational grid. In *GRID'00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214–227.

[Heymann et al., 2000b] Heymann, E., Senar, M. A., Luque, E., and Livny, M. (2000b). Adaptive scheduling for master-worker applications on the computational grid. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214–227, London, UK. Springer-Verlag.

[Hill, 1990] Hill, M. D. (1990). What is scalability. *SIGARCH Computer Architecture News*, 18(4):18–21.

[Hochbaum, 1997] Hochbaum, D. S., editor (1997). *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA.

[Huang et al., 2004] Huang, C., Sebastine, S., and Abdelzaher, T. (2004). An architecture for real-time active content distribution. In *ECRTS'04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*.

[Huang et al., 2007] Huang, J., Liu, H., Beck, M., Gaston, A., Gao, J., and Moore, T. (2007). Dynamic sharing of large-scale visualization. *IEEE Computer Graphics and Applications*, 27(1):20–25.

[Hulaas and Binder, 2004] Hulaas, J. and Binder, W. (2004). Program transformations for portable cpu accounting and control in java. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 169–177, Verona, Italy.

[Innocente, 2003] Innocente, V. (2003). CMS data analysis: Current status and future strategy. In *Computing in High Energy and Nuclear Physics*, La Jolla, CA.

[Jackson, 1963] Jackson, J. R. (1963). Jobshop-like queueing systems. *Management Science*, 10:131–142.

[Jacobson, 1988] Jacobson, V. (1988). Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY. ACM.

[Jagatheesan, 2004] Jagatheesan, A. S. (2004). DGL: The assembly language for grid computing. In *HPDC'13: Demo at the 13th IEEE International Symposium on High Performance Distributed Computing*, Honolulu, HI.

[Jain and Sekar, 2000] Jain, K. and Sekar, R. (2000). User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS'00: In Proceedings of the 7th Annual Network and Distributed System Security Symposium*, San Diego, CA.

[Johnson et al., 2000] Johnson, K. L., Carr, J. F., Day, M. S., and Kaashoek, M. F. (2000). The measured performance of content distribution networks. In *5th International Web Caching Workshop and Content Delivery Workshop*.

[Lam, 1988] Lam, M. (1988). Software pipelining: an effective scheduling technique for vliw machines. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328.

[Larson et al., 2003] Larson, S. M., Snow, C. D., Shirts, M., and Pande, V. S. (2003). Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Modern Methods in Computational Biology, Horizon Press*.

[Litzkow et al., 1998] Litzkow, M., Livny, M., and Mutka, M. (1998). Condor – a hunter of idle workstations. In *ICDCS'98: Proceedings of the 8th International Conference of Distributed Computing Systems*.

[Maymounkov and Mazières, 2002] Maymounkov, P. and Mazières, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS'02: Proceedings for the 1st International Workshop on Peer-to-Peer Systems*.

[McCance, 2003] McCance, G. (2003). Metadata management in the EU datagrid. In *MMGPS'03: Proceedings of the 1st IST Workshop on Metadata Management in Grid and P2P Systems*.

[Morris et al., 2001] Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, San Diego, CA.

[Page et al., 2004] Page, A., Keane, T., and Naughton, T. J. (2004). Adaptive scheduling across a distributed computation platform. In *ISPDC'04: Proceedings of the 3rd International Symposium on Parallel and Distributed Computing*, Cork, Ireland.

[Patterson and Hennessy, 1998] Patterson, D. A. and Hennessy, J. L. (1998). *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc.

[Plank et al., 2003] Plank, J. S., Atchley, S., Ding, Y., and Beck, M. (2003). Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224.

[Price, 1976] Price, T. G. (1976). A comparison of queuing network models and measurements of a multiprogrammed computer system. *SIGMETRICS Perform. Eval. Rev.*, 5(4):39–62.

[Ramakrishnan, 1986] Ramakrishnan, K. K. (1986). A model of file server performance for a heterogeneous distributed system. *ACM SIGCOMM Computer Communication Review*, 16(3):338–347.

[Ramakrishnan and Emer, 1989] Ramakrishnan, K. K. and Emer, J. S. (1989). Performance analysis of mass storage service alternatives for distributed systems. *IEEE Transactions on Software Engineering*, 15(2):120–133.

[Ranganathan and Foster, 2003] Ranganathan, K. and Foster, I. T. (2003). Simulation studies of computation and data scheduling algorithms for data grids. *Journal of Grid Computing*, 1(1):53–62.

[Reed et al., 1998] Reed, D., Saltzer, J., and Clark, D. (1998). Comment on active networking and end-to-end arguments. *IEEE Network*, 12(3):69–71.

[Reiser, 1977] Reiser, M. (1977). Numerical methods in separable queueing networks. *Studies in Management Science*, (7):113–142.

[Rhea et al., 2005] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. (2005). Opendht: a public dht service and its uses. In *ACM SIGCOMM '05*, pages 73–84, Philadelphia, PA.

[Rhea et al., 2001] Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. (2001). Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49.

[Riedel et al., 2001] Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. (2001). Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74.

[Rowstron and Druschel, 2001] Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350.

[Saltzer et al., 1984] Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288.

[Sarmenta, 2001] Sarmenta, L. F. G. (2001). *Volunteer computing*. PhD dissertation, Dept. of Electrical Engineering and Computer Science, MIT, Boston, MA. Supervisored by Stephen A. Ward.

[Sarmenta, 2002] Sarmenta, L. F. G. (2002). Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572.

[Semke et al., 1998] Semke, J., Mahdavi, J., and Mathis, M. (1998). Automatic tcp buffer tuning. *SIGCOMM Comput. Commun. Rev.*, 28(4):315–323.

[Shao et al., 1998] Shao, G., Wolskiy, R., and Berman, F. D. (1998). Performance effects of scheduling strategies for master/slave distributed applications. Technical Report CS98-598, University of California, San Diego.

[Shoshani et al., 1998] Shoshani, A., Bernardo, L., Nordberg, H., Rotem, D., and Sim, A. (1998). Storage management for high energy physics applications. In *Computing in High Energy Physics*.

[Sit and Morris, 2002] Sit, E. and Morris, R. (2002). Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA.

[Sivathanu et al., 2002] Sivathanu, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2002). Evolving rpc for active storage. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 264–276, San Jose, CA.

[Song et al., 2000] Song, D. X., Wagner, D., and Perrig, A. (2000). Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 44, Washington, DC. IEEE Computer Society.

[Srinivasan and Singh, 2002] Srinivasan, S. H. and Singh, P. (2002). Active network file system for data mining and multimedia. In *ICCC'02: Proceedings of the 15th international conference on Computer communication*, pages 714–729, Mumbai, India.

[Sujithan, 1996] Sujithan, K. R. (1996). Towards a scalable, parallel object database - the bulk synchronous parallel approach. Technical Report PRG-TR-17-96, Programming Research Group, Oxford University Computing Laboratory.

[Tennenhouse and Wetherall, 1996] Tennenhouse, D. L. and Wetherall, D. J. (1996). Towards an active network architecture. *ACM SIGCOMM Computer Communication Review*, 26(2):5–17.

[Thain et al., 2002] Thain, D., Tannenbaum, T., and Livny, M. (2002). Condor and the grid. In Berman, F., Fox, G., and Hey, T., editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc.

[Verma, 2002] Verma, D. C. (2002). *Content Distribution Networks: An Engineering Approach*. John Wiley & Sons, Inc., New York, NY.

[Wagner, 1999] Wagner, D. A. (1999). Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, University of California, Berkeley.

[Weiser, 1984] Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

[Wen et al., 2002] Wen, S., Griffioen, J., and Calvert, K. (2002). Building multicast services from unicast forwarding and ephemeral state. *Computer Networks*, 38(3):327–345.

[Whittle, 1967] Whittle, W. P. (1967). Nonlinear migration processes. *Bulletin, The Institute of International Statistics*, 42:642–647.

[Wickremesinghe et al., 2002] Wickremesinghe, R., Chase, J. S., and Vitter, J. S. (2002). Distributed computing with load-managed active storage. In *HPDC'11, Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*.

[Wolski et al., 1999] Wolski, R., Spring, N. T., and Hayes, J. (1999). The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Comp. Syst.*, 15(5-6):757–768.

Appendix

Appendix

7.1 Simulation Results of the Queuing Network Model

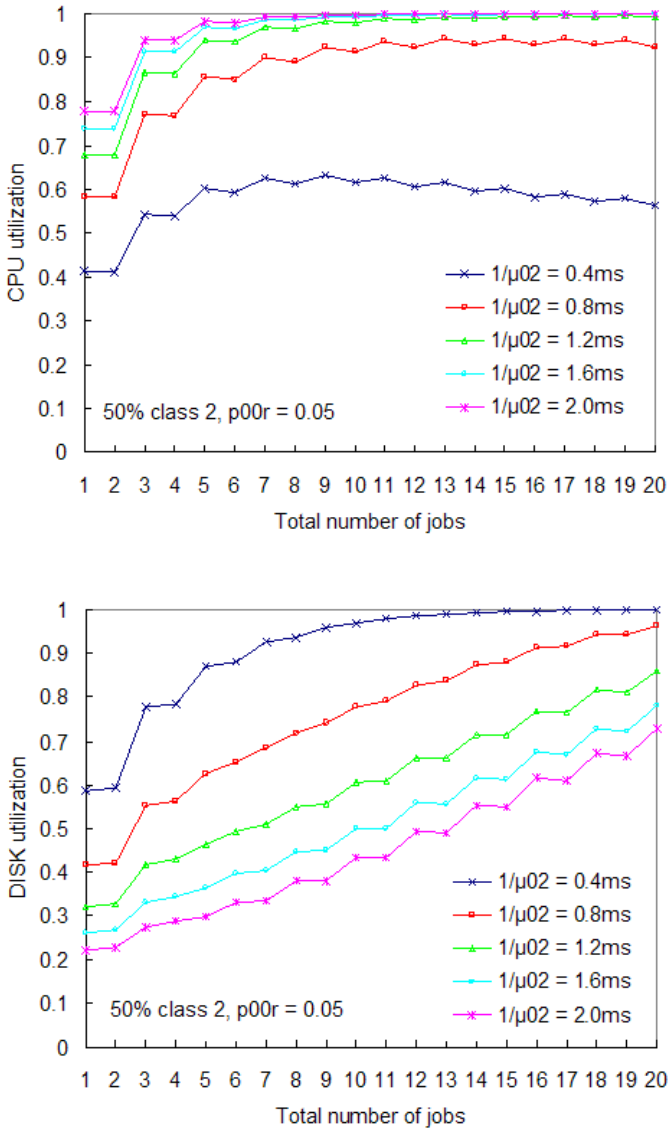


Figure 7.1: CPU and disk utilization with various mean service times of class two jobs.

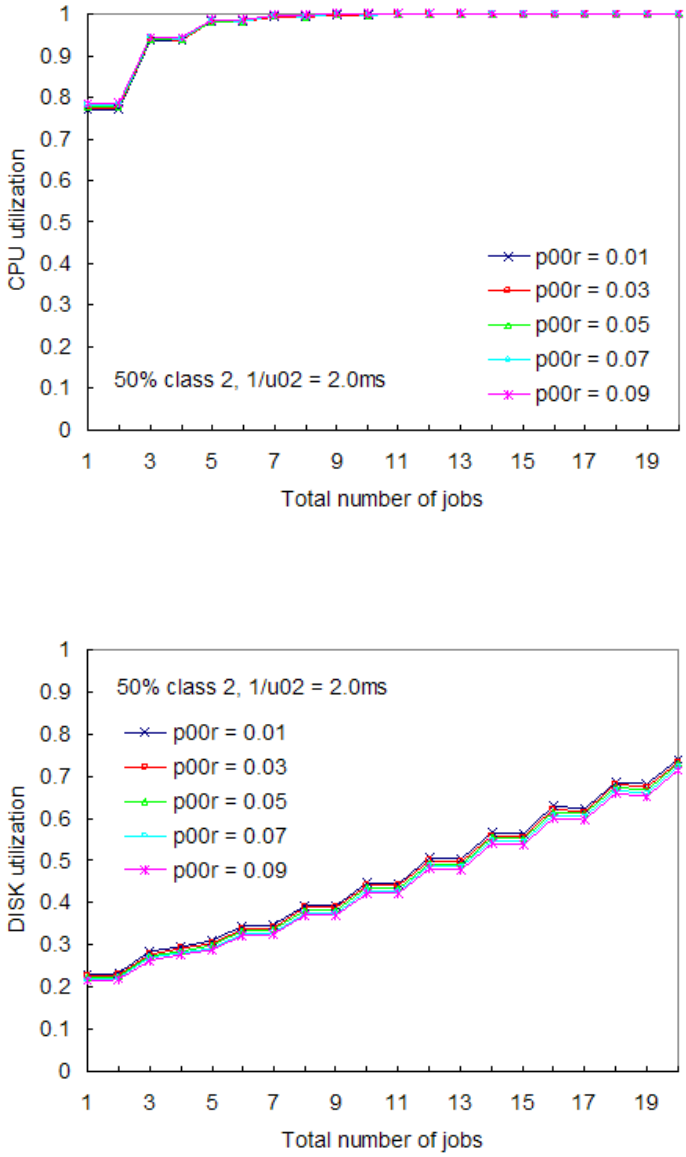


Figure 7.2: CPU and disk utilization with different class one and class two job lengths.

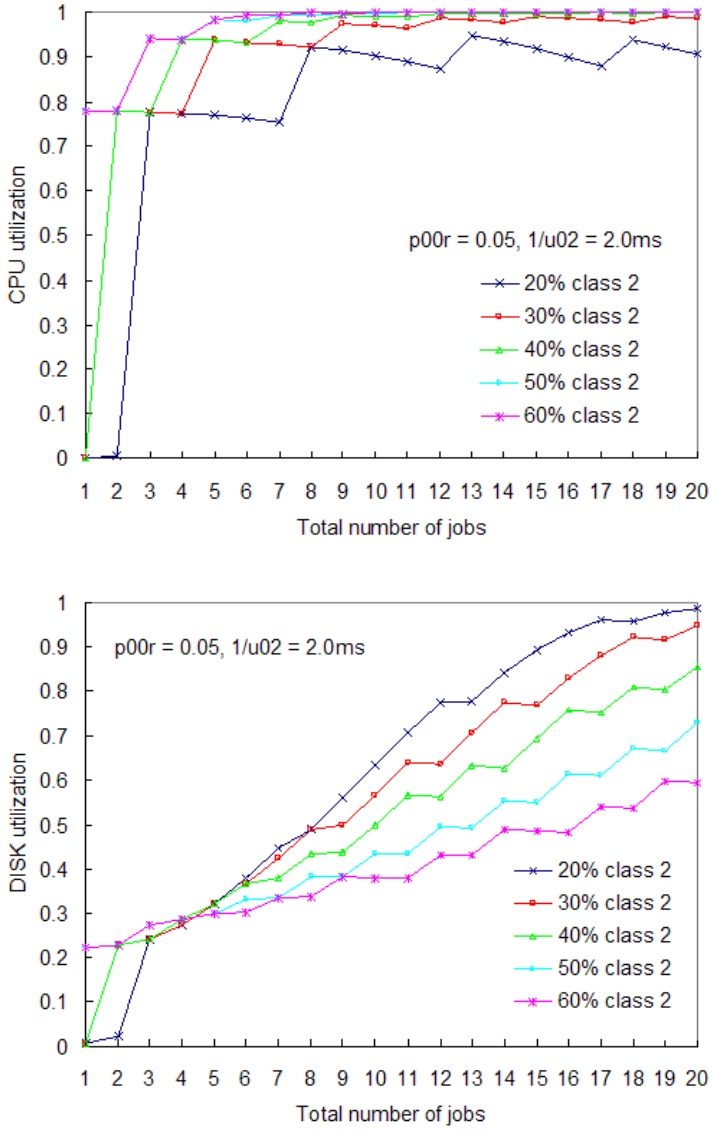


Figure 7.3: CPU and disk utilization with different job mixes.

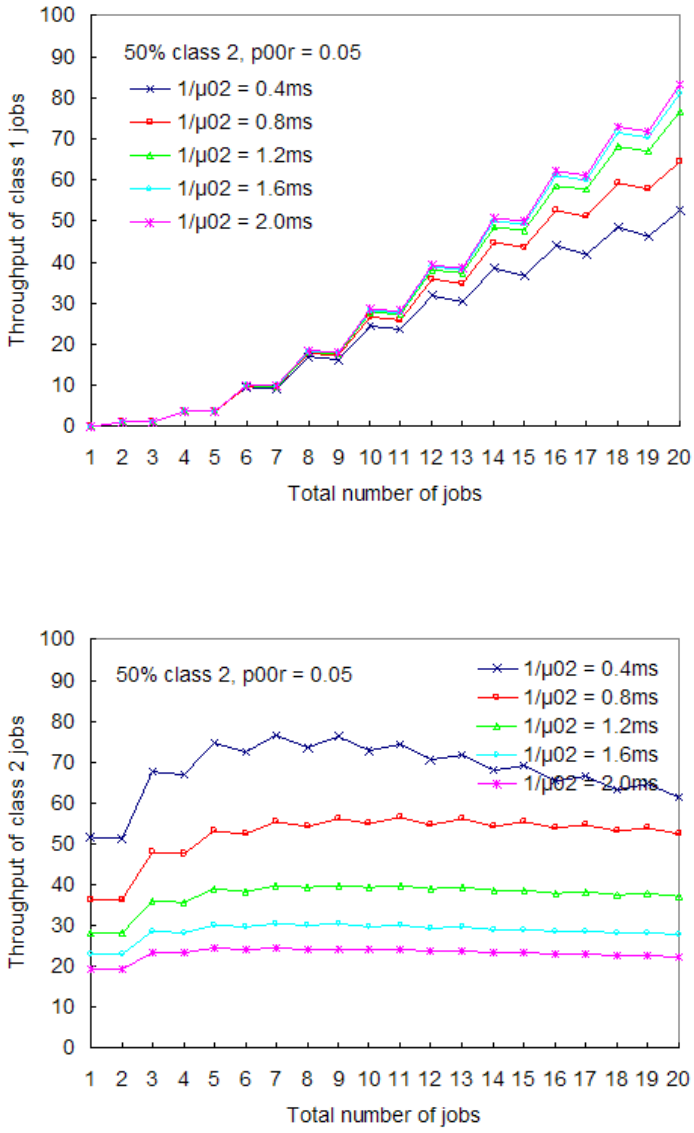


Figure 7.4: Throughput of class one and class two jobs with various mean service times of class two jobs.

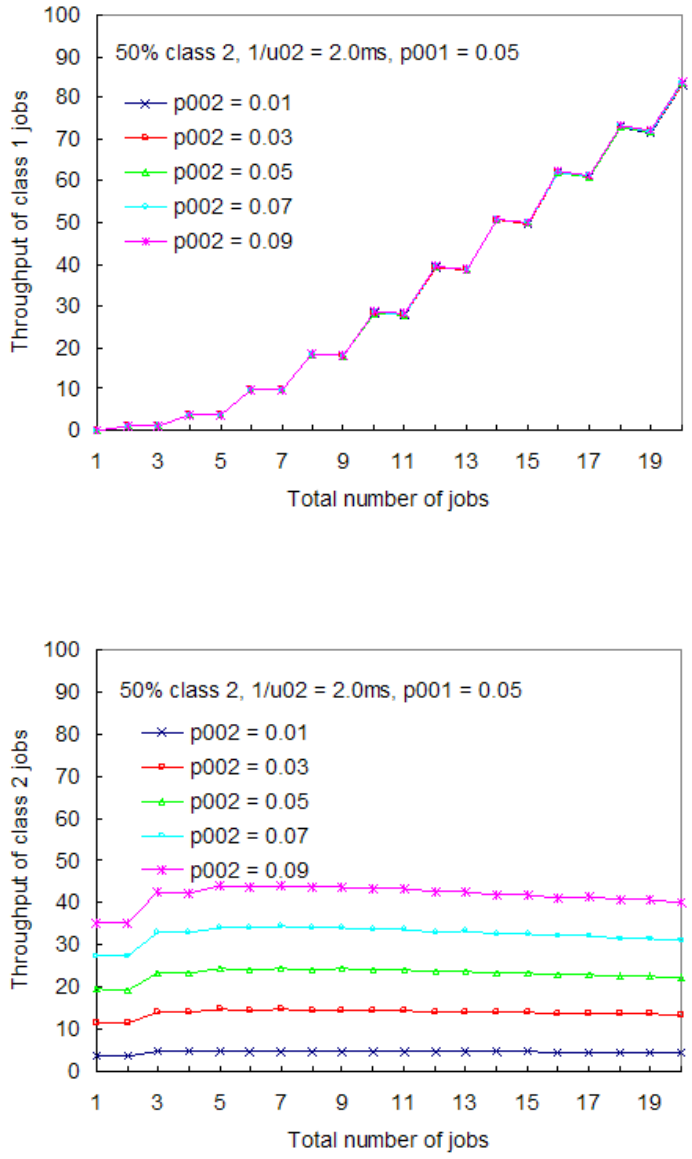


Figure 7.5: Throughput of class one and class two jobs with different job lengths.

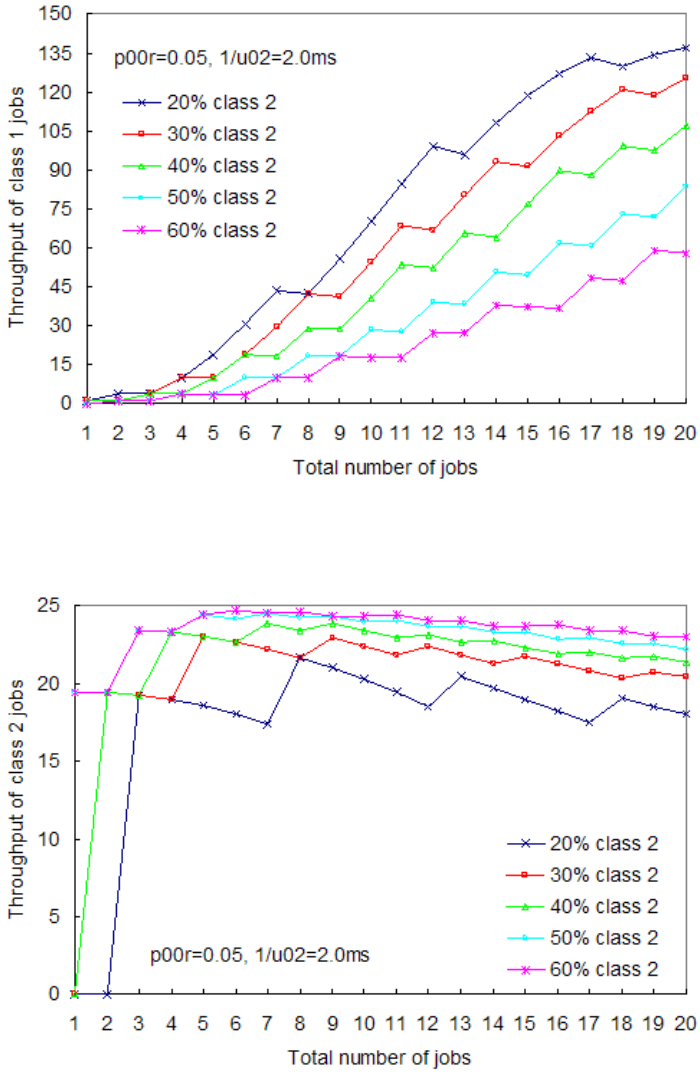
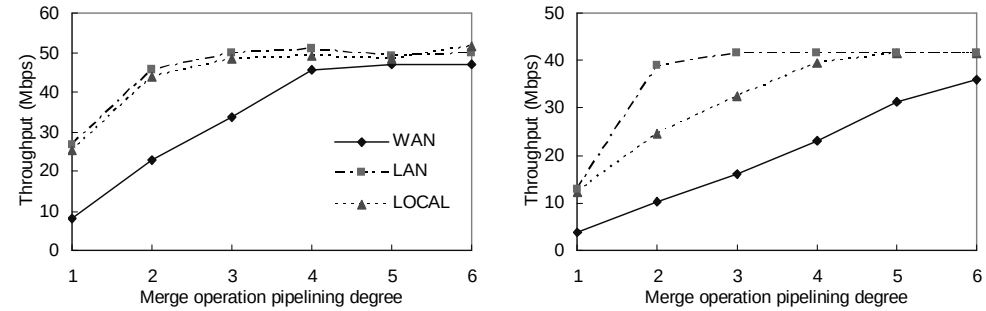


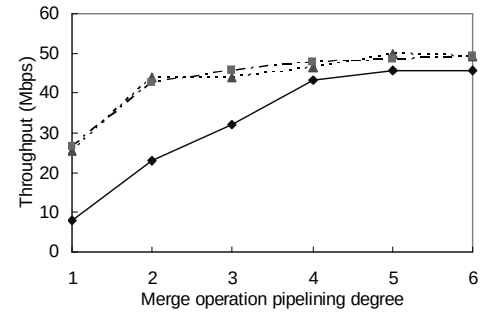
Figure 7.6: Throughput of class one and class two jobs with different job mixes.

7.2 Experiment Results of the Distributed Merge Example

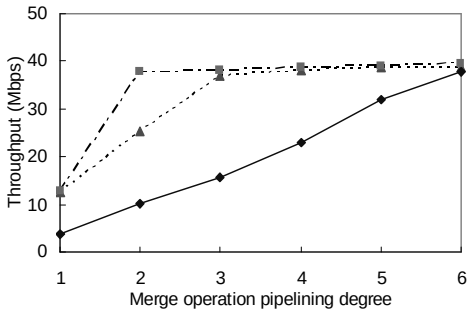


(a) Data pre-fetching degree 8, block size 512KB

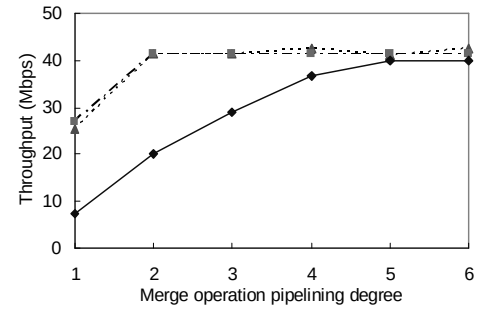
(e) Data pre-fetching degree 8, block size 256KB



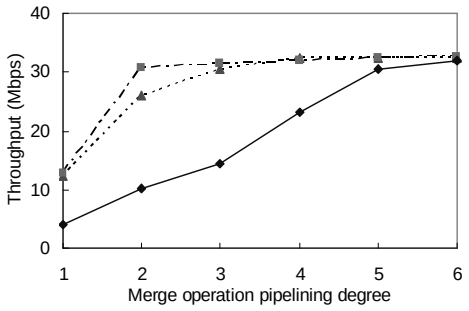
(b) Data pre-fetching degree 6, block size 512KB



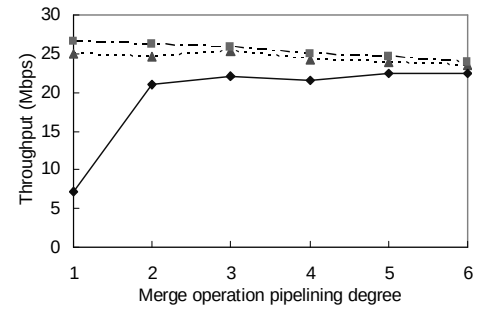
(f) Data pre-fetching degree 6, block size 256KB



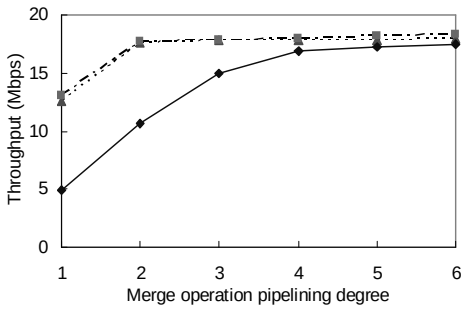
(c) Data pre-fetching degree 4, block size 512KB



(g) Data pre-fetching degree 4, block size 256KB

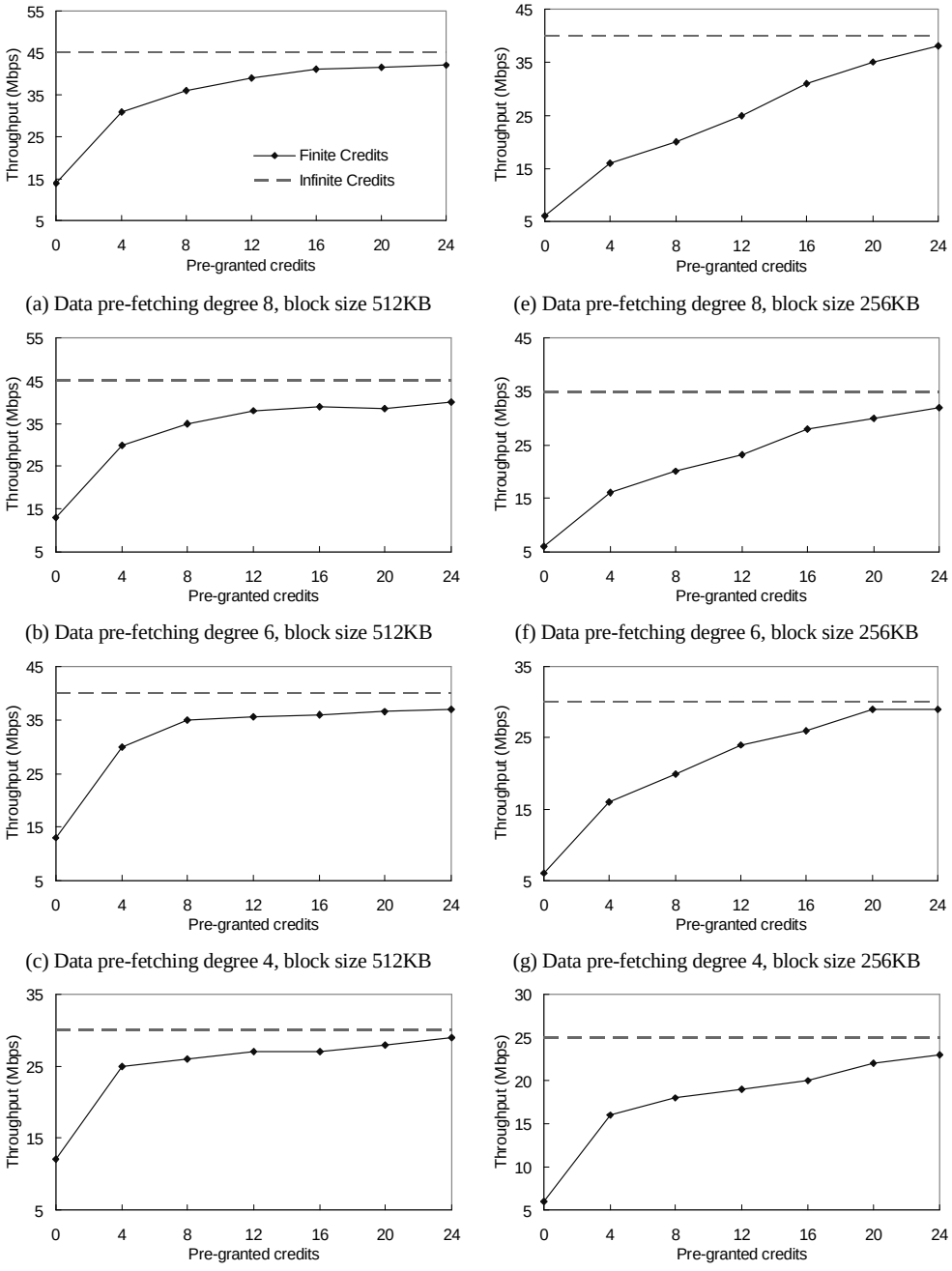


(d) Data pre-fetching degree 2, block size 512KB



(h) Data pre-fetching degree 2, block size 256KB

Figure 7.7: Merge throughput as a function of pipelining depth



(a) Data pre-fetching degree 8, block size 512KB

(e) Data pre-fetching degree 8, block size 256KB

(b) Data pre-fetching degree 6, block size 512KB

(f) Data pre-fetching degree 6, block size 256KB

(c) Data pre-fetching degree 4, block size 512KB

(g) Data pre-fetching degree 4, block size 256KB

(d) Data pre-fetching degree 2, block size 512KB

(h) Data pre-fetching degree 2, block size 256KB

Figure 7.8: Merge throughput vs. excess credits issued

Vita

Huadong Liu was born in Jiangsu, China, on February 27, 1977. After graduating from Funing High School, Jiangsu, China in 1995, he attended Nanjing University of Post and Telecommunications in Nanjing, China, where he received a Bachelor of Engineering degree in 1999 and a Master of Engineering degree in 2002 from the Computer Science and Technology department. During his masters studies, he worked for one year at Global Software Group Motorola China as an intern. In the fall of 2002, Huadong started his doctoral study at the University of Tennessee in Computer Science. In spring 2003, he joined the Logistical Computing and Internetworking Laboratory as a graduate research assistant where he completed his Doctor of Philosophy degree in 2008. During his Ph.D. study, he worked as a Google Summer of Code in summer 2005 and research intern at NEC Labs America in summer 2006.