SATISFYING STRONG APPLICATION REQUIREMENTS IN DATA-INTENSIVE CLOUD COMPUTING ENVIRONMENTS

BY

BRIAN CHO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Indranil Gupta, Chair
Professor Tarek F. Abdelzaher
Assistant Professor P. Brighten Godfrey
Dr. Marcos K. Aguilera, Microsoft Research Silicon Valley

Abstract

In today's data-intensive cloud systems, there is a tension between resource limitations and strict requirements. In an effort to scale up in the cloud, many systems today have unfortunately forced users to relax their requirements. However, users still have to deal with constraints, such as strict time deadlines or limited dollar budgets. Several applications critically rely on strongly consistent access to data hosted in clouds. Jobs that are time-critical must receive priority when they are submitted to shared cloud computing resources.

This thesis presents systems that *satisfy strong application requirements,* such as consistency, dollar budgets, and real-time deadlines, *for data-intensive cloud computing environments, in spite of resource limitations,* such as bandwidth, congestion, and resource costs, *while optimizing system metrics,* such as throughput and latency. Our systems cover a wide range of environments, each with their own strict requirements. Pandora gives cloud users with deadline or budget constraints the optimal solution for transferring bulk data within these constraints. Vivace provides applications with a strongly consistent storage service that performs well when replicated across geo-distributed data centers. Natjam ensures that time-critical Hadoop jobs immediately receive cluster resources even when less important jobs are already running. For each of these systems, we designed new algorithms and techniques aimed at making the most of the limited resources available. We implemented the systems and evaluated their performance under deployment using real-world data and execution traces.

To dad, mom, and David.
Thank you for your constant love and support.

Acknowledgments

I would like to thank my advisor, Indranil Gupta. Thank you for your guidance and belief in me. Our meetings have always lifted up my spirits and productivity. I will miss them.

I am grateful to my committee members, Marcos Aguilera, Brighten Godfrey, and Tarek Abdelzaher. The preliminary and final exams were especially rewarding because of their sharp insights.

I would like to thank the members of DPRG. Ramses Morales, Jay Patel, and Steve Ko inspired me, each in their distinct ways. Watching them, I shaped the habits that carried me throughout my research. Discussions in the office with Imranul Hoque, Muntasir Rahman, and Simon Krueger provided new insights and encouragement that pushed my projects along.

I am thankful to have collaborated with many brilliant, fun folks. The aforementioned members of DPRG were a constant presence, and I had the privilege to work on projects big and small with all of them.

Marcos's theoretical insights and persistent day to day hard work were inspiring. I can only hope some of it rubbed off on me during our work together.

I picked up a lot of system building while working with Abhishek Verma and Nicolas Zea. Abhishek has remained a reliable sounding board throughout my research. I had fun working with Liangliang Cao, Hyun Duk Kim, Min-Hsuan Tsai, Zhen Li, ChengXiang Zhai, and Thomas Huang. Especially, Liangliang and Hyun Duk showed great persistence in presenting our work to the world. I'd like to thank Cristina Abad and Nathan Roberts for their expertise and hard work that led to successful large-scale cluster experiments. Special thanks are due to Derek Dagit for his timely support with the CCT cluster.

I would like to thank my internship mentors: Mike Groble at Motorola, Sandeep Uttamchandani at IBM Almaden, Nathan at Yahoo, and Marcos at

Microsoft Research Silicon Valley.

Finally, I'd like to thank family and friends. Creating a list of personal acknowledgments would be an impossible task, even more so than creating the list of academic acknowledgments listed here (where I have undoubtedly missed many important people). I omit such a list. I hope instead that God blesses me with the opportunity to extend my gratitude in person to each special individual.

Table of Contents

Chapter 1

Introduction

Cloud computing has taken off as a multi-billion dollar market, worth $40.7 billion in 2010, and predicted to grow up to $241 billion by 2020 [116]. This growth has been fueled by the emergence of cloud infrastructures where customers pay on-the-go for limited resources such as CPU hours used, network data transfer, and disk storage [6, 17, 26, 65], rather than paying an upfront cost [44]. The workloads in cloud infrastructures are largely data-intensive. For example, web applications such as photo sharing or social network services store, interpret, and propagate large amounts of user-generated data. Other data includes click data, and increasingly scientific or business data. These are routinely processed by parallel data processing programming frameworks such as MapReduce/Hadoop [62, 19], Pig Latin [111], and Hive [127], at scales of terabytes and petabytes.

In handling this data, there is a fundamental tension between resource limitations and user requirements. At the same time, there is a desire to optimize certain runtime metrics. Thus, in this dissertation we design solutions that meet strong user requirements, such as time, money, consistency, and resource assignment. At the same time, we design our solutions to be practical in resource-limited cloud environments, by optimizing on key metrics.

Many of today's cloud offerings were designed to only optimize key metrics within the specific resource limits of their environment, without meeting many strong user requirements. These systems force users to relax critical requirements. For example, the Dynamo key-value store [64] was designed for low latency and high availability, in a failure-prone environment. The users of Dynamo are forced to have data follow only a (weak) eventual consistency model.

Yet, there is a pervasive demand from users to have their applications meet a strong requirement. One such requirement is a time deadline. During the 2008 elections, a newspaper reporter turned to cloud computing to quickly

System	Strong user requirement	Key optimized metric	Key resource limitations
Pandora-A (Ch 2)	Deadline	Low $ cost	Bandwidth, latency, and cost of various transfer options
Pandora-B (Ch 3)	$ Budget	Short transfer time	
Vivace (Ch 4)	Consistency	Low latency	Bandwidth of cross-site links
Natjam (Ch 5)	Queue priority	High throughput	Cluster capacity

Figure 1.1: Contributions of the thesis.

process the records of a candidate that had just been released [34]. Cloud infrastructure that allowed the reporter to specify and meet deadlines would have been invaluable in publishing the results within the news cycle.

Another requirement is a dollar budget. A group of academic researchers who wish to process a large dataset in the cloud should avoid financial loss by using a system that treats their budget (e.g., funds from a grant) as a strong requirement.

A third requirement is strong consistency guarantees. A storage system that meets this requirement can make developers of a large-scale messaging application more effective [109]. Finally, a cluster operator can offer shared cluster resources to many users at a time, without worrying about interference, when the cluster ensures that important jobs are given priority.

1.1 Thesis and Contributions

Our work shows that *it is feasible to satisfy strong application requirements,* such as consistency, dollar budgets, and real-time deadlines, *for data-intensive cloud computing environments, in spite of resource limitations,* such as bandwidth, congestion, and resource costs, *while simultaneously optimizing runtime metrics,* such as throughput and latency.

Our contributions, as summarized in Figure 1.1, are the design and implementation of systems that solve problems with strong requirements. These systems are *practical*, i.e., they feature good runtime performance and optimally use limited resources. They cover a wide range of application requirements, practical features, and resource limitations.

Pandora-A and Pandora-B create solutions for transferring bulk data across many shipping and Internet links that have unique bandwidth, latency and dollar costs. Each produces optimal solutions while meeting a different re-

quirement: Pandora-A optimizes dollar cost under a deadline constraint, and Pandora-B optimizes transfer time under a dollar budget constraint.

Vivace provides applications with a strongly consistent storage service that maintains low latency even when replicated across geo-distributed data centers that have congested links between each other.

Natjam ensures that time-critical jobs immediately receive cluster resources even when less important jobs are already running, while maintaining high average throughput at all jobs in the cluster. We briefly summarize each system below.

1.1.1 Pandora: Deadline and budget-constrained bulk data transfer via Internet and shipping networks (Chapter 2-3)

Cloud computing promises its users a vast amount of computational resources at their fingertips. Yet, this alone is not enough for many large computations, because data transfer has requirements.

For example, a security researcher may be asked to investigate logs from multiple networks that have been attacked by a botnet. These networks are important to national security, so a strong requirement is placed on the time it takes to transfer the logs from multiple sites to a single datacenter that has the compute resources to process them.

As another example, an astronomer wishes to combine TBs of telescope data from multiple observation points around the world. The amount of money that the astronomer can use to transfer data to the computation site is limited by her research grant. Staying within her grant budget is a strong requirement for the astronomer.

We tackle the distributed bulk data transfer problem with respect to the user's particular strong requirements. For data transfer, a user may have:

- A time deadline for when the transfer must be completed (Pandora-A, Chapter 2), or
- A dollar cost budget that cannot be exceeded to pay for the transfer (Pandora-B, Chapter 3).

Using Pandora, a user can determine whether it is feasible to transfer data for a cloud computation, and is given the transfer plan that should be executed. The service creates better transfer plans by combining multiple Internet and shipping transfer options together. The service empowers cloud users by allowing them to satisfy strong requirements, by making the best use of data transfer resources in the pay-as-you-go cloud environment.

Our Pandora system creates a feasible transfer plan that meets the requirement. Not only does it meet the requirement, but the transfer plan is optimized for another metric. With a time deadline, we produce a transfer plan that meets the deadline while minimizing the dollar cost of the transfer. Likewise, when given a dollar budget, we give a plan that minimizes the time required for the transfer.

The challenge of bulk transfer is that the data is large, and distributed in multiple geographic locations. Pandora's key insight is that there are multiple, competing, options for transferring data in bulk. The Internet provides a convenient option, but in situations can be too slow or expensive. As an alternative, data can be transferred first to a disk drive, and then shipped over transport networks (e.g. via FedEx). These shipment options can be used together to create many different flexible plans. We formulate the various parameters in the cooperative bulk transfer problem as a graph, and develop algorithms to create a transfer plan.

We evaluate Pandora using trace data from actual Internet and shipment networks between academic sites. We find that the optimal transfer plans created by Pandora are significantly better than Internet-only and shipping-only strategies. For example, Pandora meets a deadline that is less than 40% of the time a transfer would take via direct Internet, while costing more than 10% less. Similarly, by setting a deadline within 25% of the fastest but most expensive direct shipping times, Pandora creates a solution that costs 70% less than the direct shipping cost.

Using the algorithms, we have created a Pandora webservice, which can be used by cloud users to plan bulk transfers [95].[1] The webservice produces transfer plans using live data, e.g., shipping times and rates queried from FedEx.

[1]The Pandora service is located at `http://hillary.cs.uiuc.edu`.

1.1.2 Vivace: Consistent data for congested geo-distributed systems (Chapter 4)

As web applications expand their reach to a worldwide audience (e.g. Facebook, Twitter), they must increasingly span multiple data centers in different geographical locations. The limited bandwidth and high latency of links between data centers pose a challenge to keeping data fault-tolerant, available, and consistent with practical data access times. Ad-hoc solutions, such as designating a data center as the single writer [121], are not scalable when there are many geographic locations. Existing approaches often choose to relax the consistency of data. For example, data is cached at remote locations in an asynchronous manner [66], or replicated to be eventually consistent [64].

Yet, many applications require a strong notion of consistency, such as those designed for traditional databases [130], or a new scalable messaging system [109]. Our system, Vivace [55] is a key-value storage system for these applications. It consists of two strongly consistent algorithms, which are designed for low-latency when deployed across data centers, even during periods of congestion which can message delay times to increase.

Vivace's key insight is that the high latency and low bandwidth across geographic regions can cause replication to slow down significantly when there is congestion on the links, while this is not the case for networks within each data center. Taking congestion into account, we design consistent algorithms by separating small control information from data, and prioritizing messages containing only the control information. In the common case, the algorithms can proceed with only the small control information. Messages containing the control information are not delayed, because they are prioritized at outgoing routers.

We have implemented Vivace and evaluated it in a geo-distributed setting. We show that Vivace's prioritization scheme is feasible using commercial routers. The experiments show that Vivace is effective at maintaining low delays in congested networks, reducing delays of up to a second or more compared to prior algorithms on congested networks.

1.1.3 Natjam: Strict queue priority in Hadoop (Chapter 5)

MapReduce, and the open-source Hadoop implementation, has emerged as a general purpose data-intensive computation tool for businesses. Organizations require that production jobs, when submitted, receive priority over research jobs, which are mostly batch jobs. This is because production jobs directly affect revenue, e.g., a production job that counts the number of clicks per advertisement from current ad-click logs.

On the other hand, delays in research jobs can be tolerated, although they too are valuable to the organization. For example, in a research job the ad provider may investigate the change in click habits over the past year, leading to the detection of new click-fraud patterns.

Today, time-sensitive production jobs are typically run in lightly loaded clusters that have tight oversight on job submissions, while batch jobs that are run for research purposes are run on separate clusters with less restrictive job submission policies.

In Natjam, we design techniques which ensure that research and production jobs can share the same cluster. Time-critical production jobs are immediately given cluster resources even when batch research jobs are already running. By meeting this strong requirement we pave the way for organizations to consolidate production and research jobs into a single cluster, which has the benefit of higher resource utilization and more capacity. Production jobs have a higher peak throughput because the consolidated cluster is larger. Research jobs are able to sustain higher average throughput by making use of slack in the larger cluster.

Natjam addresses several challenges. First, production jobs should receive resources in a timely manner. This is met by asking research jobs to *suspend* tasks that are currently executing. Second, research jobs should not lose work. To address this, we save the state at the jobs' tasks through a low-overhead suspend mechanism. When these tasks are later *resumed*, they continue from where they left off.

Finally, research jobs must avoid starvation or unnecessary delays. As tasks do not make any progress when they are suspended, the choice of which tasks to suspend and resume can critically affect the progress of research jobs. We develop eviction policies that decide which tasks will be suspended.

We have implemented Natjam on Hadoop, and evaluated its effectiveness in

a cluster. Our experiments show that Natjam meets its goals of prioritizing production queues while still providing fast completion times for research queues. Natjam allocates resources to jobs in the production queue with little delay, even when the research queue are using all the resources in the cluster. Thus, the completion time of production jobs are minimally affected by the research queue. At the same time, eviction policies help keep the impact of suspending tasks low. Thus, research jobs complete quickly, even when some of their tasks are suspended. In our experiments that compare with existing techniques on a single cluster, we found production jobs can complete in up to 150 seconds less than techniques that fail to prioritize production jobs, and research jobs can complete in up to 750 seconds less than existing techniques that prioritize production jobs at the expense of research jobs.

1.2 Related Work

Data-intensive cloud computing is made possible today, through a variety of systems and services. Large infrastructures include pay-as-you-go services, such as EC2 [4], AppEngine [17], and Azure [26], ready-made infrastructure [12, 31, 24], and dedicated facilities [28]. Several data processing systems have emerged to run data-intensive computations. These include proprietary systems, such as MapReduce [63] and BigTable [50], as well as open source systems, such as Hadoop [19], Pig [111], and Hive [127]. Scalable and low-latency data services, including S3 [5] and Azure Storage [96], and systems, including Dynamo [64], GFS [74], and Cassandra [1], support the back-end storage of the data-intensive cloud. Each of these systems and services are continually optimized to provide competitive performance metrics within restricted environments.

At the same time, recent experiences have suggested the importance of data-intensive systems with strong requirements. For example, the designers of Facebook's Messages framework found their previous system's "eventual consistency model to be a difficult pattern to reconcile [109]." Due to user demand, Amazon introduced new strong consistency features into its SimpleDB service [130]. Finally, to satisfy the requirements of US government agencies and contractors, Amazon now offers fixed price billing plans [9].

Meeting strong application requirements has been the catalyst behind diverse fields such as real-time systems, database systems, and the Internet. Real-time systems must meet scheduling deadlines, while at the same time having practical performance and cost [102, 103]. Many successful database systems provide strong ACID semantics [81]. TCP guarantees against lost, duplicated, and out of order packets, allowing users of FTP and HTTP to receive data as expected [49]. In this dissertation, we are the first to pursue a diverse set of problems characterized by strong requirements in data-intensive cloud settings.

Chapter 2

Pandora-A: Deadline-constrained Bulk Data Transfer via Internet and Shipping Networks

In this chapter, we discuss the deadline-constrained bulk data transfer problem. We formulate the deadline-constrained problem as a graph problem, and then transform it to a Mixed Integer Problem (MIP). By solving the MIP formulation, our system Pandora-A satisfies the deadline constraint while optimizing for a minimum dollar cost.

2.1 Motivation

Cloud computing is enabling *groups* of collaborators to come together in an ad-hoc manner to collect their datasets to a single cloud location and quickly run computations there [44, 73]. In a typical cloud computation project, very large datasets (each measuring several GBs to TBs) are originally located at multiple geographically distributed sources. They need to be transferred to a single sink, for processing via popular tools such as Hadoop, DryadLINQ, etc. [62, 19, 111, 136]. The participants in such a group may be academic collaborators in a project, business partners in a short-term venture, or a virtual organization.

Significant obstacles to the growth of cloud computing are the long latency and high cost involved in the data transfer from multiple sources to a single sink. Using *Internet transfer* can be cheap and fast for small datasets, but is very slow for large datasets. For instance, according to [73], a "small" 5 GB dataset at Boston can be transferred over the Internet to the Amazon S3 storage service in about 40 minutes, but a 1 TB forensics dataset took about 3 weeks! At Amazon Web Services (AWS) [6], which has data transfer prices of 10 cents per GB transferred, the former dataset would cost less than a dollar, but the latter is more expensive at $100.

However, there is an alternative called *shipping transfer*, first proposed

by Jim Gray [79], the PostManet project [132], and DOT [128]. At the source site, the data is burned to a storage device (e.g., external drive, hot-plug drive, SSD, etc.), and then shipped (e.g., via overnight mail, or 2-day shipping) to the sink site (e.g., via USPS [32], FedEx [14], UPS [33], etc.). AWS now offers an Import/Export service that allows data to be shipped and uploaded to the AWS storage infrastructure (S3) in this manner. Using shipping transfer is expensive for small datasets, but can be fast and cheap for large datasets. For instance, the same 5 GB dataset above costs about $50 via the fastest shipping option, which is overnight. The 1 TB dataset, on the other hand, would also cost $50 and reach overnight, which is both cheaper and faster than the Internet transfer.

In this chapter, we focus on the problem of satisfying a *latency* deadline constraint (e.g., transfer finishes within a day) while minimizing *dollar cost*. The choice between using shipping vs. Internet for a given scenario is complex because there are: (1) multiple sites involved, (2) heterogeneity in sizes of datasets, (3) heterogeneity in Internet bandwidth from each source to the sink, and (4) heterogeneity in the shipping costs from each source to the sink. The extended example at the end of this section illustrates this.

Thus, it would be unwise for each participant site in the group to independently make the decision of whether to "ship physically or transfer using the Internet?" Instead, transferring data *via* other sites might be preferable. In other words, we can leverage an *overlay consisting of many sites and both shipping links and Internet links*. This motivates us to model the data transfer as a graph and formulate the problem as finding optimal *flows over time*.

After formulating the problem, we prove its NP-Hardness. Then, we present a solution framework using *time-expanded networks*. A network is represented by a Mixed Integer Program (MIP) and directly solving the problem can give optimal solutions for small problem instances. For large problem instances, we further optimize the solution framework to make the computation time practical. In particular we use knowledge about the transfer networks to reduce the computation time of the branch and bound method used for finding a solution to the MIP. Finally, we build our solution into a system called *Pandora-A* (People and Networks Moving Data Around). Using the implementation, we present experimental results, based on real Internet traces from PlanetLab and real shipping costs from FedEx. Our ex-

periments show that optimal transfer plans can be significantly better than naive plans, and realistic-sized input can be processed in a reasonable time using our optimizations.

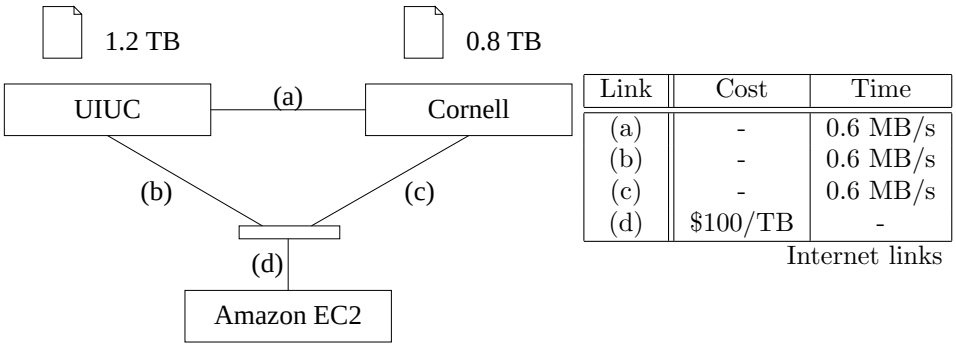
The Pandora planning system takes in as input the following: the dataset sizes at source sites, the inter-connectivity amongst different sites including sources and sink (bandwidth, cost and latency of both Internet and shipping links). Constraints are also specified – in this chapter, we look at latency deadline constraints. The system then outputs a cooperative transfer plan. The plan meets the latency deadline and minimizes dollar cost, by choosing an optimal set of links and a schedule for each link specifying the amount of data that should be transferred. The Pandora system can be extended to solve other constraints as well, e.g., we solve budget constrained problems in [56].

This work represents a significant step forward from the state of the art in several fields. Our results extend not only the extensive class of cooperative network transfer systems [112, 10], but also shipping-only approaches such as Jim Gray's SneakerNet [79], the PostManet project [132], and DOT [128]. In addition, our algorithmic results solve a stronger (and more practical) version of the problems addressed in the theoretical literature on network flow over time [70, 91].

Extended example

Before delving into the technical details of Pandora, we present an extended example to illustrate how different constraints can result in different solutions, even for a fixed topology and dataset sizes. Consider the topology of Figure 2.1, where there are two source sites (UIUC and Cornell) and one sink site (Amazon EC2). The costs and latencies for both Internet transfers (using AWS prices) as well as shipping transfers (using AWS Import/Export prices, with 2TB disks) are shown with the figure.

In the dollar cost minimization version of the problem, the sole objective is to minimize the total dollar cost. The optimal solution to this problem is: send data from Cornell to UIUC via the Internet (no cost), load data at UIUC onto a disk and ship to EC2. The total cost of $120.60 is significantly



Link	Cost	Time
(a)	-	0.6 MB/s
(b)	-	0.6 MB/s
(c)	-	0.6 MB/s
(d)	$100/TB	-

Internet links

Item	Cost	Time
Overnight	$42	24 hrs
Two-Day	$17	48 hrs
Ground	$6	96 hrs

(a)

Cost	Time
$51	24 hrs
$27	48 hrs
$7	96 hrs

(b)

Cost	Time
$59	24 hrs
$28	48 hrs
$8	120 hrs

(c)

Item	Cost	Time
Device Handling	$80/device	-
Data Loading	$18.13/TB	40 MB/s

(d) Amazon EC2 Import/Export Service

Figure 2.1: An example network, with Internet and shipment link properties. The cost of shipment is for a single 2 TB disk (weighing 6 lbs).

lower than other options such as transferring all data directly to EC2 via the Internet ($200) or via ground shipment of a disk from each source ($209.60).

However, this does not work for the latency-constrained version of the problem - the above solution takes 20 days! With a deadline of, say 9 days, the optimal solution is: ship a 2 TB disk from Cornell to UIUC, add in the UIUC data, and finally ship it to EC2. This takes far less than 9 days while also incurring a reasonably low cost of $127.60.

Given an even tighter latency constraint, the best options are: send a disk from Cornell through UIUC to EC2 using overnight shipping, or send two separate disks via 2-day shipping from Cornell and UIUC each to EC2. Given the prices in our input, the latter ($207.60) gives us a price advantage over the former ($249.60). However, it is important to note that even small changes in the rates could make the former a better option.

Finally, we present the case when all data cannot fit into a single 2 TB disk – consider the above example where UIUC has 1.25 TB (an extra 50 GB). Figure 2.2 shows the shipment and handling costs (for overnight shipping) when the number of disks increases. The cost jumps by over $100 when

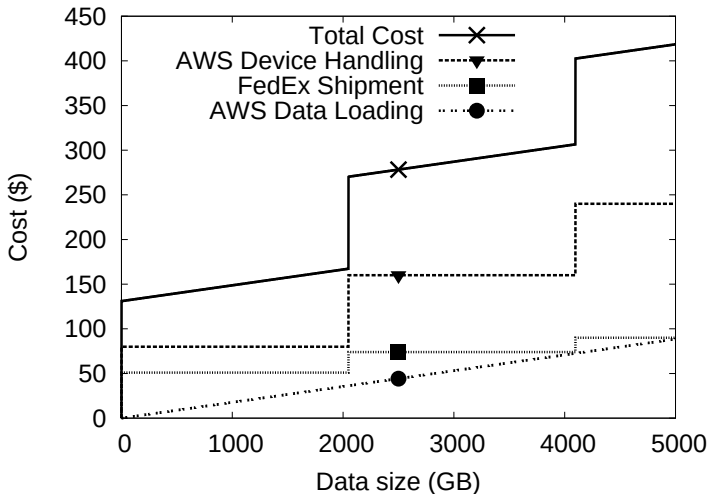


Figure 2.2: Cost of sending data from UIUC to Amazon, on disks with 2 TB capacity. The cost includes FedEx overnight shipping and Amazon handling charges.

shipping two disks instead of one. Thus, for the 50 GB of extra data that doesn't fit into a disk, it is better to send it through the Internet than to combine it on disk.

2.2 Problem Formulation

In this section, we present a formal definition of the deadline-based data transfer problem. We model our problem as a network flow graph with both shipping links and Internet links connecting all pairs of sites. We formalize the flow and constraints on the graph *over time*, because data movement is dynamic and time-sensitive.

2.2.1 Graph Model

We model our problem as a directed graph, where edges have capacities, costs, and transit times. The edge properties are based on the characteristics of the links connecting sites, and the bottlenecks that appear within sites.

Internet and Shipping links

Graph edges represent the transportation of data through the network, composed of Internet and disk shipment links. Each link has a *capacity*, a *cost*, and a *transit time*. An Internet link has a constant capacity equal to the average available bandwidth, a transit time set to zero (since millisecond latencies are negligible), and a cost of zero (except when terminating in the sink).

Shipping disks in packages has entirely different properties. The first thing to note is that there are many levels of service (e.g. Overnight, Two-day, Ground, etc.) when shipping between sites. We treat each level of service as a distinct link, and the discussion below pertains to any one of these links.

The price paid (cost) for shipment with 2 TB disks is depicted by the "FedEx Shipment" line in Figure 2.2. This cost is a *step function* of the amount of data transferred. The cost grows with the number of disks, but not linearly with the amount of data inside each disk. For example, sending either 0.2 TB or 1.8 TB has the same cost because a single disk is used, but sending 2.2 TB has a higher cost because an additional disk is needed.

The time that a package takes to reach its destination (transit time) is on the order of hours or days from the time it was sent, however the time in transit is not fixed. Rather, it depends on the time of day – for instance, with overnight shipping, all packages sent from UIUC anytime before 4pm may arrive at Cornell the next day at 8am. The amount of data shipped at once (i.e., capacity) grows with the number of disks sent, on which shipping links impose practically no limits.

In summary, a shipment link has a cost that follows a step function of the amount of data transferred, capacity that is infinite, and a transit time that depends on the time sent.

Site bottlenecks

However, knowing the capacity, cost, and transit time of each link is not sufficient. Our model also combines end-site constraints. The combined model is depicted in Figure 2.3 and elaborated below.

Data sent by disk does not enter a site instantaneously. Rather, an individual at the receiving end must remove the disk from its packaging, then

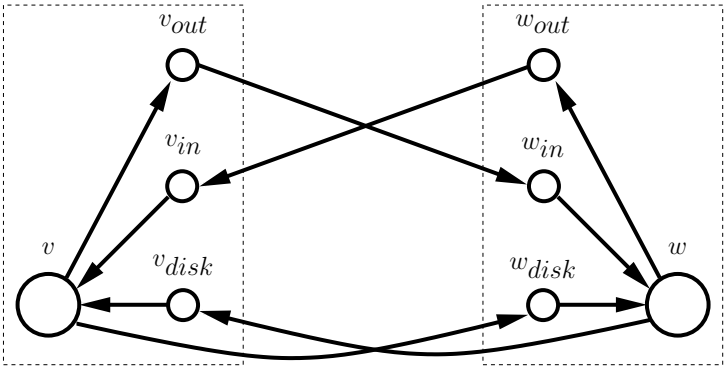


Figure 2.3: Two connected sites v, w in our model. A site v is represented by vertices v, v_{out}, v_{in}, and v_{disk} and the edges between them.

plug in the disk and transfer the actual bytes. The transfer is done through a disk interface such as eSATA, which has a typical transfer rate of 40 MB/s. Only when this transfer is finished is the data transfer considered complete. In addition, this process is not always free. Cloud services charge per-disk and per-data fees, e.g., see the "AWS Device Handling" and "AWS Data Loading" lines of Figure 2.2.

We formulate our model to include this stage of the data transfer at node v in the following way. We first add a new vertex v_{disk} and new edge (v_{disk}, v). This edge represents both the capacity constraint (e.g. 40 MB for eSATA transfer) and the per-data linear cost (the data loading fee, if this is a sink node); these properties are similar to an Internet edge. A shipment link from node w to node v is represented by the edge (w, v_{disk}). This edge has the shipment link properties (step function cost, infinite capacity, and send-time dependent transit time).

There are also end-site constraints for data transferred through the Internet. Most often, not all connections can be used to capacity at once, because there is a common bottleneck at the incoming or outgoing ISP. We model this by including two extra vertices v_{in} and v_{out} and corresponding edges (v_{in}, v) and (v, v_{out}). The Internet connection between sites v and w is depicted with (w_{out}, v_{in}) and (v_{out}, w_{in}). These edges take the properties of Internet links, i.e., capacity equal to available bandwidth, zero transit time, and zero cost.

2.2.2 Data Transfer Over Time

Our graph model is formalized as a flow network $\mathcal{N}$ consisting of the set of directed edges A and vertices V discussed above, along with the following attributes on these elements. Each edge in the graph $e \in A$ has a capacity u_e, cost function c_e, and transit time function τ_e. Each vertex $v \in V$ has a demand attribute D_v. Demand represents the data that originates from that vertex, and needs to be transferred to the sink. Vertices having non-zero values of D_v constitute a terminal set $S \subseteq V$. This set is further partitioned into source terminals $v \in S^+$ which have $D_v > 0$ and sink terminals $v \in S^-$ with $D_v < 0$ such that $\sum_{v \in S} D_v = 0$. In this chapter, we focus only on the single sink problem, where $|S^-| = 1$.

Now, expanding to consider time as well, flow is assigned to each edge at time unit θ, denoted as $f_e(\theta)$. A flow that is feasible and satisfies demands within deadline T adheres to the following four constraints:

i) Capacity constraints: $f_e(\theta) \leq u_e$ for all $\theta \in [0, T), e \in A$

This ensures that at any time, flow through an edge does not exceed its capacity.

ii) Conservation of flow I:

$$\int_0^{\xi} \left(\sum_{e \in \delta^+(v)} f_e(\theta) - \sum_{e \in \delta^-(v)} f_e(\theta - \tau_e(\theta)) \right) d\theta \leq 0$$

$$\text{for all } \xi \in [0, T), v \in V \setminus S^+$$

This ensures that any non-terminal vertex sends out only as much flow as it has received until then. This does not preclude the storage of flow at vertices.

iii) Conservation of flow II:

$$\int_0^{T} \left(\sum_{e \in \delta^+(v)} f_e(\theta) - \sum_{e \in \delta^-(v)} f_e(\theta - \tau_e(\theta)) \right) d\theta = 0$$

$$\text{for all } v \in V \setminus S$$

This ensures that at time T, there is no leftover flow at any vertex other than

the sink.

iv) Demands:

$$\int_0^T \left(\sum_{e \in \delta^+(v)} f_e(\theta) - \sum_{e \in \delta^-(v)} f_e(\theta - \tau_e(\theta)) \right) d\theta = D_v$$

for all $v \in S$

This ensures that the total amount of flow that has left and entered a terminal node by time T is equal to the amount specified.

Under the above constraints, we would like to find flows that are feasible and satisfy demand, and in addition meet monetary and performance goals. In particular, the problem we focus on in this chapter is to minimize dollar cost while satisfying a deadline. That is:

$$\text{Minimize } c(f) := \sum_{e \in A} \int_0^T c_e(f_e(\theta)) d\theta$$

while completing transfer within deadline T. Here c_e is either a linear function or a step function.

Unfortunately, this problem turns out to be NP-Hard.

Theorem 2.2.1. *Solving the data transfer problem is NP-Hard.*

Proof. Our problem is NP-Hard because it is a generalization of the known NP-Hard problem of minimum cost flows over time with finite horizon [91]. That problem has only linear cost functions, where the cost on an edge is defined as a single coefficient k_e on an edge, such that $c_e(f_e(\theta)) = k_e * f_e(\theta)$, while our problem additionally considers costs that are defined as step functions. Also the transit times in [91] are fixed constants τ_e, while in our problem the transit time is a function τ_e that depends on the sending time. □

2.3 Deadline-constrained Solution

In this section, we obtain transfer plans by deriving an *optimal solution* to the data transfer problem. For small problem sizes this computation can

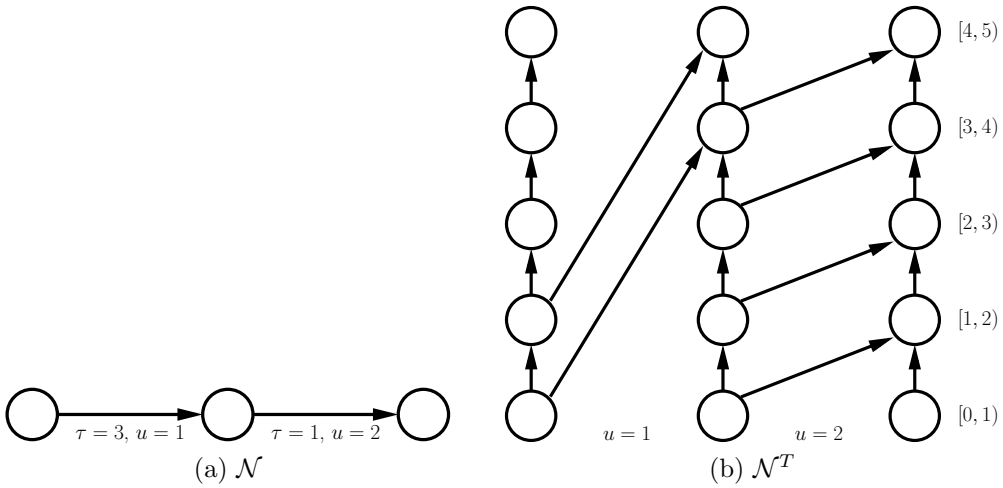


Figure 2.4: An example network, and its corresponding time-expanded network with $T = 5$. Intuitively, a vertex in the time-expanded network represents the original vertex at a point in time. The time range to the right reflects that time passes as we advance up the network.

be done quickly. We later explore optimizations for larger problem sizes (Section 2.4).

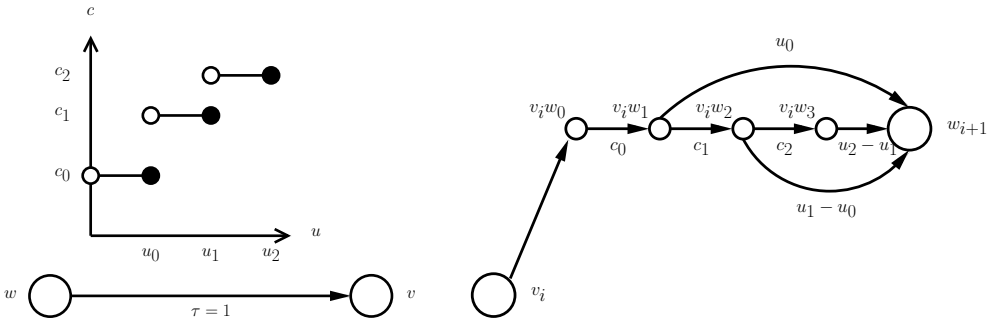
Our approach uses the following 4 steps to solve the problem:

- **(Step 1: Formulate)** Formulate inputs into a data transfer problem with flow network $\mathcal{N}$.
- **(Step 2: Transform)** Transform $\mathcal{N}$ into a static T-time-expanded network $\mathcal{N}^T$.
- **(Step 3: Solve)** Solve static min-cost flow problem; resulting flow is x.
- **(Step 4: Re-interpret)** Re-interpret x as flow over time f.

We showed in Section 2.2 how the problem is formulated (Step 1). We expand on the other individual steps in the rest of this section.

2.3.1 Building a Time-Expanded Network

In the previous section, we formulated the data transfer problem as a flow network. However, this representation is unwieldy because there are many



(a) Edge (v, w) in $\mathcal{N}$ with step cost function.

(b) Decomposed edge $(v_i, w_{i+\tau})$ in $\mathcal{N}^T$. Unlabeled costs are 0, and capacities are ∞.

Figure 2.5: Decomposition of an edge with a step cost function. The costs in $\mathcal{N}^T$ are fixed costs that must be paid if at least a single unit of flow goes through the edge. The vertices in $\mathcal{N}^T$ are v_i, v_iw_0, v_iw_1, v_iw_2, v_iw_3, and w_{i+1}.

attributes for each edge, including the transit time and cost function. Thus, we transform the flow network into a new representation with edge attributes that are easier to handle. We remove the transit time attribute from the graph by transforming the flow network into a time expanded network [70]. We apply the canonical time-expanded network conversion to edges with linear cost. On the other hand, for edges with a step function cost we develop a novel conversion. When applied, this conversion further simplifies edge attributes, by removing step function costs from the graph.

Intuitively, a T-time-expanded network represents the possible flows through all edges and vertices, from 0 up to T time units. Figure 2.4 depicts an example with 3 nodes and time moving upwards. A canonical T-time-expanded network $\mathcal{N}^T$ creates T copies of each vertex (labeled v_i) in the network. It then replaces edge $e = (v, w)$ in the original graph $\mathcal{N}$ with edges $(v_i, w_{i+\tau_e})$ for $i = 0, 1, ..., (T - \tau_e)$. Intuitively, flow on this edge represents flow originating at v in the original graph at exactly time i. In addition, in order to account for the storage of flow at node v, $T - 1$ holdover edges are added between v_i and v_{i+1} with infinite capacity.

For step cost edges, we develop a new conversion that decomposes the edge into multiple intermediary vertices and edges. Similar to linear cost edges, we explicitly depict a path for sending flow between an edge $e = (v, w)$ in the original graph for each time unit. However, instead of a single edge, the path is composed of multiple edges. In addition to eliminating transit time, the conversion eliminates step costs by breaking them into arcs with linear cost and fixed cost.

The conversion is illustrated with an example in Figure 2.5. The first intermediary edge $(v_i, v_i w_0)$ represents the transit time τ_e. Then, for each step in the cost function, we add an intermediary vertex and two intermediary edges. The first intermediary edge $(v_i w_0, v_i w_1)$ represents the fixed cost c_0 that must be paid to ship at least one more unit of flow. The second intermediary edge $(v_i w_1, w_{i+\tau_e})$ constrains the amount of flow that can be sent while paying the fixed cost to capacity u_0. In the next step the intermediary edges are $(v_i w_1, v_i w_2)$ with cost c_1 and $(v_i w_2, w_{i+\tau_e})$ with capacity $u_1 - u_0$, and so on.

As a whole, the edges between the original vertices v_i and $w_{i+\tau_e}$ can hold flow originating at v in the original graph at exactly time i. All flow must arrive at the destination w by time $i + \tau_e$. Therefore, unlike the original vertices, the intermediary vertices cannot store flow, and so no holdover edges are drawn between them.

2.3.2 Solving the Static Network Problem

In literature, the static time-expanded network is solved using polynomial time minimum cost flow algorithms [69, 78]. However, these solutions do not apply to our problem because of its unique characteristics. Concisely, in our static problem some of our edges (depicted in Figure 2.5b) have fixed cost defined by a constant cost k_e that must be paid in full when at least one unit of flow goes through the edge:

$$c_e(f_e) = \begin{cases} k_e & \text{if } f_e > 0 \\ 0 & \text{if } f_e = 0 \end{cases}$$

Given the presence of these edges, we solve the static network problem exactly as a Mixed Integer Program (MIP). We define integer (binary) variables y_e defined on the set of fixed cost edges $e \in F$. The value of y_e is 1 when the edge is used for at least one unit of flow and 0 when it is not used at all. Formally, the problem becomes:

$$
\begin{aligned}
\text{Minimize} \quad & c(f) := \sum_{e \in A \setminus F} k_e f_e + \sum_{e \in F} k_e y_e & \\
\text{s.t.} \quad & f_e \leq u_e & \text{for all } e \in A \setminus F \\
& f_e \leq u_e y_e & \text{for all } e \in F \\
& \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e = D_v & \text{for all } v \in V \\
& y_e \in \{0, 1\} & \text{for all } e \in F
\end{aligned}
$$

Unfortunately, a solution for the MIP formulation may not be computable within a reasonable time because of two factors. First, the static T-time-expanded network has a size of $O(n \cdot T)$ edges, where T is a numeric value of the input. Thus, even if a polynomial time algorithm on the static network size existed, the network size itself may be exponential in the input of the flow over time problem.[1]

Second, solving the MIP can be time consuming even with a small T, for a large network. The existence of integer variables suggests that the problem is hard to solve. In fact, it is impossible to remove the integer variables (thus making the problem a Linear Program) because the problem is NP-Hard.

Theorem 2.3.1. *Solving the min-cost flow on a static network with fixed cost edges is NP-Hard.*

Proof. The proof is by reduction from the Steiner Tree problem in graphs. To solve a Steiner Tree problem, we can convert the original graph to a directed graph by replacing each undirected edge (u, v) with two directed edges $u \to v$, $v \to u$; each directed edge is given unlimited capacity and unit fixed cost. All but one of the terminal vertices are set as source vertices with unit demand, and the remaining terminal vertex is set as a sink vertex with negative demand equal to the number of source vertices.

It is easy to see that a demand-satisfying feasible flow in the converted graph defines a connected tree in the original graph (with same cost): each source is connected to the sink, and thus all terminals are connected. Like-

[1] This is intuitively why the min-cost flows over time problem is NP-Hard despite the existence of polynomial time min-cost flow algorithms.

wise, each connected tree must be a demand-satisfying feasible flow (with same cost): there is a single path between a terminal and each of the other terminals, and this can be used as the path that flow goes from a source to the sink.

The min-cost flow in the directed graph must be a Steiner Tree (i.e., min-cost connected tree) in the original graph. Assume this is not true, and there is a connected tree with less cost; then there must be a corresponding flow with less cost, which is a contradiction. □

We use the GNU Linear Programming Kit (GLPK) [16] to solve the MIP. We present the branch-and-bound method used by GLPK and techniques we developed to improve the computation time of the MIP in Section 2.4.

Finally, we describe how we re-interpret the solution in Step 4. When the static solution is solved, Pandora finally transforms the solution x back onto the original network. The transformation is straightforward: take the flow $x_{e=(v_i,w_{i+\tau})}$ going through an edge $(v_i, w_{i+\tau})$ in the static network; in the flow over time, this becomes the flow $f_{e=(v,w)}(i)$ that initiates at v at time $\theta = i$. For edges with step functions, we can take as $f_e(\theta)$ the amount of flow $x_{e'=(v_i,v_iw_0)}$ going through the first edge in the decomposition.

2.4 Optimizations

The previous section showed how to convert the data transfer problem to a Mixed Integer Program (MIP) which in turn can be solved by plugging into a general MIP solver. However, the NP-Hardness result and our evaluation in Section 2.5 show that processing time can become impractically long for larger problems. In this section we present optimizations made in Pandora to improve computation time. Characteristics specific to the data transfer flow networks are identified and used to reduce the computation time of the MIP.

Branch and Bound Example: As background for the presentation of our techniques, we discuss via Figure 2.6 an example of the branch and bound method. In our small example, we consider sending 2.01 TB of data from a

$$
\begin{aligned}
\text{Minimize} \quad & 17y_1 + 42y_2 + 0.1f_3 \\
\text{s.t.} \quad & f_1 + f_2 + f_3 = 2010 \\
& f_1 \leq 1872y_1 \quad f_2 \leq 2000y_2 \quad f_3 \leq 140 \\
& y_1, y_2 \in \{0, 1\}
\end{aligned}
$$

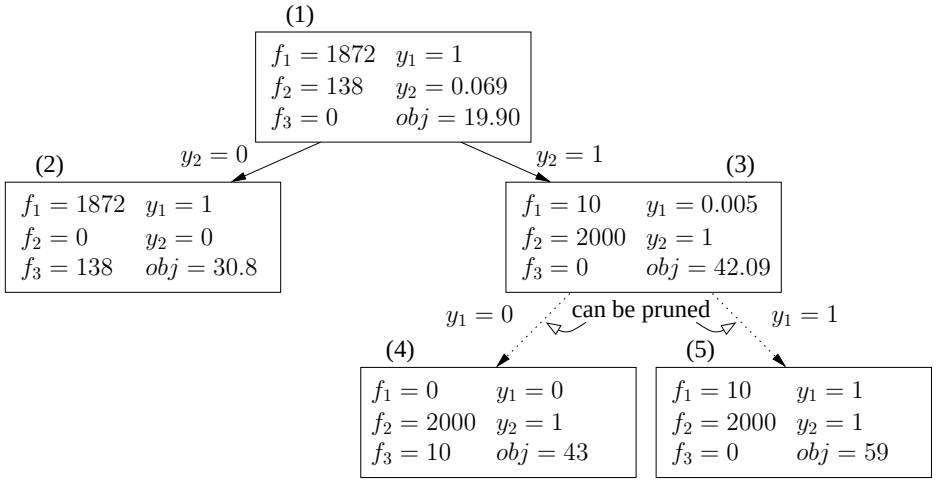


Figure 2.6: A branch and bound tree for an MIP with two binary variables. The LP solution is shown for each subproblem. The optimal solution of the MIP is found at subproblem (2). If the subproblems are solved in numerical order, the subtree of subproblem (3) can be pruned.

single source site to a destination site within a tight time bound (61 hrs).[2] We must choose among three methods of transfer, and because the time bound is tight, there are constraints on how much data can be sent within the time bound – 2 TB via overnight, 1.872 TB via two-day, and 0.14 TB via Internet. The binary variables y_1, y_2 represent the shipping options.

To solve a MIP with the branch and bound method, the original problem is solved as a Linear Program (LP) by removing integer restrictions. This solution may not be a solution to the MIP, because the latter must have integer values for integer variables. In order to find such a solution, an integer variable that is not assigned an integer value in the LP solution is selected to be *branched*, creating new subproblems. In our example, as a result of solving the LP for the original problem (1) we obtain an objective value of 19.90, while y_2 has a non-integer value. To remove the non-integer value, the problem is branched by selecting variable y_2 and setting the variable y_2 to 0

[2]This particular MIP has been simplified from the time-expanded MIP in Section 2.3.2. The simplification is possible because there are only two sites and this is not the case in general. A more formal description of branch and bound can be found in [100].

or 1. This results in the two subproblems (2) and (3). Each subproblem must then be solved as an LP, and if there are non-integer values, again branched.

In the worst case, an exponential number of subproblems must be solved. With binary variables, this corresponds to a complete binary tree with height equal to the number of binary variables. Fortunately, the number of subproblems that must be solved can be reduced by *bounding* the possible solutions of a subproblem and its branches with its LP solution. If this bound is worse than a known solution, the branches of the subproblem can be pruned without solving their LPs.

Continuing our example, an LP solution is found at (2) with objective value 30.8. This solution is a solution to the MIP problem because the integer variables have integer values. Thus it does not have to be branched further. Next the LP for subproblem (3) is solved. Its objective value 42.09 has a larger value than the known MIP solution 30.8. This proves that none of the subproblems branched from (3) can have a solution better than the known solution at (2), and thus both (4) and (5) are pruned without solving their LPs.

From this example, we can see that a possibly large number of subproblems must be solved as LPs in order to compute the optimal MIP solution. We can improve the computation time by reducing the number of subproblems solved by making informed branching and bounding decisions, or reducing the time needed to solve each subproblem.

Our optimizations are divided into two categories of techniques: the first category (A) preserves an optimal solution while the second (B) trades off computation time by producing a possibly sub-optimal solution. These optimizations are not mutually exclusive, rather combinations of them can be used, as we do in the Pandora system.

2.4.1 Optimal Solution Techniques

Below, we describe our first set of optimizations, whose goal is to still maintain an optimal solution. The first two optimizations (1) & (2) make direct changes to the MIP formulation, based on characteristics specific to shipping and Internet data transfer networks. In the third optimization (3), an ad-

ditional MIP is formulated and solved. Pandora uses information from this solution to improve the branch and bound execution for the original MIP.

Reducing shipment links

When shipping a package between sites, we have observed from our experiments with FedEx that there are a small number of possible package arrival times during a day. For example, an overnight package from UIUC sent anytime between noon and 4pm will arrive at Cornell the next day at 10am. Thus, it does not matter whether the package is sent at noon or four hours later.

We use this observation to our advantage. This allows us to reduce the number of shipment links represented in the MIP. When shipping has the same cost and arrival time for different send times, we only need to represent one of these shipment edges in the time-expanded graph. We can ensure this does not affect the deadline or cost by choosing the one with the latest send time. Intuitively, a flow that would have been sent earlier can be stored and sent at the chosen send time and still arrive at the same time. By reducing the number of shipment links present in the MIP, we crucially reduce the number of integer variables y_e. This results in a faster branch and bound computation. The number of possible branches is reduced, which reduces the number of subproblems solved. It also reduces the size of each subproblem, and thus the time required to solve each LP.

Adding negligible amounts of cost to Internet links

Our second optimization relies on the following observation: if data is to be sent over Internet links, it makes sense to send data as soon as possible. In other words, there is no gain in storing data at the node. In contrast, when shipping disks we need to wait until a lot of data can be sent at once, because of the fixed costs involved. We use this as an assumption in formulating the MIP, by adding per each Internet edge (v_i, w_i) in the time-expanded graph, a negligible cost proportional to the time i (e.g., $\frac{i}{T} \cdot 0.00001$ \$/GB in our

experiments). This gives a hint to the MIP solver to send via Internet edges as soon as data is available. It is not obvious how this optimization should speed up the computation, but we observe in our experiments in Section 2.5.2 that the computation time per subproblem LP is reduced.

Using a lower-bound solution

In the two previous optimizations, we used characteristics of the transfer problem to change the MIP problem formulation. In this optimization, we improve the branching and bounding methods without changing the MIP formulation. Instead, before solving the MIP, we create and solve a separate lower-bound MIP. This solution helps make wise branching decisions and creates a lower bound that can be used to prune subproblems in the original MIP. The lower-bound MIP itself can be computed much faster than the original MIP because it is much smaller.

The lower-bound problem is constructed by removing, from the time-expanded network all Internet edges $(v_t, w_t), t = 0, 1, ..., T-1$ between each node. All these edges together are replaced by a single edge (v_{T-1}, w_0) with a capacity equal to T times the capacity of each edge. Likewise, bottleneck edges are only defined at v_{T-1} and w_0 and the capacity of the bottleneck Internet edges are increased by a factor of T.

These edges represent the total amount of data that can be transferred from v to w within the deadline period. The minimum cost solution of the new problem is thus a lower bound of the original MIP. This is because all flow between $(v_t, w_t), t = 0, 1, ..., T-1$ in the original MIP can be summed as the corresponding flow across (v_{T-1}, w_0) in the new problem. Please note that the converse however may not be true. A solution in the new formulation may contain data transfer that is impossible in the original MIP. Because edges are drawn from time $T-1$ to time 0, data is allowed to be transferred backwards in time according to the time-expanded network.

After solving the lower-bound problem, we leverage the solution to solve the original MIP. The computation time is made faster by using information from the lower-bound solution – this is done by us in two ways. First, the minimum dollar cost found by the lower-bound problem is used as an absolute

lower bound for the original MIP. If a solution is found during the original MIP branch and bound process that is equal to the lower bound, all remaining subproblems are pruned because they cannot contain solutions with lower dollar cost. This reduces the number of subproblems that are solved.

Second, the shipping edges chosen by the lower bound solution are used to make branching decisions in the original MIP. Intuitively, the use of a shipping edge in the lower-bound MIP can indicate that shipping should be used instead of Internet transfer between v and w. Of course, this information is not perfect, because the lower-bound MIP gives optimistic times for data arrival at each site. For instance, any data transferred through Internet edges arrives at the destination at time 0. Thus, we make use of the information in the following way. If a shipping edge is chosen between v and w in the lower-bound solution, all shipping edges between v and w are prioritized when choosing branching variables for the original MIP. We choose to prioritize all shipping options and times instead of just the exact shipping edge chosen because we expect a shipment to be made between the sites, but do not know the exact time or option that will be used.

2.4.2 Approximate Solution Techniques

In the previous optimizations, we computed optimal transfer solutions. However, computing an optimal solution may take a long time for large MIP problems, e.g. due to many sites, long deadline time, or a large amount of data to send. In these cases, it may be more practical to trade guarantees on the optimality of the solution with reduced computation time. The optimizations in this section provide methods of applying this trade-off. In optimization (1), a solution still maintains the lowest dollar cost, but may overstep its deadline by a bounded amount of time. In optimization (2), the deadline is met, but the lowest dollar cost is not guaranteed.

Δ-condensed time-expanded network

The size of a time-expanded network grows with T. To reduce the dependence of network size to T, we utilize Δ-condensed networks. These networks can find fast approximate solutions for the flow over time problem [70]. In-

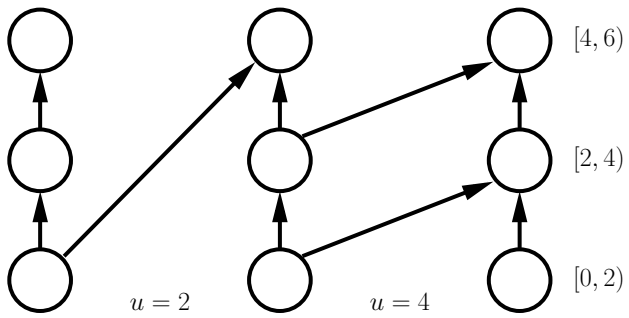


Figure 2.7: The canonical Δ-condensed time-expanded network $\mathcal{N}^T/\Delta$ for network in Figure 2.4, with $\Delta = 2$. Intuitively, Δ consecutive time units at each vertex are compressed into a single vertex. The time ranges to the right reflect this.

tuitively, a Δ-condensed network compresses each group of Δ consecutive time units at each vertex in the time expanded network into a single vertex. Further, this is done synchronously across all vertices.

As illustrated in Figure 2.7, a canonical Δ-condensed T-time-expanded network $\mathcal{N}^T/\Delta$ is constructed of $\lceil T/\Delta \rceil \Delta$ copies of vertices in $\mathcal{N}$. The transit times in a condensed network are rounded up to the nearest multiple of Δ. An edge $e = (v, w)$ in $\mathcal{N}$ is now transformed into edge $(v_i, w_{i+\lceil \tau_e/\Delta \rceil})$. This edge represents all the flow through e originating during a Δ timespan, thus its capacity is set to $u \cdot \Delta$.

We define a modified Δ-condensed network conversion for step cost edges. An infinite capacity step cost edge is decomposed in the same way as in a T-time-expanded network, except that the destination vertex is $w_{i+\lceil \tau_e/\Delta \rceil}$. The capacities u_i do not change because they represent the cost function, rather than the actual capacity of the link (which is infinite).

By condensing the network, we are able to reduce the number of edges by a factor of Δ. However, to insure that we still get a transfer plan with minimum cost, the time expansion of the static network must be increased. The corresponding Δ-condensed network has an increased deadline $T(1+\varepsilon)$, where $\varepsilon = \frac{n \cdot \Delta}{T}$ as we prove in Theorem 2.4.1. The resulting network has $O(n^2/\varepsilon^2)$ vertices, which is less than the $O(n \cdot T)$ vertices in a regular time-expanded network when T is large and ε is not too small.

Thus, we substitute the time-expanded network in the transform step of the solution in Section 2.3 with a Δ-condensed network with time expansion $T(1+\varepsilon)$. Concretely, step 2 becomes:

- **(Step 2: Transform*)** Transform $\mathcal{N}$ into a static Δ-condensed time-

expanded network $\mathcal{N}^T/\Delta$, where $\Delta := \frac{\varepsilon}{n}T$; and time expansion $T' := T(1+\varepsilon)$.

Step 4 (re-interpret) is modified in the following way: Consider flow going through an edge $(v_i, w_{i+\lceil \tau_e/\Delta \rceil})$. Define τ'_e such that $\tau'_e + \tau_e = \lceil \tau_e/\Delta \rceil \Delta$. Notice that $\tau'_e + \tau_e = \lceil \tau_e/\Delta \rceil \Delta$ is the (rounded) transit time in the Δ-condensed network. For a linear cost edge, we convert to flow over time by holding the flow at v for τ'_e and sending $1/\Delta$ of the flow for Δ time units in interval $[\tau'_e + i\Delta, \tau'_e + (i+1)\Delta)$. Fixed cost edges are converted by holding the flow for $\tau'_e + \Delta - 1$ time units and sending the entire flow at once at $\tau'_e + (i+1)\Delta - 1$. These conversions obey flow conservation and capacity. The cost remains the same because the cost function and flow for the edge in $\mathcal{N}^T/\Delta$ and $\mathcal{N}$ remain the same.

Theorem 2.4.1. *A solution f to the data transfer over time problem that finishes within time $T(1+\varepsilon)$ with a cost of C can be obtained from a flow x of the Δ-condensed network with cost C and time expansion $T(1+\varepsilon)$. When C is the minimum cost of the Δ-condensed network, it is also the minimum cost of the original flow over time with deadline T.*

Proof. We must show that (a) given a feasible solution to the flow over time problem $\mathcal{N}$ with cost C and time T, there exists a solution to the static Δ-condensed time-expanded network in Step 2 with cost C and time $T(1+\varepsilon)$; and (b) a flow x in the Δ-condensed network can be modified to a flow over time f with the same cost and time.

The conversions in Step 4 prove (b), by showing such a modification.

We now show (a). Because our flow over time network has non-negative costs, there exists an optimal flow f^* with no cycles. We show there exists a feasible flow $\hat{f}$ with cost at most C satisfying demands D by time $T^*(1+\varepsilon)$ in the network with transit times rounded up to the nearest multiple of $\Delta := \frac{\varepsilon}{n}T$; define a topological ordering $\{v_0, v_1, ..., v_{n-1}\}$ of V on the graph with edges in f^*, and for an edge $e = (v_i, v_j)$, set $\hat{f}_e(\theta) = f^*_e(\theta - i\Delta)$. The completion time of $\hat{f}$ is $T^* + \Delta(n-1) \leq T^*(1+\varepsilon)$, with same cost and demands (since flow travels on the same paths). Using storage, we make this a flow in the Δ-rounded network, by holding flow sent on e at v_j for an additional $\Delta(j-i) - \tau'_e \geq 0$ units of time. □

Compacting flow in holdover edges: Using Δ-condensed networks, it is possible that the finish time will be beyond the original deadline. The MIP treats each solution within the extended deadline as equally good, even if there is unnecessary storage of flow that increases the finish time i.e., when the final byte of data enters the sink. We compact the storage of flow, by adding a negligible amount of cost to all the holdover edges, except for those at the sink (e.g, 0.0001 \$/GB in our experiments). Intuitively, at each time step that flow is stored at a vertex, if sending that flow to the sink does not worsen the solution, the cost at holdover edges causes this option to be chosen. While compacting the transfers does not guarantee that the deadline is met, it does cause the finish time to be as soon as possible given the sequence of transfers. We use compaction in all of our Δ-condensed network experiments.

Using partial results

In all of our previous discussion, we assumed that all computations must be run to completion. However, when the computation time required to find the optimal solution is too long, Pandora can stop the computation and produce a partial result. This can be done by outputting the best integer solution found so far, which the branch and bound method keeps track of. This forcefully reduces the number of subproblems solved. Because a solution can be produced at any time, this optimization can give Pandora great flexibility when solving large problems. In our experiments, we find that the partial result solution can be quite good if given even a fraction of the time needed to run the computation to completion.

2.5 Experimental Results

We implemented Pandora and ran trace-driven experiments to evaluate the system. We present two sets of experimental results. We first show in Section 2.5.1 the benefit of using Pandora's flexible graph formulation that produces a cooperative data transfer plan. In Section 2.5.2 and 2.5.3, we present

Index	Site	BW	Index	Site	BW
Sink	uiuc.edu	-	5	rochester.edu	6.9
1	duke.edu	64.4	6	stanford.edu	5.3
2	unm.edu	82.9	7	wustl.edu	2.0
3	utk.edu	6.2	8	ku.edu	6.4
4	ksu.edu	65.0	9	berkeley.edu	7.1

Table 2.1: Sites used in experiments. BW is the measured available bandwidth (Mbps) to the Sink.

microbenchmarks to evaluate the optimizations made to the MIP formulation.

We evaluated Pandora using trace-driven experiments. The basic topology we used had a single sink at uiuc.edu and 9 additional sites (sometimes used as sources) at .edu domains as listed in Table 2.1. The Internet bandwidth between the sites was derived from PlanetLab available bandwidth traces measured using the Spruce measurement tool [122] by the Scalable Sensing Service (S^3) [135] (at 12:32 pm on Nov 15, 2009).

We obtained real shipping cost and time data between all sites by using FedEx SOAP (Simple Object Access Protocol) web services [14], with site addresses provided by a whois lookup to the domains. For service charges at the sink, we used Amazon AWS's published costs.

We solve the MIP formulation of the static problem using the GNU Linear Programming Kit (GLPK) [16]. Except when explicitly mentioned, GLPK's default branch and bound parameters were used. The execution times were taken on a machine with 4GB RAM and two Quad-core 1.86 Ghz Intel Xeon processors; GLPK only used one core.

2.5.1 Pandora Transfer Plans

To evaluate the quality of Pandora transfer plans, we ran Pandora on the topology of 10 PlanetLab sites. For the experiment labeled Source $1-i$, the sites used as source nodes are 1 through i. The total dataset was fixed at 2 TB. The data was spread uniformly over the source nodes. The results are shown in terms of time and cost in Figures 2.8 and 2.9.

We compare Pandora's transfer plan to two baseline plans that make independent choices at each node. In the first, called "Direct Internet," all the sites send their data to the sink over the Internet. This incurs a total cost

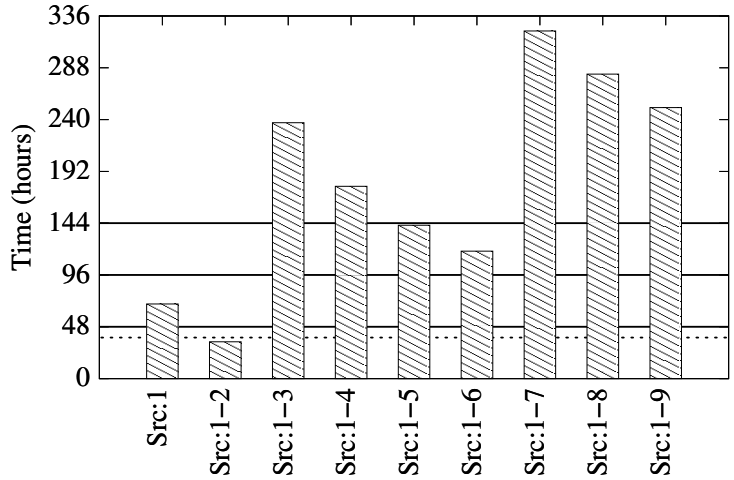


Figure 2.8: Time required for Direct Internet transfers for each experiment. Lines are for comparison with Direct Overnight shipment (38) and Pandora deadline experiments (48, 96, 144).

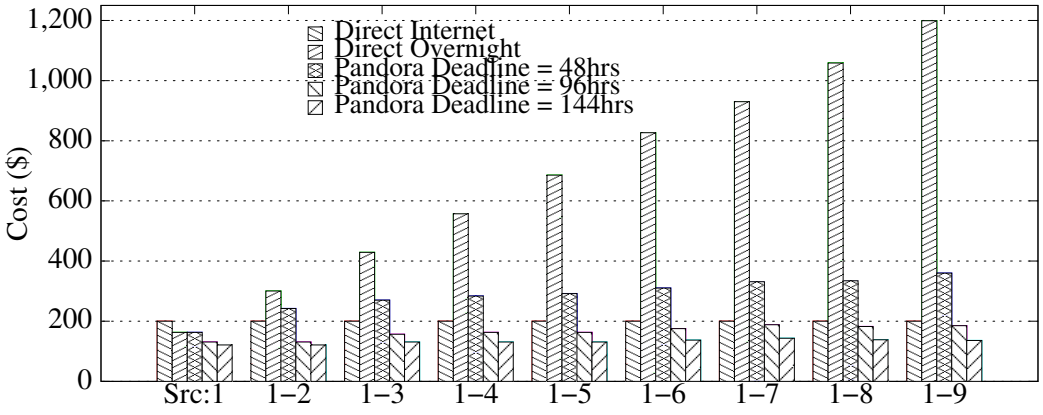


Figure 2.9: Cost comparison of transfer plans. Pandora produces low-cost solutions under different conditions and deadlines. Direct transfer approaches are inflexible, and do not adapt well to data being transferred from many locations.

Deadline (hrs)	Cost ($)	to Sink (MB)		to Intermediate (MB)	
		Internet	Shipping	Internet	Shipping
48	334.33	1747043	252957	327360	136192
96	182.57	75560	1924440	1530440	144000
144	138.26	87794	1912206	1688333	0
240	121.25	0	2000000	1750000	0

Table 2.2: Scheduled data transfers in Pandora solutions for Source 1-8 settings under different time deadlines. Direct or indirect, and Internet or shipping transfers are used in various combinations according to time deadlines to find the solution with lowest cost.

of $200 for the total data regardless of the number of sources. To calculate the time required, we assume optimistically that there is no data bottleneck at the sink. Thus, the time required is equal to the amount of data at the slowest source, divided by the available bandwidth to the sink, as shown in Figure 2.8.

The second baseline plan, called "Direct Overnight," ships a disk overnight immediately from each source site. While this gives a very fast transfer time of 38 hours, the price of transfer grows increasingly with the number of sources, as shown in Figure 2.9. This is because the cost of sending a disk is incurred at each source.

Pandora differs from the above two approaches. The flexibility of using both Internet and shipping data transfer gives Pandora a wide range of possible plans to choose from. We compare results for deadlines of 48, 96, and 144 hours against the direct approaches in Figures 2.8 and 2.9.

We compare a few of the results given 9 source sites. The 48 hour deadline is within 25% of the fastest but most expensive direct shipping times. When given this deadline, Pandora creates a solution that costs 70% less than that direct shipping cost. A relaxed deadline of 96 hours is less than 40% of the time a transfer would take via direct Internet. Under this deadline, Pandora finds a solution that has the added benefit of costing 10% less than direct Internet transfer. Relaxing the deadline to 144 hours gives us a 30% cheaper alternative to direct Internet transfer.

The above results show that Pandora adapts to different constraints in the transfer network to produce good solutions. To give a sense of this flexibility, we present more details in Table 2.2 and Figure 2.10 of a small set of solutions. The optimal transfer plans produced by Pandora vary in their use of direct or indirect, and Internet or shipping transfers. In Table 2.2, we total the

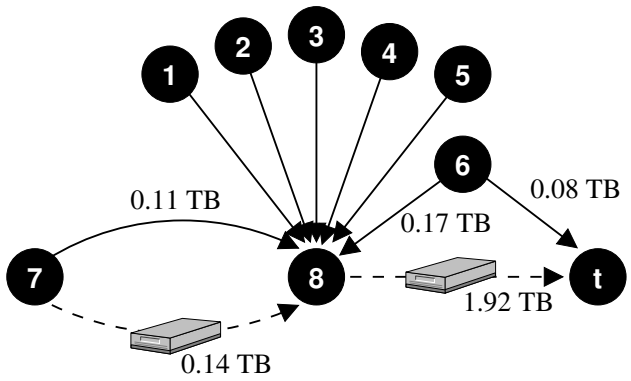


Figure 2.10: Pandora solution for deadline 96 hours in Source 1-8 settings. The illustration depicts the second solution in Table 2.2, using indices corresponding to Table 2.1. Dashed edges indicate disk shipment, and solid edges indicate Internet transfer. 0.25 TB is transferred on each unlabeled edge.

number of bytes that were sent from a site to the sink ("to Sink") through Internet and shipping, and do the same with those bytes that are sent from one source to another ("to Intermediate"). Figure 2.10 depicts in more detail the links used in the solution with deadline 96 hours.

In Figure 2.10, Source 8 aggregates data from sources 1-6 via the Internet, and sends it via disk to the sink. For the solutions in Table 2.2, we observe a similar pattern. Data appears to be aggregated at a single source site. A single 2TB disk is sent from this site to the sink. Other sources choose to relay their data to this source through the Internet because it allows them to amortize the fixed cost of shipping and handling that the single disk shipment incurs.

With a tight deadline however, not all data can be aggregated via Internet before it is sent to the sink. In Figure 2.10, this is the case for Sources 6 and 7. Source 6 cannot send all its data to Source 8 before the deadline, so it sends data through Internet transfer directly to the sink. This kind of transfer is seen for other solutions with tight deadlines (see the "to Sink"-Internet column in Table 2.2). Alternatively, disk shipment can be used as a fast but costly way to aggregate data when Internet bandwidth is limited to the aggregating site and sink. For example, Source 7 had to send data through both Internet and shipment links to meet the deadline. This behavior is also seen in solutions with tight deadlines (see the "to Intermediate"-Shipping column). With less restrictive deadlines, these two kinds of transfers are avoided, and more free Internet links between sources are used. Thus cheaper solutions are found.

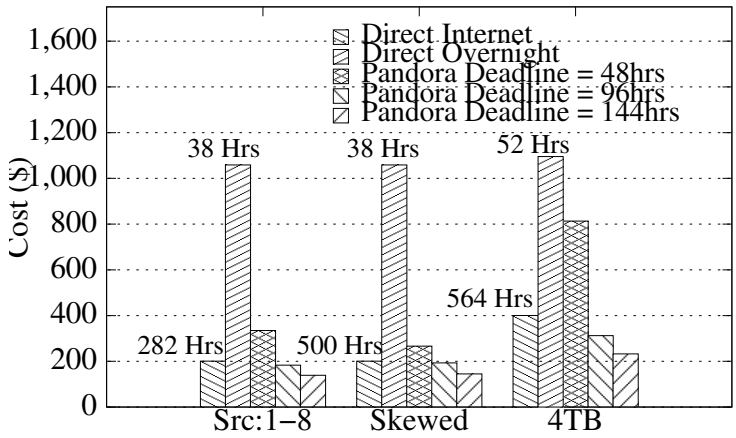
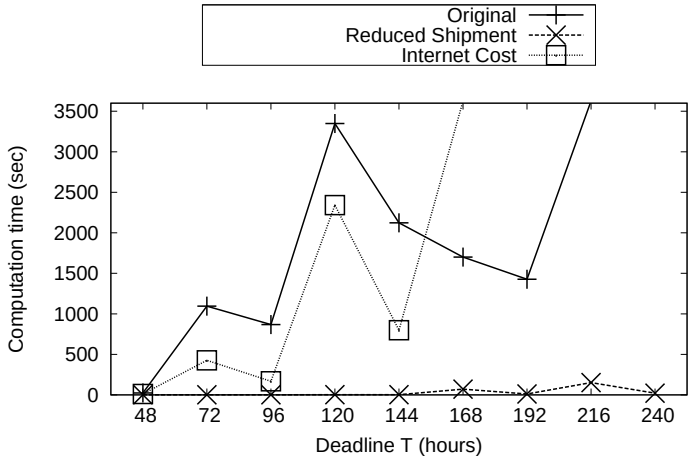


Figure 2.11: Cost comparison of solutions given different data placement and sizes. The time required is also shown for direct transfer plans.

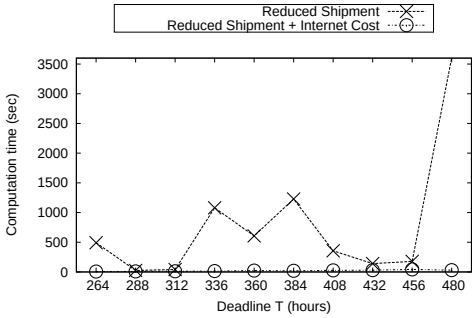
Pandora's flexibility also helps it find good solutions given different data distributions. In Figure 2.11, we look at two data distributions in the Source 1-8 setting, that differ from the uniform, 2 TB distribution. In the skewed distribution, 2 TBs are spread in a 9:1 ratio between sources 1,3,5,7 – with 0.45 TB each – and sources 2,4,6,8 – with 0.05 TB each. As expected, the direct Internet and disk transfer strategies do not adjust to the skewed distribution. In contrast, the Pandora solutions take advantage of the placement of data. This results in the 48 hour deadline solution having a reduced cost. We reason that Pandora remains flexible, even with skew in data distribution.

In the next distribution, the data is increased to 4 TBs, placed uniformly as before. In this case, the direct Internet solution increases in cost and time by a factor of two. The direct shipment strategy, by contrast, does not increase much with the data size. The extra data amortizes the shipment costs, while only increasing the time by a small amount. In other words, the disks used in Direct Overnight transfer are more fully utilized. Thus, when given a tight constraint of 48 hours, Pandora uses many of the same shipment links, and the cost incurred is relatively close to Direct Overnight. However, even in this case, we see two advantages of using Pandora. The time is reduced, even compared to the expensive direct shipment strategy, by making use of Internet links. At the same time, the Pandora solution also reduces the dollar cost of the solution, by making use of cooperation between sites. This shows that Pandora is useful with tight time deadlines and can scale with the amount of data.

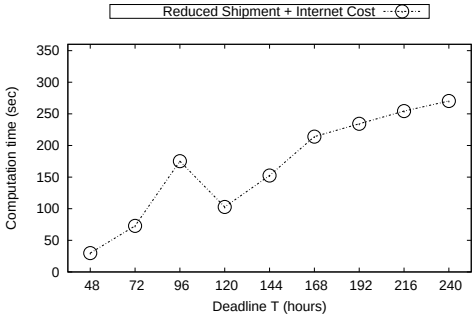
In summary, we conclude that the results show Pandora can significantly



(a) Computation times for the original MIP formulation, reduced shipment optimization, and Internet cost addition optimization, under Source 1-2 settings.



(b) Computation times when only the reduced shipment optimization is applied, and when both the reduced shipment and Internet cost addition optimization are applied together, under Source 1-2 settings with relatively large T.



(c) Computation times when both reduced shipment and Internet cost addition optimizations are applied, under Source 1-9 settings. The y-axis is scaled to $1/10^{\text{th}}$ of previous experiments.

Figure 2.12: Reduced shipment and Internet cost optimization microbenchmarks.

improve bulk data transfer on many different realistic topologies and data distributions.

2.5.2 Pandora Optimization Microbenchmarks – Optimal Solution Techniques

We now investigate the computation time taken to create Pandora solutions. Here, the critical metric is computation time that needs to remain low, so the transfer plans can be executed as soon as possible.

We begin with the optimizations that maintain an optimal solution. For

each optimization, we first look at the changes in computation time. We then discuss how each optimization affects computation time. As discussed in Section 2.4, these changes can be traced to either a reduction in the number of subproblems solved, or the time needed per LP.

Reducing shipment links and adding negligible amounts of cost to Internet links

In Figure 2.12a, we show the computation time taken to solve the problem in the experiment with Sources 1 and 2. The computation times show a general trend of increasing as the deadline increases. This is expected because a deadline increase means the MIP problem size increases. The original MIP computation time increases to above an hour once the deadline is set to be beyond 220 hours. Using our shipment link reduction optimization ("Reduced Shipment"; presented in Section 2.4.1), the computation time decreases by a large amount, staying below 3 minutes for deadline up to 240 hours.

The results for adding costs on Internet links ("Internet Cost"; presented in Section 2.4.1) are mixed. Below a deadline of 150 hours, the computation time is reduced from the original problem, however beyond 150 hours the computation times increase to over an hour.

Figure 2.12b shows the computation time for the same experiment at larger deadlines. We see that the Reduced Shipment optimization keeps the computation time at a reasonable level. In addition, we apply the Internet Costs to this already reduced MIP and find that the computation time is reduced substantially, and remains below 10 seconds. We show that the combined Reduced Shipment and Internet Cost optimization continues to do well with larger problems in Figure 2.12c. In the Source 1-9 setting with 9 sources, the computation time remains fast and stays below 300 seconds with a deadline of up to 240 hours.

As discussed in Section 2.4, the improvement in computation time can be traced to either a reduction in the number of subproblems solved, or the time needed per LP. In Figure 2.13, we show how each optimization affects the number of subproblems solved and the time needed to solve each subprob-

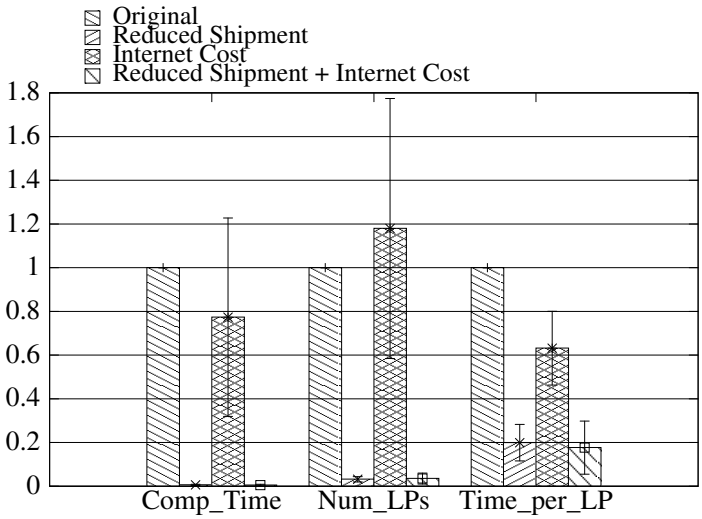


Figure 2.13: Change in number of subproblems solved and LP computation time per subproblem for reduced shipment and Internet cost addition optimizations, against the original MIP formulation. The gain in computation time for reduced shipment is due to reduction of the number of subproblems, while the gain in computation time for Internet cost is due to the reduction in LP computation time.

lem's LP. We average the data points in Figure 2.12a across the different deadlines, and normalize them in terms of the original MIP computation time. The same is done for the number of subproblem LPs solved, and the average computation time for solving each LP.

From Figure 2.13, in the Reduced Shipment optimization, the number of binary variables are reduced. This cuts down the total number of subproblems, and thus cuts down on the number of subproblems for which LPs must be solved. Also, each LP itself has a smaller number of variables, reducing the time required to compute each LP. For the added Internet Costs optimization, the time required to compute each LP is reduced, showing that the optimization successfully hints at the LP solution for each subproblem. In contrast, the number of subproblems solved actually increases, although there is a large variance. This uncertainty is reflected in Figure 2.12a, where it is difficult to determine if Internet Cost optimization improves over the Original MIP. However, from Figure 2.12b we can conclude that adding Internet Cost to Reduced Shipment does, in fact, significantly improve the computation time.

Using a lower-bound solution

An optimization that further reduces the computation time is the use of the lower bound solution ("Lower Bound Branching"; presented in Section 2.4.1). We combine this optimization with the previous two optimizations. Figure 2.14a shows that adding the Lower Bound Branching optimization, the computation time is reduced in many cases. The improvement is clearer for deadlines that are 240 hours or longer.

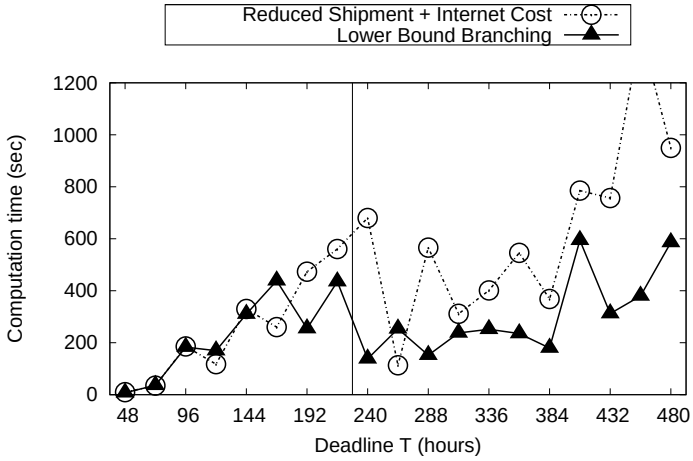
As Figure 2.14b shows, the reason for the improvement is because the number of subproblems to be computed is reduced when a lower bound solution is leveraged. Again, the behavior is more pronounced when considering the deadlines that are 240 hours or longer. At 240 hours and up, the lower bound solution and the eventual optimal solution converge to the same objective value. Therefore, the number of subproblems that need to be computed is reduced. As soon as the optimal solution is found, the branch and bound method can prune all remaining subproblems, effectively completing the computation. Using the lower-bound solution retards the growth of the computation time with respect to the deadline. Thus, given a loose deadline constraint, we can find the solution within a reasonable time.

2.5.3 Pandora Optimization Microbenchmarks – Approximate Solution Techniques

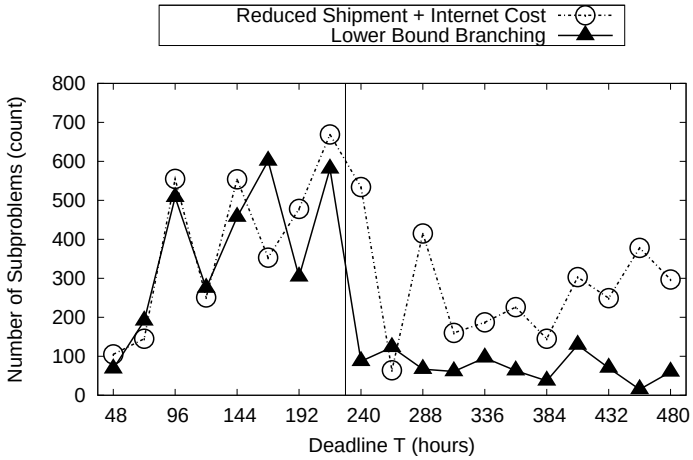
Δ-condensed time-expanded network

We next turn our attention to Δ-condensed networks (Section 2.4.2). Figure 2.15a compares the Δ-condensed MIP to the original MIP solution. As expected, the Δ-condensed MIP has a faster running time than the original solution.

Given these results, we expected to get even greater gains by combining Δ-condensed optimization with the reduced shipment optimization. However, this turns out not to be the case, as we see in Figure 2.15b. We believe this can be explained by looking at the structure of the networks. Applying Δ-condensing on a network that has already reduced shipment edges to a minimum will not reduce shipment edges. In fact, by extending the time expansion to $T(1+\varepsilon)$ on the network, we may add a number of shipment



(a) Computation times for reduced shipment and Internet cost optimizations only, and when they are applied together with the lower bound optimization, under Source 1-8 settings. For deadlines after the vertical line, the solutions to the lower bound and original MIP are identical.

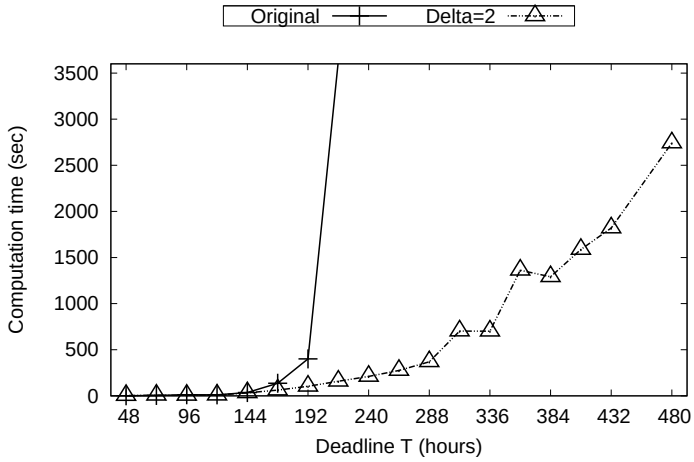


(b) Number of subproblems for reduced shipment and Internet cost optimizations only, and when they are applied together with the lower bound optimization, under Source 1-8 settings. For deadlines after the vertical line, the solutions to the lower bound and original MIP are identical.

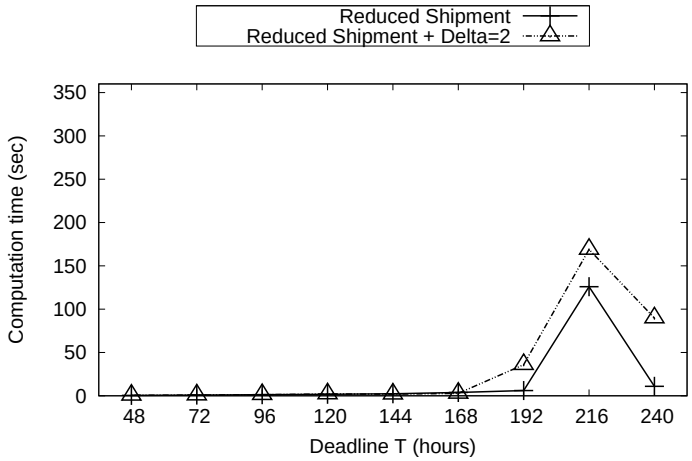
Figure 2.14: Lower bound optimization microbenchmarks

Deadline	Solution
48 hrs	43 hrs
72 hrs	55 hrs
96 hrs	61 hrs
120 hrs	78 hrs
144 hrs	85 hrs

Table 2.3: The deadline and finish time of the transfer plan solution, given $\Delta = 2$ in the Sources 1-2 setting. In our results the Δ solutions managed to stay within the deadline.

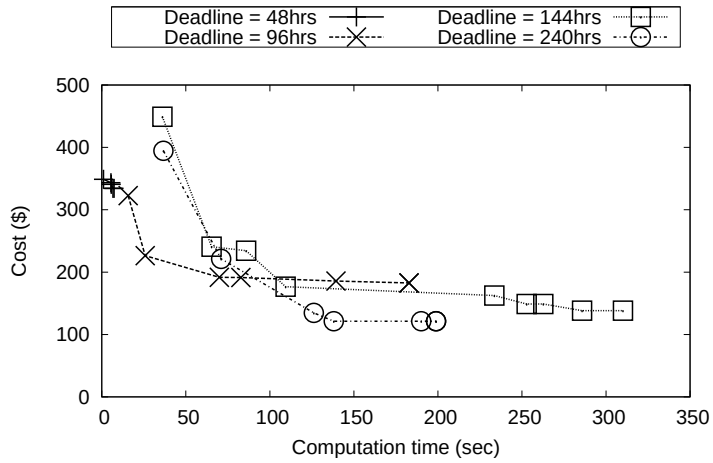


(a) Computation times of original MIP and Δ-condensed MIP with Δ=2 using Source 1 settings.

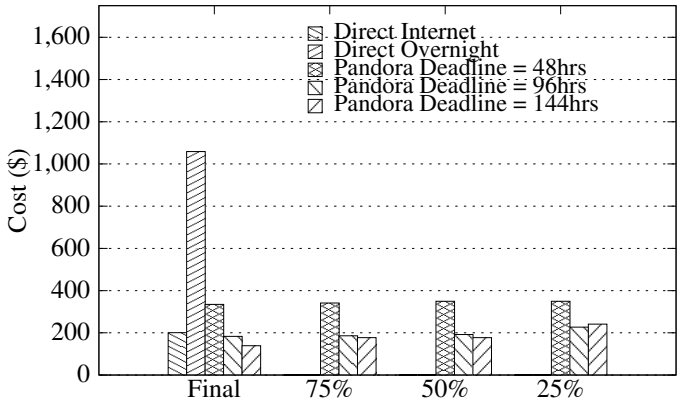


(b) Computation times of reduced shipment optimization and, when the optimized MIP is additionally Δ-condensed, using Source 1 settings. The y-axis is scaled to $1/10^{\text{th}}$ of previous experiment.

Figure 2.15: Δ-condensed network microbenchmarks



(a) Progression of the best candidate solution during the branch and bound algorithm. Source 1-8 settings are used.



(b) Quality of the best candidate solution when computation time is 75%, 50% and 25% complete. Source 1-8 settings are used.

Figure 2.16: Partial result optimization microbenchmarks

edges, and thus integer variables.

Finally, we present in Table 2.3 the finish time of the solutions for Δ-condensed MIPs. These experiments were run using the compaction of flow in holdover edges (Section 2.4.2). We were able to meet all time T deadlines, even though the worst case time is $T(1+\varepsilon)$.

Using partial results

Finally, we look at the effectiveness of using partial results of the MIP computation as a transfer plan. We first plot the progression of known solutions

found by the branch and bound method in Figure 2.16a. After an initial solution is found, solutions with a much lower cost are found in a relatively short time span. The gains made by continuing the computation then level off significantly. We observed that good solutions were found even when only a small portion of subproblems in the branch and bound are solved. In Figure 2.16b we compare the results of Pandora when the MIP computation is allowed to run up to 75%, 50%, and 25% of the total computation time. We observe that the solutions are still close to the optimal solution, and much better than direct transfer strategies. This suggests that even when Pandora must use a partial result for very large problems, the resulting solution can be better than naive solutions and close to the optimal.

2.6 Related Work

There is a long list of success stories running ad-hoc computations in industry clouds [7]. Meanwhile, work is under way to allow better academic research on the cloud. In the scientific domain, one of the main obstacles is that data sets are very large and distributed across many organizations [114, 133]. Pandora can alleviate the long latency and large cost of transferring this data.

Previous work has looked at bulk network data transfers in the Grid [18], and on PlanetLab [29]. Researchers have found that the use of intermediate nodes can help transfer bulk data on these networks [89, 112]. Yet, the scale of cloud data in the TBs challenges researchers to look for new approaches.

Likewise, the shipping of physical media for data transfer is not new. Jim Gray [79], the PostManet project [132], DOT [128] and others, have explored the feasibility of writing datasets onto a storage drive and shipping it as a means of data transfer. Amazon AWS [6] gives end users the option to use disk shipments to upload data, using the Import/Export service.

At the same time, the economic and performance trade-offs of cloud services, and data transfer in particular, has been gaining interest in the research community [44, 73]. Our work is the first to combine both data networks and shipping physical media, and to devise a single transfer plan that is optimal in respect to economic and performance goals.

Many algorithms for network flows over time using time-expanded net-

works have been studied since the seminal work of Ford and Fulkerson [71]. [70] introduces Δ-condensed networks that can be used to approximate time-expanded networks in polynomial time. Our work builds on the theoretical literature by applying it to a novel domain. At the same time, we solve a variation of the problem that has not been previously studied.

2.7 Summary

In this chapter, we have presented and evaluated a solution to the group-based data transfer problem where multiple source sites wish to send disjoint datasets to a sink site. Our solution, Pandora, is the first ever to account for both Internet as well as shipping links. Our algorithms satisfy deadlines while simultaneously minimizing dollar costs, using these diverse links. As a result, the transfer planning algorithms outperform both Internet-only and shipping-only transfers. Our experimental results involving PlanetLab traces and real data from FedEx show these benefits. In our experiments, Pandora meets a deadline that is less than 40% of the time a transfer would take via direct Internet, while costing more than 10% less. Similarly, by setting a deadline within 25% of the fastest but most expensive direct shipping times, Pandora creates a solution that costs 70% less than the direct shipping cost. We also found that we can reduce computation time of Pandora solutions significantly by using specific characteristics of the data transfer networks to improve the branch and bound search.

Chapter 3

Pandora-B: Budget-constrained Bulk Data Transfer via Internet and Shipping Networks

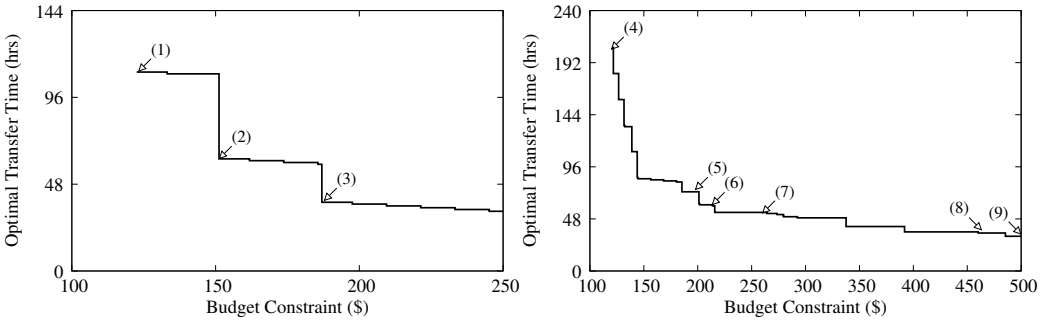
In this chapter, we discuss the budget-constrained bulk data transfer problem. Our solution, Pandora-B, finds the optimal budget-constrained solution by searching among deadline-constrained solutions, that were solved in the previous chapter. We bound the search space, allowing Pandora to quickly converge to good and optimal solutions.

3.1 Motivation

In this chapter, we solve the new problem of finding the fastest transfer plan given a strict constraint on the dollar cost budget. This *budget-constrained* problem requires new solution techniques that are explored in-depth in this chapter.

For instance, finding a fast transfer time within a given budget is important for research collaborations. Consider a researcher who would like to use sets of geological data collected at distributed sites. She would like to get the data transferred to a cloud service provider (data center) to run computations as quickly as possible. On the other hand, she has a limited amount of money to spend on the data transfer, e.g. she may be working under a grant with limited funds. By solving the budget-constrained transfer problem, Pandora can help the researcher get timely results with a limited budget. Similar examples apply to other cloud users such as consultants, freelancers, newsroom staff, etc. In our algorithms, we solve for the *optimal* transfer plan that meets the constraints while minimizing total transfer latency, so that cloud users can make the most of their funds.

Our algorithms for finding an optimal solution must contend with several major challenges. First, there are many different transfer strategies, and therefore the solution space is massive. Second, each transfer option has



(a) (b)

	$	Hrs	Shipment Links		TB
1	120	109	unm.edu→uiuc.edu	Ground	2.00
2	150	59	unm.edu→uiuc.edu	Two-Day	2.00
3	185	31	unm.edu→uiuc.edu	Overnight	2.00
4	120	206	indiana.edu→uiuc.edu	Ground	2.00
5	200	73	unc.edu→uiuc.edu	Overnight	1.88
6	215	60	wustl.edu→indiana.edu	Two-Day	0.09
			unc.edu→uiuc.edu	Overnight	1.73
7	260	54	wustl.edu→uiuc.edu	Two-Day	0.13
8	460	36	wustl.edu→uiuc.edu	Overnight	0.19
			unc.edu→uiuc.edu	Overnight	0.50
9	500	32	indiana.edu→uiuc.edu	Overnight	0.20
			wustl.edu→uiuc.edu	Overnight	0.19
			unc.edu→uiuc.edu	Overnight	0.19

(c)

Figure 3.1: Optimal transfer times given budget constraints. Figure (a) shows the transfer of 2 TB from site 1 in Table 3.1, while (b) shows the same data sent from sites 1-11. Table (c) summarizes the disk shipments involved for points labeled in (a) and (b). Internet transfers are not shown.

different characteristics: an Internet link between a pair of sites has a unique *bandwidth*; further there are several shipment options between a pair of sites (e.g. Ground, Two-Day, Overnight), each with its own *transit time* and *cost* depending on the geographic location of the sites. Third, each site also has characteristics: the time it takes to unpack, plug in, and transfer data from a disk varies by location. If the site is a service provider, charges may apply to incoming Internet bandwidth and disk handling. Fourth, the *scheduled time* when a transfer should begin is important. Finally, we observed that the optimal solution strategy is highly sensitive to the budget constraint. The best transfer plan may look very different for a transfer budget that is slightly smaller or slightly larger.

Figure 3.1 shows an example that highlights these challenges. We plot the optimal transfer time as a function of the specified budget constraint, for two different transfer scenarios. The sites for these scenarios are shown in

Table 3.1 (with more details in Section 3.4). In Figure 3.1a we transfer 2 TB of data from a single source at unm.edu, to a single sink at uiuc.edu, while in Figure 3.1b we transfer 2 TB of data from eleven different sources (sites 1 through 11 in Table 3.1), each with 0.18 TB. The two plots show that the optimal transfer time differs widely between different budget constraints. We highlight some of the points in these plots and summarize their use of disk shipment in Figure 3.1c.

Focusing on a single source and sink, Figure 3.1a shows the general tradeoffs between shipment options. Faster shipment options can result in a faster transfer time, but require a larger cost budget. This tradeoff affects the shape of the optimal budget-constrained solution curve. The solutions can be divided into three sections by the points (1), (2), and (3). As the budget constraint is increased, these solution points show, respectively, the cut-off when the budget is sufficient to complete the transfer via Ground shipment (from (1) to (2)), a faster Two-Day shipment (from (2) to (3)), and the fastest Overnight shipment ((3) and beyond). Within each section, the transfer time continues to fall as the budget is increased, and more and more money is spent sending data on Internet links, in parallel to the disk shipments.

Figure 3.1b shows optimal transfer solutions for a network with many sites. The transfer options used in Figure 3.1c shows a sliver of the massive solution space. Deciding which of the diverse transfer options to use is sensitive to the budget constraint and has a critical impact on the shipment time. For example, point (5) and (7) only have a difference of $60 in transfer budget, yet the shipment time is reduced by 19 hours. The strategies differ in many ways – not only are the shipment options used different, Overnight in (5) and Two-Day in (7), but also the size of data in the disks, 1.88 GB and 0.13 GB respectively, and even the where the shipment originates, unc.edu and wustl.edu.

Before we discuss our solutions, we would like to clarify a couple of points. First, our back-of-the-envelope calculations for the CCT testbed [25] show that only one to two disk sizes are preferable for ease of management (e.g. to enable hot-swapping disks into CCT). Since the focus of this chapter is on solving the planning problem, a further discussion of the disk population is beyond our scope. Secondly, we aim to derive optimal solutions no matter what the setting. Due to the heterogeneity of dataset distribution and bandwidth across sites (as seen in Table 3.1), degenerate solutions (e.g. ship all

or transfer all over Internet) would be sub-optimal.

3.2 Problem Formulation

Section 2.2 described how we model problem inputs as a concise graph representation. We use the same graph representation which is formalized as a data transfer over time problem, for the budget-constrained solutions in this chapter.

3.3 Budget-constrained Solution

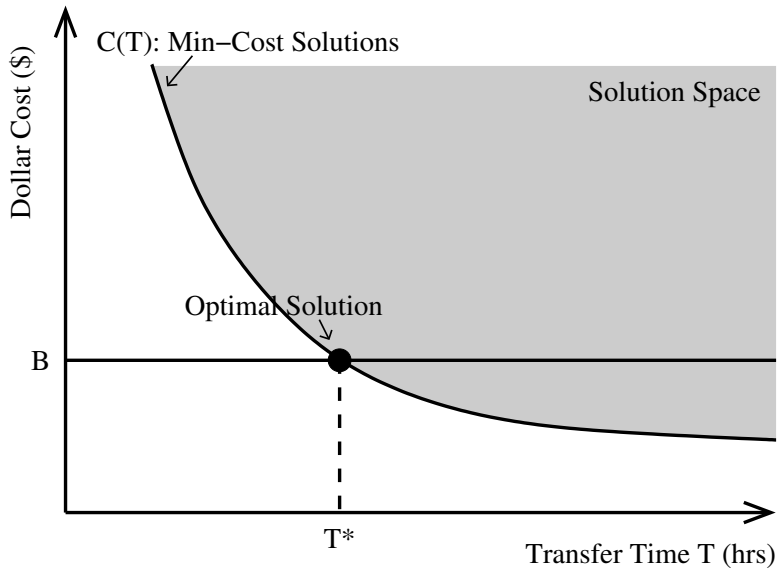


Figure 3.2: A sketch of the solution space according to transfer time and dollar cost. The optimal solution satisfies the budget constraint B in the shortest possible transfer time T^*.

From the solution space of data transfer plans, we are looking for a solution that minimizes transfer latency subject to a budget constraint. In Figure 3.2 we sketch the solution space according to the transfer time and dollar cost of the solutions. The entire solution space is colored in gray. The horizontal line shows the budget constraint B of the problem. The solutions we are interested in are in the area on or below the line. The optimal solution is precisely the solution with the smallest transfer time in this area. We denote the time of the optimal solution as T^*.

Algorithm 3.1 Binary Search Functions

```
// type = {orig, lb, ub}
function binary_findinterval(init_head, budget, type):
  head = tail = init_head
  while cost > budget do
    cost, endtime = solve_mincost(deadline, type)
    head = tail
    tail = deadline
    deadline = deadline * 2
  end while
  return head, tail
function binary_search(head, tail, budget, type):
  while head < tail do
    midpoint = ⌊(head+tail)/2⌋
    cost, endtime = solve_mincost(midpoint, type)
    if cost > budget then
      head = midpoint+1
    else
      tail = endtime
    end if
  end while
  return head, tail
```

Algorithm 3.2 Two-Step Min-Cost Binary Search

```
1: function twostep_mincost_binary_search(budget):
2:   head, tail = binary_getinterval(1, budget, orig)
3:   head, tail = binary_search(head, tail, budget, orig)
4:   return tail
```

Our problem is to find the optimal solution among the solution space. The strategy we adopt is to make use of the line in Figure 3.2 that separates the solution space from the non-solutions space. We denote as a function, $C(T)$, the minimum cost among all solutions with transfer time T. This allows us to leverage the algorithms in Chapter 2 to find the value $C(T)$ for any given T. Using these algorithms, one naive approach may be to compute $C(T)$ for each value of $T = 1, 2, ...$ until we find the smallest value T such that $C(T) \leq B$. In fact, this strategy produces the optimal solution at T^*. However, this approach can be prohibitively expensive. This motivates our development of faster search strategies, which we do next.

3.3.1 Two-Step Binary Search using Deadline-Constrained Minimum Cost Solutions

The function $C(T)$ is monotonically decreasing. This can be seen by con-

sidering that the solution $C(T)$ can be replicated at time $T+1$, so the value of $C(T+1)$ by definition will be at most $C(T)$. We make use of this property to create an efficient *binary search algorithm* on $C(T)$ to find the optimal solution.

This binary search is illustrated in Algorithm 3.1 and 3.2. It runs in two main steps defined in line 2 and 3 in Algorithm 3.2. The first part of our search finds both an upper bound and lower bound on the optimal transfer time T^*. It does so by computing values of $C(T)$ for *exponentially increasing values* of T until a solution that meets the budget constraint is found. The first value of T that meets the budget constraint is used as the upper bound, while the immediately previous tried value of T is the lower bound. Notice that this requires computing exactly $\lceil logT^* \rceil + 1$ deadline-constrained minimum cost solutions.

Thereafter, a binary search is performed in the interval between the lower bound and upper bound. After each iteration of the search, the search interval for the next iteration is selected depending on the value of $C(T)$: the upper half of the current interval is selected when $C(T)$ is greater than B, and then lower half is selected otherwise.

The binary search step can add, at most, $\lceil logT^* \rceil$ deadline-constrained minimum cost computations. Thus, the total number of minimum cost computations required is only $2\lceil logT^* \rceil + 1$ in the worst case.

However, while the *number* of computations grows only logarithmically with the optimal transfer time T^*, the actual *time* of computation grows at a much higher rate with T^*. This is because the time required to compute each constrained minimum cost solution grows with the size of the problem which is determined by the deadline time T (see Section 2.3.2). Thus performing a search by solving many of these minimum cost computations can become expensive.

3.3.2 Bounded Binary Search using Strong Lower and Upper Bounds

We can reduce the computation time of binary search if we reduce the number of computations of the function $C(T)$, especially when T becomes large. In this section, we show how to reduce the binary search interval around T^*,

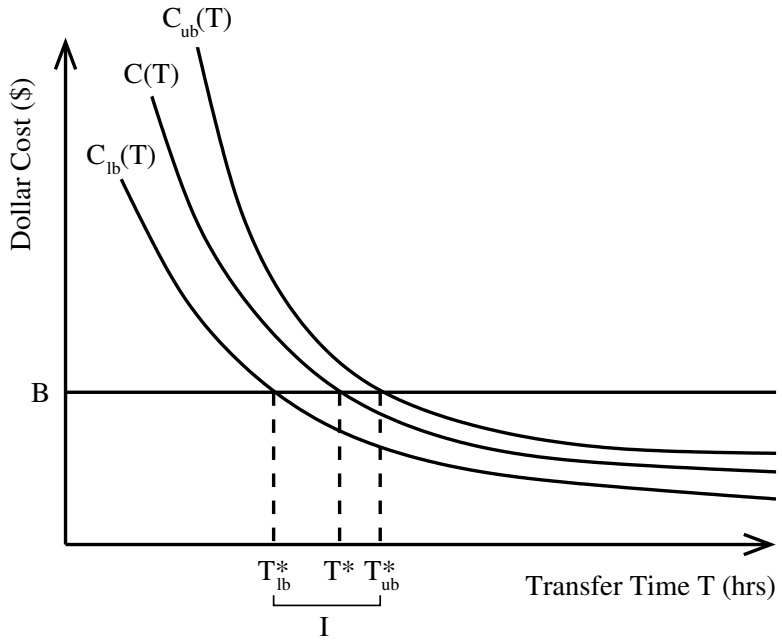


Figure 3.3: $C_{ub}(T)$, $C_{lb}(T)$, and $C(T)$ minimum cost solution curves.

Algorithm 3.3 Bounded Min-Cost Binary Search

```
1: function bounded_mincost_binary_search(budget):
2:     head_ub, tail_ub = binary_getinterval(1, budget, ub)
3:     head_ub, tail_ub = binary_search(head_ub, tail_ub, budget, lb)
4:     head_lb, tail_lb = binary_search(1, tail_ub, budget, lb)
5:     head, tail = binary_search(tail_lb, tail_ub, budget, orig)
6:     return tail
```

which is when T becomes the largest. We accomplish this by introducing two forms of *bounding functions*, $C_{ub}(T)$ and $C_{lb}(T)$. These bound the original function from above and below, respectively. More concretely, these bounding functions obey the relationship $C_{lb}(T) \leq C(t) \leq C_{ub}(T)$. Like $C(T)$, they are monotonically decreasing functions of T.

Figure 3.3 sketches these bounding functions. We denote the time T where each bounding function intersects with the budget constraint B as T^*_{lb}, and T^*_{ub}. From the definition of the bounding functions, we have $T^*_{lb} \leq T^* \leq T^*_{ub}$. Thus, if we know the values of T^*_{ub} and T^*_{lb}, they define an interval $I = [T^*_{lb}, T^*_{ub}]$ where T^* must reside in.

We can build an efficient binary search framework around these bounding functions, as shown in Algorithm 3.3. This algorithm replaces the exponentially increasing search done directly on $C(T)$ in Algorithm 3.2 with a pair of searches for T^*_{ub} and T^*_{lb}. First, the value of T^*_{ub} is found in a similar way to

T^* in Algorithm 3.2. The exponentially increasing search done to find initial bounds for $C_{ub}(T)$ involves exactly $\lceil logT^*_{ub} \rceil + 1$ computations of C_{ub}. The binary search phase given the initial bounds takes at most $\lceil logT^*_{ub} \rceil$ computations of C_{ub}. Next, after we have found T^*_{ub}, we search for T^*_{lb}. We can skip the exponentially increasing search by using T^*_{ub} itself as the upper bound. The binary search phase involves at most $\lceil logT^*_{ub} \rceil + 1$ computations of C_{lb}. Finally, we perform a binary search for T^* on the interval $[T^*_{lb}, T^*_{ub}]$. This involves at most $\lceil log(T^*_{ub} - T^*_{lb}) \rceil + 1$ computations of C.

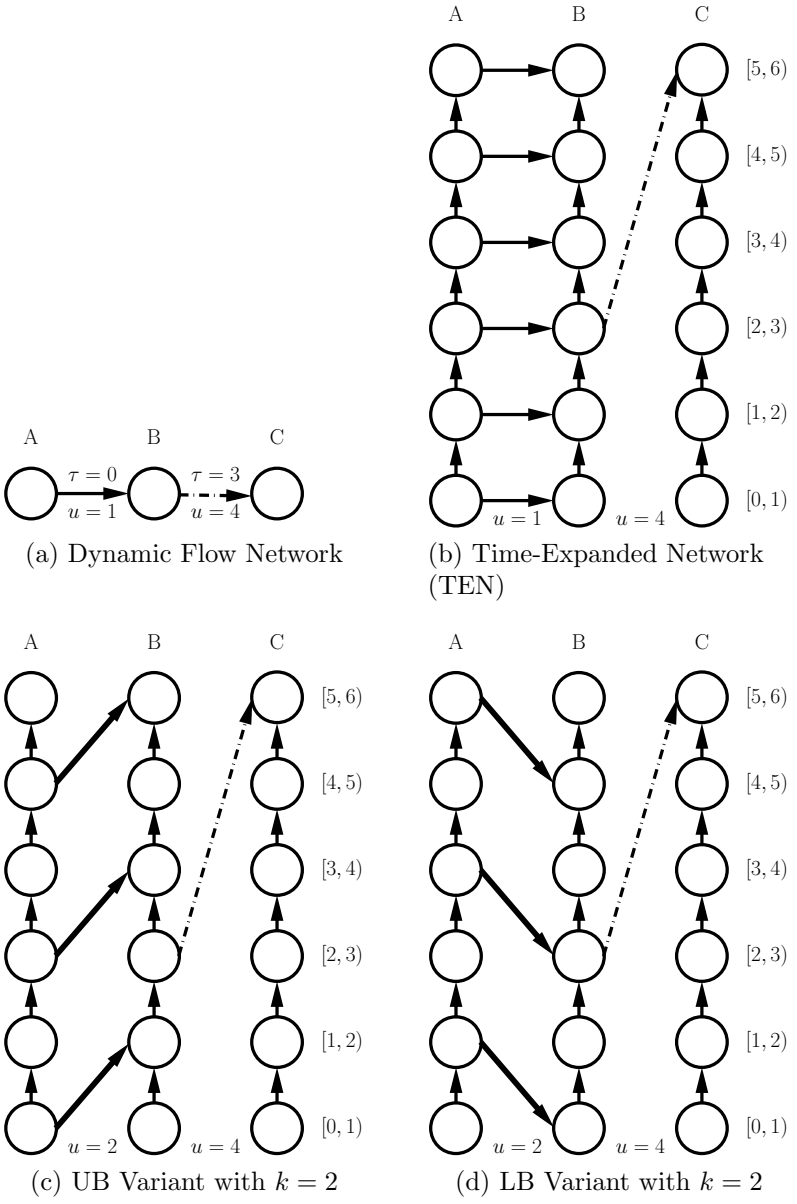
Using this strategy, the worst case number of deadline-constrained minimum cost computations required for the original function is changed from $2\lceil logT^* \rceil + 1$ to $\lceil log(T^*_{ub} - T^*_{lb}) \rceil + 1$. On the other hand, additional computations to solve bounding functions are required. Thus, for bounding functions to reduce the binary search computation time, they must be cheap to compute, and their values must not be too far from each other.

3.3.3 Lower-bound and Upper-bound Networks

If the bounding functions are cheap and tight, the bounded binary search solution approach could be very fast. In this section, we construct bounding functions for the deadline-constrained minimum cost problem. We prove that these functions strongly bound the original problem. Later, in Section 3.4 we show that these constructions are indeed cheap and tight.

Before we present the bounding formulations, we first review relevant parts of the time-expanded networks concept used to solve the minimum cost problem with a deadline T. More details are in Section 2.3.1.

In Figure 3.4a we show a data transfer network that contains a single Internet transfer link between node A and B that can transfer one unit of flow at each time unit, and a single shipping option between node B and C that can transfer four units of flow with a transit time of four time units. This network is expanded into a time-expanded network in Figure 3.4b, which explicitly represents the passing of time while transferring data in the network. The time-expanded network consists of *holdover edges* between every time point that represents the holding of data during that time (vertical edges), *Internet transfer edges* at every time point (solid horizontal edges), and *shipping transfer edges* at each relevant pick-up and delivery time (dashed slanted



(a) Dynamic Flow Network

(b) Time-Expanded Network (TEN)

(c) UB Variant with $k = 2$

(d) LB Variant with $k = 2$

Figure 3.4: An example dynamic flow network and the construction of its time-expanded network and variants.

edges). In this network, the capacity constraints allow a total of three units of data to be transferred from node A to node C during the transfer time of $T = 6$.

We produce *lower bound (LB)* and *upper bound (UB)* variants to the original time-expanded network (denoted as *TEN*) by combining Internet transfer edges at neighboring time points. We systematically group k successive Internet transfer edges into a single edge with a capacity k times the original. Combining edges makes the problem smaller, thus meeting the desirable property of cheap computation. We construct each of the variant networks in a way such that the solutions of a deadline T-constrained minimum cost solution on the variants obey the relationship $C_{lb}(T) \leq C(t) \leq C_{ub}(T)$. This makes them suitable as bounding functions. In addition, we also argue that the construction allows variant solutions to be close to the solutions of the original TEN network.

We show the construction of the bounding variants using the example in Figure 3.4. We first walk through the UB variant. Groups of k Internet edges are combined into a single edge. For example with $k = 2$, the edges $(A_{[0,1)}, B_{[0,1)})$ and $(A_{[1,2)}, B_{[1,2)})$ with capacity 1 are combined into the single edge $(A_{[0,1)}, B_{[1,2)})$ with capacity 2. The combined single edge faithfully represents the total amount of flow that can be sent during the entire k time steps. Intuitively, this is why the UB variant can act as a bound for the TEN solutions. This relationship is given formally in the following Theorem.

Theorem 3.3.1. *The UB variant is an upper bound to a TEN network. In other words, the relationship $C(T) \leq C_{ub}(T)$ holds for all T, where $C(T)$ is the T-deadline constrained minimum cost solution to the TEN network, and $C_{ub}(T)$ is the T-deadline constrained minimum cost solution to the UB network.*

Proof. The theorem is proved by showing that any solution on the UB network can be transformed into a solution in the TEN network with the same cost. Consider the Internet transfer edges $e(t)$ with capacity u_e and cost per flow c_e at each time point $t \in [0, T)$ in the TEN network. Each group with k of these edges with $t = [ak, a(k + 1))$ make up a UB edge $e_{ub}(a)$ with capacity $u_e k$ and cost per flow c_e. Consider a solution with flow through $e_{ub}(a)$ of size $F \leq u_e k$. We can convert this flow into the TEN network by assigning $f_e(t) = c_e$ to the edges with $t = [ak, ak + \lfloor F/k \rfloor]$, and then assigning any

remaining flow to $f_e(ak + \lfloor F/k \rfloor + 1)$. This obeys the capacity constraints of the TEN network edges, while maintaining the same total cost on these edges.

Thus, if we have the T-deadline constrained minimum cost solution to the UB network, $C_{ub}(T)$, we can transform this solution into a solution with the same cost in the TEN network. This solution must have a cost greater than or equal to $C(T)$, by the definition of $C(T)$ as the minimum cost with transfer time T. We have shown $C(T) \leq C_{ub}(T)$. □

On the other hand, the combined edge in the UB network misses opportunities for sending flow between some nodes. This is the tradeoff we pay for constructing a smaller network that is easier to compute. For example, $(A_{[2,3)}, B_{[3,4)})$ does not allow the flow across A and B at time $[2, 3)$ that the TEN network allows. Still, our technique of grouping edges in distinct groups of k mitigates the effect of missing opportunities by keeping them within a fixed time frame k. This is aimed at maintaining the UB solution close to that of the TEN network.

The LB variant is constructed in a similar way to the UB variant. However, in the case of the LB variant, the combined edge is drawn in the opposite direction of time. In the case of the LB variant, any flow in the TEN network can be faithfully represented by the combined edges of the LB network. This gives us:

Theorem 3.3.2. *The LB variant is a lower bound to a TEN network. In other words, The relationship $C_{lb}(T) \leq C(T)$ holds for all T, where $C_{lb}(T)$ is the T-deadline constrained minimum cost solution to the LB network, and $C(T)$ is the T-deadline constrained minimum cost solution to the TEN network.*

Proof. The theorem is proved similarly to the previous one. We show that any solution on the TEN network can be transformed into a solution in the LB network with the same cost. Consider the Internet transfer edges $e(t)$ with capacity u_e and cost per flow c_e at each time point $t \in [0, T)$ in the TEN network. Each group with k of these edges with $t = [ak, a(k+1))$ make up a LB edge $e_{lb}(a)$ with capacity $u_e k$ and cost per flow c_e. Consider a solution in the TEN network with flow through the edges in $t = [ak, a(k+1))$. The sum of

the flows is less than the combined capacities, i.e., $F = \sum_{t \in [ak, a(k+1))} f_e(t) \leq u_e k$. We can convert this solution into a LB solution by assigning in the LB network $f_e(a) = F$. This obeys the capacity constraints of the LB network edges, while maintaining the same total cost on these edges.

Thus, if we have the T-deadline constrained minimum cost solution to the TEN network, $C(T)$, we can transform this solution into a solution with the same cost in the LB network. This solution must have a cost greater than or equal to $C_{lb}(T)$, by the definition of $C_{lb}(T)$ as the minimum cost with transfer time T. We have shown $C_{lb}(T) \leq C(T)$. □

However, in the case of the LB variant, the combined edge can artificially model flow being sent backwards in time. In Figure 3.4d we can send two units of flow on the edge $(A_{[3,4)}, B_{[2,3)})$ while obeying the flow constraints in the LB network. Yet, this is physically impossible. We must be particularly careful that the LB variant solution does not stray far from the TEN solution because of these time-traveling edges. Thus, our construction uses the same global k value across all pairs of sites. This restricts the time that flow can travel back in time to only k units. For example, once a flow has reached the time $[3, 4)$ at any site, it cannot then travel back to a time before $[2, 3)$.

3.3.4 Using Partial Results

Our solution techniques are developed to produce optimal solutions. When planning a potentially long and expensive transfer, we believe that it is useful to solve for an optimal solution. However, users may still want the option to begin a transfer with sub-optimal results for transfer plans that take a very long time to compute. In this case, we can stop the binary search, and present the user with the solution with the shortest transfer time, among those that meet the budget constraint. In Algorithm 3.2, at least one constraint satisfying solution will be available when the exponential search stage is complete. In Algorithm 3.3, a constraint satisfying solution is also available after the first stage. This is due to the result of Theorem 3.3.1. A minimum cost solution to the UB variant can be converted to a constraint satisfying solution of the TEN network. In our experiments, we observe how fast partial results become available, and how close to optimal their transfer

times are.

3.4 Experimental Results

In this section, we use trace-driven experiments to evaluate Pandora's techniques presented in Section 3.3. We focus on computation time as the main metric. The value of optimal solutions solved by Pandora were presented in Figure 3.1 – in addition, we evaluate the quality of partial results.

3.4.1 Experimental Setup

Index	Site	BW	Index	Site	BW
Sink	uiuc.edu	-	6	mtu.edu	33.1
1	unm.edu	82.9	7	ufl.edu	7.6
2	unc.edu	78.5	8	rochester.edu	6.9
3	indiana.edu	73.8	9	umn.edu	5.9
4	utexas.edu	70.7	10	ncsu.edu	4.6
5	duke.edu	64.6	11	wustl.edu	2.0

Table 3.1: Sites used in experiments. BW is the measured available bandwidth (Mbps) to the Sink.

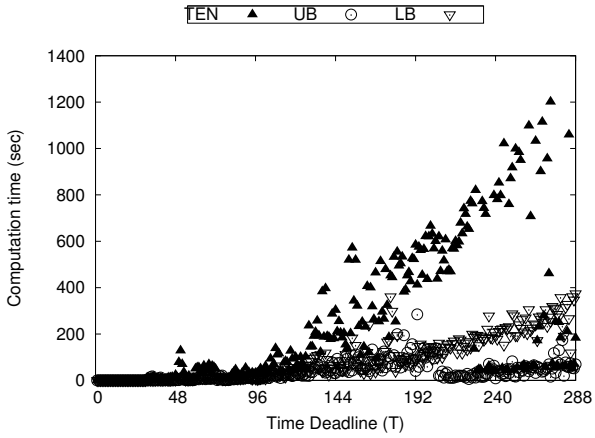
Setting	Site Index	Data Size (per site)
Uniform and *Half Shipment*	1-11	0.18 TB
Skewed	1, 3, 5, 7, 9, 11 2, 4, 6, 8, 10	0.3 TB 0.04 TB

Table 3.2: Size of source data at each site for the various experimental settings.

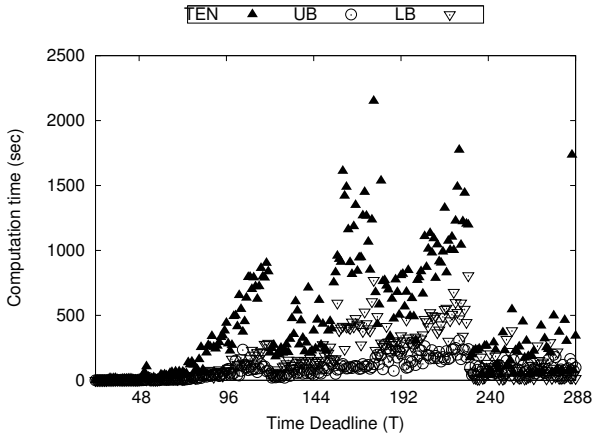
Our experiments were driven by trace data on actual Internet and shipment networks between academic sites. The basic topology we used had a single sink at uiuc.edu and 11 additional sites at .edu domains as listed in Table 3.1.

The Internet bandwidth between the sites was derived from PlanetLab available bandwidth traces measured using the Spruce measurement tool [122] by the Scalable Sensing Service (S^3) [135] (at 12:32 pm on Nov 15, 2009). We obtained real shipping cost and time data between all sites by using FedEx SOAP (Simple Object Access Protocol) web services [14], with site addresses provided by a whois lookup to the domains. For service charges at the sink, we used Amazon AWS's published costs.

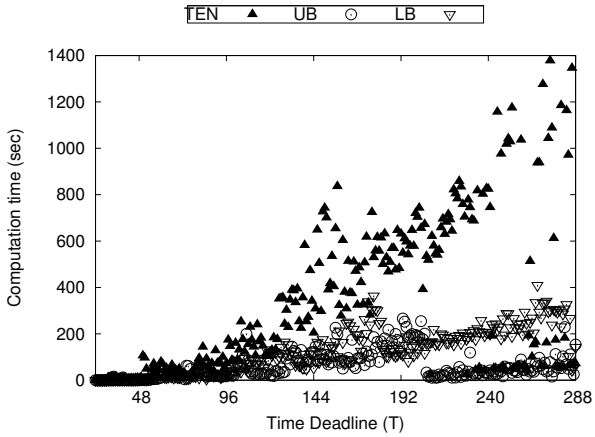
Sites 1 through 11 were chosen evenly by taking one site from each of 11 quantiles based on the measured bandwidth between .edu domains and the



(a) Uniform



(b) Skewed



(c) Half Priced Shipment

Figure 3.5: Computation times for computing the T-deadline constrained minimum cost for TEN, UB, and LB networks with $k = 4$.

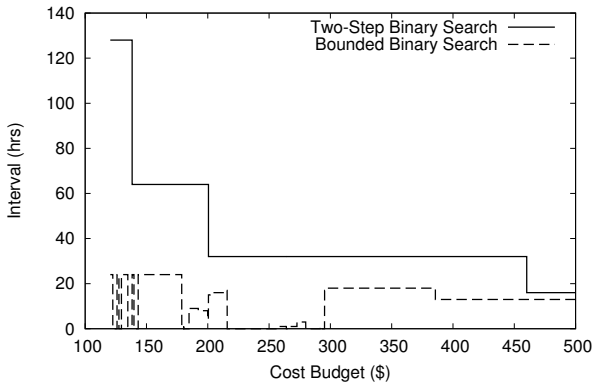


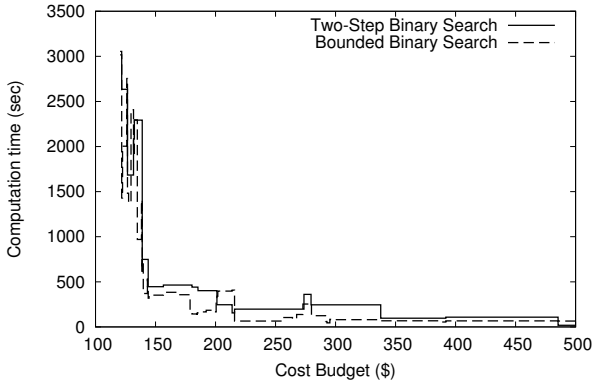
Figure 3.6: The difference in the binary search interval, using the Two-Step Binary Search in Algorithm 3.2 and the Bounded Binary Search in Algorithm 3.3.

sink. They serve as our data sources. We use three different experimental settings. These are chosen to show how our algorithms cope with diverse, realistic environments. As shown in Table 3.2, they are: *Uniform*, which places 2 TB of data uniformly at each source (0.18 TB each); *Skewed*, which places 1.8 TB at sources 1, 3, 5, 7, 9, and 11 (0.3 TB each) and 0.2 TB at sources 2, 4, 6, 8, and 10 (0.04 TB each); and *Half Shipment*, which places data in the same way as the Uniform setting, but cuts the shipment link costs to half of their real values. Where not mentioned, we use the Uniform setting.

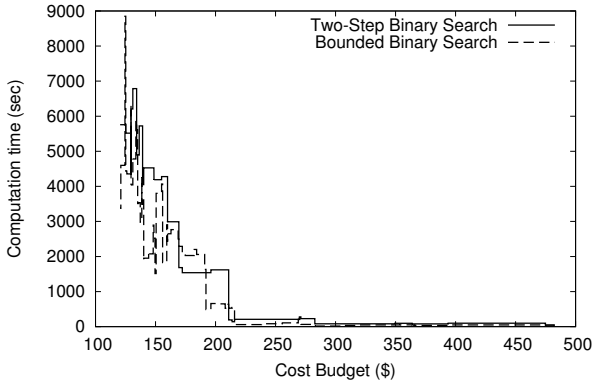
3.4.2 UB and LB network Microbenchmarks

Our first set of experiments, shown in Figure 3.5, compares the computation times for solving the T-deadline constrained minimum cost problem on the original time-expanded networks (TEN) against the same computation on UB and LB networks. There is much variation in the measured computation times. Still, generally for all three settings, the computation time forms an upward trend with increasing T. Also, the TEN network computations make up the majority of high computation times. The UB and LB network computations are comparatively cheaper. Thus, the UB and LB networks meet the criteria for a bounding function of being cheaper to compute than the original function.

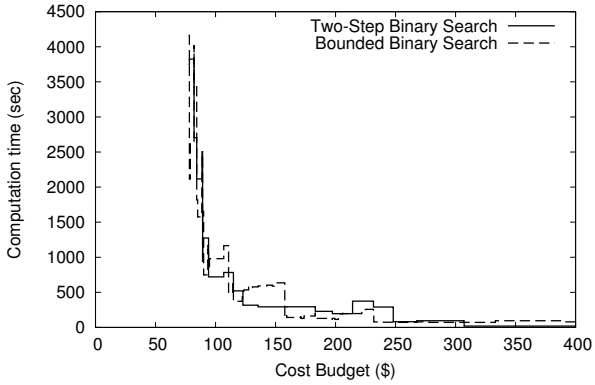
In Figure 3.6 we investigate whether the bounding functions are sufficiently tight. We show the interval that is searched by the TEN network for both



(a) Uniform

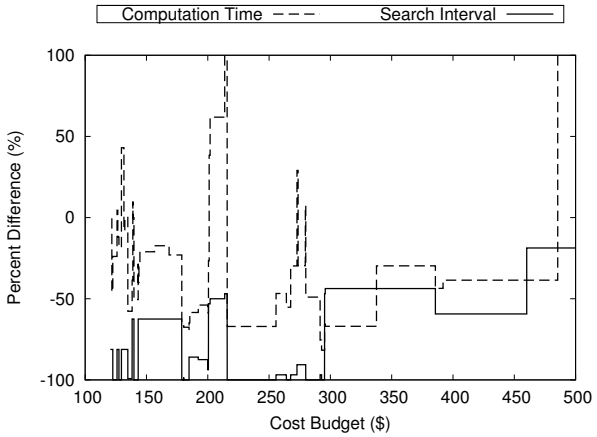


(b) Skewed



(c) Half Priced Shipment

Figure 3.7: Computation times using different search strategies.



Budget Range	Pearson Coefficient
$[0, 150]$	0.30
$[0, 175]$	0.28
$[0, 200]$	0.53
$[0, 225]$	0.50
$[0, 250]$	0.50
$[0, 275]$	0.53
$[0, 300]$	0.60
$[0, 325]$	0.60
$[0, 350]$	0.11

Figure 3.8: The difference between the computation time of bounded and two-step binary search strategies is correlated with the difference in the size of the interval between the strategies. The positive correlation is strong in the first half of the budget range.

algorithms. For Algorithm 3.3, this is the distance between T^*_{lb} and T^*_{ub} (used as parameters in line 5), while for Algorithm 3.2, this is the interval found in the first step (used as parameters in line 3). This interval is large when the budget constraint is strict, because it grows with T^*. In contrast, the interval found with the bounding functions does not grow with T^*. Thus, the bounding functions should be useful, especially when there is a very tight budget constraint.

3.4.3 Binary Search Strategies

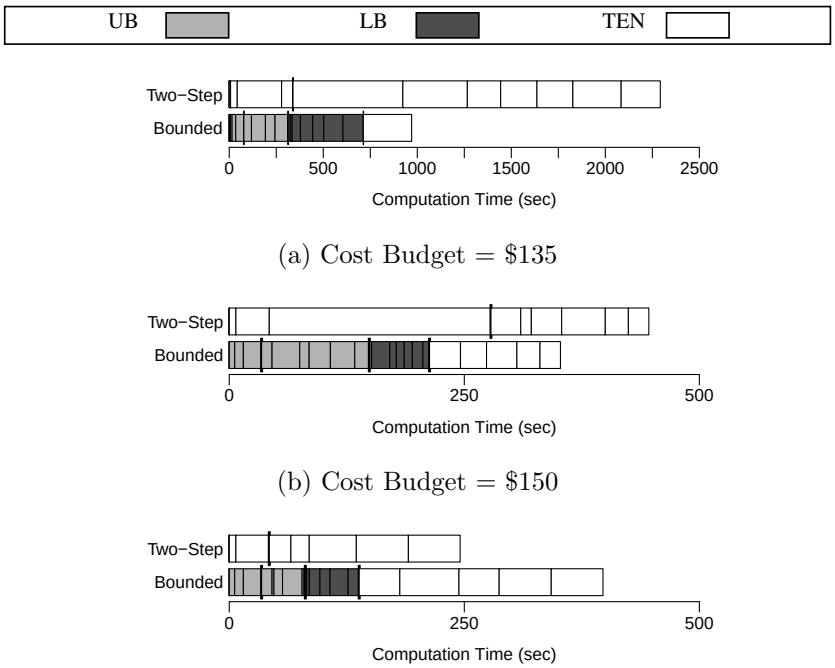
In this section, we look at the computation time for finding an optimal solution to the budget-constrained transfer problem using our binary search strategies. We show the computation times of the two-step binary search and bounded binary search in Figure 3.7. We find that, in general, more stringent cost budgets imply a longer time to compute. This is intuitive because a stringent cost budget will have a larger value of T^*, which in turn means that both the number of search iterations increases, and larger time

points for the minimum cost problem will have to be solved.

When we compared the effectiveness of both strategies, we found that in the majority of cases the computation time of using the bounded binary search strategy is less than that using the two-step binary search strategy. This pattern is true for all three experimental settings, despite the differences in the computation time for each deadline T in Figure 3.5. This suggests that the bounded binary search strategy is useful for solving transfer problems on a wide range of networks.

The comparison also shows variance in the relative difference in computation time for the bounded binary search and two-step binary search strategies. Some of this variance can be seen as a natural consequence of the variance we observed in Figure 3.5 for minimum cost computation running times. Yet, we are able to determine in Figure 3.8 that the difference in computation is correlated to the difference in interval size for the final minimum cost binary search. The plot shows the relative percentage difference of both the computation time and interval length. The shape of the curves look roughly correlated. We quantify this correlation using the Pearson product-moment correlation coefficient statistic [88]. The Pearson coefficient is a value between -1 and 1, that is used to measure the strength of linear dependence. A strong positive correlation would have a coefficient close 1. We apply the Pearson correlation across various ranges of the budget constraint. We observe from Figure 3.8 that there is a somewhat strong correlation of 0.5 when looking at the first half of the budget range (up to \$320), for which computation times are relatively high. Since values are significantly positive, we can conclude that the bounded binary search is effective because the bounding functions limit the number of computations required for the final minimum cost binary search.

In Figure 3.9, we break down the computation time of the binary search techniques into their various stages, for a few specific cost budgets. Each slot is color-coded to represent the minimum cost computation of a network variant. The computations are divided into stages of the binary search algorithms by vertical lines. Algorithm 3.2 has two stages (lines 2 and 3) while Algorithm 3.3 has four stages (lines 2 through 5). Both Figure 3.9 (a) and (b) show the common case where the bounded binary search is better. In (a), finding the value of T^*_{ub} for the bounded binary search finishes a little before the upper limit is found in the two-step binary search. Then, given



(a) Cost Budget = \$135

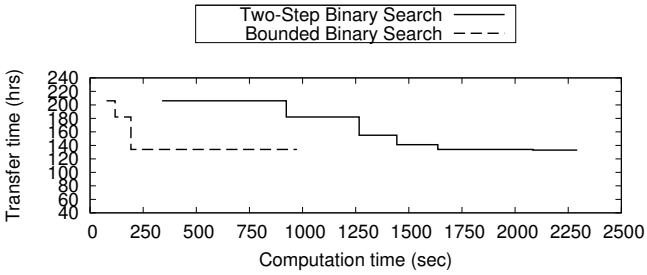
(b) Cost Budget = \$150

(c) Cost Budget = \$210

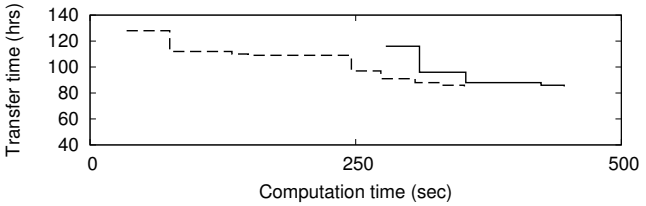
Figure 3.9: Timeline of the minimum cost computations taken in Algorithm 3.2 and Algorithm 3.3. Each slot represents a computation. The slots are grouped according to stage, which are presented in lines 2-3 of Algorithm 3.2 and lines 2-5 of Algorithm 3.3.

these upper limits, the binary search for T^*_{lb} reduces the binary search interval much faster than the binary search on the original problem. In this case, the interval is small enough that only a single minimum cost computation of the original problem is required in the last stage. In (b), the first three stages of the bounded binary search finish before the first stage of the two-step binary search. In this case, using the bounding functions allowed us to find a tighter interval in a shorter amount of time for the final stage. Finally, (c) shows an example of the less common case where the computation time for the bounded binary search takes longer than the two-step version. In this case, the exponential search stage of the two-step binary search finishes much earlier than the first three stages of the bounded binary search. Then, in the final stage of the bounded binary search, the computation time of each minimum cost took longer than the unbounded search. This can happen because of the variations in computation times shown in Figure 3.5. However, despite these variances, we have observed (previously in Figure 3.7) that the bounded binary search is a better option in most cases.

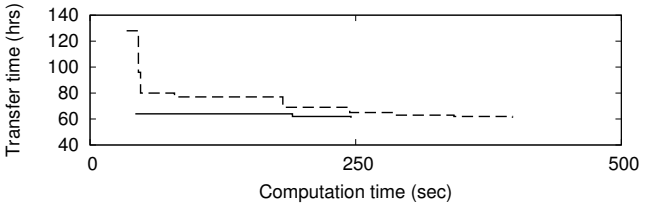
3.4.4 Partial Result Solutions



(a) Cost Budget = $135



(b) Cost Budget = $150



(c) Cost Budget = $210

Figure 3.10: Evolution of best transfer time during binary search.

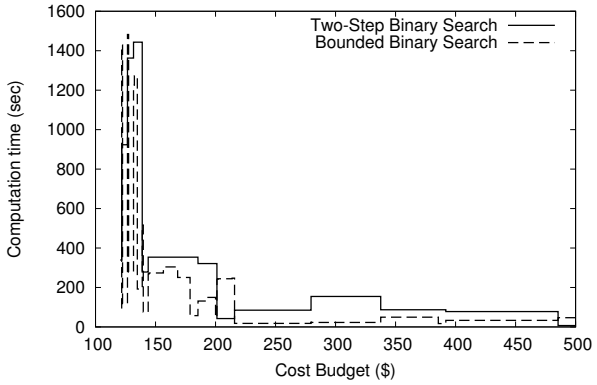


Figure 3.11: Computation time needed to find a solution within 10% of the optimal.

As our final set of experiments, we look at the effectiveness of our binary search algorithms in producing non-optimal partial results. Recall that for

both binary search strategies, the first feasible solution is found after their respective first stages. The results in Figure 3.9 show that at least one sub-optimal solution that meets constraints becomes available much earlier than the final optimal solution. For example, in Figure 3.9a, a solution becomes available in less than 350 seconds for the two-stage binary search, and in less than 100 seconds for the bounded binary search.

For partial results, we are interested not only in when a solution becomes available. It is important also to consider the quality of the solution. We show this in Figure 3.10. We plot the evolution of the best solution's transfer time as the binary search progresses. We observe that the bounded binary search finds solutions with short transfer time much faster than the two-stage strategy. Intuitively, this is because the feasible solutions on the UB network are feasible solutions on the original TEN network.

In Figure 3.11 we show how quickly the binary search strategies converge to near-optimal solutions across all cost budgets. We plot the computation time needed to get a solution that is within 10% of the optimal transfer time. Compared with Figure 3.7a, we see a significant decrease in computation time required. For lower cost budgets, the computation time is less than half of finding the optimal time. Thus, using partial results is an attractive technique for users that wish to decrease computation time.

3.5 Related Work

The budget-constrained data transfer problem presented is related to the quickest transshipment and evacuation problems posed on dynamic flow networks [71, 85, 70]. Our problem is different, because it contains edges with step function costs. Step function costs are considered in similar problems in operations research [123], such as [84] and [90]. Yet these problems do not take into account transfer latencies. Thus, finding an optimal solution is more challenging than the aforementioned works. Supply chain management considers the efficient transporting of physical goods across many different transportation networks as a key strategic goal [58]. As far as we know, this literature has not considered either bulk data or the Internet as a transportation network.

We covered related work on bulk data transfers and ad-hoc computations

in Section 2.6.

3.6 Summary

In this chapter, we have formulated and solved the problem of finding the fastest bulk data transfer plan given a strict budget constraint. We first characterized the solution space, and observed that the optimal solution can be found by searching through solutions to the deadline-constrained minimum cost problem, solved in Chapter 2. Based on these observations, we devised a two-step binary search method that will find an optimal solution. We then developed a bounded binary search method that makes use of bounding functions that provide upper- and lower bounds. We presented two instances of bounding functions, based on variants of our data transfer networks, and proved that they do indeed provide bounds. Finally, we evaluated our algorithms by running them on realistic network inputs. We found that our techniques significantly reduce the time needed to compute solutions.

Chapter 4

Vivace: Consistent Data for Congested Geo-distributed Systems

In this chapter, we consider achieving strong consistency on congested links across geo-distributed data centers. Our solution, Vivace, solves the requirement of strong consistency in a key-value store, through two strongly consistent algorithms. These algorithms are designed to provide low latency even during periods of congestion between data centers.

4.1 Motivation

Web applications such as web mail, search, online stores, portals, social networks, and others are increasingly deployed in *geo-distributed* data centers, that is, data centers distributed over many geographic locations. These applications often rely on key-value storage systems [64] to keep persistent user data, such as shopping carts, profiles, user preferences, and others. These data should be replicated across data centers or *sites*, to provide disaster tolerance, access locality, and read scalability.

Traditional replication solutions fall in two categories:

- *Local replication,* designed for local-area networks, which provide strong consistency and easy-to-use semantics. Protocols for local replication include ABD and Paxos [45, 97]. These protocols perform poorly when replicas are spread over remote sites.
- *Remote replication,* designed for replicas at many sites, which provides good performance in that case, by propagating updates asynchronously (e.g., [64, 61, 104]). These protocols provide weaker forms of consistency, such as eventual consistency [125].

Vivace is a key-value storage system that combines the advantages of both categories, by providing strong consistency, while performing well when deployed across multiple remote sites. Strong consistency is desirable: although

weak consistency may be adequate in some cases (e.g., [64, 61]), strong consistency is necessary in others, when reading stale data is undesirable.[1] The difficulty with providing strong consistency is that it requires coordination across sites to execute storage requests. At first glance, this coordination appears to be prohibitive, due to the higher network latencies across sites. However, typical round-trip latencies across sites are not too high—around 50–200 ms—which can be reasonable for certain storage systems. In fact, Megastore [46] has demonstrated that synchronous replication can sometimes work across sites.

The real problem while providing strong consistency occurs when the cross-site links become congested, causing the round-trip latency to increase to several seconds or more—as observed in systems like Amazon's EC2 (Section 4.2). In that case, Megastore and existing synchronous replication solutions suffer from delays of several seconds, which are unacceptable to users. A study by Nielsen [110] indicates that web applications should respond to users within one second or less.

A solution to this problem is to avoid network congestion, by provisioning the cross-site links for peak usage. This solution can be expensive and wasteful, especially when the system operates at peak usage only rarely, as in many computer systems. A better solution is to provision cross-site links more conservatively—say, for typical usage—and design the system to deal with congestion when it appears. This is the approach taken in Vivace.

The key innovation in Vivace is two novel strongly consistent replication algorithms that behave well under congestion. The first algorithm is simpler and implements a storage system with read and write atomic operations. The second algorithm is more complex and implements a state machine [120], which supports generic read-modify-write atomic operations. These algorithms rely on network prioritization of a few latency-critical messages in the system to avoid delays due to congestion. Because the size and number of prioritized messages is small, only a tiny fraction of the total network bandwidth comprises prioritized data. We evaluate Vivace and its algorithms to show that they are feasible and effective at avoiding delays due to congestion, without consuming resources significantly.

[1]For this reason, cloud services such as Google AppEngine, Amazon SimpleDB, and others support both weak and strong consistency.

4.2 Setting and Goals

We consider a system with multiple data centers or *sites*, where each site consists of many machines connected by a local-area network with low latency and high bandwidth. Sites are connected to each other via wide-area network links which have lower bandwidth and higher latencies than the links within a site. Machines or processes are subject to crash failures; we do not consider Byzantine failures. Network partitions may prevent communication across sites. Such partitions are rare, because the cross-site links are either provided by ISPs that promise very high availability, or by private leased lines that also provide very high availability. Nevertheless, partitions may occur; we represent them as larger network delays that last until the partition is healed. The system is subject to disasters that may destroy an entire site or disconnect the site from the other sites. We represent such a disaster as a crash of many or all the machines in the affected site.

Processes have access to synchronized clocks, which are used as counters. These clocks can be realized with GPS sensors, radio signals, or protocols such as NTP. Alternatively, it is possible to replace the use of synchronized clocks in Vivace with a distributed protocol that provides a global counter. However, doing so would reduce performance of Vivace due to the need of a network round-trip to obtain a counter value.

The network is subject to intermittent congestion across sites, because the cross-site bandwidth is provisioned based on typical or average usage, not peak usage. This is so due to cost considerations. For example, small or medium data centers may use MPLS VPNs[2], which can provide a fixed contracted bandwidth priced accordingly (Figure 4.1). Large data centers might use private leased lines or other solutions, but the bandwidth could still be much smaller than needed at peak times, causing congestion. For example, in Amazon EC2, simple measurements of network latencies across locations show that congestion occurs often, causing the round-trip latencies of messages sent over TCP to grow from hundreds of milliseconds to several seconds or more [94].

We assume the ability to prioritize messages in the network, so that they are transmitted ahead of other messages. We evaluate whether this assumption holds in Section 4.6.2. Support for prioritization is required only at the

[2]MPLS VPN is a technology offered by ISPs to connect together sites [2].

Mbps	K$/month	Mbps	K$/month	Mbps	K$/month
2	3.3	8	11.6	40	31.2
4	6.4	10	13.7	50	37.3
5	7.8	20	21.4	60	43.4
6	9.1	30	25.1	100	67.9

Figure 4.1: Sample cost to connect sites using MPLS VPNs, as a function of the contracted maximum bandwidth [15].

network edge of a site—say, at the router that connects the site to the external network—not in the external network itself, because congestion tends to occur at the site edge.

We wish a distributed key-value storage system that replicates data across sites and provides strong consistency, defined precisely by linearizability [83]. Roughly speaking, linearizability requires each storage operation—such as a read or write—to appear to take effect sequentially at an instantaneous point in time.

The key-value storage system should support two types of data objects: (1) RW objects, which provide read-write storage, and (2) RMW objects, which provide state machines; RMW stands for read-modify-write, which are operations that can modify the object's state based on its previous state. Objects of either type are identified by a *key*. A RW object has two operations, *write*(*value*) and *read*(), which stores a new value and retrieves the current value of the object, respectively. The size of object keys tend to be small (bytes), whereas the values stored in an object can be much larger (KBs).

RMW objects have an operation *execute*(*command*), which applies the command. RMW objects are state machines [120]—which are implemented by protocols such as Paxos—and the command can be an arbitrary deterministic procedure that, based on the current state of the object, modifies the state of the object and returns a result. We consider a slightly weaker type of state machine, where if many concurrent commands are executed on the same RMW object, the system is allowed to abort the execution of the commands, returning a special value $\perp$. Aborted operations may or may not take effect; if an aborted operation does not take effect, the user can reissue the operation after a while. By using known techniques, such as exponential random back-off or a leader election service, it is possible to guarantee that

the operation takes effect exactly once [38].

4.3 Design and Algorithms

We now explain the design of Vivace, with a focus on the replication algorithms that it uses.

4.3.1 Architecture

Vivace has a standard architecture for a key-value storage system, which we now briefly describe. There is a set of storage servers, which store the state of objects, and a set of client machines, which run applications. Applications interact with Vivace via a client library.

Each object has a *type*, which is RW or RMW (Section 4.2), and a *replica set*, which indicates the storage servers and sites where the object is replicated. In addition, for each site, each object has a set of local storage servers, called the *local replica set* (or simply *local set*) of the object, which we explain in Section 4.3.2. Our algorithms can make progress despite the crash of any minority of replicas in the replica set, plus any minority of replicas in the local set. Replica sets and local sets can be provisioned accordingly. For example, in our evaluation we provisioned three replicas for every replica set and local set, so the system tolerates one failure in each set.

The type, replica set, and local set comprise the *metadata* of an object. To reduce the overhead of storing metadata, objects are grouped into containers, and all objects in the container share the same metadata. The container of an object is fixed and the container id is a part of the object's key. A directory service stores the metadata for each container. Clients consult the directory service rarely, since they cache the metadata. The directory service is itself implemented using Vivace's replication algorithms, except the metadata for the directory objects is fixed: the type is a RW object, and the replica set is a fixed set of DNS names.

4.3.2 Algorithm for RW objects

Vivace's algorithm for RW objects is based on the *ABD* algorithm by Attiya, Bar-Noy, and Dolev [45]. It is a simple algorithm that provides linearizable read and write operations that always succeed when a majority of replicas are up. Moreover, the algorithm ensures safety and progress in a completely asynchronous system. In Section 4.3.3, we present a more complex algorithm that implements a state machine for RMW objects. That algorithm requires some partial synchrony to ensure progress—as with any other state machine algorithm.

We now briefly describe the ABD algorithm. To write value v to an object, the client obtains a new timestamp ts, asks for the replicas to store (v, ts), and waits for a majority of acknowledgments[3]. To read the latest value, the client asks the replicas to send their current pairs of (v, ts). The client waits for a majority of replies, and picks the reply (v', mts) with the largest timestamp mts. The client then executes a write-back phase, in which it asks replicas to store (v', mts) and waits for a majority of acknowledgments. This write-back phase is needed to provide linearizability: it ensures that a subsequent read operation sees v' or a more recent value.

If the replicas are in different sites, the ABD algorithm sends and receives messages across sites before the operation can complete. If remote network paths are congested, this remote communication can take a long time. We propose to avoid these congestion delays, by having the client use prioritized messages that are transmitted ahead of other messages, thereby bypassing the congestion. Prioritized messages must be small, otherwise these messages themselves will congest the network. To obtain small messages, we modify the ABD algorithm by breaking up its messages into two parts: critical fields—such as timestamps, statuses, and acks—that must be sent immediately, and the other fields. We restructure the ABD algorithm to operate correctly when the messages are broken up, and then we use prioritized messages for sending the critical fields. The challenge in doing so is threefold. First, we must still continue to provide linearizability (strong consistency) when the messages have been split; in fact, we define linearizability as the correctness criteria for the new algorithm. The difficulty here is that a message that is split in

[3]In the algorithm in [45], there are no synchronized clocks, so there is an extra round of communication to obtain a new timestamp. Here we obtain the timestamp from the synchronized clocks.

two parts may be interleaved with the split messages of other clients, creating concurrency problems. To address such problems, the new algorithm includes some additional phases of communication and coordination. Second, we must find a very small amount of critical information to prioritize, otherwise the prioritized data will congest the network; we later analyze and evaluate the new algorithm to show that the prioritized fields we chose indeed comprise a small proportion of the total data sent. Third, we must not impose significant extra overhead in the new algorithm, otherwise it will perform worse than the original algorithm when the network is not congested; in particular, the new algorithm has extra phases of communication; we later evaluate this overhead and show that it is very small and worth the benefit, because the extra communication occurs in the local area network.

We now describe the algorithm in more detail. To read a value, the client uses a small prioritized message to ask replicas to send their current timestamp. Replicas reply with another small prioritized message. Once the client has a majority of replies, it computes the highest timestamp *mts* that it received. It then asks replicas to send the data associated with timestamp *mts*. The reply is a large non-prioritized message with data but, in the common case, a replica in the local site has the data, so this replica responds quickly without remote communication. Thus, the client can read the value without being affected by the congestion on the remote path. The client then performs a fast write-back phase, by sending a small prioritized message with only the highest timestamp, not the data value.

To write a value v, the client obtains a new timestamp *ts*. We want to avoid sending v to remote sites in the critical path. The client stores (v, ts) at temporary replicas located in the same site as the client; the set of temporary replicas is called the *local replica set*, and each replica is called a *local replica*. The client also stores the timestamp *ts* at the (normal) replicas—which are typically at remote sites—using small prioritized messages; these messages carry only the timestamp *ts*, not the data value, and they bypass congestion on the remote paths. Once the client receives enough acknowledgments, the operation completes. Meanwhile, in the background, each local replica propagates the data value to the (normal) replicas. These larger messages do not delay the client, even if there is congestion, because they are not in the critical path. Furthermore, the larger messages are not prioritized.

Detailed pseudocode. The pseudocode is given by Algorithm 4.1. We

Algorithm 4.1 Vivace algorithm for RW objects

```
function read(key):
  acks ← sendwait(∗⟨R-TS, key⟩, nodes[key], ⌈(n+1)/2⌉)
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  data ← sendwait(∗⟨R-DATA, key, mts⟩, nodes[key], 1)
  sendwait(∗⟨W-TS, key, data[0].ts⟩, nodes[key], ⌈(n+1)/2⌉)
  return data[0].val

function write(key, val):
  ts ← clock()
  sendwait(⟨W-LOCAL, key, ts, val⟩, local_nodes[key], ⌈(n+1)/2⌉)
  sendwait(∗⟨W-TS, key, ts⟩, nodes[key], ⌈(n+1)/2⌉)

function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  return the replies in an array acks[0..num_acks − 1]

upon receive ⟨R-TS, key⟩:
  return ∗⟨ACK-R-TS, ts[key]⟩

upon receive ⟨R-DATA, key, ts⟩:
  wait until ts[key] ≥ ts and val[key] ≠⊥
  return ⟨ACK-R-DATA, ts[key], val[key]⟩

upon receive ⟨W-TS, key, ts⟩:
  if ts > ts[key] then
    ts[key] ← ts
    if remote_buf[key][ts] exists then
      val[key] ← remote_buf[key][ts]
      delete remote_buf[key][x] for all x ≤ ts
    end if
    else val[key] ←⊥
  end if
  return ∗⟨ACK-W-TS⟩

upon receive ⟨W-LOCAL, key, ts, val⟩:
  local_buf[key][ts] ← val
  async
    sendwait(⟨W-REMOTE, key, ts, val⟩, nodes[key], ⌈(n+1)/2⌉)
    delete local_buf[key][ts]
  return ∗⟨ACK-W-LOCAL⟩

upon receive ⟨W-REMOTE, key, ts, val⟩:
  if ts = ts[key] then val[key] ← val
  else if ts > ts[key] then remote_buf[key][ts] ← val
  return ⟨ACK-W-REMOTE⟩
```

denote by *read*(*key*) and *write*(*key*, *v*) the operations to read and write an object with the given *key*; n is the number of replicas for the object, and f is a fault-tolerance parameter indicating the maximum number of replicas that may crash. The algorithm requires that $f<n/2$, that is, only a minority of replicas may crash. The replica set of an object with a given *key* is denoted *nodes*[*key*], while the local replica set at a given *site* is denoted *local_nodes*[*key*, *site*]. We omit *site* from *local_nodes*[*key*, *site*] when the site is local (where the client is). That is, *local_nodes*[*key*] refers to the local replica set at the client's site. A prioritized message m is denoted $*\langle m \rangle$, and a normal message m is denoted $\langle m \rangle$.

The communication in the algorithm occurs via a simple function *sendwait*, which sends a given message *msg* to a set of *nodes* and waits for *num_acks* replies. The replies are returned in an array.

To write a value, the client obtains a new timestamp and executes two phases. In the first phase, the client sends the value and timestamp to the local replicas, and waits for an acknowledgment from a majority. In the second phase, the client sends just the timestamp to the replicas, using prioritized messages. When the client receives acknowledgments from a majority, it completes the write operation. Meanwhile, the local replicas propagate the value to the (regular) replicas. The client need not wait for the propagation to complete, because a majority of the local replicas already store the data and a majority of the replicas store the timestamp: if another client in a different site were to execute a read operation, it would observe the timestamp from at least one replica and know what value it needs to wait for.

To read a value, the client executes three phases. In the first phase, the client retrieves the timestamp from a majority of the replicas, and picks the highest timestamp *mts*. In the second phase, the client asks the replicas to send the data associated with this timestamp, if they have it. A replica replies only if it has a timestamp at least as large as the requested timestamp. This phase completes when the client obtains its first reply. In the common case, this reply arrives quickly from a replica in the same site as the client. Once the second phase has completed, the client knows the value that it must return for the read operation. In the third phase, the client writes back the timestamp to the replicas using prioritized messages. When the client receives a majority of acknowledgments, it completes the read operation.

Note that the client returns from a read operation without having to write

back the value v. This is possible because the client that originally sent timestamp *mts* to the replicas did so only after it had stored v at a majority of local replicas. When the read operation returns, a majority of replicas has seen the timestamp *mts* of v, but not necessarily v itself. However, we are guaranteed that a majority of replicas subsequently receive v from the local replicas.

4.3.3 Algorithm for RMW objects

We now present Vivace's algorithm for state machines, to implement RMW objects. We apply the same principles as in Section 4.3.2: the basic idea is to break-up the protocol messages into critical and non-critical fields, restructuring the algorithm so that, in the critical path, remote communication involves only the critical fields. The non-critical fields are stored at a majority of local replicas and later propagated to the (regular) replicas in the background.

Our starting point is an algorithm for RMW objects similar to the algorithms in [72, 57, 38], which we now briefly describe—we later explain how we apply the above principles to this algorithm. The base RMW algorithm is not new; we explain it here for completeness.

To execute a command, a client first obtains a new timestamp *ts*, and sends it to the replicas. Each replica stores the timestamp as a tentative ordering timestamp and subsequently rejects smaller timestamps. The replica replies with its current value and associated timestamp. The client waits for a majority of responses, and picks the value v with the highest timestamp. It applies the command to the value v to obtain a new value v' and a response r. The client then asks the replicas to store v' with the new timestamp *ts*. Each replica accepts the request if it has never seen a higher timestamp; otherwise, the replica returns an error to the client. The client waits for a majority of responses, and if any of them is an error, the client aborts by returning $\perp$; otherwise, if no responses were an error, the client returns r.

The pseudocode is given by Algorithm 4.2. The first message sent by a client has a tag *OR*, which asks each replica to store a tentative ordering timestamp and reply with its current value and timestamp. The second message sent by a client has a tag *OW*, which asks each replica to store the

Algorithm 4.2 Algorithm for RMW objects in a LAN

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(⟨OR, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if acks = ⊥ then return ⊥
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  mval ← acks[i].msg.val where acks[i].msg.ts = mts
  ⟨val, r⟩ ← apply(mval, command)
  w_acks ← sendwait(⟨OW, key, ots, val⟩, nodes[key], ⌈(n+1)/2⌉)
  if w_acks = ⊥ then return ⊥
  else return r
```

```
function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]
```

```
upon receive ⟨OR, key, ots⟩:
  if ots > ots[key] then
     ots[key] ← ots
     return ⟨ACK-OR, true, ts[key], val[key]⟩
  end if
  else return ⟨ACK-OR, false⟩

upon receive ⟨OW, key, ots, val⟩:
  if ots ≥ ots[key] then
     ots[key] ← ots
     ts[key] ← ots
     val[key] ← val
     return ⟨ACK-OW, true⟩
  end if
  else return ⟨ACK-OW, false⟩
```

Algorithm 4.3 Vivace algorithm for RMW objects (1/2)

```
function execute(key, command):
  ots ← clock()
  acks ← sendwait(∗⟨OR-TS, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if acks =⊥ then return ⊥
  mts ← max_{1≤i≤⌈(n+1)/2⌉} acks[i].msg.ts
  data ← sendwait(∗⟨OR-DATA, key, mts⟩, nodes, 1)
  if data =⊥ then return ⊥
  ⟨val, r⟩ ← apply(data[0].val, command)
  sendwait(⟨W-LOCAL, key, ots, val⟩, local_nodes[key], ⌈(n+1)/2⌉)
  w_acks ← sendwait(∗⟨OW-TS, key, ots⟩, nodes[key], ⌈(n+1)/2⌉)
  if w_acks =⊥ then
    return ⊥
  else
    return r
  end if
function sendwait(msg, nodes, num_acks):
  send msg to nodes
  wait for num_acks replies
  if any reply has status = false then return ⊥
  else return the replies in an array acks[0..num_acks − 1]
```

new value and timestamp. These messages can have data values that are large, and they are sent in the critical path.

We now explain how to send just small messages in the critical path, so that we can prioritize these messages and make the algorithm go faster when there is congestion. The *OR* message in Algorithm 4.2 itself does not have a data value, but its response carries data. We modify the algorithm so that the response no longer carries any data, just a timestamp. The client then picks the largest received timestamp *mts* and must now retrieve the value v associated with it, so that it can apply the command to v. To do so, the client sends a separate *OR-DATA* message to all replicas asking specifically to retrieve the value associated with *mts*. In the common case, a replica at the local site has the appropriate value and replies to the client quickly. The client can now proceed as before, by applying the command to obtain a new value v' and a response r that it will return to the caller when it is finished.

The *OW* message in Algorithm 4.2 carries the new value v', so we must change it. The client uses the local replica set as in the write function of Algorithm 4.1. The client proceeds in two phases: it first sends v' to the local replicas in a *W-LOCAL* message and waits for a majority of replies. In the background, the local replicas send the data to the (regular) replicas. The client then sends just the new timestamp *ts* to the replicas, knowing that

Algorithm 4.3 Vivace algorithm for RMW objects (2/2)

```
upon receive ⟨OR-TS, key, ots⟩:
  if ots > ots[key] then
    ots[key] ← ots
    return ∗⟨ACK-OR-TS, true, ts[key]⟩
  else
    return ∗⟨ACK-OR-TS, false⟩
  end if

upon receive ⟨OR-DATA, key, ts⟩:
  wait until ts[key] ≥ ts and val[key] ≠⊥
  if ts[key] = ts then
    return ⟨ACK-OR-DATA, true, val[key]⟩
  else
    return ⟨ACK-OR-DATA, false⟩
  end if

upon receive ⟨OW-TS, key, ots⟩:
  if ots ≥ ots[key] then
    ots[key] ← ots
    ts[key] ← ots
    if remote_buf[key][ots] exists then
      val[key] ← remote_buf[key][ots]
      delete remote_buf[key][x] for all x ≤ ots
    else
      val[key] ←⊥
    end if
    return ∗⟨ACK-OW-TS, true⟩
  else
    return ∗⟨ACK-OW-TS, false⟩
  end if

upon receive ⟨W-LOCAL, key, ts, val⟩:
  local_buf[key][ts] ← val
  async
    sendwait(⟨W-REMOTE, key, ts, val⟩, nodes[key], ⌈(n+1)/2⌉)
    delete local_buf[key][ts]
  return ∗⟨ACK-W-LOCAL⟩

upon receive ⟨W-REMOTE, key, ts, val⟩:
  if ts = ts[key] then
    val[key] ← val
  else if ts > ts[key] then
    remote_buf[key][ts] ← val
  end if
  return ⟨ACK-W-REMOTE⟩
```

they will eventually receive the value from the local replicas. Algorithm 4.3 presents the pseudocode with these ideas.

4.3.4 Optimizations

We now describe some optimizations to Algorithms 4.1 and 4.3, to further reduce the latency and bandwidth of operations.

Read optimizations

We present two optimizations for the three-phase read operation of Algorithm 4.1. The first optimization reduces the number of phases, while the second removes the need for a client to communicate with a majority of replicas.

Avoiding or parallelizing the write-back phase. Recall that Algorithm 4.1 has a write-back phase, which propagates the largest timestamp *mts* to a majority of replicas. This phase can be avoided in some common cases. In the original ABD algorithm, if (C1) the client receives the largest timestamp *mts* from a majority of replicas in the first phase, it can skip the write-back phase. Similarly, we can skip this phase in Algorithm 4.1, provided that the same condition (C1) is met and another condition is also met: (C2) the timestamp of the data received in phase two is also *mts*. (Condition (C2) is not needed in the original ABD algorithm because the data and timestamps are together.) With this optimization, in the common case when there are no failures or concurrent writes, a read completes in two phases. It is also possible to read in two phases when (C1) is not met, but (C2) is. To do this, we add a parallel write-back in the second phase, triggered when the client observes that (C1) does not hold. The client proactively writes back *mts* in phase two, in parallel with the request for data. When a data reply is received, the client checks (C2)—it compares the data timestamp with *mts*—and if it holds the read operation can complete in two phases. There are two rare corner cases, when (C2) does not hold. In that case, if (C1) was met in phase one, the read remains as in Algorithm 4.1; if neither (C1) nor (C2) hold, then the read can use the parallel write-back in phase two.

Reading data from fewer replicas. This optimization reduces bandwidth us-

age, by having the client send *R-DATA* requests to only some replicas. After the first read phase, the client considers the set of replicas for which it received *ACK-R-TS* containing the large timestamp *mts*. If a local replica exists in this set, the client sends a single *R-DATA* request to that replica. If not, the client sends the request to a subset of replicas, based on some policy. For example, the client can keep a history of recent latencies at remote links, and send a single request to the replica with the lowest latency. The policy used affects performance, but policy choice is orthogonal to the algorithm.

Role change optimizations

In these optimizations, the role played at a node is moved to another node: the first optimization moves the execution of commands from the client to a replica, and the second moves the client role from outside to inside a replica.

Executing commands at replicas. In the execute operation of Algorithm 4.3, the client reads the current value of the object from the replicas, applies the command to obtain the new value, and writes that value to the replicas. Doing so involves transferring the value from the replicas to the client and back to the replicas. If the value is large, but the command is small, it is more efficient for the client to send the command to the replicas and thereby avoid transferring the value back and forth. To do this, the client first sends the *OR-TS* message and finds the largest timestamp *mts*—this determines the state on which the command should be applied. Then, rather than retrieving the data, the client sends the command to the replicas and the timestamp *mts*. The replicas apply the command to the value with timestamp *mts* (they may have to wait until they obtain that value, but they eventually do), or they reject the command if they see a larger timestamp. If a replica applies the command, it stores the new value in a temporary buffer together with the new timestamp. The client waits for a majority of responses, and if none of these are rejects, the client can send *OW-TS* messages as before. A replica then retrieves the value from its temporary buffer. This optimization reduces the bandwidth consumption of remote links when the command is smaller than the data value.

Delegating to a replica. In Vivace, the client library does not directly execute Algorithms 4.1 and 4.3. Rather, the library contacts one of the replicas,

Message type	normal max delay	priority max delay
Local, within site	δ	δ
Remote, across sites	D	d

Figure 4.2: One-way delay parameters for latency analysis.

which then executes the algorithms on behalf of the client. Doing so is a common technique that saves bandwidth of the client, at the expense of the added latency of a local round-trip. It also makes it possible to modify the algorithms without changing the client library.

4.4 Analysis

We now analyze the algorithms of Section 4.3. We compare the latency of the new algorithms with the prior algorithms that they are based on. We then compare the size of prioritized and normal messages within the new algorithms. Next, we discuss the fault tolerance provided by the new algorithms.

4.4.1 Latency

We first consider latency, when a majority of the replicas are on sites different from the client's. (If a majority of the replicas are in the client's site, clients complete their operation locally.) One-way message delays are represented by a few parameters, depending on whether the delay is within or across sites, and whether the message is normal or prioritized, as shown in Figure 4.2. Within a site, normal and prioritized messages have the same delay δ, due to lack of congestion. The delay for messages sent to remote sites are represented as D when sent normally, and d when prioritized. All the delay parameters incorporate the time to send, transmit, receive, and process a message.

Figure 4.3 summarizes the results. An execution of an operation can have different latencies, depending on the set of live replicas and the object state at those replicas. We analyze two cases: common and worst. The common case represents a situation with no failures, so that there is a live replica at the local site (the site where the client is). The worst case occurs when

(a) all replicas at the local site are failed, and (b) the latest value is not yet stored at any replicas in the replica set; it is stored only at temporary, local set replicas, in a site remote to the client.

We first consider writes of RW objects. The ABD algorithm has a round-trip between the client and replicas (latency: $2D$). In the new Algorithm 4.1, this round-trip is replaced by two phases: the first one has a round-trip with local replicas (2δ), and the second one has a prioritized round-trip to remote sites ($2d$). By adding these delays, we obtain the total write latencies in Figure 4.3.

The read operation of ABD has two phases, each with a round-trip ($2D + 2D$). Algorithm 4.1 has three phases. The first phase has a prioritized round-trip ($2d$). The second phase depends on whether there is a replica at the client's site. If there is, the phase has a local round-trip (2δ); otherwise, there are three message delays: (1) the client sends a prioritized request to the replicas (d); (2) the replicas may not have the most recent value and must wait for it from a remote temporary replica (D); and (3) a remote replica sends the value to the client (D). The final write-back phase has a prioritized round-trip to remote replicas ($2d$). By adding these delays, we obtain the total of read latencies in Figure 4.3.

From Figure 4.3, we can see that reads and writes of RW objects are faster with the new Algorithm 4.1 than with ABD, because typically $\delta \ll d \ll D$. Also note that, in the common case, the latency of Algorithm 4.1 is independent of D, which is not true for ABD.

We now consider the execute operation for RMW objects. Algorithm 4.2 has two remote round-trips ($2D + 2D$). Algorithm 4.3 has four phases. The first and fourth phases each have a prioritized round-trip ($2d + 2d$). Without failures, the second and third phases have a local round-trip ($2\delta + 2\delta$). When replicas at the local site are not live, the read phase is prolonged, as in the read operation of Algorithm 4.1, which we analyzed above (it takes $d + 2D$ instead of 2δ). By adding these delays, we obtain the total of execute latencies in Figure 4.3. We can see that the new Algorithm 4.3 is significantly better than Algorithm 4.2 and that, in the common case, the latency of Algorithm 4.3 does not depend on D.

Algorithm	Operation	Common Case	Worst Case
ABD Algorithm (prior work)	read	$4D$	$4D$
	write	$2D$	$2D$
Algorithm 4.1 (new)	read	$2\delta + 4d$	$5d + 2D$
	write	$2\delta + 2d$	$2\delta + 2d$
Algorithm 4.2 (prior work)	execute	$4D$	$4D$
Algorithm 4.3 (new)	execute	$4\delta + 4d$	$2\delta + 5d + 2D$

Figure 4.3: Message delays: common and worst cases.

4.4.2 Size of prioritized and normal messages

Prioritized messages should be a small fraction of the traffic, otherwise they become useless. We now analyze whether that is the case with the new algorithms. Prioritized messages in Algorithms 4.1 and 4.3 have up to four pieces of information: message type, key, timestamp, and an accept bit indicating if the request is accepted or rejected. The message type and accept bit are stored in one byte. The timestamp is eight bytes, which is large enough so that it does not wrap around. The key has variable length, but is typically small, say 16 bytes. Adding up, the size of a prioritized message is up to 25 bytes. Each data message (which is not prioritized) has the preceding information and a value with several KBs, which is orders of magnitude larger than a prioritized message.

This difference in size is advantageous. Let k be the factor by which normal messages are larger than prioritized messages, and B be the bandwidth of a remote link. Then clients can issue storage operations with peaks of throughput equal to $P = k \times B$ without affecting the performance of the system: at the peak throughput, the entire remote link bandwidth is consumed by prioritized messages, and their message delays are still d, so Algorithms 4.1 and 4.3 perform as expected. Since k can be large (with data values of size 1 KB, $k \approx 40$), the benefit is significant.

4.4.3 Fault tolerance

The new Algorithms 4.1 and 4.3 are fault tolerant: they tolerate up to f replica crashes. Even if a site disaster destroys more than f replicas in a site, the algorithms safeguard most of the data: only data written in a small

window of vulnerability is lost (data held by the temporary local replicas but not yet propagated remotely). Furthermore, the algorithms allow administrators to identify the lost data quickly: these are the keys for which the remote replicas store a timestamp but not the data itself.

4.5 Implementation

Vivace consists of 6000 lines of Java. Clients and servers communicate using TCP. The system can be configured to use any of the four algorithms of Section 4.3: the ABD algorithm, Algorithm 4.1, Algorithm 4.2, and Algorithm 4.3. We implemented the optimizations, in Section 4.3.4 of avoiding the write-back phase and delegation to a replica, for the experiments in Section 4.6.4. We did not implement the directory service: currently, the metadata for containers is kept in a static configuration file. This does not affect our performance evaluation, because metadata lookups are rare due to caching.

4.6 Evaluation

We now evaluate Vivace. After describing the experimental setup (Section 4.6.1), we validate the assumption we made in Vivace that prioritized messages are feasible and effective (Section 4.6.2). We then consider the performance of the new algorithms of Vivace (Section 4.6.3). Finally, we demonstrate the benefits of Vivace in a real web application (Section 4.6.4).

4.6.1 Experimental setup

The experimental setup consists of machines in Amazon's EC2 and a private local cluster in Urbana-Champaign. In EC2, we use extra large virtual machine instances with 4 CPU cores and 15 GB of memory, in two locations, Virginia and Ireland. We use the private cluster for experiments that require changing the configuration of the network. The local cluster has three PCs with 2.4 GHz Pentium 4 CPU, 1 GB of memory, and an Intel PRO/100 NIC. The cluster is connected to the Internet via a Cisco Catalyst 3550 router,

which has 48 100 Mbps ports. The median round-trip latencies were the following (in ms):

	Local cluster	EC2 Virginia	EC2 Ireland
Local cluster	<1	23	109
EC2 Virginia		<1	93
EC2 Ireland			<1

4.6.2 Message prioritization schemes

Vivace assumes the existence of an effective mechanism to prioritize messages in the network. In this section, we examine whether this assumption holds. We consider network-based and host-based schemes to achieve prioritized messages, and evaluate their overhead and effectiveness in the presence of congestion. The network-based scheme relies on prioritization support by network devices, while the host-based scheme implements prioritization in software using a dedicated server. Both schemes can be set up within a site, without assumptions on the external network that connects sites.[4]

For each scheme, we answer four questions: What is required to use it? How to set it up? What is the overhead? How effective is it?

Network-based solution

What is required? The scheme requires devices and appropriate protocols that support prioritization of messages. Prioritized messages are available at several levels:

Layer	Device	Mechanism
IP	IP router	RFC 2474 (DiffServ)
MPLS VPN	Edge router	RFC 2474 (DiffServ)
Ethernet	Switch	IEEE 802.1p

One way to connect sites is via a private leased line, using a modem and an IP router or switch at each end of the line. An alternative cost-effective

[4]Another scheme is TCP Nice [129], which de-prioritizes traffic. We can conceptually prioritize messages by de-prioritizing all others using TCP Nice, but doing so requires de-prioritizing traffic outside our system. This could be hard and it fails if other systems use UDP.

scheme is to use VPNs, such as MPLS VPNs.

With private leased lines, prioritization is possible via IP or Ethernet solutions. IP solutions were first available using the Type of Service (ToS) bits in the IP header, which were later superseded by the 6-bit DSCP field in IP DiffServ. The DSCP field defines a traffic class for each IP packet, which can be used to prioritize traffic at each network hop. Ethernet solutions are based on the IEEE 802.1p standard, which uses a 3-bit PCP field on an Ethernet frame. Such solutions are available even on commodity low-end switches, where a use case is to prioritize video or real-time gaming traffic in a home network connected to a broadband modem.

With MPLS VPNs, prioritization is available via IP prioritization at the edge routers [2].

How to set it up? The simpler and lower-end devices are configured via web-based user interfaces. Higher-end routers and switches have configuration interfaces based on command lines. We set up traffic prioritization in the Cisco Catalyst 3550 router by configuring it to classify packets based on DSCP bits at ingress and place them accordingly into egress priority queues [35, Chapter 28].

What is the overhead? We evaluate the overhead of the scheme, by configuring the Cisco Catalyst 3550 router in the local cluster. We measured the round-trip latency of null requests sent from the cluster to EC2 (Ireland), with DSCP bit prioritization enabled and disabled. We found no detectable differences, indicating that the overhead of the prioritization mechanism in the IP router is small.

How effective is it? We perform a simple experiment: two clients in the private cluster periodically measure the round-trip time of their established TCP connection to a machine outside the cluster. One client was configured to set the DSCP field to a priority value, while the other used the default DSCP field. 10 seconds into the experiment, two machines in the cluster generate congestion by running iperf, which sends UDP traffic at a rate of 60 Mbps each to two machines. The congestion continues for 20 seconds and then stops. Figure 4.4 shows the round-trip latency observed by both clients. We can see that prioritized messages are effective: their latency is unaffected by the congestion. In contrast, congestion causes regular messages to be dropped, which triggers TCP retransmissions—sometimes multiple times—causing large delays due to the TCP back-off mechanism.

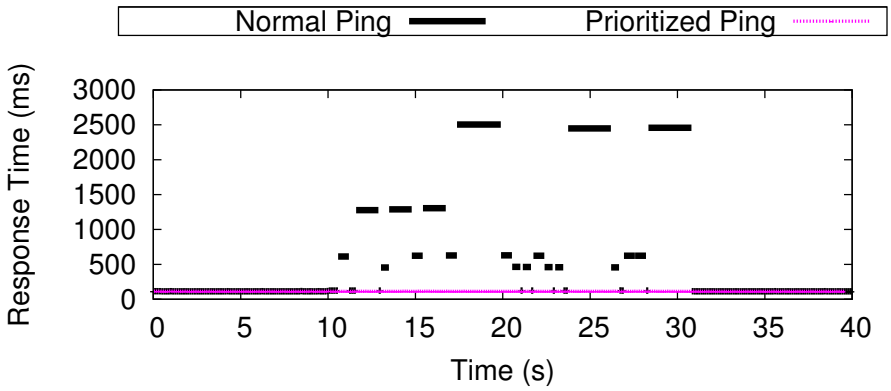


Figure 4.4: Round-trip delay for regular and prioritized messages using a router with IP DiffServ support.

Host-based solution

What is required? This scheme requires one or more proxy machines to handle traffic between sites. Messages targeted to machines outside the local site are first sent to a local proxy machine. The proxy forwards the message to a proxy machine in the remote site, which finally forwards the message to the destination. In each proxy, packets are prioritized by placing them into queues according to their DSCP bits. The proxy is a dedicated Linux instance, running SOCKS Dante server processes [11] and using outbound priority queues configured with the HTB packet scheduler in the *tc* tool [23].

How to set it up? To reduce internal traffic at a site, the proxy is placed close to the site's external network connection. Machines are configured to use the proxy, using the SOCKS protocol.

What is the overhead? We measure the round-trip latency of null requests sent between the Virginia and Ireland EC2 locations, with and without the proxy, to determine the extra latency added by the proxy. There was no congestion in the network. Without the proxy, the average round-trip latency is 93 ms, while with the proxy, it is 99 ms. The overhead of the proxy is dwarfed by the much larger network latencies across sites.

How effective is it? To evaluate the scheme, we place four machines at each of two EC2 locations (Virginia, Ireland). One machine runs the proxy, one runs two clients, and two run iperf to generate congestion. The experiment is the same as for the network-based solution, except that it uses an extra machine for the proxy, and it uses machines in EC2 only. The results are

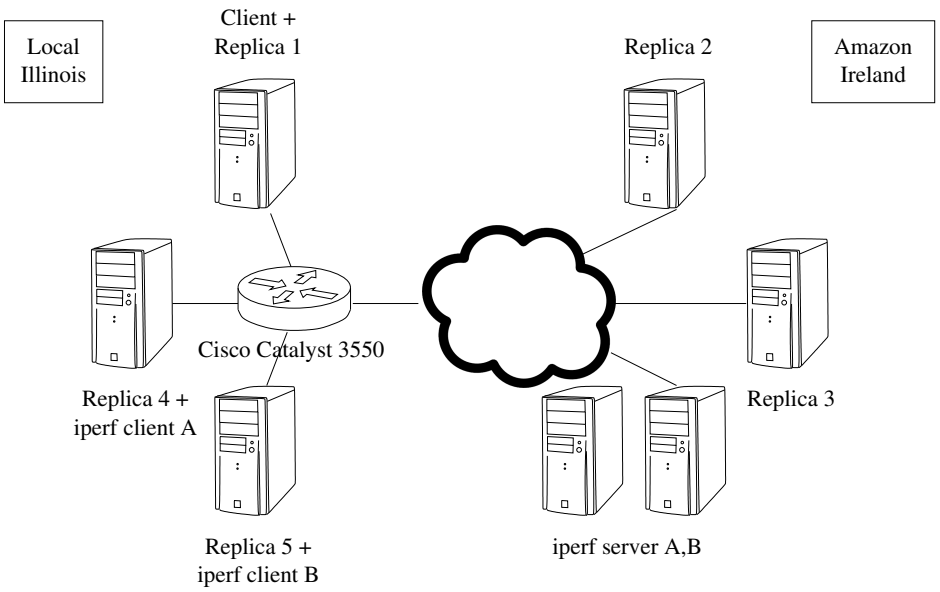


Figure 4.5: Network setup.

also similar and demonstrate that the prioritized messages are quite effective (not shown).

Applicability of each solution

The host and network-based solutions are applicable to a small or medium private data center connected by leased lines or MPLS VPNs. Larger data centers or the cloud have larger external bandwidths, and so the host-based scheme creates a bandwidth bottleneck at the proxy, making the network-based scheme a better alternative.

4.6.3 Storage algorithms

In the next experiments, we consider the performance of the new storage algorithms in Vivace. To do so, we configure Vivace to use either prior algorithms (ABD and Algorithm 4.2) or the new algorithms (Algorithm 4.1 and 4.3). The goal is to understand what are the overheads and benefits of the new algorithms.

The experimental setup consists of five server processes placed in two sites—the local cluster and EC2 Ireland—as shown in Figure 4.5. Replicas

Algorithm	Operation	Min latency	Max latency
ABD Algorithm (prior work)	read	221	228
	write	111	120
Algorithm 4.1 (new)	read	221	231
	write	113	121
Algorithm 4.2 (prior work)	execute	221	231
Algorithm 4.3 (new)	execute	222	231

Figure 4.6: Round-trip latency with no congestion (in ms).

1, 2, 3 are used as the replica set, with replica 1 placed locally and replicas 2 and 3 placed in Ireland. The local replicas consist of replicas 1, 4, 5, which are in the local cluster. The client is co-located with replica 1. We use the network-based prioritization scheme available in the IP router.

In each experiment, the client issues a series of operations of a given type on a 1 KB object for 20 seconds, and we measure the latency of those operations. For the RW algorithms (ABD and Algorithm 4.1), the operation types are *read* or *write*; for the RMW algorithms (Algorithms 4.2 and 4.3), the operation type is *execute*.

Overhead under no congestion

The new algorithms are designed by deconstructing some existing algorithms to prioritize critical fields in their messages. This deconstruction increases the number of communication phases of the algorithms, and raises the question of whether they would perform worse than prior algorithms. To evaluate this point, we run an experiment where there is no congestion in the network and we compare performance of the different algorithms.

The results are shown in Figure 4.6. We find that the algorithms perform similarly, which indicates that the overhead of the extra phases in the new algorithms are not significant. This result confirms the analysis in Section 4.4.1 when $\delta \ll d$.

Benefit under congestion

In the next experiment, we evaluate the performance of the algorithms under network congestion to understand the benefits of prioritizing critical messages in the new algorithms.

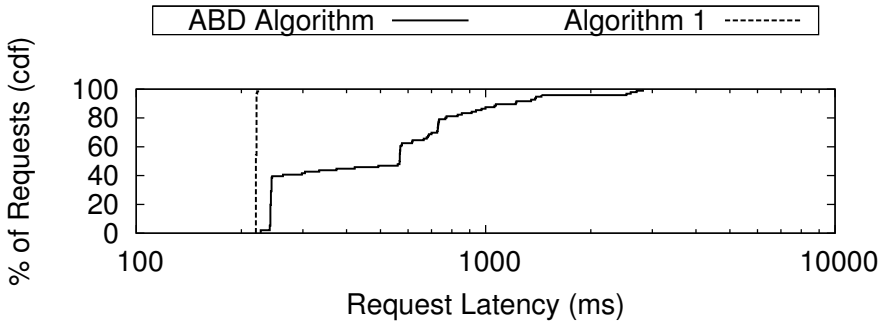
The results are shown in Figure 4.7. Note that the x-axis has a logarithmic scale. As can be seen, the new algorithms perform significantly better than the equivalent prior algorithms. The continuous congestion causes large latencies in the execution of the prior algorithms. The difference in median latency is over 300ms for all operations. Perhaps more significantly, the differences in the higher percentiles are large. The difference at the 90th percentile for read and write operations is nearly 1s, while for the execute operation it is over 1s. This is particularly relevant because online services tend to have stringent latency requirements at high percentiles: for instance, in Amazon's platform, the latency requirements are measured at the 99.9th percentile [64]. With the use of priority messages, the new algorithms are better suited for satisfying such requirements.

4.6.4 Effect on a web application

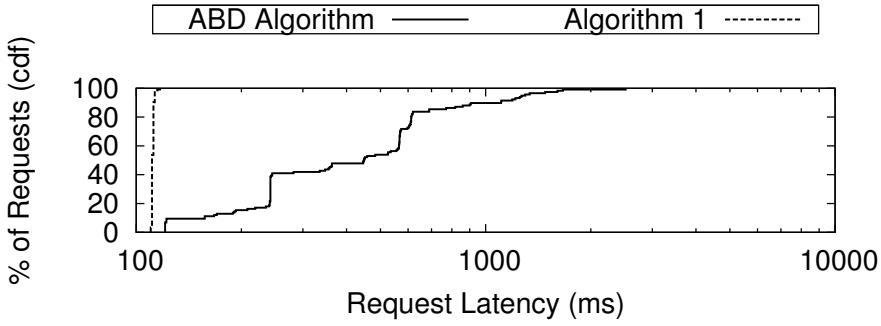
We now consider how the new algorithms in Vivace can benefit a real web application. We use Vivace as the storage system for a Twitter-clone called Twissandra [3], which is originally designed to use the Cassandra [1] storage system. We replace Cassandra with Vivace, to obtain a system that employs the new algorithms with prioritization. More precisely, Twissandra uses Cassandra to store account settings, tweets, follower lists, and timelines. In the modified Twissandra, we store account settings and tweets in Vivace RW objects, and we store follower lists and timelines in Vivace RMW objects.

We evaluate the benefit of Vivace's algorithms, by measuring the latency of common user operations in Twissandra. As in Section 4.6.3, we configure Vivace to use the prior and new algorithms, and compare the difference in performance.

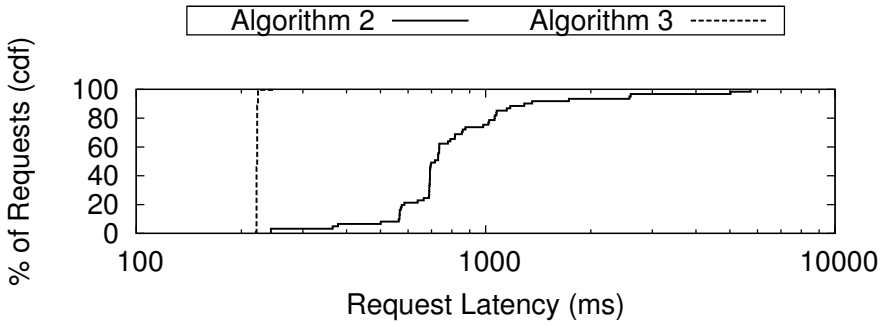
We run the experiments with two sites—the local cluster and EC2 Ireland—using the IP router to provide network-based prioritization. A load generator issues a sequence of 200 requests to Twissandra of a given type, one request at a time. We consider three request types: (R1) post a new tweet, (R2) read



(a) Read Algorithms



(b) Write Algorithms

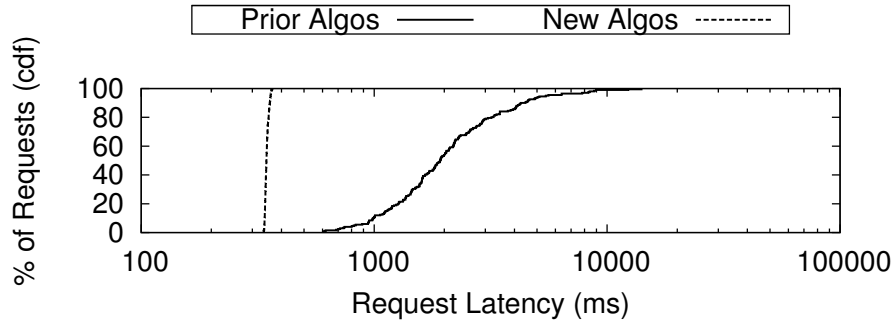


(c) Execute Algorithms

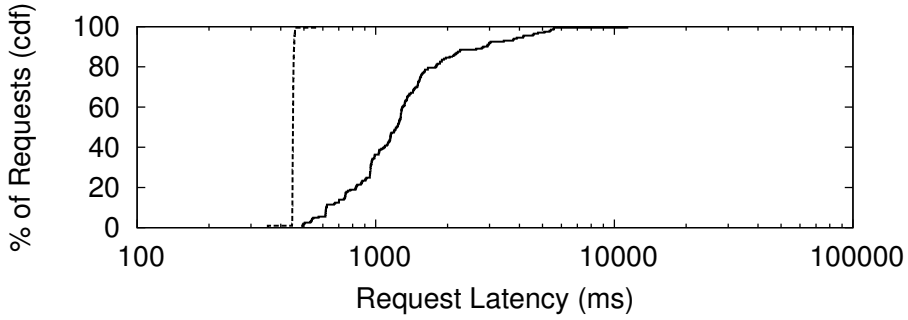
Figure 4.7: Latency of Vivace under congestion.

the timeline of a single user, and (R3) read the timeline of a user's friends. Each of these application requests result in multiple Vivace requests. We measure the latency it takes to process each request while the network is congested with background traffic generated by two machines running iperf (as in other experiments).

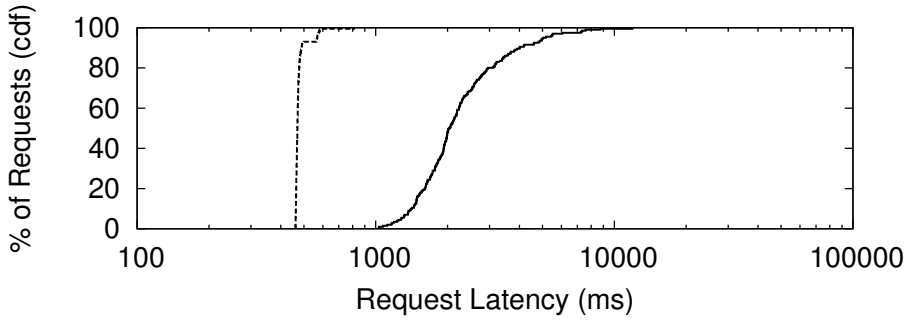
The results are shown in Figure 4.8. As can be seen, when Vivace is configured to use the new algorithms, the system is much more resilient to congestion. With the prior algorithms, latency for user operations often grew



(a) Post Tweet (R1)



(b) Read User Timeline (R2)



(c) Read Friends Timeline (R3)

Figure 4.8: Latency of Twissandra-Vivace under congestion.

well above 1 second, with a median latency of 2.0s, 1.2s, 2.0s, and maximum latency of 14.1s, 11.3s, 11.9s, for requests of types R1, R2, R3, respectively. In contrast, with the new algorithms, the median latency was 0.3s, 0.4s, 0.5s, and maximum latency was 0.4s, 0.6s, 0.8s, for the same request types, respectively—showing a significant improvement of using the new algorithms of Vivace under congestion.

4.7 Related Work

There has been a lot of work on distributed storage systems in the context of local-area networks (e.g., [86, 75, 118, 134, 68, 41, 51, 22, 80, 39, 99, 126, 105]). In contrast, we are interested in a geo-distributed data center setting, which comprises of multiple local-area networks connected by long-distance links. There is also work on distributed file systems in the wide area [131, 108, 119], which is a setting that in some ways resembles geo-distributed systems. These distributed file systems provide a weak form of consistency that requires conflict resolution [108] or other techniques to handle concurrent writes at different locations [131, 119]. In contrast, we provide strong consistency (linearizability), and our focus is on performing well despite congestion of remote links. Dynamo [64] is a key-value storage system where replicas can be located at multiple data centers, but it provides only relaxed consistency and applications need to deal with conflicts. Cassandra [1] is a storage system that combines the design of Dynamo [64] with the data model of BigTable [51]; like Dynamo, Cassandra provides relaxed consistency. PNUTS [61] is a storage system for geo-distributed data centers, but it too resorts to the technique of providing relaxed consistency to address performance problems. COPS [104] is a key-value storage system for geo-distributed data centers, which provides a consistency condition that is stronger than eventual consistency but weaker than strong consistency, because it allows stale reads.

There has been theoretical work on algorithms for read-write atomic objects or arbitrary objects (e.g., [76, 40, 77, 117, 107]). There has also been practical work on Byzantine fault tolerance (e.g., [48, 36, 93, 60, 82]) which considers the problem of implementing a service or state machine that can tolerate Byzantine failures. These algorithms were not designed with the

geo-distributed data center setting in mind. In this setting they would see long latencies under congestion, because processes send large messages across long-distance links in the critical path.

In the context of wide-area networks, there have been proposals for more efficient protocols for implementing state machines. The Steward system [42] builds Byzantine fault tolerant state machines for a system with multiple local-area networks connected by wide-area links. They use a hierarchical approach to reduce the message complexity across the wide-area. Mencius [106] is another system that builds a state machine over the wide-area. The use of a multi-leader protocol and skipping allows the system to balance message load according to network conditions. Hybrid Paxos [67] is another way to reduce message complexity. It relies on the knowledge of non-conflicting commands as specified by the application [113, 37, 98]: for instance, commands known to be commutative need not be ordered across replicas, allowing for faster processing. Steward, Mencius, and Hybrid Paxos can reduce or amortize the bandwidth consumed by messages across remote links. Yet, the systems can experience high latency if congestion on these links reduces the available bandwidth to below the message load. Our work addresses this problem by deconstructing algorithms and prioritizing a small amount of critical information needed in the critical path. As long as there is enough bandwidth available for the small fraction of load produced by prioritized messages, bursts of congestion will not slow down our algorithms.

PRACTI [47] separates data from control information to provide partial replication with flexible consistency and data propagation. Vivace also separates certain critical fields in messages, but this is done differently from PRACTI for many reasons. First, Vivace has a different purpose, namely, to improve performance under congestion. Second, Vivace uses different algorithms, namely, algorithms based on majority quorum systems, while PRACTI is based on the log exchange protocol of Bayou [125]. Third, Vivace provides a different storage service to clients, namely, a state machine [120], while PRACTI provides a more limited read-write service.

Megastore is a storage system that replicates data synchronously across multiple sites. Unlike Vivace, Megastore supports some types of transactions, but it has no mechanisms to cope with cross-site congestion.

Other related work includes peer-to-peer storage systems, which provide weak consistency guarantees rather than linearizability.

4.8 Summary

In this chapter, we presented Vivace, a distributed key-value storage system that replicates data synchronously across many sites, while being able to cope with congestion of the links connecting those sites. Vivace relies on two novel algorithms that can overcome congestion by prioritizing a small amount of critical information. In our experiments, deployed across two sites, we found Vivace avoids delays experienced under congestion by prior algorithms of up to a second or more for read, write, and execute operations.

We believe that the volume of data across data centers will increase in the future, as more web applications become globalized, which will worsen the problem of congestion across sites. But even if that does not happen, Vivace will still be useful, by allowing remote links to be provisioned less aggressively. More broadly, we believe that geo-distributed systems that make judicious use of prioritized messages will become more relevant, not just for storage systems as we considered here, but also in a wider context.

Chapter 5

Natjam: Strict Queue Priority in Hadoop

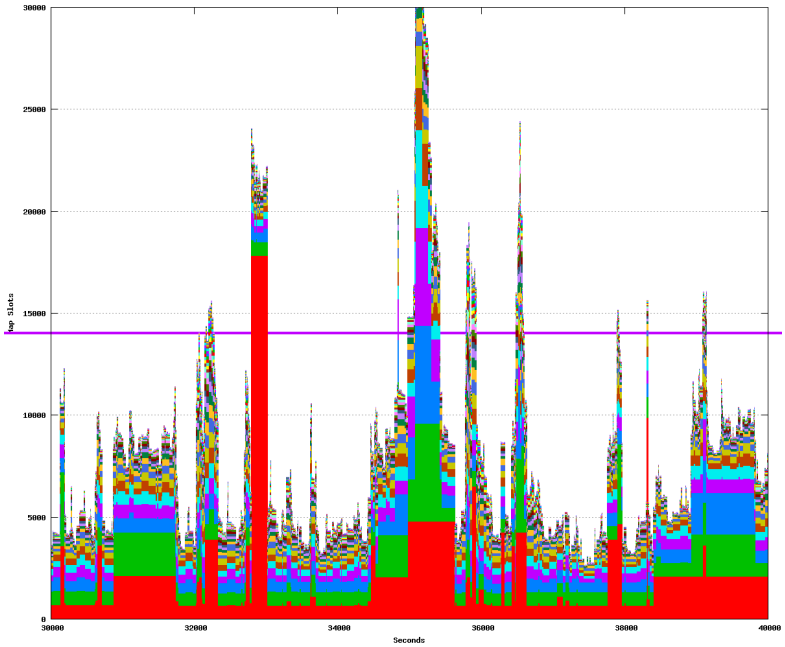
In this chapter, we discuss the prioritization of production jobs within a Hadoop cluster. Our solution, Natjam, provides job priority through a suspend and resume mechanism.[1] Our design of suspend and resume allows production jobs to immediately gain resources, while minimally impacting the completion time of research jobs.

5.1 Motivation

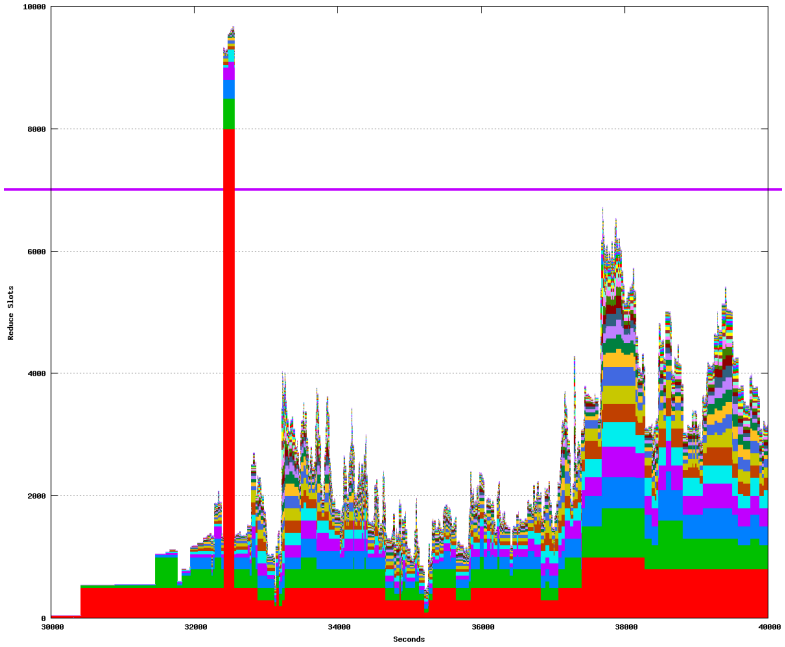
Organizations use the MapReduce [62] distributed computation framework, and its Hadoop open source implementation [19], to process enormous amounts of data. Many of these computations are long-running batch jobs for which Hadoop was originally designed. We call these *research* jobs. Increasingly, organizations run time-sensitive computations on these same datasets. These jobs also run as Hadoop jobs [52], or on distributed real-time analytics systems [30]. We call these *production* jobs.

These organizations require that their production jobs receive priority over research jobs. This is because the running time of a production job can directly affect revenue. Consider the workload of an advertisement provider, which contracts with each of its advertisers to receive payment for a fixed number of customer click-throughs. The ad provider has many ad servers placed at various geo-distributed locations, to reach a global customer base. To keep track of clicks and analyze them, logs of click data containing time, user, IP address, etc. are transferred to a Hadoop cluster. A production job runs periodically every five minutes, counting the number of clicks per advertisement from the most recent logs. If this information is inaccurate or dated, the ad provider may lose revenue by showing an ad that has already

[1] Natjam is the Korean word for nap.



(a) Map tasks



(b) Reduce tasks

Figure 5.1: Trace of 10K seconds in a production cluster.

reached the number of clicks that the advertiser has agreed to pay for, instead of an alternative ad that would have resulted in payment from another advertiser.

On the other hand, research jobs are not as time-sensitive, but they too affect revenue. They require a cluster with a large capacity, to complete in a reasonable time. At the ad provider, an analyst runs a research job to identify more lucrative ad placement patterns, by computing a machine learning algorithm on a larger subset of the historical click data (e.g. over the entire past year). Because the dataset is large, and the algorithm is complex, the job will take an extremely long time to process without a large cluster that has enough capacity to provide high throughput.

In today's organizations, a popular solution for achieving strict priority between production and research jobs is to provision physically-separate production and research clusters, and tightly restrict the workloads that are allowed on the production clusters. However, this conservative approach has two disadvantages: a production workload may: (a) inefficiently utilize resources, by having long periods when resource usage is below the cluster capacity, while at the same time it (b) has periods of time when the resource requirement is larger than the cluster capacity, which can lead to extended job completion times.

Figure 5.1 is a 10K second trace of job submission and runtimes that captures both of these disadvantages in a production cluster.[2] The number of slots requested at a given time, on the y-axis, is plotted against time, on the x-axis. Each block of a different shade depicts a job – the height shows the number of task slots requested, the left-most point shows the time that job was submitted, and the width shows the average task execution time observed. The dark horizontal line shows the total capacity of task slots in the cluster. In this trace, the two disadvantages are shown as follows: (a) time intervals with a white area under the horizontal line show when cluster utilization is below capacity; and on the other hand, (b) time intervals with a shaded area above the horizontal line show when the resource requirement for the submitted jobs is larger than the cluster capacity.

To deal with both of these disadvantages, we design Natjam, which provides strict priority between queues in a single cluster, allowing production and research jobs to co-exist. A combined cluster with an effective strict queue priority scheme can have clear benefits to efficiency: (a) research jobs can fill in areas of under-utilization, running as low-priority jobs that yield

[2]This plot is reproduced with permission from Yahoo.

resources to incoming high-priority production jobs; (b) combining the capital investment of separate clusters into a single cluster can raise the capacity, increasing the total available resources for research jobs, and especially raising the amount of resources that production jobs can request before being constrained by the cluster capacity.

5.2 Solution Approach and Challenges

Our approach to achieving strict queue priority is to preempt low-priority tasks, through a *suspend and resume* mechanism. Our support for suspend allows the resource scheduler to revoke task slots from research jobs when production jobs require them. When resources become free again, suspended tasks are resumed from where they left off. With this support in place, we are able to assign research jobs to idle slots.

For example, in Figure 5.1b, we could add research jobs between times 30000s and 32200s, when the production jobs take up less than a third of the reduce slots. The tasks running in these slots can then be suspended when production jobs request more resources.

This suspend/resume approach can preserve work at research jobs while quickly allocating resources to production jobs. In contrast, support for naive preemption through killing will not preserve work, which can lead to wasted work and low effective utilization. Consider what happens to research jobs that are running just before the spike of production job submissions at 32200s. If tasks are simply killed at this spike, the work done at each task is lost. Thus, the effective utilization, i.e., the progress made toward job completion, would dip to zero for the tasks that were preempted.

On the other hand, running without preemption can delay jobs while they wait for resources. According to [138], in a Facebook cluster, the median Reduce task is 231s. As a result, only 3 (of 3100) Reduce slots are freed up every second. Based on these statistics, jobs executed on well-utilized clusters must wait for a long time before being allocated all its requested reduce slots. On a fully loaded cluster, a small job with 30 reduces would wait on average 10 seconds, a medium job with 300 reduces would wait 100 seconds, and a large job with 3000 reduces would end up waiting 1000 seconds.

Thus, we propose that a low-priority Reduce task should be *suspended*

– saving all in-progress data, and then freeing up memory resources – and when resources again become available for this task, it should be *resumed* – picking up where it left off by reading from the previously saved in-progress state. We focus solely on Reduce tasks, because: (a) waiting to get allocated a Reduce task can take longer than for a Map task, leading to a large delay when executing time-sensitive jobs, and (b) each Reduce task runs longer than a Map task, leading to a lot of lost work if these tasks are killed. This is less of a concern for Map tasks, for which the median task in [138] is 19s long and 27.1 (out of 3100 total) Map slots are freed up every second.

Natjam's Reduce task suspend/resume approach, must tackle several major challenges:

1. It must avoid excessive overhead when suspending tasks. A high overhead suspend would require production tasks to wait for resources, delaying their execution.

2. It must keep task resume overhead small. A large resume overhead would impact the execution of research tasks, because in this case their execution would be delayed waiting to restart.

3. It must optimize the tasks evicted by Natjam to optimize for job completion times. Hadoop users can only make use of computation results when their submitted jobs have completed, i.e., all the tasks in the job have completed. Accordingly, the metric that Hadoop users are interested in is job completion time, not the individual task completion times. Thus, we look for eviction policies with the goal of choosing tasks and jobs that will have the least impact on job completion times.

The rest of this chapter covers how we have designed Natjam to meet these challenges (Section 5.3–5.8), and our evaluation of those decisions (Section 5.9).

5.3 Design: Overview

In the following sections, we discuss the design of Natjam. We start with a brief look at Hadoop's architecture and Capacity Scheduler (Section 5.4).

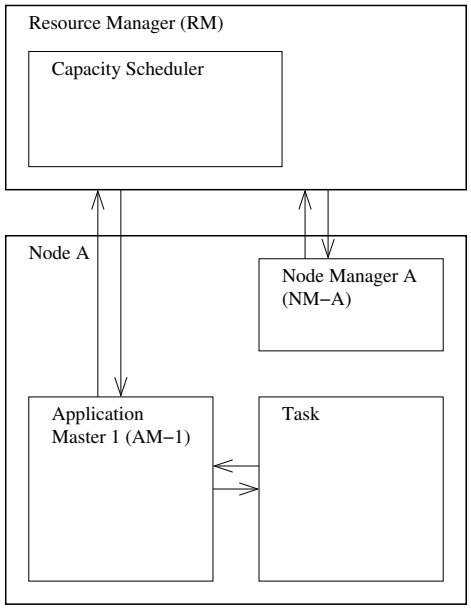


Figure 5.2: Hadoop YARN architecture depicted with a Resource Manager and single slave node. The arrows depict periodic heartbeat messages. Although shown here, an Application Master and its Tasks do not have to reside on the same node.

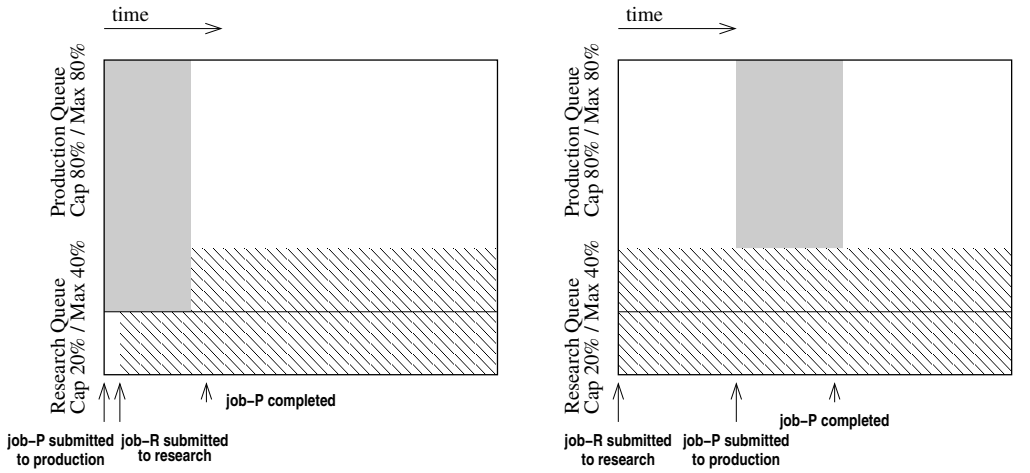
Next, we discuss Natjam's suspend/resume architecture, showing how we design an efficient suspend/resume mechanism on top of Hadoop (Section 5.5). We then discuss details of Natjam's implementation within Hadoop (Section 5.6). We then look at Natjam's eviction policies, which choose the jobs and tasks to suspend within a research queue (Section 5.7). Finally, we present possible extensions to the design (Section 5.8).

5.4 Design: Background

In this section, we present parts of the Hadoop design that are essential to understanding Natjam. Natjam is designed to require only minimally intrusive changes to the Hadoop design. For example, we piggyback off existing periodic messages to remain scalable, and our configurations remain compatible with Hadoop's Capacity Scheduler.

5.4.1 Hadoop YARN architecture

The Hadoop YARN architecture [8] divides resource management from application management. The architecture scales because resource management



(a) Production job followed by research job

(b) Research job followed by production job

Figure 5.3: Queue allocations given by the Capacity Scheduler.

deals solely with a finite amount of resources, while for each application, an independent process is spawned to manage it. In addition, resource management is done through periodic heartbeat messages to avoid overloading these components when there is lots of activity.

We depict key components in Figure 5.2. A single Resource Manager (RM) performs resource management with the help of one Node Manager (NM) per node. The RM receives status updates from each NM about resource usage and availability at its node via periodic heartbeats.

Resource usage and allocations are accounted in units of memory called *containers*. Each Map or Reduce task is allocated a container to execute on.

A container is also allocated to a single Application Master (AM) per application, which performs job management. The AM has two main roles. First, it requests and receives from the RM container allocations needed for tasks for its. The AM assigns Map and Reduce tasks to each container it receives, and sends launch requests to the NM of each container. Second, it deals with job management. It monitors running tasks via heartbeats, and performs complex functions, such as creating speculative tasks for slow tasks.

5.4.2 Hadoop Capacity Scheduler

The RM follows the allocation decisions of a scheduler to arbitrate between applications. Under the popular Capacity Scheduler, applications are submitted to one of multiple queues. Each queue is meant to receive a guaranteed capacity of resources, while some are allowed to expand beyond their guaranteed capacities.

To configure these queues, an administrator configures two capacities per queue, a fixed capacity and a maximum capacity. Figure 5.3 shows how the Capacity Scheduler will split workload onto queues given two sequences of job submissions. For both sequences, there are two queues, production and research, that are given a capacity/max-capacity of 80%/80% and 20%/40% respectively. In the first scenario, job-P is submitted to the production queue, and receives 80% of resources. It maintains this guaranteed capacity until it finishes, even after job-R is submitted to the research queue.

However, the next scenario shows how the Capacity Scheduler can fail to guarantee capacities. The Capacity Scheduler can allow a long-running application to indefinitely hold on to resources above its fixed capacity, even when jobs later request resources in queues that are under their capacity. This is shown by job-R in the research queue, which holds on to 40% of resources even after the job-P is submitted to the production queue. This limits job-P to 60%, less than the queue's capacity of 80%, which can slow down the job. Because queues cannot scale down the capacity the currently hold, it is extremely difficult for an administrator to optimize the queue configuration to satisfy both capacity sharing and capacity expansion.

5.4.3 Hadoop container allocation example

We conclude this introduction to the Hadoop architecture with an example of a container allocation. The AM for Job 1 (AM-1) requires a container. The example starts directly after the NM of node A (NM-A) has learned that a container has become empty because a task has completed.

Step 1. On NM-A's next heartbeat, it alerts the RM that the container has completed.

Step 2. When AM-1 sends its next heartbeat, it is now allocated the open

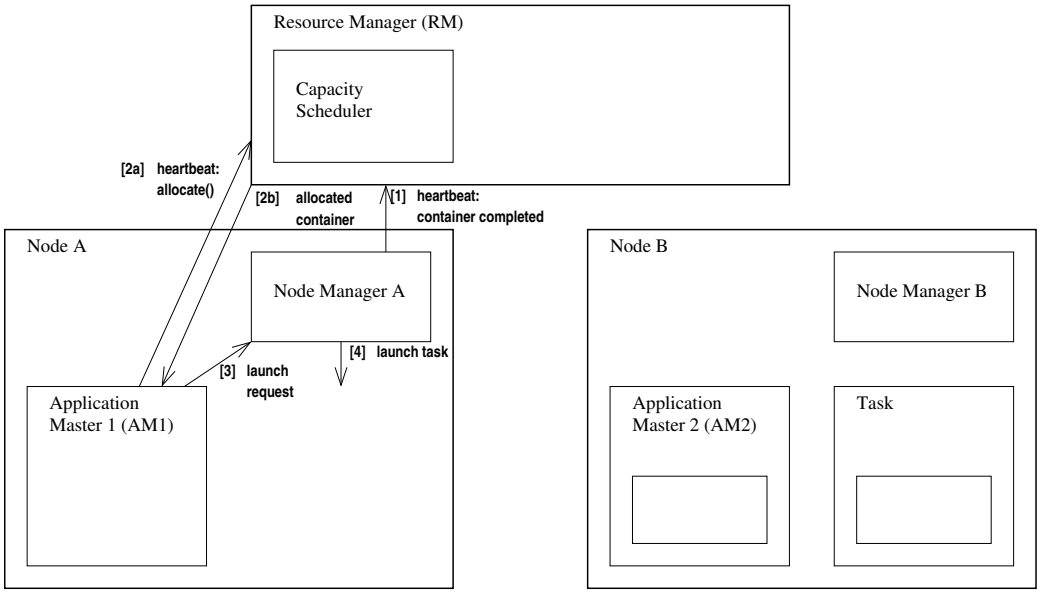


Figure 5.4: Container allocation in Hadoop.

container.

Step 3. AM-1 sends NM-A a request to run a task on the container.

Step 4. NM-A launches the task on the container.

This example shows how the RM, NM and AMs extensively use periodic heartbeat messages for communication. We show a similar example in Section 5.5.2 that highlights the changes made to Hadoop in Natjam.

5.5 Design: Suspend/resume Architecture

Natjam augments the Hadoop YARN architecture. It adds three new components to the Hadoop YARN architecture, as depicted in Figure 5.5. Briefly, these are:

- *Preemptor:* As a component inside the Capacity Scheduler, it finds queues to preempt resources from, by periodically comparing queue capacities and allocated resources. If a queue is chosen, this component also makes the decision of which jobs to suspend within that queue.
- *Releaser:* As part of the AM, decides which tasks to suspend among those running in the current job.

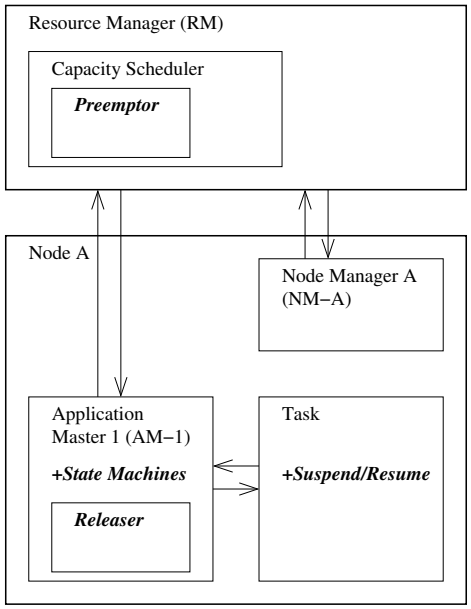


Figure 5.5: Natjam architecture depicted with a Resource Manager and single slave node. The arrows depict periodic heartbeat messages. New components and modifications are shown in bold italics.

In addition, changes are made within existing YARN components. Briefly, the major changes are:

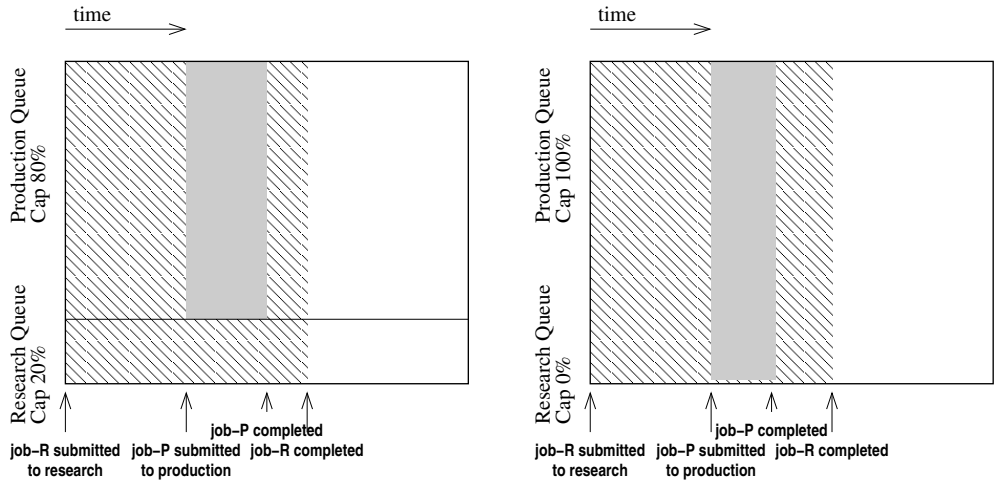
- *AM State Machines:* The job, task, and task attempt state machines are modified to save and use suspend states.
- *Reduce Task:* On a suspend, the reduce task saves state and sends it to the AM. On a resume, a reduce task looks for local input data, and skips past keys computed by previously suspended task attempts.

We go over these modifications in more detail in this section.

5.5.1 Natjam Capacity Scheduler and Preemptor

The Preemptor is a new component inside the Capacity Scheduler. Its role is to make suspend requests to AMs, thus forcing jobs in a queue to scale down its usage when it is taking up resources from another queue.

Figure 5.6a depicts the Capacity Scheduler's operation when the Preemptor is used. The max-capacity setting is no longer used, while guaranteed capacity is set to 80% and 20% at the production and research queues, respectively. When job-R is running on the research queue, it takes up the



(a) 80-20 (b) 100-0

Figure 5.6: Queue allocations given by Natjam.

entire cluster capacity. Then, when job-P is submitted to the production queue, job-R scales down to the research queue's guaranteed capacity of 20%, allowing job-P to take the remaining 80%.

This immediate scale-down allows for more flexible configurations. In Figure 5.6b we allow the production queue to take 100% of the capacity. This 100%/0% capacity configuration strictly prioritizes jobs in the production queue. If job-P is large enough, as depicted in this example, it takes over all resources when submitted, and gives them up when it is done. If job-P is small, it takes only as much resources as needed, and job-R takes the rest.

The examples depicted in Figure 5.6 show simplified jobs with equal task running times, under an ideal Natjam scheduler with no overheads. In the evaluation in Section 5.9, we use the same 100%/0% configuration, and show how the execution times behave with jobs that have a variable number of tasks and task running times, on our implementation of Natjam.

5.5.2 Container suspend and allocation example

We now show where the changes in Natjam fit in, by taking a look at the steps involved in suspending a task. In our scenario, a production job (Job 1) is submitted to a cluster already running a research job (Job 2) that takes up the entire cluster. A container allocation request is made by Job 1, forcing a container in Job 2 to be suspended, followed by Job 1 receiving this container.

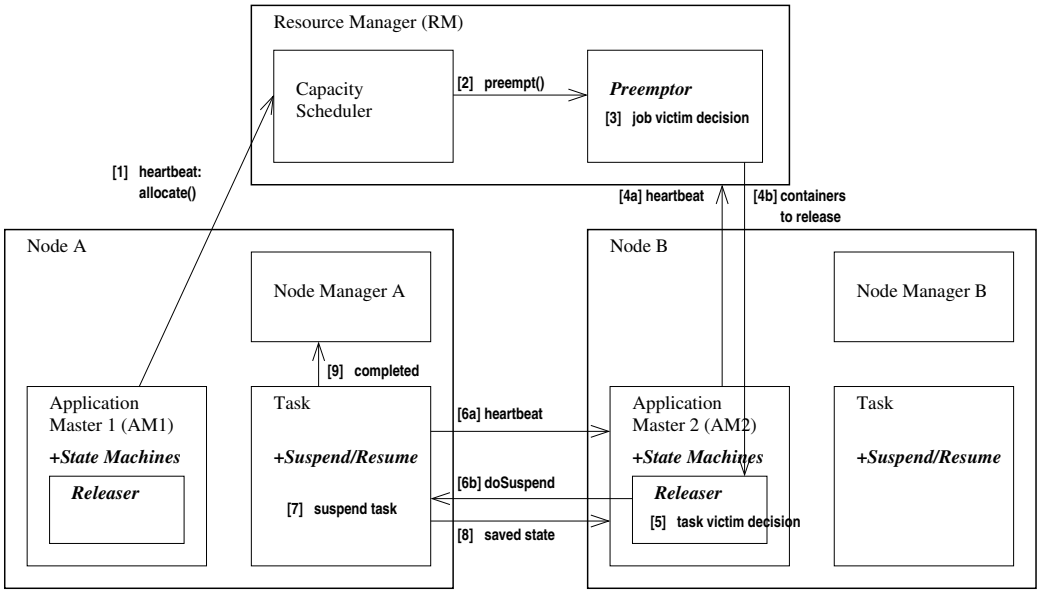


Figure 5.7: Container suspend in Natjam. After suspend, the completed container is allocated to AM-1, using the steps in Figure 5.4.

The sequence of steps is depicted in Figure 5.7, and explained here.

Step 1. On AM-1's heartbeat, it asks the RM to allocate a container. The RM does not immediately have a resource available.

Step 2. When the Preemptor runs its periodic search, it decides that AM-2 should release resources.

Step 3. On AM-2's next heartbeat, it receives the number and type of containers to be released.

Step 4. To satisfy this request, the Releaser finds a suitable victim task in Job 2.

Step 5. When the running task sends its heartbeat, it receives a request to suspend.

Step 6. The running task immediately suspends.

Step 7. It then sends the saved suspend state via RPC to AM-2.

Step 8. The task is now done with its container, and indicates to NM-A that it has completed, then promptly exits.

This ends the Natjam-specific steps. For completeness, we continue with the steps for allocating the newly open container to AM-1. These steps are identical to Hadoop shown in Figure 5.4.

Step 9. On NM-A's next heartbeat, it alerts the RM that the container has completed.

Step 10. When AM-1 sends its next heartbeat, it is now allocated the open container.

Step 11. AM-1 sends NM-A a request to run a task on the container.

Step 12. NM-A launches the task on the container.

5.5.3 Suspend state and intermediate data

When suspended, Reduce tasks send state information to the AM, to be used when the same task is resumed. We keep this state information small. This allows fast suspend times, and keeps the load on the AM from becoming large. The suspend state saved by Natjam is:

- List of suspended container IDs
- Key counter
- Reduce input paths (stored locally)
- Hostname

The suspend state can be small, because it is used as an index to larger intermediate data, which is already stored within Hadoop [92]. The intermediate data used by Natjam is:

- Reduce inputs, stored at a local host.
- Reduce outputs, stored on HDFS.

We now show how this saved state and intermediate data is used during suspend, resume, and commit.

Suspend

The Reduce task keeps track of two pieces of state while it is executing the Reduce stage. First, it keeps track of paths to files in the local filesystem that hold Reduce input. Second, it keeps track of the key counter, i.e., the number of keys that have been processed by the Reduce function so far.

When a suspend request is received from the AM, the Reduce task first waits for the currently executing key to finish execution. It then writes the input file paths to a local log file. The Reduce output file on HDFS that holds the output computed up this point is closed. We call this a partial data file. Finally, the Reduce task sends the current key counter to the AM. The task process is then exited to make its container available to other jobs.

Resume

The AM launches a task as a resumed task by sending the saved state as launch parameters to the NM. If the resumed task is assigned on the same host as the last suspend, the paths for Reduce input files are parsed from local disk, and read in as Reduce input. If the resume is done on a different host, the Reduce input is assembled from Map task outputs, just like a new task. (Our extension in Section 5.8.1 allows Reduce input to be transferred instead of using Map task outputs.)

Once the Reduce inputs are assembled, the Reduce creates a new output file on HDFS. It then skips over a number of input keys, equal to the key counter. The Reduce processing then proceeds like a normal Reduce task.

Commit

In Hadoop, Reduce tasks write their output directly to HDFS. To ensure that users see only completely processes output, the output is first written to a file within a temporary directory with the name of its container ID. Once the task is finished, it is then committed, i.e., the file is moved to a user-accessible output directory.

For a resumed Reduce task, this commit phase does some extra work. It finds all partial output files by using the container ID of all previously suspended tasks to make recover temporary directory paths. Then, along

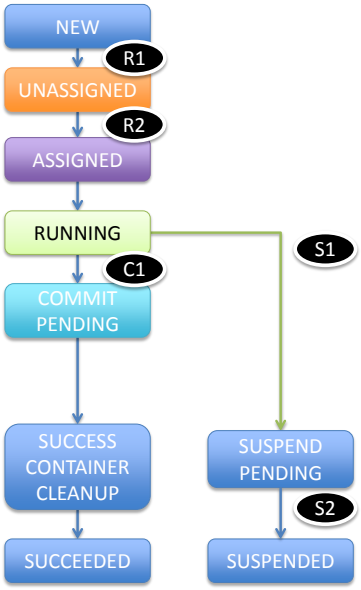


Figure 5.8: Task Attempt state machine at Application Master. (Failure states are omitted.)

with the recently written output, it moves all the partial output to the final output directory.

When moving these partial output files, the sequence in which the partial outputs were written are indicated by an integer suffix in the filename. When the outputs are read sequentially in this order, they are guaranteed to match the output of the same task that executed without any interruption, as long as the Reduce function does not use state saved across keys. (Our extension in Section 5.8.2 extends this guarantee to Reduce functions that save state across keys, which we call stateful Reducers.)

5.6 Design: Implementation

In this section, we take a more detailed look at Natjam's implementation. Our implementation builds off of Hadoop components, and uses periodic messages and operations to avoid bottlenecks.

5.6.1 Application Master state machine

In Hadoop, the AM uses state machines to keep track of the job and its tasks. A state machine is kept for each job, and for each task within the job, and for each task attempt within the task. To execute a job, a job state machine is created, along with its individual tasks. For each remote task process, a task attempt state machine is created to: assign the container, setup execution parameters, monitor progress, and commit output. The current state of the task attempt are used when making task-wide and job-wide decisions.

In Natjam, suspend and resume takes place during the Running state of both the job and task state machines. These state machines are changed to handle suspend and resume information, but their structure remains the same as Hadoop.

For the task attempt state machine, there are two new states, Suspend-pending and Suspended. In Figure 5.8, we show this modified state machine. The task attempt has a state of Suspend-pending when it wishes to suspend a task but has not received confirmation from the local task that the suspend has been performed (Steps 5-7 of Section 5.5.2). It is transitioned to Suspended when it has received the saved suspend state (Step 8).

The Natjam task attempt state machine makes use of, and passes along saved suspend state during its state transitions. In particular, the following transitions, shown in Figure 5.8, deal with suspended state:

- S1: The AM asks the task to suspend, and requests its suspend state.
- S2: When the AM receives the suspend state, it saves it within the task attempt state machine.
- R1: When the task is resuming, the AM asks the RM for a container with the hostname of the previously suspended task attempt.
- R2: Once a container is allocated, the AM passes along saved state, which is passed into the new task.
- C1: On commit, the state machine finds partial output file paths in HDFS by constructing them from the previous container IDs.

5.6.2 Preemptor

The Preemptor is implemented as a thread within the Capacity Scheduler, and it runs at the RM. Our Preemptor implementation suspends tasks, but also kills tasks if resources have not been reclaimed for a timeout period, for example because of a misbehaving AM. It is based on the Hadoop preemptor, which only implemented killing of tasks, and has since been removed [21].

The Preemptor periodically runs a reclaim algorithm, that chooses which queues to reclaim resources from. Intuitively, a queue can reclaim resources from another, when: (Condition 1) the queue is under its capacity, and (Condition 2) there is not enough idle resources to satisfy its outstanding resource requests, up to its capacity.

However, an implementation using this formulation may result in more resource reclaim requests than necessary. This is because the suspension of a container and its subsequent allocation to another job carries a delay. For example, in Section 5.9.3, we observed an average delay of 4.7 seconds between the first allocation request of a production job and when it gets satisfied. During this delay, the Preemptor may run multiple times, and create an identical reclaim request each time.

To prevent identical reclaim requests, we keep track of them until resources are allocated at the requesting job. We then change the condition for when a queue can reclaim resources, to take into account the resources that are currently being reclaimed:

- (Condition 1') including outstanding reclaim requests it has made, the queue is still under its capacity, and
- (Condition 2') there is not enough idle resources including outstanding reclaim requests it has made, to satisfy its outstanding resource requests, up to its capacity.

The Preemptor keeps track of outstanding reclaim requests by updating a list of reclaim requests per job. The amount of outstanding resources at a queue is calculated by summing up the requests stored in the reclaim lists for all its jobs. These lists are updated as follows. Reclaim requests are added to a job's reclaim list when a suspend request is sent to an AM due to that job's resource request. A reclaim request is removed from the job's reclaim

list each time a container allocation is given to that job in the Capacity Scheduler.

Once a queue is selected to reclaim resources from, the Preemptor must decide which job to remove the resources from. The Preemptor makes this choice based on one of the job-level eviction strategies discussed in Section 5.7.2.

Killing containers

Killing a container is a last resort, when an AM has remained unresponsive to a suspend request for longer than a configured killing interval. The decision to kill a container is made if a reclaim request has remained in the reclaim list for longer than the killing interval. When this happens, a kill request is sent directly to an NM to kill the container. By bypassing the AM, this ensures that a container will indeed be killed.

Accounting for delays in killing is done with the one more level of lists, named expired lists. When a kill request is sent, the reclaim request is now added to the expired list, and remains there for an expired interval, at which point it is assumed the container was killed and the request is removed. Killing a container can potentially increase job execution time for the victim by a large amount. Yet killing should be extremely rare, likely due to problems at the AM or task. For example, with a killing interval set to 12 seconds, we have never had to kill tasks in our experiments.

5.6.3 Releaser

The Releaser decides at the AM which tasks to suspend. To implement the task eviction strategies in Section 5.7.1, we need an accurate estimate of the remaining time of the tasks. Estimating task times from a task's observed progress during its execution has proven to be effective in Hadoop [137], so we use Hadoop's default exponentially smoothed task runtime estimator. Because jobs can have many tasks, it may not be feasible for tasks to be estimated on-demand, whenever a release request comes from the RM. To alleviate this concern, the AM periodically estimates all the tasks in the job, and uses the latest complete set of estimates for task selection.

5.7 Design: Eviction Policies

In this section, we look at Natjam's eviction policies. There are two kinds of eviction policies, dealing with tasks and jobs, respectively. Task eviction policies are implemented at the Releaser within the AM, and job eviction policies are implemented at the Preemptor within the RM.

5.7.1 Task eviction policies

The goal of a task eviction policy is to choose the order in which tasks within a research job are suspended. Our policies are based on the time remaining in a task. We showed how remaining time in tasks can be estimated at the AM in Section 5.6.3.

Time remaining in a task is important for two reasons.

1. Job completion time: The last task to finish in a job decides the job completion time. In other words, a long tail, or even a single task that finishes late will extend the job completion time until it has finished.

2. Cluster resource usage: A task that finishes early is able to release its used resources, for use by other jobs, earlier than a task that finishes late.

Based on these factors, we propose two contrasting eviction policies that optimize on either of the above.

Shortest Remaining Time (SRT)

With this policy, tasks that have the shortest remaining time are selected to be suspended. This policy optimizes on job completion time. This is because tasks suspended with SRT will finish early once they have been resumed. Thus, by selecting these tasks we are shortening the tail.

Note that SRT is different from the Shortest Job First scheduling strategy in traditional operating systems, which is optimal in terms of average job completion time. Instead of choosing to run the shortest task first, SRT chooses to delay it. Yet, this policy is effective in terms of job completion time in Hadoop. In fact, we have the following optimality result.

Theorem 5.7.1. *SRT results in an optimal job completion time for a victim job, if each suspended task within the job is resumed simultaneously.*

While suspended tasks are not likely to resume at the exact same time, this is a reasonable assumption because Hadoop does not have information about job submissions that will happen in the future.

This strategy is selfish, because it can delay the release of resources. The tasks selected by SRT are those that would have released resources early, had they not been suspended. Thus, the start time of tasks waiting for resources at another research job can be delayed by using this policy.

Longest Remaining Time (LRT)

In this policy, the tasks with the longest remaining time are chosen to be suspended. The strategy may help jobs waiting within the same queue receive resources. However, it may lengthen the tail, resulting in increased job completion times.

5.7.2 Job eviction policies

The goal of a job eviction policy is to choose the order in which jobs within a research queue are requested to suspend tasks. Our policies for selecting a job are based on the amount of resources held at the job, leading to three different policies: *Least Resources, Most Resources, and Probabilistically-weighted on Resources*. We discuss these policies below.

Least Resources (LR)

LR chooses to evict tasks from the job that holds the least resources. Tasks for small research jobs are evicted first, so large research jobs benefit.

On the other hand, LR is a harsh policy for small jobs. In fact, LR can cause starvation even from small production jobs. Consider a scenario on a cluster running with no spare capacity, where a research job with a single task is running, and a production job with a single task is re-submitted every time it completes. Because LR will choose the small research job to suspend each time the production job appears, it will suffer from starvation.

Most Resources (MR)

MR chooses to evict tasks from the job with the most resources. Small research jobs benefit.

If there is a single large job, and a small production job needs resources, evicting from only this job may be preferable. Doing so prevents the production job from delaying many other jobs. However, if a large production job is submitted instead of a small one, the large research job's completion time may be delayed for a long time.

Probabilistically-weighted on Resources (PR)

This policy chooses probabilistically among jobs, with the probability of choosing a job weighted by the resources it holds. As we discussed above, LR can starve small jobs, even with small production jobs. MR may force the largest job's completion time to increase by a large amount.

In contrast, PR avoids biasing tasks based on their job's size, i.e., assuming the task eviction policy is random, the chance of eviction for each task is identical regardless of its job. However, by spreading out evictions across jobs, this may result in many jobs being delayed by suspension, even when the number of suspensions is smaller than any single job.

In fact, restricting the number of victim jobs to a minimum may be desirable because of the following result.

Theorem 5.7.2. *Choosing the minimum number of victim jobs results in the shortest average job completion time, if tasks within each job require the same amount of processing time.*

However, the assumption that tasks within a job are the same length does not always hold. In this case, the effectiveness of job eviction policies also depend on the underlying task eviction policy, as explored in our evaluation in Section 5.9.5.

5.8 Design: Extensions

In this section, we discuss possible extensions to Natjam.

5.8.1 Rack-level locality when resuming reduce tasks

In our current implementation of Natjam, a resumed task only re-uses Reduce input files when it is assigned to the same node as the latest suspended task attempt. This repeats the network transfers from all Map tasks, which were already done in the previous task attempts. If this becomes a bottleneck, an alternative approach that we can apply is to save suspended state on HDFS.

Storing on HDFS will make the Reduce input globally accessible. When a resume task attempt is assigned, it can follow the same process used by Hadoop to assign Map tasks to local data: the first preference is the local node, followed by a node in the same rack, and finally any node in the cluster. Reduce tasks will be able to receive node-local or rack-local data with the same high frequency as Map tasks.

The copy to HDFS must incur low overhead. Since there is the backup strategy of reconstructing Reduce input from Map tasks, we can tell HDFS to store a single replica. HDFS will store the single replica within the writing node, so putting the data on HDFS involves only a local disk copy.

5.8.2 Suspending stateful reduce tasks

In Hadoop, a Reduce function may use state saved from computations done on previous keys. We call Reduces implemented in this way stateful Reducers. They break away from the purely functional mode of Reduce, where Reduce invocations on different keys are independent of one another. In limited cases, stateful Reducers may be preferred because they can make MapReduce applications more efficient. For example, in [101] saving state across keys allows a computation of relative frequencies that previously required a sequence of two MapReduce jobs to be expressed as a single job.

We can extend Natjam's API to call serialize and deserialize methods while suspending and resuming. A developer will only have to implement the serialize and deserialize methods to fit the particular Reduce function.

When a suspend is called, the inter-key state datastructures are serialized and then copied to HDFS. Storing on HDFS makes the serialized datastructures globally accessible. When a resumed task is assigned, Natjam deserializes the state before skipping the amount of the key counter to the current key.

In this way, the state at the suspended Reduce task is faithfully reproduced at the resuming task. The serialize and deserialize can run with little overhead, because the state to be saved can be very small.

As an example, we discuss the stateful Reducer used to compute relative frequency across word co-occurrences, from [101]. For this computation, the Reduce input keys are pairs of words. The application customizes input key ordering such that a special input key (w, Σ), that is used to compute the sum of all occurrences for w, appears just before the input pairs $(w, *)$ of words that co-occur with w. Each subsequent pair containing w is divided by this sum to produce the relative frequency output, and thus the only state that must be saved is the latest sum. To allow suspend and resume for this job, a developer will implement serialize and deserialize methods for just this integer value.

5.8.3 Supporting multiple priorities

In Natjam, we consider jobs with either of two priorities. A job is either a production job or research job. Where finer priority control is required [115], Natjam can be extended to support multiple priorities.

Multiple priorities behave in the following way. The highest priority queue takes up all its required resources, up to the cluster capacity. Each successive priority takes up all its required resources, up to the remaining cluster capacity.

The queues are arranged in a binary tree. Jobs are only submitted to leaf queues, while interior queues are used to restrict their children queue's capacities. The height of the tree is equal to the number of priorities. The configuration discussed in Section 5.5.1 is the specific case with height two, containing the root and its children: the high priority production queue and low priority research queue.

At the root of the tree is the root queue, configured with a guaranteed capacity of 100%. It has two children queues, a high priority queue and low priority queue. The high priority queue is given a guaranteed capacity of 100%, while the low priority queue is given 0%.

The low priority queue is a leaf queue, where jobs can be submitted to. It has the lowest priority. Except at the lowest level, the high priority queue

Job	# Reduces	Avg Time (s)
Research-XL	47	192.3
Research-L	35	193.8
Research-M	23	195.6
Research-S	11	202.6
Production-XL	47	67.2
Production-L	35	67.0
Production-M	23	67.6
Production-S	11	70.4

Figure 5.9: Jobs run during experiments and their average execution time across five runs.

has two children of its own, which again are configured with guaranteed capacities of 100% and 0%. At the lowest level, the high priority queue is a leaf queue. This queue has the highest priority.

5.9 Evaluation

In this section, we present our evaluation of Natjam, which is implemented based on Hadoop 0.23. We discuss microbenchmarks (Section 5.9.1 to 5.9.5), and small and large Natjam deployments (Section 5.9.6 to 5.9.8), which are run using trace-driven workloads taken from both a Yahoo production cluster and research cluster.

We first describe the experimental setup of the microbenchmarks run on CCT [25] (Section 5.9.1). We then show the overall benefits of Natjam, comparing job execution times to ideal settings and traditional Hadoop techniques (Section 5.9.2). After that, we discuss the overheads of the system (Section 5.9.3). We then investigate the effectiveness of task eviction policies (Section 5.9.4) and job eviction policies (Section 5.9.5).

We then look at the deployment results. We describe the setup of these deployments and their trace-driven workloads (Section 5.9.6). We then discuss small-scale results run on CCT (Section 5.9.7), followed by large-scale results executed on a 250-node Yahoo cluster (Section 5.9.8).

5.9.1 Microbenchmarks: Setup

For our microbenchmarks, we deployed Natjam on a homogeneous 7-node cluster in CCT. Each node has two quad-core processors, and 16 GB of RAM, of which 8 GB were configured to run Hadoop containers. A single node acts as the Resource Manager, while the other six are slaves, combining for 48 GB of memory available for containers. Each container – whether it is used for an Application Master, Map task, or Reduce task – is allocated with a size of 1 GB, so a total of 48 containers can be allocated at any point in time. The scheduler is configured with two queues, research and production, to which research jobs and production jobs, respectively, are submitted.

Each microbenchmark's workload consists of both research and production jobs. At the beginning of each workload, one or more research jobs are submitted to fill up the available slots. Production jobs are later submitted on this fully loaded cluster. These production jobs have a shorter execution time, to represent the use case of timely production jobs that run on recent data, and for the same reason most production jobs in the experiments have a small size compared to research jobs. The sizes of jobs submitted in our experiments, as well as their execution time running on our cluster when it is empty are shown in Figure 5.9.

Each job has a small Map execution time, and is dominated by the Reduce execution time. To model variance in task running times, the workload of Reduce tasks within a job are given following a uniform distribution in $(0.5, 1]$, where 1 represents the normalized workload for the largest Reduce task. We used a distribution generated from the same random seed across all experiments to ensure each job type did the same amount of work across multiple runs. To simulate computations, each job creates random keys and values, with thread sleep called in between keys. It was implemented based on the SWIM work generation framework [53].

Each experiment is run five times.

5.9.2 Microbenchmarks: Comparing Natjam to other techniques

The goal of Natjam is to allow production jobs to finish in a timely manner, while minimizing the impact on research jobs. In Figure 5.10, we show that

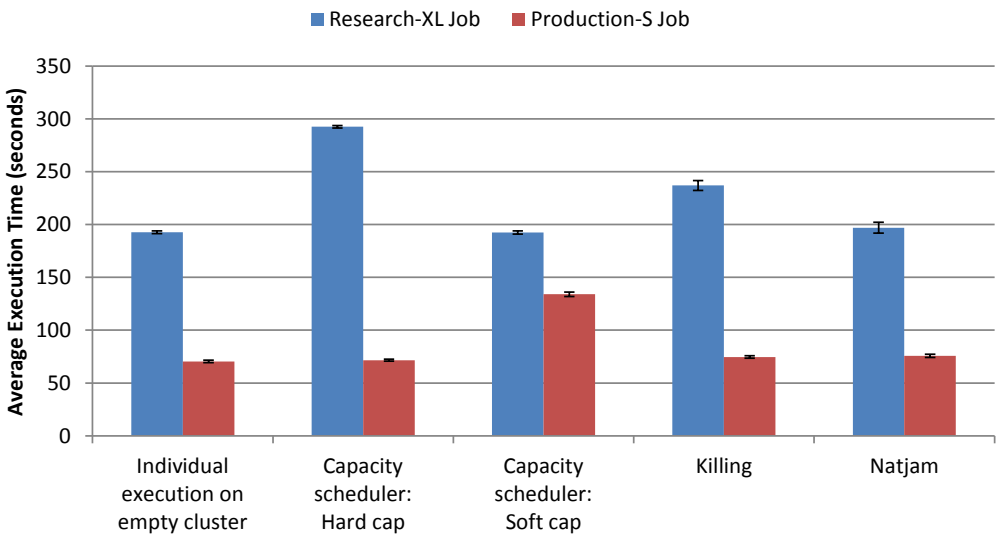


Figure 5.10: Comparison of running times with different scheduling strategies. At t=0s Research-XL job submitted; at t=50s Production-S job submitted.

Natjam's suspend/resume approach achieves this goal. The experiment is run as follows. At t=0s a Research-XL job is submitted; then at t=50s a Production-S job is submitted.

The first bars show the ideal execution time of each job on this cluster, i.e. its execution time when it is run individually on an empty cluster (these are the same values as Figure 5.9).

The next two experiments are run under Hadoop's Capacity Scheduler, specifically configured to fit this workload: the research queue is set with a 75% capacity (36 containers), and the production queue with a 25% capacity (12 containers). In the first experiment, the queue capacities are given as a hard limit. The production job finishes within its ideal execution time. However, the research job's execution time increases, because it only runs on 75% of the cluster's capacity, for the entire time of the execution. The observed increase is roughly 50%.

The second experiment configures queues with a soft limit, allowing them to expand when there is unused space. This time, the research job fills up the unused space, so its execution time matches the ideal, but the production job can only gain containers once research tasks finish. The observed increase in production job completion time is 90%.

For the next experiment, we re-introduce preemption via killing tasks to the Hadoop 0.23 code, which had been removed because of performance

concerns [21]. With killing enabled, tasks in the research queue are killed by the Resource Manager when its assigned containers are preventing the production queue from reaching full capacity. Thus, the production job's observed execution time is close to ideal. On the other hand, the research job's execution time increases nearly 50 seconds, or 23%. This is because the work done at tasks in the first 50 seconds before getting killed are repeated from scratch when the tasks are re-started.

In the last experiment, we use Natjam and observe that both the execution time of the research job and production job are close to the ideal execution time. This is true for the production job, as it obtains containers with little overhead compared with that on an empty cluster. We observe the completion time increase is 7% compared to executing in an empty cluster. We take a closer look at these overheads in Section 5.9.3. For the research job, although the execution of some tasks are suspended, the execution time of the job is not effected very much. We observe the percent increase as merely 2% compared to executing in an empty cluster. This is because we use the SRT eviction strategy for tasks. We investigate these effects further in the next section.

Natjam's observed performance is better than the previous techniques, which either fail to prioritize production jobs, or do so at the expense of research jobs. Compared to the Capacity Scheduler with a soft limit, the production job's completion time is cut by more than 40%. Compared to the Capacity Scheduler with a hard limit, the research job's completion time is cut by more than 30%, and compared to killing, its completion time is cut by more than 15%.

5.9.3 Microbenchmarks: Suspend overhead

Natjam's suspend mechanism was designed to free up containers for production tasks in a timely manner. These experiments show that this is indeed the case.

Figure 5.11 shows three different times pertaining to task assignment. First shown is the time it takes to assign a task on Hadoop, on a cluster that has spare containers. The time measured is the time that the task attempt state machine remains in the Unassigned state (see Figure 5.8), i.e., the time

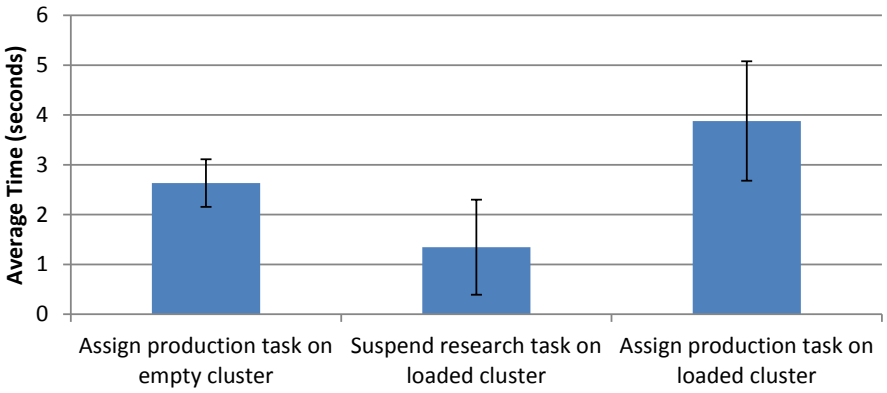


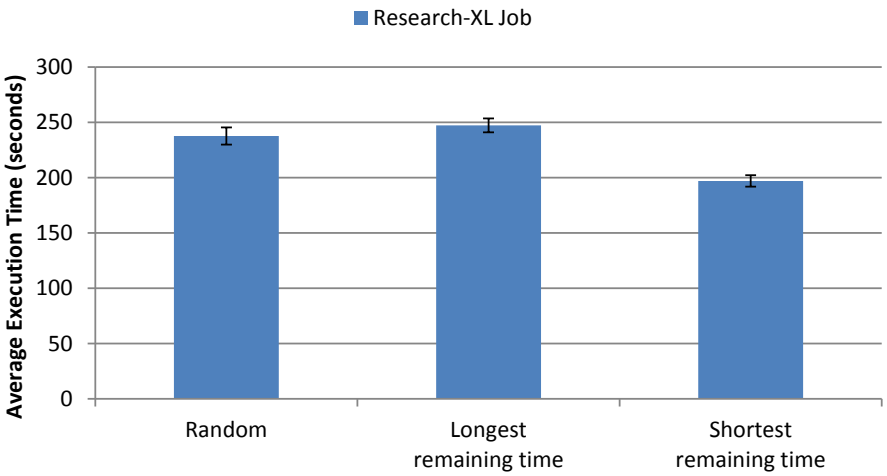
Figure 5.11: Overheads for assigning containers and suspending tasks.

between when the internal state machine has passed the container assignment request on to the AM but has not yet received the assignment. The average observed delay was 2.63 seconds.

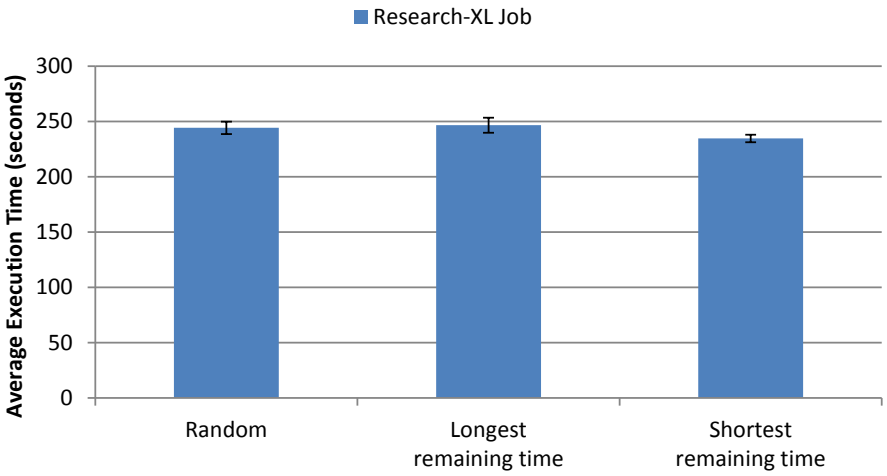
The second time shown is the time it takes to suspend a research task. The time measured is the time that the task attempt state machine remains in the Suspend-pending state, i.e., the time between when the internal state machine has passed on the suspend request to the AM but has not yet received the saved suspend state. The average observed delay was 1.35 seconds.

The last time measured is the task assignment time, which is the same as the first time. Yet, this time we measure the task assignment time of a production task which requires a research task to be suspended. The observed time, an average of 3.88 seconds, is within a standard deviation of the sum of the first two delays.

In summary, suspending a task is observed to increase task assignment time by 1.25 second, which is nearly a 50% increase. This may seem like a large impact, yet there is not a large impact on job completion times. For example, in the case of Figure 5.10, the percent increase in job completion time was only 4.7 seconds, or 7%. This is because task assignments happen mostly in parallel, so the time increase for assigning tasks does not cumulatively affect the job completion time. Thus, the average increase of 4.7 seconds is attributable to roughly three assignment delays: (i) assigning the Application Master, (ii) assigning the Map tasks, and (iii) assigning the Reduce tasks. Within (ii) and (iii) the assignments happen in parallel.



(a) At t=0s Research-XL job submitted; at t=50s Production-S job submitted.



(b) At t=0s Research-XL job submitted; at t=50s Production-L job submitted.

Figure 5.12: Comparison of job completion times using task eviction policies.

5.9.4 Microbenchmarks: Task eviction policies

In these experiments, we look at the effect of using a good task victim selection strategy. We run a Research-XL job at t=0, and start a production job at t=50s. We run two sets of experiments. The production job is a small Production-S job for the first set of experiments, and a large Production-L job for the second set.

Figure 5.12 shows the resulting execution time for the research job, depending on the task eviction strategy. In the first set of results, we observe the following.

When selecting a random task for eviction, we see a 45 second increase in research job execution time. This is because the 1/4th fraction of tasks that are suspended lead to a long tail of tasks when the job nears completion. Using the LRT eviction policy, we see an even larger 55 second increase. By choosing the longest remaining time, the tail only grows longer.

On the other hand, using SRT, we see a much smaller increase in research job execution time. In this experiment, the difference is only 4.7 seconds. This is a 17% decrease in execution time compared to random selection. The decrease is because by using SRT we are suspending the tasks that we estimate will finish the earliest once they have been resumed. Thus, with SRT we shorten the tail.

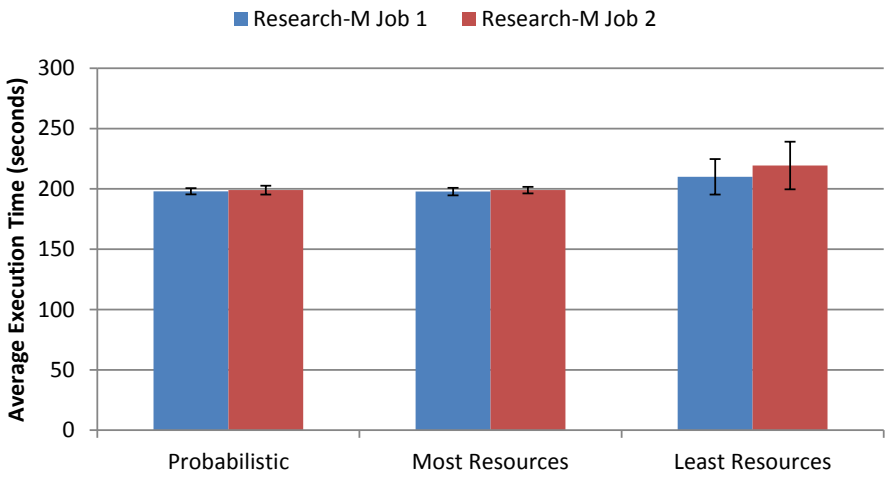
In the second set of experiments, a Production-L job is started, leading to more suspended tasks. We observe the random eviction and LRT eviction policy results are similar to the first set. The tail was already long for a small job, and does not grow much with a long job. For SRT, the completion time increases significantly. The average completion time is only 4% less than random.

In summary, we observe that SRT is effective at limiting the increase of job completion time due to task evictions. It is most effective when a small fraction of tasks are suspended at the job.

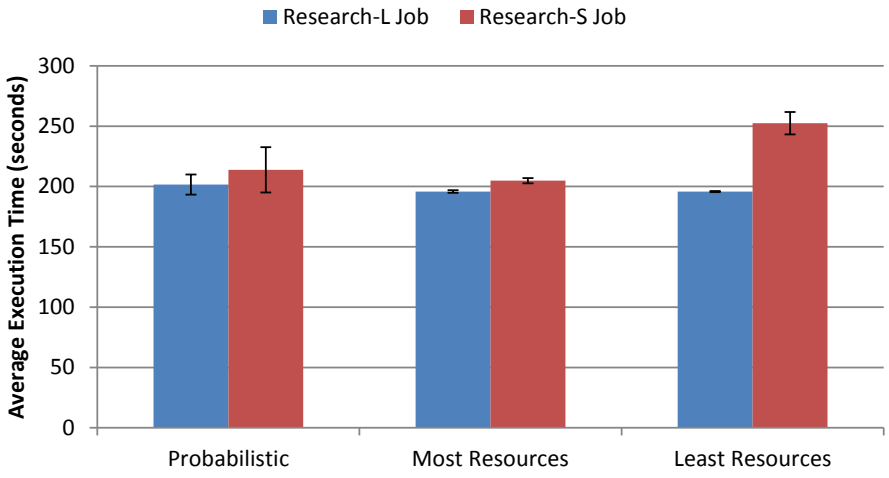
5.9.5 Microbenchmarks: Job eviction policies

In this section, we evaluate job eviction policies. We run two sets of experiments. In the first, we run two Research-M jobs at t=0s, and then run a Production-S job at t=50s. In the second, we run a Research-L job and Research-S job at t=0s, and then run a Production-S job at t=50s. Within each job, we run SRT.

The resulting research job execution times are shown in Figure 5.13 In the first set, we label the Research-M job with shorter execution time as Job 1. In this scenario, where the research jobs start with an equal size, the Probabilistically-weighted on Resources (PR) policy chooses roughly an equal amount of evictions from both jobs. By spreading out the tasks chosen with SRT among both jobs, the resulting job completion time remains within 10 seconds, or 5% of the ideal.



(a) At t=0s two Research-M jobs submitted; at t=50s Production-S job submitted. We label the Research-M job with shorter completion time as Job 1 and longer completion time as Job 2.



(b) At t=0s Research-L and Research-S job submitted; at t=50s Production-S job submitted.

Figure 5.13: Comparison of job completion times using job eviction policies.

The same is seen for the Most Resources (MR) strategy, which chooses an equal amount of evictions from both jobs. In contrast, the Least Resources (LR) strategy chooses evictions from a single job, showing a significant increase in the completion time of that job. The resulting completion time increase is 36.5 seconds from the ideal, a 19% increase.

In the second set of experiments, the effect of LR is more pronounced. For LR, we observe a completion time increase of 50 seconds, because all Reduce tasks in Research-S are suspended.

With PR, the effect of Reduce task suspensions on Research-S is less. This is because only a fraction of its tasks are suspended. With MR, there is no effect on Research-S because none of its tasks are suspended. Even so, PR and MR have little effect on the Research-L job. This is because of SRT's effectiveness when evicting a small fraction of jobs, that we showed in the previous section.

From these observations, we believe the PR and MR job eviction policies are better fits with SRT task eviction, than the LR policy.

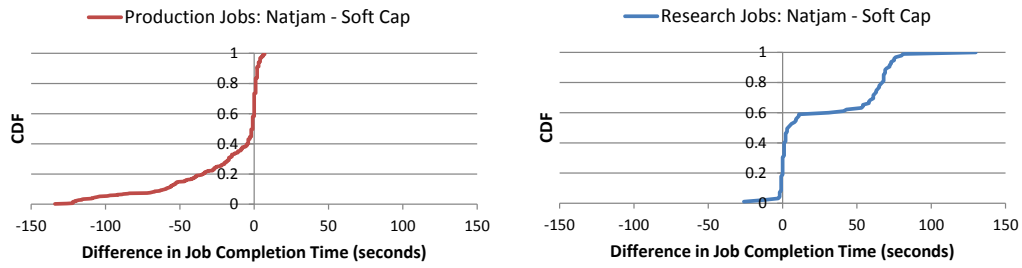
5.9.6 Deployment evaluation: Setup

We deploy Natjam and compare it to existing Soft Cap and Killing scheduling configurations using real workload traces. We start with two, hour-long traces of job submissions at a Yahoo production cluster and research cluster. We combine the two cluster workloads by scaling them. A combined trace scaled to fit a small cluster is executed on a deployment on a small cluster of 7 nodes (Section 5.9.7) and a combined trace scaled to fit a large cluster is executed on a deployment on a large cluster of 250 nodes (Section 5.9.8).

Each trace consists of an hour of job submissions, including job submission time, the number of Map tasks and Reduce tasks, and the sum of execution time across Map tasks and across Reduce tasks. The production trace consisted of jobs submitted to a single queue, among many queues, in a Yahoo production cluster. The production cluster consists of roughly 1000 machines, of which this queue takes an unknown portion.

Likewise, the research trace consisted of jobs submitted to a single queue among many in a Yahoo research cluster. The production cluster consists of roughly 400 machines, of which this queue takes an unknown portion.

We scale the traces in the following way. We first scale the number of tasks across all production jobs by a scaling factor. For a job in the resulting scaled trace, the number of tasks run is equal to the product of the scaling factor and the number of tasks in the original trace. The scaling factor is chosen to prevent the production job submissions from overloading the cluster, by choosing a factor that causes the maximum number of tasks run at a time to nearly reach the number of containers The factor used is 0.0015 for the CCT cluster and 0.5 for the Yahoo test cluster.



(a) (b)

Figure 5.14: Comparison of Natjam to Soft Cap under small-scale trace-driven workload. Negative values are better.

We then choose a scaling factor to scale the number of tasks across research jobs. This factor is chosen so that research jobs take up a significant amount of idle capacity. The factor used is 0.23 for the CCT cluster and 2.0 for the Yahoo test cluster.

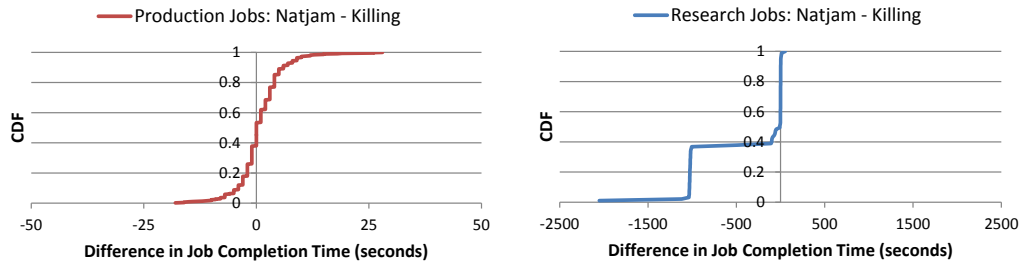
Each job is configured to do work according to the sum of execution time across Map tasks and Reduce tasks provided in the traces. These jobs simulate the execution time by running a job that creates random keys and values, while sleeping in between keys, as presented in Section 5.9.1.

We present the results of Natjam configured to use SRT task eviction and Probabilistic job eviction policies. This Natjam configuration is compared to two existing approaches. In the first, Soft Cap, the Capacity Scheduler is configured with soft limit queues of 80% for production jobs and 20% for research jobs. In the second, Killing, the Capacity Scheduler is configured with soft limit queues of 100% for production jobs and 0% for research jobs, and tasks are killed to enforce these limits. As a baseline, we also compare Natjam job completion times with an experiment, Production Only, that runs only production jobs from the same trace.

5.9.7 Small-scale deployment evaluation

Our small-scale deployment consists of a cluster of 7 nodes on CCT, that is configured to run 72 containers. The scaling factor used is 0.0015 for production jobs and 0.23 for research jobs.

The job completion times observed in this setup are summarized in Figure 5.14 to 5.16. The plots show the CDF of completion times for each



(a) (b)

Figure 5.15: Comparison of Natjam to Killing under small-scale trace-driven workload. Negative values are better.

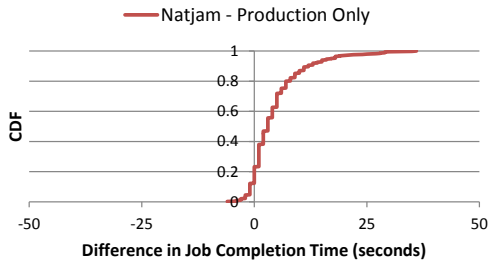


Figure 5.16: Comparison of Natjam to Production Only under small-scale trace-driven workload. Negative values are better.

job when running Natjam, subtracted by the completion time for the same job when running under the configuration it is compared to. Negative values mean Natjam performed better, while positive values mean Natjam performed worse than the configuration it is compared to.

Natjam compared to Soft Cap

Figure 5.14a compares production job completion times in Natjam to Soft Cap. 40% of production jobs perform 5 seconds or better in job completion time using Natjam, 20% perform 35 seconds better, and 10% perform 60 seconds better. Only 3% of jobs perform 5 seconds or worse in Natjam.

The way Natjam prioritizes production jobs is by suspending research jobs. This behavior is reflected in Figure 5.14b. 60% of research jobs perform 30 seconds or worse in Natjam compared to Soft Cap. This tradeoff is worthwhile because production jobs are time-sensitive.

Natjam compared to Killing

Figure 5.15 compares Natjam to Killing. Production job performance is comparable between the two approaches. The mean and median are within a second of each other, and the absolute decrease and increase in performance at the 1st percentile and 99th percentile are within 2 seconds.

Research job performance for Natjam is considerably better than Killing. 36% of jobs perform 1000 seconds or better using Natjam over Killing. In fact, when using Killing many of these jobs finish well after the hour-long job submission trace is over, suggesting that they made very little progress due to the wasted work of killed tasks triggered by new production job submissions.

Natjam compared to Production Only

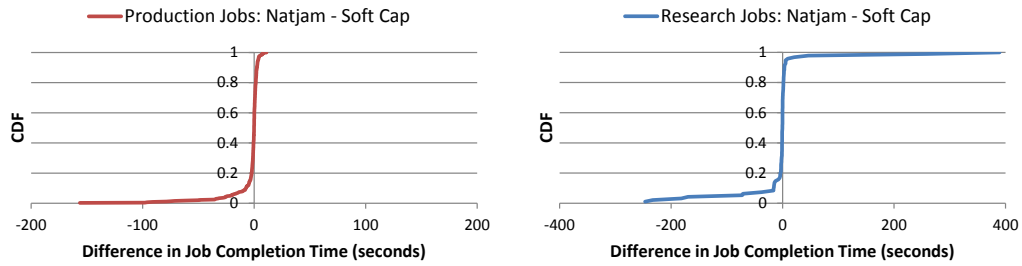
Figure 5.16 shows Natjam compared to the Production Only configuration. The median Natjam job performs within 3 seconds of Production Only, while the mean performs within 4.5 seconds of Production Only. However, the maximum difference in performance is 36 seconds.

This is due to two factors. First, Natjam has an overhead for suspending tasks as investigated in Section 5.9.3. Second, an issue in Natjam's implementation can cause concurrent requests to start Application Masters for new production jobs to be satisfied sequentially. These factors are amplified because the evaluation is run on a small cluster with a low rate of containers becoming available. In the large scale results in Section 5.9.8, these factors are mitigated.

5.9.8 Large-scale deployment evaluation

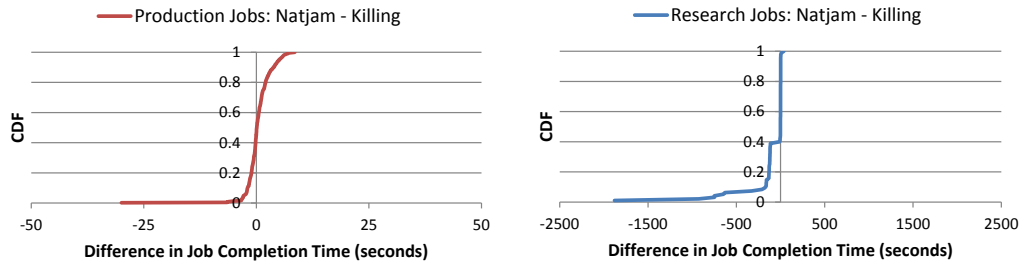
Our large-scale deployment is executed on a Yahoo test cluster consisting of 250 nodes, that is configured to run 2000 containers. The scaling factor used is 0.5 for production jobs and 8.0 for research jobs.

The job completion times observed in this setup are summarized in Figure 5.17a to 5.19. Using a larger cluster and workload, the results maintain their overall characteristics but show a significant change in distributions. We believe the main difference is because there are more containers and thus containers become available at a steadier rate.



(a) (b)

Figure 5.17: Comparison of Natjam to Soft Cap under large-scale trace-driven workload. Negative values are better.



(a) (b)

Figure 5.18: Comparison of Natjam to Killing under large-scale trace-driven workload. Negative values are better.

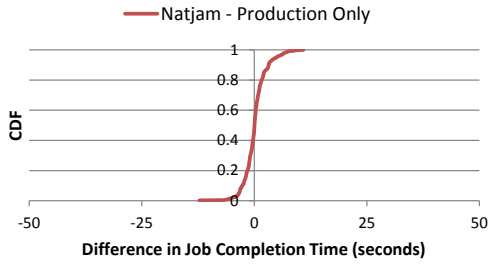


Figure 5.19: Comparison of Natjam to Production Only under large-scale trace-driven workload. Negative values are better.

Natjam compared to Soft Cap

Figure 5.17a compares production job completion times in Natjam to Soft Cap. 12% of production jobs perform 5 seconds or better in job completion time using Natjam, while less than 3% of jobs perform 5 seconds or worse in Natjam. Thus, the fraction of jobs that perform 5 seconds better in Natjam is four times those that perform worse.

More importantly, of those production jobs that perform better in Natjam,

the difference is significant. At the 5th percentile jobs perform 20 seconds or better in Natjam, at the 2nd percentile 60 seconds or better, and at the 1st percentile 80 seconds or better. The biggest improvement is greater than 150 seconds.

In Figure 5.17b, the effect of suspending research jobs is particularly pronounced for two jobs that are close to 260 and 390 seconds worse in Natjam than Soft Cap. However, in these large-scale results, there are also several jobs that perform better in Natjam than Soft Cap. The Application Master for these jobs are allocated earlier in Natjam than Soft Cap, using space created by suspending tasks in already running research jobs.

Natjam compared to Killing

Figure 5.18 compares Natjam to Killing. Production job performance is comparable between the two approaches. The mean and median are within a fraction of a second of each other, and the absolute decrease and increase in performance at the 1st percentile and 99th percentile is also within a second.

Research job performance for Natjam is considerably better than Killing. 38% of jobs perform 100 seconds or better using Natjam over Killing. At the 5th percentile, this is more pronounced, with Natjam performing nearly 750 seconds better. The biggest improvement is 1880 seconds.

Natjam compared to Production Only

Figure 5.16 shows Natjam compared to the Production Only configuration. The median Natjam job performs within 40 ms of Production Only, while the mean performs within 200 ms of Production Only. In these large-scale results, the effects of suspend overhead are mostly hidden, in contrast to the observed small-scale results.

5.10 Related Work

The problem of sharing finite resources between multiple applications is a fundamental issue in operating systems [124]. Natjam deals with these issues in large compute clusters running Hadoop. Our suspend/resume mechanism is

analogous to context switching between processes, while our eviction policies are related to cache replacement policies.

Natjam does not, however, rely on prioritization at the operating system. Doing so across Hadoop clusters, where tasks are I/O intensive with frequent disk and memory accesses, would require tight integration between the operating system and Hadoop scheduling mechanisms. By working at the Hadoop scheduler level, Natjam is able to prioritize memory allocations across the entire cluster, in a manner that cleanly integrates with Hadoop's existing scheduling mechanisms.

Much of the work on scheduling jobs in MapReduce clusters has looked at providing fairness between jobs. Quincy solves an optimization problem which considers both fairness and locality [87].

The Capacity Scheduler [20] and Fair Scheduler [13], included in Hadoop, provide average fairness by allowing an administrator to configure queue (or pool) capacities. The Capacity Scheduler is concerned solely with assigning resources, and does not allow resources to be preempted [21]. In contrast, Natjam can suspend tasks that take up resources needed by another job, so queue configurations are strictly met. The Fair Scheduler does support preemption via killing of tasks. However, killing a task results in wasted work. In Natjam, we preempt by suspending, so the work done at tasks is not wasted.

Recently, Amoeba was presented as a solution to providing instantaneous fairness with elastic queues [43]. Amoeba's checkpointing mechanism is similar to our suspend/resume mechanism. Natjam additionally employs job and task eviction policies, choosing evicted tasks based on job size and task progress rates. While Amoeba makes use of the prototype Sailfish system that stores intermediate data in a distributed file system. Natjam is designed to require minimal changes to production Hadoop 0.23 code and makes use of existing local intermediate data.

Other work has touched upon the problem of wasted work due to preemption. Delay Scheduling is implemented in the Fair Scheduler, to avoid killing many tasks while achieving high data locality [138]. Delay Scheduling is applied only to Map tasks, which are shorter on average, and thus resources become available at a high rate. In our work we focus instead on Reduce tasks, which may unavoidably have to be preempted because they are longer and thus their resources become available at a slower rate. For these tasks,

waiting for resources can be very time consuming, so instead Natjam suspend tasks.

The authors in [54] also observe that preemption negatively impacts Hadoop clusters by significantly delaying the completion time of long-running tasks. They propose a Global Preemption technique that selects tasks to kill across all jobs. In Global Preemption, preference is given to small tasks to reduce wasted work. However, when long and large research jobs take up many resources, tasks from these jobs must also be killed. Natjam is resilient to this scenario because it suspends tasks without losing work.

Hadoop jobs may be chained together, for example when using data analysis frameworks [111, 127], or workflow schedulers [27]. To optimize job completion time when job dependencies exist, new job eviction policies can be developed within Natjam, that take into account the effect of each job's completion time on dependent jobs.

Data transfer within a data center can be prioritized to speed up time-sensitive jobs [59]. Natjam does not look at network transfer, but instead looks at prioritizing the allocation of computational resources. Network-based and allocation-based techniques are not mutually exclusive and the combination of the two could lead to better performance for production jobs.

5.11 Summary

In this chapter, we presented Natjam, a system that provides strict prioritization between job queues in Hadoop. Natjam's approach is to suspend running tasks in low-priority research queues when high-priority production queues need their resources. When resources become free again, suspended tasks are resumed from where they left off. Natjam is designed to provide fast completion times for both production and research jobs.

Our evaluation shows that Natjam meets these goals. The suspend/resume methods are fast, allowing production jobs to maintain nearly optimal completion times, within 7% of the same job run on an empty cluster in our experiments. By applying suitable eviction policies, the completion times of research jobs are kept to a minimum. When interrupted by small production jobs, we observe research job completion time increases of as little as 2% more than when run on an empty cluster.

The result is that Natjam is considerably better than existing techniques in our trace-driven experiments. In our large-scale experiments we observed that, compared to an existing technique that fails to prioritize production jobs (Soft Cap), 5% of production jobs performed 20 seconds or better with Natjam, 1% of production jobs performed 80 seconds better, and the production job with the biggest improvement performed 150 seconds better. Likewise, compared to an existing technique that prioritizes production jobs but wastes work at research jobs (Killing), we observed an improvement of 100 seconds or better for 38% of research jobs, 750 seconds or better for 5% of research jobs, and the research job with the biggest improvement performed 1880 seconds better. Thus, we believe organizations can benefit from consolidating production and research workloads into a single cluster using Natjam.

Chapter 6

Conclusions and Future Directions

This thesis has described work that shows that it is feasible to satisfy strong application requirements for data-intensive cloud computing environments, in spite of resource limitations, while simultaneously optimizing run-time metrics. Our contributions are techniques and systems that reinforce this statement in diverse data-intensive cloud settings.

Our first two contributions, Pandora-A and Pandora-B, produce optimized solutions for bulk data transfers that satisfy strong requirements. Pandora-A satisfies a deadline, while optimizing on dollar cost. Pandora-B satisfies a cost budget, while optimizing on time. The next contribution, Vivace, guarantees consistent data storage between geo-distributed data centers, while providing low latency when faced with congestion. Our last contribution, Natjam, implements prioritization of production jobs over research jobs in Hadoop, by suspending research tasks to allow production jobs immediate access to resources, without losing work at research tasks.

Several future directions of research arise out of this work. First, the Pandora webservice optimizes bulk data transfers within organizations composed of cooperative users, e.g. a coalition of scientists. For the Pandora service to be used by multiple organizations, where each organization may not wish to cooperate with organizations, new algorithms should be explored that globally optimize simultaneous transfers by different organizations.

Second, to avoid delays due to congestion, Vivace relies on novel algorithms that prioritize a small amount of critical information. Designing algorithms in this way should be investigated as a general approach for geo-distributed cloud systems. Finally, in Natjam the production queue receives priority over the research queue, by suspending tasks within research jobs. The technique of suspending and resuming tasks should be explored when a job's priority dynamically changes throughout its lifetime, e.g., based on a deadline requirement and progress at the job.

References

[1] http://cassandra.apache.org, as of Oct 2011.

[2] http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6557/prod_white_paper0900aecd803e55d7.pdf as of Oct 2011.

[3] https://github.com/twissandra/twissandra, as of Oct 2011.

[4] Amazon EC2. Website. http://aws.amazon.com/ec2/.

[5] Amazon S3. Website. http://aws.amazon.com/s3/.

[6] Amazon Web Services. Website. http://aws.amazon.com/.

[7] Amazon Web Services: Case Studies. Website. http://aws.amazon.com/solutions/case-studies/.

[8] Apache Hadoop NextGen MapReduce (YARN). Website. http://hadoop.apache.org/common/docs/r0.23.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[9] AWS GovCloud (US) FAQs. Website. http://aws.amazon.com/govcloud-us/faqs/#GovCloud_fixed_reserved_storage.

[10] BitTorrent. Website. http://www.bittorrent.com/.

[11] Dante – proxy communication solution. http://www.inet.no/dante/.

[12] Dell Modular Data Center. Website. http://content.dell.com/us/en/enterprise/by-need-it-productivity-deploy-systems-faster-modular-data-center.

[13] Fair Scheduler Guide. Website. http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html.

[14] Federal Express Developer Resources Center. Website. http://fedex.com/us/developer/.

[15] Global MPLS VPN pricing guide. http://shop2.sprint.com/assets/pdfs/en/solutions/worldwide/taiwan_global_mpls_vpn.pdf as of Oct 2011.

[16] GNU Linear Programming Kit (GLPK). Website. http://www.gnu.org/software/glpk/.

[17] Google App Engine. Website. http://code.google.com/appengine/.

[18] GridFTP. Website. http://www.globus.org/toolkit/docs/4.0/data/gridftp/.

[19] Hadoop. Website. http://hadoop.apache.org/.

[20] Hadoop. Website. http://hadoop.apache.org/docs/r0.23.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[21] HADOOP-5726 JIRA: Remove pre-emption from the capacity scheduler code base. Website. https://issues.apache.org/jira/browse/HADOOP-5726.

[22] The Hadoop distributed file system: Architecture and design. http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[23] HTB Linux queuing discipline manual—user guide. http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm.

[24] IBM: Scalable modular data center. Website. http://www.ibm.com/services/us/en/it-services/scalable-modular-data-center.html.

[25] Illinois Cloud Computing Testbed (CCT). Website. http://cloud.cs.illinois.edu/.

[26] Microsoft Azure Services Platform. Website. http://www.microsoft.com/azure/.

[27] Oozie. Website. http://incubator.apache.org/oozie/.

[28] Overview: Google Data Centers. Website. http://www.google.com/about/datacenters/.

[29] Planetlab. Website. http://www.planet-lab.org/.

[30] Storm. Website. http://storm-project.net/.

[31] Sun Modular Datacenter S20. Website. http://docs.oracle.com/cd/E19115-01/mod.dc.s20/index.html.

[32] United States Postal Web Tools. Website. `http://www.usps.com/webtools/`.

[33] UPS Online Tools. Website. `http://www.ups.com/e_comm_access`.

[34] Washington Post Case Study: Amazon Web Services. Website. `http://aws.amazon.com/solutions/case-studies/washington-post/`.

[35] *Catalyst 3550 Multilayer Switch Software Configuration Guide, Cisco IOS Release 12.1(13)EA1*. March 2003.

[36] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles*, October 2005.

[37] Marcos K. Aguilera, Carole Delporte-gallet, Hugues Fauconnier, and Sam Toueg. Thrifty generic broadcast. In *International Symposium on Distributed Computing*, October 2000.

[38] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *ACM Symposium on Principles of Distributed Computing*, 2007.

[39] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *International Conference on Very Large Data Bases*, 1(1), August 2008.

[40] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. In *ACM Symposium on Principles of Distributed Computing*, August 2009.

[41] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3), November 2009.

[42] Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *International Conference on Dependable Systems and Networks*, 2006.

[43] Ganesh Anantharanayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True elasticity in multi-tenant clusters through amoeba. In *Proc. of ACM SOCC*, 2012.

[44] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California at Berkeley, Feb 2009.

[45] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1), January 1995.

[46] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research*, January 2011.

[47] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Symposium on Networked Systems Design and Implementation*, 2006. Extended version available at `http://www.cs.utexas.edu/users/dahlin/papers/PRACTI-2005-10-extended.pdf`.

[48] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, February 1999.

[49] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, may 1974.

[50] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[51] Fay Chang et al. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, November 2006.

[52] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proc. of VLDB*, 2012.

[53] Yanpei Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 390–399, july 2011.

[54] Lu Cheng, Qi Zhang, and Raouf Boutaba. Mitigating the negative impact of preemption on heterogeneous mapreduce workloads. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 189–197, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[55] Brian Cho and Marcos K. Aguilera. Surviving Congestion in Geo-distributed Storage Systems. In *Proc. of USENIX ATC*, 2012.

[56] Brian Cho and Indranil Gupta. Budget-Constrained Bulk Data Transfer via Internet and Shipping Networks. In *Proc. of IEEE ICAC*, 2011.

[57] Gregory Chockler and Dahlia Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), July 2005.

[58] Sunil Chopra and Peter Meindl. *Supply Chain Management: Strategy, Planning, and Operation*, chapter 13. Pearson Prentice Hall, 4 edition, 2010.

[59] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. 2011.

[60] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Symposium on Networked Systems Design and Implementation*, April 2009.

[61] Brian F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases*, August 2008.

[62] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, 2004.

[63] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[64] Giuseppe DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles*, 2007.

[65] K. A. Delic and M. A. Walker. Emergence of the academic clouds. *ACM Ubiquity*, 9, Aug 2008.

[66] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, pages 50–58, September/October 2002.

[67] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. HP: Hybrid paxos for WANs. In *European Dependable Computing Conference*, April 2010.

[68] John R. Douceur and Jon Howell. Distributed directory service in the Farsite file system. In *Symposium on Operating Systems Design and Implementation*, November 2006.

[69] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[70] Lisa Fleischer and Martin Skutella. Quickest Flows Over Time. *SIAM J. Computing*, 36(6):1600–1630, 2007.

[71] Lester R. Ford and Delbert R. Fulkerson. Constructing Maximal Dynamic Flows from Static Flows. *Operations Research*, 6(3):419–433, 1958.

[72] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1), February 2003.

[73] Simson Garfinkel. An evaluation of Amazon's Grid computing services: EC2, S3 and SQS. Technical Report TR-08-07, Aug 2007.

[74] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[75] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, October 2003.

[76] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *International Conference on Dependable Systems and Networks*, June 2003.

[77] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), December 2010.

[78] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989.

[79] Jim Gray and David Patterson. A conversation with Jim Gray. *ACM Queue*, 1(4):8–17, 2003.

[80] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable distributed data structures for internet service construction. In *Symposium on Operating Systems Design and Implementation*, 2000.

[81] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[82] James Hendricks. Efficient Byzantine fault tolerance for scalable storage and services. Technical Report CMU-CS-09-146, Carnegie Mellon University, School of Computer Science, July 2009.

[83] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[84] K. S. Hindi, A. Brameller, and K. M. Hamam. Solution of fixed cost trans-shipment problems by a branch and bound method. *International Journal for Numerical Methods in Engineering*, 12(5):837–851, 1978.

[85] Bruce Hoppe and Èva Tardos. The Quickest Transshipment Problem. *Math. Oper. Res.*, 25(1):36–62, 2000.

[86] John H. Howard et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[87] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. of ACM SOSP*, pages 261–276, 2009.

[88] Maurice George Kendall and Alan Stuart. *The advanced theory of statistics*, volume II. Hafner Publishing Co., 1961.

[89] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P. Sadayappan, Ian Foster, and Joel Saltz. Using overlays for efficient data transfer over shared wide-area networks. In *Proc. of ACM/IEEE SC*, 2008.

[90] Hak-Jin Kim and John N. Hooker. Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. *Annals of Operations Research*, 115:95–124, 2002.

[91] Bettina Klinz and Gerhard J. Woeginger. Minimum Cost Dynamic Flows: The Series-Parallel Case. In *Proc. of IPCO*, pages 329–343, 1995.

[92] Steve Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *Proc. ACM Symposium on Cloud Computing (SOCC)*, pages 181–192, 2010.

[93] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles*, October 2007.

[94] Tim Kraska, Gene Pang, Michael J. Franklin, and Sam Madden. MDCC: Multi-Data Center Consistency. 1203.6049, March 2012.

[95] Simon Krueger. Engineering and evaluating bulk data transfer planning over wide area networks. Master's thesis, University of Illinois, at Urbana-Champaign, 2011.

[96] Bruce Kyle. Getting Acquainted with NoSQL on Windows Azure. Blog, Apr 2012. `http://blogs.msdn.com/b/usisvde/archive/2012/04/05/getting-acquainted-with-nosql-on-windows-azure.aspx`.

[97] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[98] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[99] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[100] Eva K. Lee and John E. Mitchell. Integer Programming: Branch-and-Bound Methods. In *Encyclopedia of Optimization*, volume II, pages 509–519. Kluwer Academic Publishers, 2001.

[101] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[102] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment. 20:46–61, Jan 1993.

[103] J. S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[104] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. Don't settle for eventual: Stronger consistency for wide-area storage with COPS. In *ACM Symposium on Operating Systems Principles*, October 2011.

[105] Prince Mahajan et al. Depot: Cloud storage with minimal trust. In *Symposium on Operating Systems Design and Implementation*, 2010.

[106] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Symposium on Operating Systems Design and Implementation*, 2008.

[107] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic Byzantine storage. In *International Conference on Dependable Systems and Networks*, June 2004.

[108] Lily B. Mummert, Maria R. Eblig, and Mahadev Satyanarayanan. Exploiting weak connectivity for mobile file access. In *ACM Symposium on Operating Systems Principles*, December 1995.

[109] Kannan Muthukkaruppan. Facebook Engineering's Notes: The Underlying Technology of Messages. Blog, Nov 2010. `http://www.facebook.com/note.php?note_id=454991608919`.

[110] Jakob Nielsen. *Designing Web Usability: The Practice of Simplicity.* New Riders Publishing, 1999.

[111] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. of ACM SIGMOD*, 2008.

[112] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. of USENIX NSDI*, pages 29–44, 2006.

[113] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2), April 2002.

[114] Lavanya Ramakrishnan, Keith R. Jackson, Shane Canon, Shreyas Cholia, and John Shalf. Defining future platform requirements for e-Science clouds. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[115] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical Report ISTC-CC-TR-12-101, Intel Science and Technology Center for Cloud Computing, Carnegie Mellon University, Apr 2012.

[116] Stefan Ried, Holger Kisker, Pascal Matzke, Andrew Bartels, and Miroslaw Lisserman. Sizing the Cloud: Understanding And Quantifying The Future Of Cloud Computing. Technical report, Forrester Research, April 2011.

[117] Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable Byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, December 2003.

[118] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[119] Yasushi Saito, Christos Karamanolis, Manus Karlsson, and Mallik Mhalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Symposium on Operating Systems Design and Implementation*, December 2002.

[120] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[121] Jason Sobel. Facebook Engineering's Notes: Scaling Out. Blog, Aug 2008. `http://www.facebook.com/note.php?note_id=23844338919`.

[122] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. of ACM SIGCOMM IMC*, pages 39–44, 2003.

[123] Hamdy A. Taha. *Operations Research: An Introduction*, chapter 5. Pearson Prentice Hall, 8 edition, 2007.

[124] Andrew S. Tanenbaum. The Operating System as a Resource Manager. In *Modern Operating Systems*, chapter 1.1.2. Pearson Prentice Hall, 3e edition, 2008.

[125] Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles*, December 1995.

[126] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *ACM Symposium on Operating Systems Principles*, October 1997.

[127] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629, August 2009.

[128] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An architecture for internet data transfer. In *Proc. of USENIX NSDI*, pages 19–19, 2006.

[129] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP nice: A mechanism for background transfers. In *Symposium on Operating Systems Design and Implementation*, December 2002.

[130] Werner Vogels. All Things Distributed: Choosing Consistency. Blog, Feb 2010. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html.

[131] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, October 1993.

[132] Randolph Y. Wang, Sumeet Sobti, Nitin Garg, Elisha Ziskind, Junwen Lai, and Arvind Krishnamurthy. Turning the postal system into a generic digital communication mechanism. In *Proc. of ACM SIGCOMM*, pages 159–166, 2004.

[133] Paul Watson. e-Science in the Cloud with CARMEN. In *Proc. of International Conference on Parallel and Distributed Computing Applications and Technologies*, 2007.

[134] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Symposium on Operating Systems Design and Implementation*, November 2006.

[135] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sung-Ju Lee, and Sujoy Basu. S^3: A scalable sensing service for monitoring large networked systems. In *Proc. of ACM SIGCOMM INM (Workshop)*, September 2006.

[136] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of USENIX OSDI*, 2008.

[137] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. Usenix OSDI*, Dec 2008.

[138] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.