ENHANCING AVAILABILITY IN LARGE SCALE STORAGE SYSTEMS AND SERVICES: ARCHITECTURES AND TECHNIQUES

A Thesis
Presented to
The Academic Faculty

by

Sangeetha Seshadri

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2009

ENHANCING AVAILABILITY IN LARGE SCALE STORAGE SYSTEMS AND SERVICES: ARCHITECTURES AND TECHNIQUES

Approved by:

Professor Ling Liu, Committee Chair
College of Computing
Georgia Institute of Technology

Professor Ling Liu, Advisor
College of Computing
Georgia Institute of Technology

Dr Brian Cooper
Yahoo! Research
Georgia Institute of Technology

Professor Karsten Schwan
College of Computing
Georgia Institute of Technology

Professor Calton Pu
College of Computing
Georgia Institute of Technology

Professor Douglas Blough
College of Computing
Georgia Institute of Technology

Date Approved: August 2009

To my family

Kaayena Vaachaa Manasendriyairvaa

Buddhyaatmanaa Vaa Prakriteh Svabhaavaatah

Karomi Yadhyadh Sakalam Parasmai

Naaraayanaayeti Samarpayaami

ACKNOWLEDGEMENTS

This thesis would not have been possible without the motivation, guidance, support and blessings of a number of individuals. Next to these, my contributions stand dwarfed. I take this opportunity to express my sincere thanks to all of them.

First, I would like to thank my advisor Prof. Ling Liu. Incredibly energetic and versatile, Ling has been a great source of inspiration and an awesome advisor. Her constant support, guidance and feedback have been crucial not only in shaping my dissertation, but also in helping me make the right career choices. The flexibility she afforded me, both in choosing my research topics and making working arrangements, has made this a smooth journey. To me, she is not only a guide but also a good friend. Next, I express my sincere thanks to Dr. Brian Cooper. Brian helped me jump-start my PhD, showing me how to approach problems and how to conduct systematic research. I have always admired his clarity of thought and elegance of expression. I hope some of it has rubbed on to me. I am also very grateful to Prof. Karsten Schwan for his patience, guidance and encouragement. Karsten always gave me sound advice and new perspectives on my research. I consider myself lucky for having worked with such talented and prolific researchers. I also thank my dissertation committee members Prof. Calton Pu and Prof. Douglas Blough for their encouragement and insightful comments and also for accommodating me, sometimes on very short notice. I would also like to thank Susie McClain for her administrative help throughout my time at Georgia Tech.

A significant portion of my dissertation research evolved during my summer internships at the IBM Almaden Research Center. I thank IBM for the generous financial support through the IBM PhD fellowship. My heartfelt thanks to Lawrence Chiu and

the entire Storage Systems group. Larry has been one of my strongest supporters and a very trusted guide. The four summers that I spent with him and his group have been some of the most productive times during my PhD. An astute researcher and a great manager, working with him has been both a tremendous learning experience and a great pleasure. I have also had the opportunity to collaborate with and learn from other great researchers at IBM. Many thanks to Paul Muench, Karan Gupta, Cornel Constantinescu, Clem Dickey, Subashini Balachandran and many others. Much work in this dissertation was possible due to the time and resources provided by KK Rao, Bruce Hillsberg, David Whitworth, Andrew Lin, Juan Ruiz, Brian Hatfield, Chiahong Chen, Joseph Hyde and David Chambliss.

I thank Bhuvan, Aameek, Mudhakar, James, Ting, Gong, Sankaran, Peter, Balaji and all present and past members of the DiSL lab for making the group meetings fun and something that I always looked forward to. Ramya made my internships in San Jose a pleasure and without Raji, I would not have been able to complete the last year of my PhD. I thank Anusha, Anitha, Sujatha, Sunitha, Nandita, Smita, Vibhore, Oneza, Adithya, Gayathri, Ellen, Deepthi, Rani, Sundar, Santhanam, Mayura, Vidyaraman, Charanya, Vijay, Anitha, Anupama, Karthik, all my wingies and many others for their support and the great times we had together. A very special thanks to Maha for being there always and to Subbu for always being on the same mobile network as me. I am extremely fortunate for having such a great set of friends.

Without the love and support of my family, all this would neither have been possible nor meaningful. I dedicate this work to my family. My brother Vijay has patiently endured my rants and raves and has always been there to tell me the right thing to do. Of course, it is a whole other story that I usually realize that only a few months later. My heartfelt thanks to Vijay, Prabha, Bhargavi and Ravishankar. I also thank my grandparents for their blessings and encouragement. Special thanks to my aunt

Meena for her prayers and belief in me. The constant prayers and blessings of my parents Vaidehi and Seshadri and my parents-in-law Lalitha and Rajaraman have been my biggest source of strength throughout this dissertation. At every turn they have been there to cheer me, inspire me and literally carry me through. Finally, I dedicate this dissertation to my dear husband Shankar for his patience, love and support. I am deeply indebted to him for his unconditional love and the many sacrifices that he has had to make along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Enterprises today are dealing with extremely large amounts of digital information that continues to grow at an astonishing rate. Online business models, regulatory compliance and business intelligence requirements have not only mandated enterprises to retain large amounts of data for significant lengths of time but have also increased the reliance on anytime and anywhere access to this information. Consequently, the storage systems that serve as repositories for these huge volumes of critical data are the foundations of today's data centers. Unavailability of these systems results in losses amounting to millions of dollars per hour and could bring organizations to a grinding halt. On the other hand, storage software (firmware, middleware) and systems are becoming much more complex and existing failure recovery mechanisms are insufficient to handle the scale of these systems while meeting high availability and service quality expectations. In addition, the concurrent development and quality assurance processes, the large number of possible test scenarios and the large scale of these systems and services imply that failures will be the norm rather than the exception. Therefore achieving high availability and reliability in storage systems remains a major concern and an open research challenge.

Most existing work in the domain of storage system availability addresses failures of the storage media (such as disks) and recoverability from these failures. However, failures at the firmware and middleware layers remain largely unaddressed. Achieving high-availability in these layers poses unique challenges. At the firmware layer, fine-grained recovery is an effective approach to reduce recovery-time. However, complex recovery semantics, dynamic interactions, recovery dependencies between large

volumes of concurrent tasks and legacy architectures pose serious challenges. At the middleware layer, a widely recognized open problem is how to provide fault-isolation and improve system availability without disrupting the system's functionality or limiting its scalability. Over the past few years, storage clusters consisting of thousands of commodity machines built specifically to serve the needs of large scale distributed data intensive applications where decentralization, high availability, and autonomy are key design principles have become common, exemplified by Amazon S3 (Simple Storage Service) [4], Google File System [69] and IBM System S [34, 76]. Another class of functionality rich dedicated storage middlewares also offer storage management and resource virtualization capabilities. While the scale of these systems result in new challenges [3], the nature of the applications present new opportunities. We can try to utilize application semantics, failure characteristics, access patterns and consistency models to define novel application-specific availability enhancing techniques at the middleware layer which go beyond traditional techniques such as replication [46] and process-pairs [72, 48].

This dissertation research addresses these challenges in depth across different storage architectures. We make the following contributions: First, we develop a recovery conscious framework for multi-core architectures and a suite of techniques for performing efficient fine-grained recovery (micro-recovery) in storage controller firmware that can be retrofitted into legacy code. The framework includes a task-level recovery mechanism, the Log(Lock) architecture that allows system state restoration during micro-recovery, and recovery-conscious scheduling algorithms that are designed to reduce the ripple effect of failure and improve recovery efficiency and system availability. Our framework also provides guidelines for system developers to perform effective mappings of system tasks to critical framework parameters aiming at improving availability by serializing dependent tasks and enhancing recovery efficiency, while sustaining high performance and system throughput.

Our second technical contribution addresses the storage middleware availability. We first develop the notion of hierarchical middleware architectures by organizing critical cluster management services into a hierarchical overlay network, which separates persistent application state from global system control state. We demonstrate that by trading some symmetry for better fault isolation, hierarchical storage middleware architectures can significantly improve availability and reliability of enterprise scale storage systems. In addition, we develop the notion of operator reuse and a suite of reuse techniques to improve data availability. The key idea of operator reuse is to efficiently utilize system resources by exploiting reuse opportunities in both operators and persistent state of computing nodes. We demonstrate our design through STREAMREUSE, a reuse-conscious store-forward network of storage nodes, which offers distributed stream query processing services. By 'reuse-conscious', we mean that the system is provided with the ability to modify operators and migrate services at runtime to maximize reuse opportunity. Our analytical and experimental results show our storage middleware solutions are efficient and effective in enhancing data availability and system availability of large scale storage systems.

CHAPTER I

INTRODUCTION

Enterprise storage systems are the foundations of most modern data centers and extremely high availability is expected as a basic requirement from these systems. With the rapid and exponential growth of both personal and enterprise digital information [9, 43, 8, 16, 13, 108], online business models [127] and service architectures [2, 4] there is increased reliance on anytime and anywhere access to this information. Consequently, the demand for large scale storage systems of extremely high availability (moving close to 7 nines) continues to grow [127]. On the other hand, storage software (firmware, middleware) and systems are becoming much more complex and existing failure recovery mechanisms are insufficient to handle the scale of these systems while meeting high availability and service quality expectations.

Although a variety of online storage alternatives exist for storing these vast quantities of data, ranging from high-end disk arrays with snapshot and remote mirroring capabilities, to scale-out storage clusters that combine smaller units of storage, to small disk appliances (e.g. RAID), achieving high availability and reliability in storage systems remains a major concern for a number of reasons. First, the storage system software is becoming much more complex, especially given the increasing functionalities of these systems and the fact that legacy systems are being adapted to new hardware platforms like multi-core architectures [112, 23, 24]. Second, the concurrent development and quality assurance processes along with the large number of possible test scenarios makes it extremely difficult to test these systems. Third, the size and functionalities of storage systems and services in modern, large scale IT installations have grown to an unprecedented scale with thousands of storage components [69, 126]

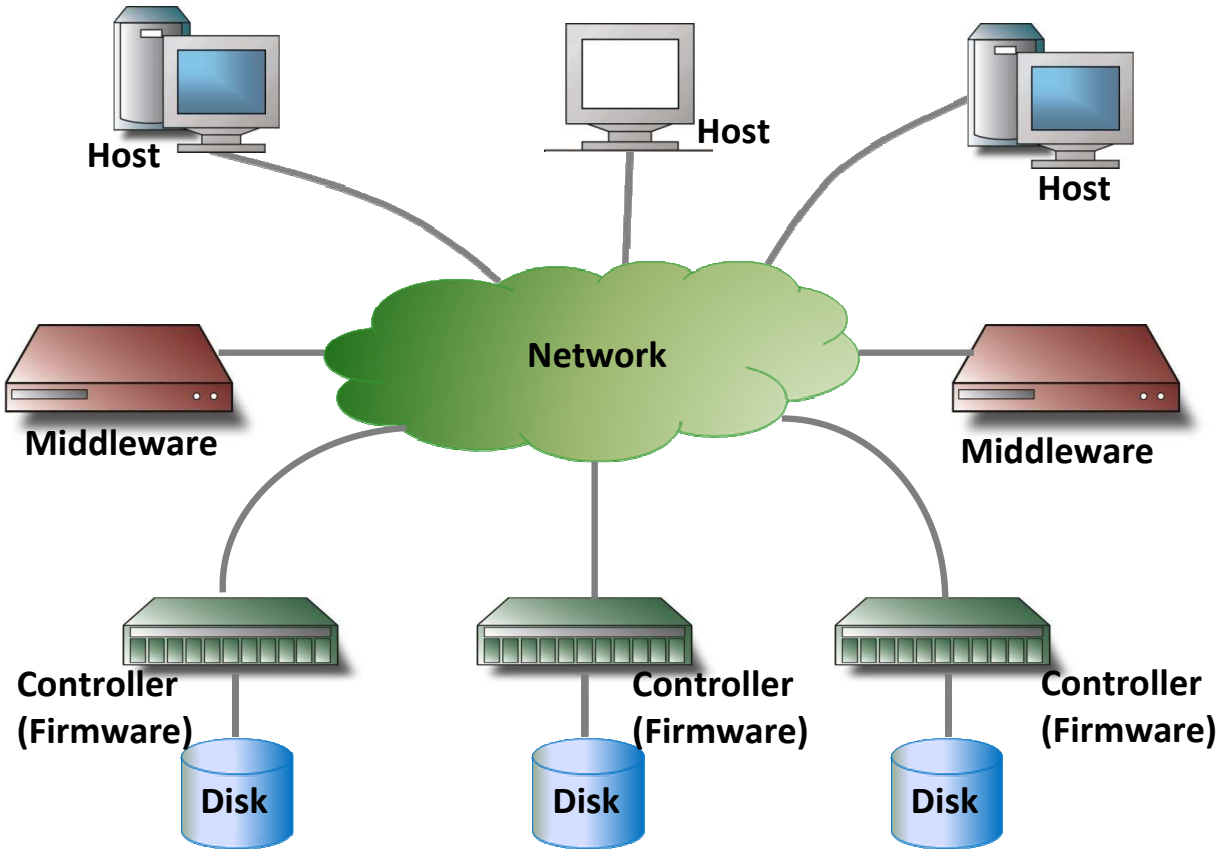


Figure 1: Storage Subsystem Architecture

and complex interacting applications. As a result, failures are the norm rather than the exception.

A storage system is composed of multiple layers. Figure 1 shows the different layers of a typical storage system architecture. First, we have the storage media (disks) with its electrical and mechanical components where the data is actually stored. Next, the embedded storage controller firmware performs most of the storage subsystem's higher level functionality such as RAID, I/O routing, error-handling and caching [70]. Storage middleware provides both management [70, 14, 21, 103] and resource virtualization capabilities [18, 10]. Finally, hosts perform I/O over this

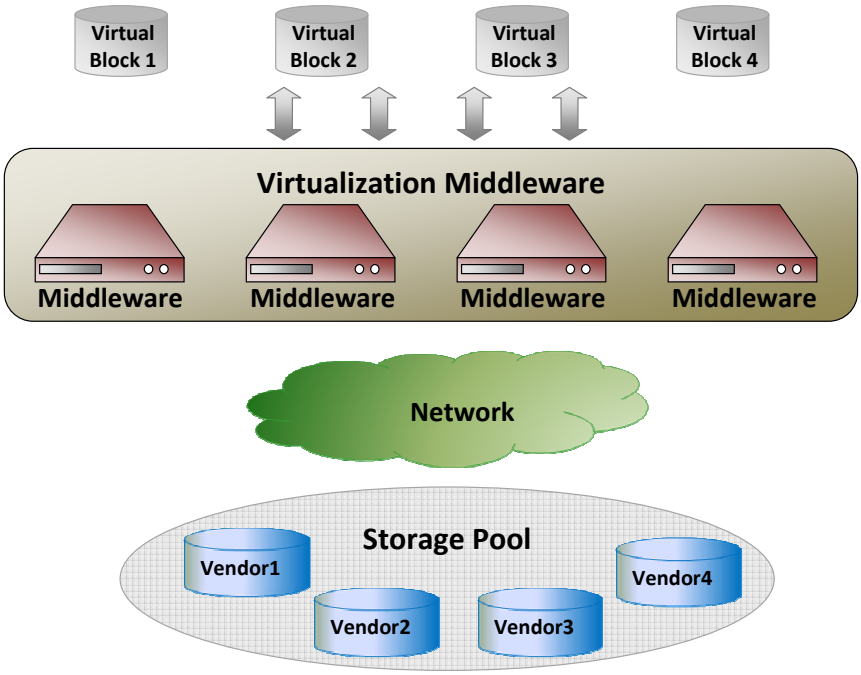


Figure 2: Scale-out storage cluster.

storage system utilizing the bandwidth provided by a shared or dedicated network [123, 17]. Each of these layers is susceptible to failures which could result in data or service loss [107, 87, 111]. Therefore, in order to build highly-available and resilient storage systems we must address the availability concerns in each layer of this system.

Most existing work in the domain of storage system availability addresses failures of the storage media (such as disks) and recoverability from these failures [139, 152, 107]. Traditionally, these studies address the problem of durability and data loss due to media failures or corruption [42, 33, 125, 41, 125]. Many mechanisms, adopted by the industry such as various levels of RAID [107], erasure coding [120] and continuous data protection [25], have emerged to address these storage media failures [31]. However, failures at the firmware and middleware layers that result in service loss remain largely unaddressed. At the same time, the firmware and middleware layers of a storage system have evolved tremendously in terms of functionality.

Modern storage controllers are highly concurrent embedded systems with millions

of lines of code of firmware [112, 129]. As a result, these systems are extremely complex and recovering from controller failures is both difficult and expensive. Similarly functionality-rich storage middleware, especially in scale-out storage systems, not only perform sophisticated storage tasks, but also support offloading of application tasks onto to the storage layer [18, 10]. Figure 2 shows an example of such a virtualization middleware which runs over a cluster of nodes, managing the underlying storage, providing a single system image (SSI) and providing storage virtualization services to the applications running above it. Such middleware may also support offloading of application tasks to the storage subsystem in order to run close to the data and utilize spare processing capabilities. The middleware provides high availability and survives node failures, utilizing a replicated state machine model [124]. Consequently, storage system middleware is susceptible to large simultaneous failures as well as application-induced failures due to its symmetric architecture and exposed interfaces [47, 52].

Over the past few years, storage clusters consisting of 1000s of commodity machines built specifically to serve the needs of a class of large scale distributed data intensive applications have become common. Systems such as Amazon S3 (Simple Storage Service) [4], Google File System [69] and System S [34, 76] where decentralization, high availability and autonomy are key design principles consist of thousands of commodity machines that manage hundreds of terabytes of shared storage and serve thousands of clients. While the scale of these systems result in new challenges [3], the nature of the applications present new opportunities. We can try to utilize application semantics, access patterns and consistency models to define novel application-specific availability enhancing techniques at the middleware layer which go beyond traditional techniques such as replication [46] and process-pairs [72, 48]. For example, text search can deal with non-determinism and the user may not notice a few missing results. Then, techniques such as failure-oblivious computing [119] which overlook failures

and return arbitrary values can be used. Likewise, in the case of applications such as data stream processing systems, the well-defined semantics of queries and intermediate data can be utilized to improve data and service availability as well as resource utilization of these systems [128].

Next we discuss the specific research challenges in different layers of the storage system stack in detail .

1.1 Technical Challenges

In this dissertation we aim to address the following broad challenges across different storage system architectures.

1.1.1 Scalable Storage Controller Failure Recovery

With software failures and bugs becoming an accepted fact, focusing on recovery and reducing time to recovery has become essential in many modern storage systems today. In current system architectures, even with redundant controllers, most microcode i.e. firmware failures trigger system-wide recovery [75, 77] causing the system to lose availability for at least a few seconds, and then wait for higher layers to redrive the operation. This unavailability is visible to customers as service outage and will only increase as the platform continues to grow in size (number of cores) using the legacy architecture.

How can failure recovery be made scalable? Partitioning the system into smaller components with independent failure modes can reduce recovery time. However, it also increases management cost and decreases flexibility, while still being susceptible to sympathetic failures. On the other hand, refactoring the software into smaller independent components, in order to use techniques such as micro-reboots [49] or software rejuvenation [85], may require sizable investments in terms of development and testing effort and cost. In the case of legacy systems, this can be unacceptable.

An alternative approach is to be able to perform fine-granularity recovery or *micro-recovery*, without re-architecting the system. Under this approach, failure recovery is targeted at a small subset of tasks/threads that need to undergo recovery while the rest of the system continues uninterrupted.

However, due to fuzzy component interfaces, complex dependencies and involved operational semantics of the system, implementing such fine-grained recovery is challenging. Therefore, firstly we must develop a mechanism to perform fine-grained recovery taking into consideration interactions between components and recovery semantics. Secondly, since localized recovery spans multiple dependent threads in reality, we must bound this localized recovery process in time and resource consumption in order to ensure that resources are available for other normally operating tasks even during recovery. Finally, in the case of storage firmware, most often we are dealing with a large legacy architecture ($>$ 2M lines of code). Therefore, in order to ensure feasibility in terms of development time and cost we should minimize changes to the architecture.

1.1.2 Storage Middleware Fault Tolerance

High-availability scale-out storage clusters such as shown in Figure 2 combine smaller units of storage to provide a scalable and cost-effective storage solution [14, 21, 70]. Current scale-out storage systems use active replication [124] based middleware to ensure consistent access to shared resources in the absence of centralized control and at the same time provide high throughput and a single system image (SSI). In order to guarantee high-availability to applications, the middleware typically maintains critical application state and check-point information persistently across nodes through active replication.

However, though the use of symmetric active replication models removes hardware as both single-points-of-control and single-points-of-failure, it causes the storage middleware itself to now become a single-point-of-failure. Moreover, the current scale-out storage architecture is also vulnerable to application-induced failures of the middleware, in addition to other issues like application-level non-determinism [141] and middleware bugs themselves. In order to achieve a truly high-availability storage system, the challenge lies in eliminating middleware as a single-point-of-failure and providing fault-isolation from application induced failures without loss of functionality or ease-of-management.

1.1.3 Fault Tolerance Through Data Reuse

An emerging class of distributed stream systems such as enterprise applications [34, 4, 104, 2, 12], scientific collaborations across wide area networks [19], and large-scale distributed sensor systems [156, 97] are placing growing demands on storage nodes to provide capabilities beyond basic data storage such as persistent state management and continuous and opportunistic processing [12]. These applications are popular across diverse domains ranging from financial management [11] to scientific computing [48].

The new challenges in systems of this scale require us to go beyond generic process-pairs [72] and checkpointing mechanisms [73] to achieve high-availability. In this dissertation we focus on distributed data stream systems, a particular class of large scale distributed storage system that offer continual query processing services using a distributed storage infrastructure. The underlying storage nodes support store and forward services acting as a store for both incoming data and intermediate results. In order to improve the availability of such a system it is essential to deal with failures as well as ensure that results are delivered to end-users in a timely manner while using resources efficiently. An effective approach to achieve both high data

availability and efficient use of resources is to minimize the amount of persistent state in the system by sharing and reusing state as much as possible. Then the challenge is to find effective reuse opportunities given the dynamic and distributed nature of applications, semantics of user requests and the variations in the underlying execution environment.

1.2 Thesis Statement

In order to effectively scale a system while retaining high availability the system must support a scalable failure recovery mechanism.

1.3 Thesis Contributions

In this dissertation we make the following contributions to address each of the challenges described in the previous section.

1.3.1 Improving Storage Firmware Availability

We develop a recovery conscious framework and a suite of techniques for improving the failure resiliency and recovery efficiency of storage firmware. The overarching goal of our framework is to enable the system to perform fine-grained recovery or *micro-recovery*, thereby reducing the recovery time and improving availability. The framework tracks recovery dependencies between concurrent threads, decides recovery actions at runtime, ensures availability of resources to normally operating tasks even during localized recovery and is designed to allow micro-recovery to be easily retrofitted into legacy systems. The framework (shown in Figure 3) achieves these goals through the three stages. The three tiers of the framework progressively answer the following questions: (1) How to we perform fine-grained recovery and restore the system state while accounting for dependencies between concurrent threads? (2) How do we map dependent tasks into groups in order to enforce scheduling constraints that can improve recovery efficiency while still delivering high performance? (3) How do

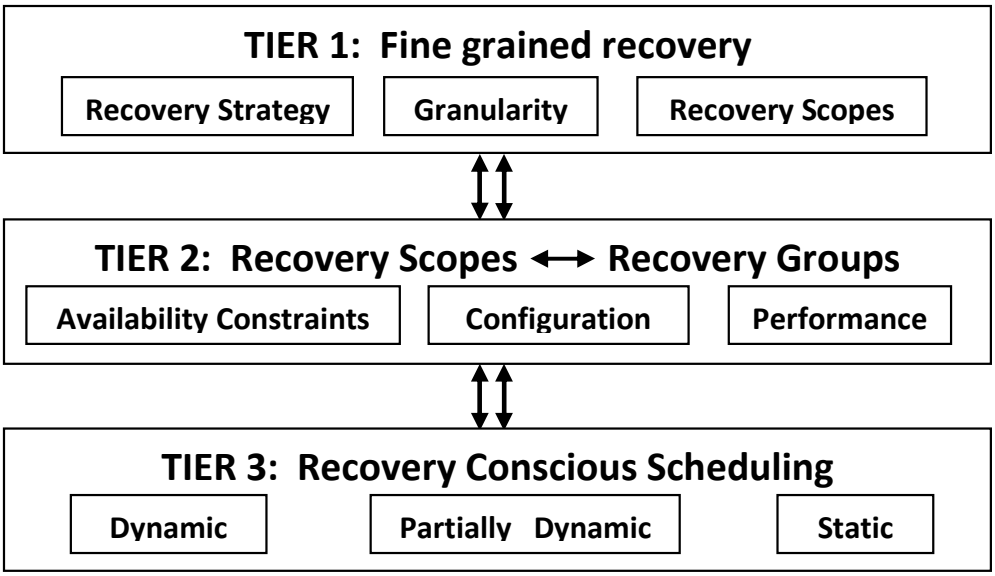


Figure 3: Recovery-Conscious Framework

we schedule these groups of dependent tasks while ensuring availability of resources to normally operating tasks during failure recovery and reducing the ripple effect of failure? The shaded boxes in Figure 3 represent the parameters under consideration at each tier of the framework which are detailed below.

- **Recovery Strategy and Scope :** In order to perform fine-grained recovery in response to storage controller failures, we must first understand recovery-dependencies between tasks i.e. concurrent threads in the firmware. When a single task encounters an exception, more than one task may need to initiate recovery procedures in order to avoid deadlocks and return the system to a consistent state. The purpose of tracking recovery dependencies between concurrent threads is two-fold. First, it allows us to perform efficient and effective state restoration, while accounting for dynamic dependencies between multiple threads in a highly concurrent environment. Next, it allows us to classify dependent threads into disjoint 'recovery scopes' over which serialization and recovery-conscious scheduling constraints can be enforced.

Our first contribution is Log(Lock), a practical and flexible architecture for tracking dynamic dependencies and performing state restoration, without re-architecting legacy code. We use a systematic approach to address the problem of system state restoration during micro-recovery, by developing state space exploration methods and the Log(Lock) execution model. In the state space exploration phase, we formally model thread dependencies based on both state and shared resources, capturing failure contexts through different 'restoration levels'. We develop recovery strategies by deriving restoration protocols in terms of recovery procedures and restoration levels. The Log(Lock) execution model tracks state changes using Log(Lock) primitives and implements state restoration based on restoration protocols. We have implemented Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient (<10% impact on performance) and effective (reduces a 4 second downtime to only a 35% performance impact) [130].

Our framework also allows explicit dependencies to be specified by the programmer. However, explicit dependencies specified by the programmer may be very coarse. Likewise, some dependencies may have been overlooked due to their dynamic nature and the immense complexity of the system. The dependency information identified using the Log(Lock) architecture can be used to refine explicit dependencies and classify firmware tasks into "recovery scopes".

- **Efficient Mappings of Recovery Scopes to System Resources :** The "recovery scopes" identified from the previous step are purely based on recovery dependencies. However, in order to improve system availability without serious performance impacts, we need to develop effective mappings of recovery scopes to system resources. Specifically, we need to realign "recovery scopes"

into "recovery groups" while taking into consideration performance and availability parameters such as: the system size, failure rates of individual recovery scopes, time required to complete recovery, scheduler overhead and performance constraints for the given workload. Each of these factors dictate the reorganization of recovery scopes into actual "recovery groups" and the right choice for the scheduling strategy and recoverability constraints. A **recoverability constraint** is specified for each group and prescribes the maximum number of concurrently executing tasks permissible for that group. The middle tier of the framework [134] is dedicated to the development of highly effective mapping of dependent tasks to system resources (processing resources in our work) in order to ensure system availability through reduced recovery time while meeting the performance requirements.

- **Recovery-Conscious Scheduling :** In spite of identifying fine-grained recovery groups and recovery dependencies between tasks, without careful design, it is possible that more dependent tasks are dispatched before a recovery process can complete. This results in an expansion of the recovery scope or an inconsistent system state. Also a dangerous situation may arise where it is possible that many or all of the threads that are concurrently executing are dependent, especially since tasks often arrive in batches. Then the recovery process could consume all system resources, essentially, stalling the entire system. In [129] we present recovery-conscious scheduling (RCS) that enforces serializability of failure dependent tasks thereby reducing the ripple effect of software failure and improving system availability. The key idea of recovery-conscious scheduling (RCS) is to ensure bounded recovery time and provide fault resiliency by optimal allocation of resources to recovery dependent tasks. We propose three alternative recovery-conscious scheduling algorithms [129]; each represents one way to trade-off between recovery time and system performance.

We have implemented and evaluated these recovery-conscious scheduling algorithms on a real industry-standard storage system. Our experimental evaluation results show that the proposed recovery conscious scheduling algorithms are non-intrusive and can significantly improve (throughput by 16.3% and response time by 22.9%) the performance of the system during failure recovery.

1.3.2 Improving Storage Middleware Availability

One obvious approach to improving availability and reliability in storage system middleware is to partition a single storage cluster into smaller independent clusters [30, 26] in order to provide application fault isolation and eliminate storage middleware as a single-point-of-failure. While application fault isolation can be achieved through this approach, without care, one may lose the SSI and the flexibility to access storage from anywhere within the system. The key challenge, therefore, is to eliminate the middleware as a single-point-of-failure and provide fault-boundaries while continuing to deliver SSI and flexible accessibility.

- **Hierarchical Middleware Architectures :** In order to address the issue of middleware availability, we introduce the notion of hierarchical middleware architectures [135]. We organize critical cluster management services into a hierarchical overlay network, which separates persistent application state from global system control state. This clean separation, on one hand, allows the cluster to maintain SSI by communicating control state to all nodes in the network, and on the other hand, provides fault isolation by replicating application state within only a subset of nodes. We demonstrate [135] that by trading some symmetry for better fault isolation, hierarchical overlay storage architectures can significantly improve system availability and reliability. We also show that such hierarchical architectures significantly reduce the number of system states for testing.

1.3.3 High Availability Through Data Reuse

One approach to effectively improve data and service availability in large scale data stream processing systems is to make efficient use of system resources and improve system recovery time in the event of failures by sharing and reusing intermediate data represented by 'operators' between multiple concurrent service requests. However in order to reuse operators, we must first scalably locate reuse opportunities and also deal with dynamic workloads, semantic differences between existing operators and new requests and variations in the underlying execution environment. To address these challenges we make the following contributions:

- **Dynamic Grouping of Similar Operators :** We present the design and evaluation of STREAMREUSE, a reuse-conscious store-forward style network of storage nodes that offer distributed stream query processing services. We develop a suite of reuse-conscious stream query grouping techniques that dynamically find cost-effective reuse opportunities based on multiple factors, such as network locality, data rates, and operator lifetime. By dynamically grouping operators based on reuse possibilities identified at runtime, our techniques enable intermediate data and persistent state to be shared between multiple concurrent operators in the system.

- **Network-Aware Operator Reuse :** We show that a static query optimization approach of plan, then deployment is inadequate for handling distributed queries involving multiple streams and node dynamics faced in distributed data stream systems and applications [131]. We propose to use hierarchical network partitions to exploit various opportunities for operator level reuse while utilizing network characteristics to maintain a manageable search space during operator placement and deployment. We develop top-down, bottom-up and hybrid

algorithms for exploiting operator-level reuse through hierarchical network partitions. Through simulations and experiments using a prototype deployed on Emulab [7] we demonstrate the effectiveness of our framework and our algorithms [132].

1.4 Organization of this Dissertation

The chapters of this dissertation are organized as follows. The first three chapters (Chapters 2, 3 and 4) address firmware availability issues. The next three chapters (Chapter 5, 6 and 7) deal with middleware and data availability issues. Below, we present a brief overview of each chapter.

Chapter 2: We present an overview of the storage controller architecture and our recovery-conscious framework for enabling micro-recovery in legacy controller microcode. We describe the mechanism to perform task-level recovery and the role of each tier of the framework. The chapter also presents a taxonomy of failure and recovery models and a discussion of related work. This chapter is intended to serve as an overview of our recovery-conscious framework with each research problem discussed in detail in subsequent chapters.

Chapter 3: In this Chapter, we present the Log(Lock) architecture for tracking dynamic dependencies and performing state restoration.

Chapter 4: We present recovery-conscious scheduling (RCS) and guidelines to effectively map dependent tasks into 'recovery groups' to ensure that the effect of fine-grained recovery percolates to the level of system availability while sustaining high performance.

Chapter 5: In this Chapter, we propose hierarchical middleware architectures that improve availability and reliability in scale-out storage systems while continuing to deliver the cost and performance advantages and a single system image (SSI).

Chapter 6: We present the STREAMREUSE system that explores operator

reuse techniques to share persistent data between multiple service requests and improve data availability. The chapter addresses the challenge of identifying similar operators and modifying operators at runtime in order to facilitate sharing of data and presents the implementation and evaluation of these techniques in a distributed stream query processing system.

Chapter 7: In this chapter we present algorithms that use hierarchical middleware overlays to scalably and efficiently identify operator placements while taking into account the dynamic nature of the system and the availability of operator reuse opportunities.

Chapter 8: We summarize the work presented in this thesis and discuss open issues and possible avenues for future work.

CHAPTER II

A RECOVERY-CONSCIOUS FRAMEWORK

This chapter presents an overview of our recovery-conscious framework for retrofitting fine-grained recovery into a highly concurrent legacy storage system. We present an overview of the storage controller architecture, failure and recovery models. The information presented in this chapter also serves as a common background for Chapters 3 and 4.

2.1 Background

We motivate this research and illustrate the problem we address by considering the storage controllers of some representative storage system architecture. We focus on system recoverability from transient software failures. Storage controllers are embedded systems that add intelligence to storage and provide functionalities such as RAID, I/O routing, error detection and recovery. Failures in storage controllers are typically more complex and more expensive to recover if not handled appropriately.

2.1.1 System Overview

Figure 4 gives a conceptual representation of a storage subsystem. This is a single storage subsystem node consisting of hosts, devices, a processor complex and the interconnects. In practice, storage systems may be composed of one or more such nodes in order to avoid single-points-of-failure. The processor complex provides the management functionalities for the storage subsystem. The system memory available within the processor complex serves as program memory and may also serve as the data cache. The memory is accessible to all the processors within the complex and holds the job queues through which functional components dispatch work. As shown

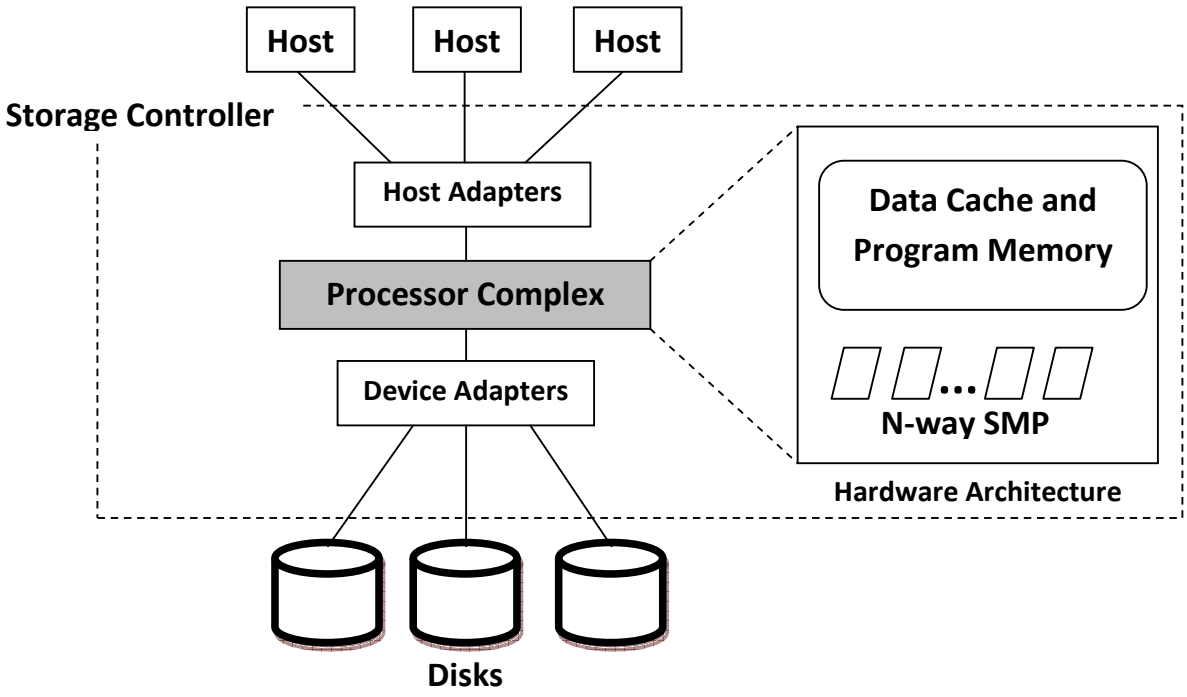


Figure 4: Storage Subsystem Architecture

in Figure 4, this processor complex has a single job queue and is an N-way SMP node. Any of the N processors may execute the jobs available in the queue. Some storage systems may have more than one job queue (e.g. multiple priority queues).

The storage controller software typically consists of a number of interacting components each of which performs work through a large number of asynchronous, short-running threads ($\sim \mu$secs). We refer to each of these threads as a '**task**'. Examples of components include SCSI command processor, cache manager and device manager. Tasks (e.g., processing a SCSI command, reading data into cache memory, discarding data from cache etc.) are enqueued onto the job queues by the components and then dispatched to run on one of the many available processors each of which runs an independent scheduler. Tasks interact both through shared data-structures in memory as well as through message passing.

With this architecture, when one thread encounters an exception that causes the system to enter an unknown or incorrect state, the common way to return the system

to an acceptable, functional state is by restarting and reinitializing the entire system. Since the system state may either be lost, or cannot be trusted to be consistent, some higher layer must now redrive operations after the system has performed basic consistency checks of non-volatile metadata and data. While the system reinitializes and waits for the operations to be redriven by a host, access to the system is lost contributing to the downtime. This recovery process is widely recognized as a barrier to achieving high(er) availability. Moreover, as the system scales to larger number of cores and as the size of the in-memory structures increase, such system-wide recovery will no longer scale.

The necessity to embark on system-wide recovery to deal with software failures is mainly due to the complex interactions between the tasks which may belong to different components. Due to the high volume of tasks (more than 20 million/minute in a typical workload), their short-running nature and the involved semantics of each task, it becomes infeasible to maintain logs or perform database-style recovery actions in the presence of software failures. Often such software failures need to be explicitly handled by the developer. However, the number of scenarios are so large, especially in embedded systems, that the programmer cannot realistically anticipate every possible failure. Also, an individual developer may only be aware of the clean-up routines for the limited scope being handled by them. This knowledge is insufficient to recover the entire system from failures, given that often interactions among tasks and execution paths are determined dynamically.

The discussion above highlights some key problems that need to be addressed in order to improve system availability and provide scalable recovery from software failures. Concretely, we must answer the following questions:

- How do we implement fine-grained recovery in a highly concurrent system?
- How do we identify recovery dependencies across tasks?
- How do we ensure availability of the system during a recovery process?

- What are important factors that will impact the recovery efficiency?

In addition to maintaining system performance while reducing the time to recovery, another key challenge in developing a scalable solution is to ensure that the recovery-conscious framework is non-intrusive i.e., does not affect performance during normal operation and minimize re-architecting of the legacy application code.

2.1.2 Taxonomy of Failures

Studies classify software faults as both permanent and transient. Gray [72] classifies software faults into *Bohrbugs* and *Heisenbugs*. Bohrbugs are essentially deterministic bugs that may be caused due to permanent design failures. Such bugs are usually easily identified during the testing phases and are weeded out early in the software life cycle. On the other hand, 'heisenbugs' which are transient or intermittent faults that occur only under certain conditions are not easily identifiable and may not even be reproducible. Such faults are often due to reasons such as the system entering an unexpected state, insufficient exception handling, boundary conditions, timing/concurrency issues or due to other external factors. Many studies have shown that most software failures occurring in production systems are due to transient faults that disappear when the system is restarted [72, 49, 94].

Our work is targeted at dealing with such transient failures in a storage software system and in particular the embedded storage controller's microcode. Below, we provide a classification of transient failures which we intend to deal with through localized recovery.

In complex systems, often code paths are dynamic and input parameters are determined at runtime. As a result many faults are not caught at compile time. On pure functions, faults may be classified as:

- **Domain errors:** are caused by bad input arguments, such as a divide by zero error or when each individual input is correct, but the combination is wrong (e.g.

negative number raised to a non-integral power in a real arithmetic system).

- **Range errors:** are caused when input arguments are correct, but the result cannot be computed (such as a result which would cause an overflow).

With actions based on system state there are additional complexities. For example, a configuration issue that appeared early in the installation process may have been fixed by trying various combinations of actions that were not correctly undone. As a result the system finds itself in an unknown state that manifests as a failure after some period of normal operation. Such errors are difficult to trace, and although transient may continue to appear every so often. We classify such system state based errors as:

- **State error:** where the input arguments are wrong for the current state of the object.
- **Internal logic error:** where the system has unexpectedly entered an incorrect or unknown state. Such an error often triggers further state errors.

Each of the above error types can lead to transient failures. Some transient failures can be fixed through appropriate recovery actions that may range from dropping the current request to retrying the operation or performing a set of actions that take the system to a known consistent state. For example, some of such transient faults that occur in storage controller code are:

- Unsolicited response from adapter: An adapter (a hardware component not controlled by our microcode) sends a response to a message which we did not send - or do not remember sending. This is an example of a state error.
- Incorrect Linear Redundancy Code (LRC): A control block has the wrong LRC check bytes, for instance, due to an undetected memory error; an example of an internal logic error.
- Queue full: An adapter refuses to accept more work due to a queue full condition;

an example of both an internal logic error and state error.

In addition, there are other error scenarios such as violation of a storage system or application service level agreements. The 'time-out' conditions are also common in large scale embedded storage systems. While the legacy system grows along multiple dimensions, the growth is not proportional along all dimensions. As a result hard-coded constant timeout values distributed in the code base often create unexpected artificial violations.

2.1.3 Recovery Models

Intuitively we can see that localized recovery may be possible for many of the failure scenarios outlined above, and thus system-wide software reboots can be avoided. Sometimes even for situations of resolving deadlock or livelock, it may be sufficient if a minimal subset of tasks or components of the system undergo restarts (e.g., deadlock resolution in transactional databases [73]). Of course there are scenarios, such as severe memory corruption, where the only high-confidence way of repairing the fault is to perform system-wide clean-up.

In production environments, techniques for fault-tolerance, i.e., coping with the existence and manifestation of software faults can be classified into two primary categories with respect to the fault repairing methods: (1) those that provide *fault treatment*, such as restarts of the software, rebooting of the system and utilizing process pair redundancy; and (2) those that provide *error recovery*, such as check-pointing and log-based recovery. Alternatively, one can categorize the recovery models based on the granularity of the recovery scopes. All the above-mentioned techniques could be applied to any recovery scope. In our context, we consider the following three types of recovery scopes:

- **System level:** Performing fault treatment at this level has proven to be an effective high-confidence way of recovering the system from transient faults [50], but

has a high cost in terms of recovery time and the resulting system downtime. On the other hand performing error recovery at the system level through checkpointing and recovery can be prohibitively expensive for systems with high volumes of workload and complex semantics.

- **Component level:** Both fault treatment and error recovery are more scalable and cost effective at this granularity. For fault treatment, the main challenge is identifying these 'component boundaries' especially in systems that do not have well defined interfaces. Again, the difficult hurdle to performing checkpoint/log-based error recovery at this level is understanding the semantics of operations.
- **Task level:** At this fine-grained level, the issue of operational semantics still remains. However, performing fault treatment at this level is efficient both in terms of cost and system availability.

The main advantage of performing error recovery or fault-treatment at the task-level as compared to the component-level, is that it allows us to accommodate cross-component interactions and define 'recovery boundaries' in place of 'component boundaries'. Our goal is to handle most of the failures and exceptions through task-level (localized) recovery, and avoid resorting to system-wide recovery unless it is absolutely necessary.

2.2 Recovery Conscious Framework

Transactional recovery in relational DBMSs is a success story of fine-grained error recovery, where the set of operations, their corresponding recovery actions and their recovery scopes are well-defined in the context of database transactions. However, this is not the case in many legacy storage systems. For example, consider the embedded storage controller in which tasks executed by the system are involved in more complex operational semantics, such as dynamic execution paths and complex interactions with other tasks. Under these circumstances, in order to implement task-level

recovery, we have to deal with both the semantics of recovery and the identification of recovery scopes.

Recovery from a software failure involves choosing an appropriate strategy to treat/recover from the failure. The choice of recovery strategy depends on the nature of the task, the context of the failure, and the type of failure. For example, within a single system, the recovery strategy could range from continuing the operations (ignoring the error), retrying the operation (fault treatment using environmental diversity) or propagating the fault to a higher layer. In general, with every failure context and type, we could associate a recovery action. In addition, to ensure that the system will return to a consistent state, we must also avoid deadlock or resource hold-up situations by relinquishing resources such as hardware or software locks, devices or data sets that are in the possession of the task.

2.2.1 Overview

With these observations in mind, we develop a recovery conscious framework for multi-core architectures and a suite of techniques for improving the failure resiliency and recovery efficiency of highly concurrent embedded storage software systems. The main contributions of our recovery conscious framework include:

1. A task-level recovery model, which consists of mechanisms for classifying storage tasks into 'recovery scopes' based on both programmer specified and system-defined recovery dependencies; all tasks that must undergo recovery simultaneously fall into the same recovery scope;

2. A recovery-conscious mapping and realignment of 'recovery scopes' (identified from the previous step) into 'recovery groups'. Recovery groups additionally take into consideration, parameters such as, system size, failure rates, recovery rates, workload distribution, performance overhead and availability requirements.

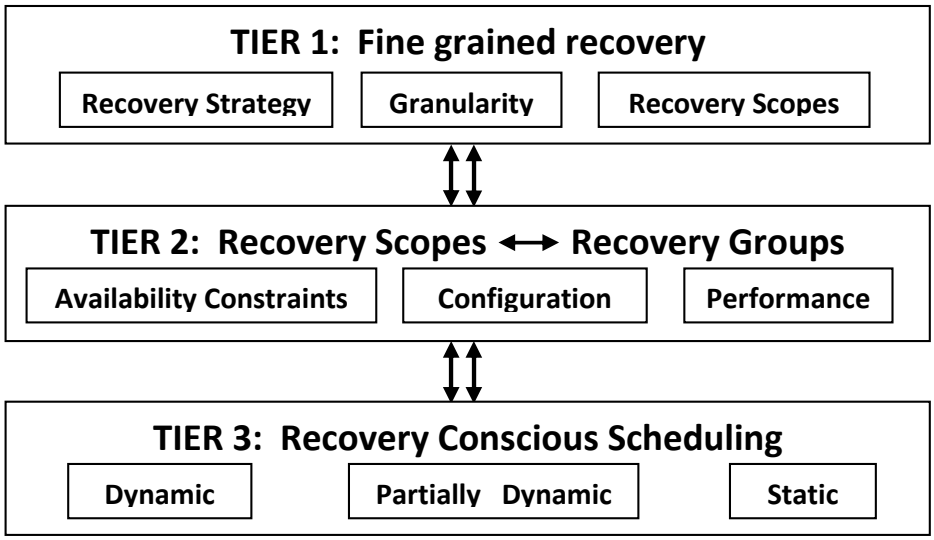


Figure 5: Recovery-Conscious Framework

3. A recovery-conscious scheduling, which enforces some serializability of failure-dependent tasks, i.e., tasks belonging to the same recovery group, in order to reduce the ripple effect of software failures and improve the availability of the system.

In this chapter we give an overview of our recovery-conscious framework, which is designed for improving recovery efficiency and system fault resilience. Here, fault-resilience refers to the ability to reduce system recovery time and sustain good performance even during failure recovery. Figure 5 provides a schematic representation of our framework. The shaded boxes in each tier represent the parameters to be considered at that tier. The framework achieves its goals progressively through three consecutive stages detailed next.

2.3 Tier 1: Fine Grained Recovery

The first tier of the framework addresses the issue of identifying recovery dependencies between concurrent tasks and defining a task-level recovery mechanism which

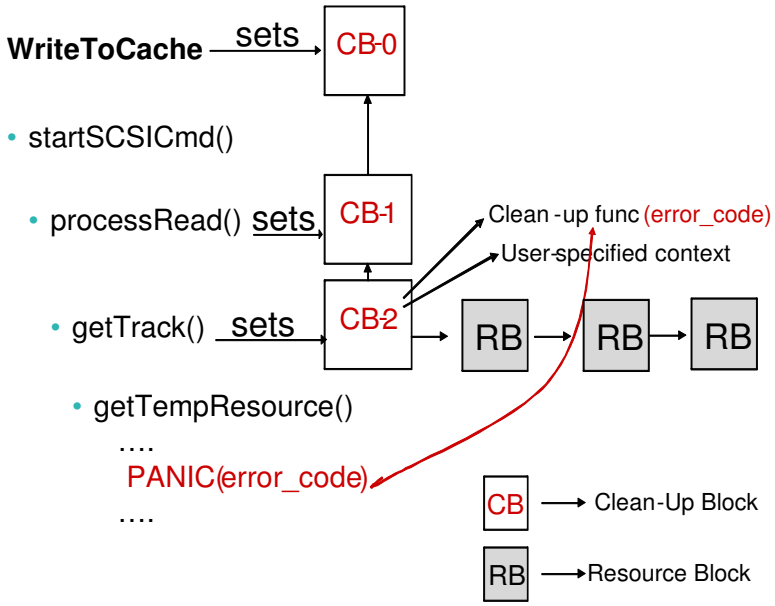


Figure 6: Framework for Task Level Recovery

allows recovery strategies to be determined dynamically based on failure and recovery context.

2.3.1 Task-level Recovery Mechanism

In our framework, we refer to the execution point to which control is returned after recovery is completed as a **recovery point**. The framework provides mechanisms for developers to define **clean-up blocks** which are recovery handlers. Each recovery point is associated with a clean-up block. The clean-up block encapsulates failure codes, the associated recovery actions, and resource information. The specification of the actual recovery actions in each of the clean-up blocks is left to the developers due to their task-specific semantics. We defer the discussion of the structure and contents of clean-up blocks and the state restoration actions to Chapter 3.

Example : We describe the selection of recovery strategy and design of clean-up blocks using an example from our storage controller implementation. Consider the error described in Figure6 which depicts relevant portions of the call stack. The failure situation described in this example is similar to the commonly used 'assert' programming construct. The error is encountered when a task has run out of a

temporary cache data structure known as a 'control block' which is not expected to occur normally and hence results in a 'panic'.

The Figure 6 shows the schematic diagram of the recovery framework using the call stack of a single task that performs a write to cache. The developer explicitly specifies the corresponding clean-up block for an identified recovery point in the code. At runtime the framework takes care of chaining clean-up blocks dynamically as the code passes through various recovery points. As the task moves through its execution path, it passes through multiple recovery points and accumulates clean-up blocks. When the task leaves a context, the clean-up actions associated with the context go out of scope. On the other hand, nesting of contexts results in the nesting of the corresponding clean-up blocks and the framework keeps track of necessary clean-up blocks.

The clean-up blocks are gathered and carried along during task execution but are not invoked unless a failure occurs. Resource information can also be gathered passively. Such a framework allows a choice of recovery strategy based on task requirements and requires minimal rearchitecting of the system.

In this particular situation, ignoring the error is not a possible recovery strategy since the task would be unable to complete until a control block is available. One possible strategy is to search the list of control blocks to identify any instances that are not currently in use, but have not been freed correctly (for example, due to incorrect flags). If any such instances exist, they could be made available to the stalled task. An alternative strategy would be to retry the operation beginning at the 'WriteToCache' routine at a later time in order to work around concurrency issues. Retrying the operation may involve rolling back the resource and state setup along this call path to their original state. Resource blocks are used to carry the information required to successfully execute this strategy. Finally, in the case of less critical tasks, aborting the task may also be an option. Alternatively, consider a situation where an error is

encountered due to a component releasing access to a track to which it did not have access in the first place. The error was caused due to a mismatch in the number of active users as perceived by the component. In this case, a possible recovery strategy would be to correctly set the count for the number of active users and proceed with the execution, effectively ignoring the error.

Note that, it is important we ensure that the interfaces with the recovery code and the recovery code itself are reliable. In our implementation, the recovery-conscious scheduler alone was implemented in approximately 1000 lines of code. A naive coding and the design effort for task level recovery would be directly proportional to the number of "panics" or failures in the code that are intended to be handled using our framework. In general, the coding effort for a single recovery action is small and is estimated to be around a few tens of lines of code (using semicolons as the definition of lines of code) per recovery action on average [15]. Note that, the clean-up block does not involve any logging or complex book-keeping and is intended to be light-weight. A more efficient handling of clean-up blocks would involve classifying common error/failure situations and then addressing the handling of the errors in a hierarchical fashion. For example, recoveries may be nested and we could re-throw an error and recover with the next higher clean-up block defined in the stack. This would involve design effort toward the classification of error codes into classes and sub-classes and identification of common error handling situations. Finally, if we are unable to address an error using our framework, existing error handling mechanisms would be used as default. The point of recovery in the stack may be determined by factors such as access to data structures and possibilities of recovery strategies such as retrying, termination or ignoring the error.

2.3.2 Recovery Scopes

Performing fine-grained recovery in response to storage controller failures, first requires an understanding of recovery-dependencies between tasks i.e. concurrent threads in the firmware. We refer to the scope of a recovery action as a 'recovery scope'. Tasks interact with each other in complex ways. When a single task encounters an exception, more than one task may need to initiate recovery procedures in order to avoid deadlocks and return the system to a consistent state. Explicit recovery-dependencies can be specified by the programmer. However, explicit dependencies specified by the programmer may be very coarse. Likewise, some dependencies may have been overlooked due to their dynamic nature and the immense complexity of the system. Therefore one way to refine explicit dependencies is to identify implicit dependencies continuously and utilize them to refine the developer-defined recovery scopes over time. The criteria for classification of tasks into recovery scopes depends on the nature of the application and failures that are intended to be handled. In our work, we identify the following three classifications of tasks into recovery scopes.

2.3.2.1 Resource-based

Tasks accessing the same resources (such as device drivers or metadata) may be classified under the same recovery scope. This classification would be effective to deal with resource-based failures. For example, consider a 'queue full condition' that occurs in storage controllers. This error occurs when an adapter refuses to accept more work due to a queue full condition. Under these circumstances, the error and the subsequent recovery action would probably affect only the tasks attempting to write to the faulty adapter. One method to identify resource-based recovery dependencies is to observe the pattern of lock acquisitions. The intuition here is that, tasks that access the same resource are likely to acquire common locks. Lock acquisitions patterns can potentially be used to further refine resource-based dependencies at runtime by

utilizing the temporal aspect of dependencies apart from the spatial aspect (discussed in Chapter 3).

Resource-based dependencies can be identified in two ways - static code analysis or through analysis of traces collected from actual workload execution. An alternative method would be to dynamically discover these dependencies during runtime. However, the disadvantage of a dynamic approach is that the dependencies (lock acquisitions) manifest only after the thread has been dispatched. Assigning recovery scopes after dispatch would be meaningless, unless the tasks can be immediately suspended and again enqueued in the appropriate recovery group queue. However, this results in a performance penalty due to the high amount of context switching. We therefore recommend an initial static assignment of tasks into recovery scopes which can then be continuously refined at runtime based on dependencies observed from locking patterns. Logging lock acquisitions at runtime is also essential in order to keep track of resource ownership and perform clean-up of resources in the event of a failure. The disadvantage of a static approach compared to a dynamic approach is the possibility of an inaccurate recovery scope assignment. However, a wrong classification of tasks into recovery scopes will not affect the consistency of results, but only the recovery efficiency and performance during failure recovery. We show in Chapter 4 that our framework can tolerate some inaccuracy.

2.3.2.2 Component-based

Even in the absence of well-defined operational boundaries between functional components certain failures may require resetting state or performing recovery actions for tasks belonging to a particular functional component. Recall the example described in Section 2.3.1 which illustrated an error encountered when a 'WriteToCache' task fails due to the unavailability of a data structure. One recovery strategy in this situation is to search the list of control blocks to identify instances that have not been freed up

correctly. Another strategy is to retry the operation at a later time, in order to work around concurrency issues. However, it is very likely that other tasks belonging to the cache component are likely to encounter the same error if executed before the issue is resolved. Moreover, modifying the data structures or checking them for consistency may require suspension of dispatch of tasks belonging to the cache component until recovery completes. In such scenarios, a functional component based classification of tasks may be effective in identifying recovery dependencies. With component-based recovery scopes, all tasks belonging to the same functional component are classified under the same recovery scope.

2.3.2.3 Request-based

To deal with errors that require aborting or recursively recovering a user-request, it may be beneficial to classify tasks on the basis of user requests or workflows; for instance, consider a situation where a read/write request fails due to an invalid address specification. In this situation we may choose a recovery strategy of performing necessary clean-up actions and then aborting the request. Then the scope of recovery is all tasks across all components that are a part of this user request.

Depending upon the nature of failures that fine-grained recovery is expected to handle, one class or a valid combination of the above classifications may be used to define recovery-scopes. The top-tier of the framework identifies recovery scopes based on such explicitly specified (as in the case of component or request based grouping) and implicitly discovered (as in the case of resource based grouping) recovery dependencies.

2.4 Tier 2: Mapping Tasks to Recovery Groups

The second tier of the framework maps the recovery scopes identified from the previous tier into recovery groups on which the scheduling of tasks during the recovery process is based.

Experiments with a state-of-the-art enterprise storage controller have shown that the clustering of tasks into recovery scopes (dependent tasks belonging to the same scope) leads to a large number of scopes over which tasks are unevenly distributed; most scopes contain a small number of tasks whereas a small number of scopes contain large number of tasks. Clearly, tracking dependencies at a coarse granularity may result in a recovery scope with many tasks whose activations have to be serialized. This is likely to decrease the opportunities for parallel execution on the multi-core architecture and also likely to increase processing overhead during recovery. At the same time, tracking dependencies at too fine a granularity increases the overhead of managing a large number of recovery scopes resulting in a performance penalty. Mapping of recovery scopes to recovery groups is intended to trade-off the performance penalty for tracking fine-granularity recovery scopes versus the recovery efficiency penalty for tracking coarse granularity recovery scopes.

In Chapter 4, based on our analysis we present guidelines for determining the recovery scopes, the recovery groups, and the mapping of recovery scopes to recovery groups for use in scheduling. We implemented this approach in a realistic environment by using an enterprise-class storage controller with minimal changes to its software. Handling of various failures in the system can be implemented incrementally. We show that by selecting appropriate values for the recovery-sensitive system parameters it may be possible to speed up the recovery of storage controllers and achieve good performance at the same time.

2.5 Tier 3: Recovery Conscious Scheduling (RCS)

An important goal for providing fine-grained recovery (task or component level) is to improve recoverability and make efficient use of resources on the multi-core architectures. This ensures that resources are available for normal system operation in spite of some localized recovery being underway and that the recovery process is bounded

both in time and in resource consumption. Without careful design, it is possible that more dependent tasks are dispatched before a recovery process can complete, resulting in an expansion of the recovery scope or an inconsistent system state. This problem is aggravated by the fact that recovery takes orders of magnitude longer (ranging from milliseconds to seconds) compared to normal operation ($\sim \mu$ secs). Also a dangerous situation may arise where it is possible that many or all of the threads that are concurrently executing are dependent, especially since tasks often arrive in batches. Then the recovery process could consume all system resources essentially stalling the entire system.

Ideally we would like to "fence" the failed and recovering tasks until the recovery is complete. In order to do so we must control the number of dependent tasks that are scheduled concurrently, both during normal operation and during recovery. In Chapter 4 we discuss how to design a recovery conscious scheduler that can control how many dependent tasks are dispatched concurrently and what measures should be taken in the event of a failure.

Chapter 4 presents three RCS algorithms, each using a different methods of mapping recovery groups to processing resources: static, partially dynamic, and dynamic. Each mapping technique represents different trade-offs between system availability and system performance under normal operation.

Static scheduling of recovery groups determines the mapping of recovery groups to processors at compile time and is effective in situations where task dependencies during recovery are well understood and the workloads are stable. With this scheme, tasks are dispatched only on processors associated with the recovery group they belong to. Dynamic scheduling of recovery groups to processing resource pools represents the other end of the spectrum. This scheme works effectively, even in the presence of frequently changing workloads. With dynamic RCS, all processors are mapped to all recovery groups. The scheduler then uses a starvation-avoiding scheme

such as round-robin to iterate through the groups and dispatch work. However, a recoverability constraint is specified for each group. A recoverability constraint prescribes the maximum number of concurrently executing tasks permissible for that group. In order to achieve acceptable utilization, the constraint is selectively violated when no task satisfying the constraint is found while resources are idle. Between the two ends of the spectrum is the partially dynamic scheduling, which involves partially static scheduling for those recovery groups whose resource demand is stable and well understood and dynamic scheduling for the remaining recovery groups.

RCS incorporates two countermeasures, one proactive and one reactive. The proactive measures comes into play during normal operation when RAS attempts to minimize the number of dependent tasks executing concurrently by dispatching tasks from different recovery groups using one of the static, dynamic and partially-dynamic algorithms. The reactive technique comes into play during failure recovery, when based on recovery dependencies information afforded by recovery groups, RAS suspends the dispatching of tasks from those recovery groups whose tasks are currently undergoing recovery.

2.6 Discussion

One of the requirements of our recovery-conscious framework is the need for the programmer to specify the recovery handler. We acknowledge that writing error-recovery code is a complex task. However, there are a number of reasons for this requirement. As a first step, our framework provides guidance for identifying dependencies (discussed in Chapter 3) and also ensures that recovery handlers are non-intrusive and have minimal impact on good-path execution.

Second, our analysis of the software shows that due to the complexity of the system, not all failures can be recovered using fine-grained recovery. The recovery strategies are often determined by the semantics of the failure and the nature of the

tasks that encountered the failure. For example, a straightforward recovery strategy for a failure during a back-ground task that is not critical, can be to simply ignore the failure, while the strategy may be different for a critical task that must complete on time. Such task specific semantic based recovery handlers are best defined by the developers of these tasks. Due to the complex semantics involved, a fully automated approach to determining the recovery strategies, without programmer assistance, is difficult and less effective. In the event that the developer specifies an ineffective recovery strategy, such that the problem causing the failure is not resolved, our framework recommends setting a recovery threshold which specifies the number of times that the micro-recovery should be attempted before falling back to system-level recovery. If the failure is not prevented by the micro recovery mechanism and the failure threshold has been reached, then system-level recovery will be performed.

Finally, we would like to point out that, in the case of identifying recovery dependencies, our framework combines the programmer assistance with system initiated learning. While the programmer is given the facility to provide explicit dependencies based on experience, the system uses the programmer's specification of dependencies as a starting point and continues to refine these dependencies throughout the life cycle of the system. In other words, the recovery conscious framework does not rely on the completeness or the correctness of developer's specified dependencies. In the next chapter we describe an access-log based architecture that utilizes the information provided by lock accesses as a guideline to understanding system state changes. Based on such interactions between concurrent tasks, the architecture dynamically identifies dependencies at runtime and alerts the developer to events like dirty reads of shared state. This log-based architecture will relieve the developer from the burden of tracking resources such as shared buffers and locks and tracking read-write conflicts on shared state.

2.7 Related Work

Techniques that improve dependability of software can be classified into those that are applied during the construction of the software and those that are applied during verification and validation of the software [113, 98]. Fault avoidance techniques aim at avoiding faults through appropriate design and development processes. Fault tolerance techniques improve dependability through appropriate construction that allows the software to tolerate or avoid faults during operation. Other techniques like fault removal, for example software testing [102], are used during the validation stage of the software (usually after completion of development).

Clear specification of system requirements, good design and software engineering processes [113, 142, 65] are critical in avoiding design faults that result in software failures. Besides these, mathematical and formal methods [58, 113] are used to verify and validate the software design and specification. However, such methodologies are complex, at least as large as the software itself and often prohibitively expensive for large software projects, especially since the methodologies are themselves susceptible to error. Both fault avoidance and fault removal techniques are orthogonal to our approach and are required at different stages of the software life cycle in order to develop robust software.

There is a large body of existing work that addresses the issue of fault tolerance in general. Techniques for fault tolerance can be classified into fault treatment and error processing. Fault treatment aims at avoiding the activation of faults through environmental diversity, for example by rebooting the entire system [72, 148], micro-rebooting sub-components of the system [50], through periodic rejuvenation [85, 68] of the software, or by retrying the operation in a different environment [114]. Error processing techniques are primarily checkpointing and recovery techniques [73], application-specific techniques like exception handling [137] and recovery blocks [116] or more recent techniques like failure-oblivious computing [119].

However we are faced with several unique challenges in the context of storage firmware and middleware. First, at the storage controller layer, the software being legacy code rules out re-architecting the system. Second, system-level reboots are expensive and this method of recovery would not scale with system size. On the other hand, the tight coupling between components makes both micro-reboots and periodic rejuvenation tricky. Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software [49]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are loosely defined. In addition, the scenario where components are stateful and interact with other components through globally shared structures (data-structures, metadata), often leads to components modifying each other's state irreversibly. Moreover, resources such as hardware and software locks, devices and metadata are shared across components. Under these circumstances, the scope of a recovery action is not limited to a single component.

Checkpointing for fault-tolerance is a well known technique [62, 63, 117, 91, 114] that has also been applied to deterministic replay for software debugging [143, 121, 122]. However, checkpointing techniques are mostly targeted at long-running applications [62] such as scientific workloads [63], or applications where the memory footprint and the system performance requirements can tolerate the overhead imposed by checkpointing [114, 91]. A number of unique challenges in the case of storage controller software make checkpointing infeasible: Unlike long-running applications, storage controllers have a high rate of short ($< 500\mu$secs) concurrent threads and are designed to support extremely high throughput and low response times. Given the highly concurrent nature of controllers, both quiescing the system in order to take the checkpoint, as well as logging the tasks in order to redrive work beyond the checkpoint is expensive in terms of time and space - especially since system state includes large amounts of metadata and cached data. Next, communication with OWPs such

as hosts and media cannot be rolled back and hence invalidates checkpoints. Finally, due to the complexity of the code, not all failures will be amenable to micro-recovery, making checkpointing too heavy weight.

Failure-oblivious computing [119] introduces a novel method to handle failures - by ignoring them and returning possibly arbitrary values. This technique may be applicable to systems like search engines where a few missing results may go unnoticed, but is not an option in storage controllers.

2.8 Summary

This chapter presented an overview of storage controller architectures and our recovery conscious framework which divides the task of retrofitting fine-grained recovery into highly concurrent legacy storage software into three stages. The stages progressively identify recovery dependencies and strategies, organize dependent tasks into groups for enforcing scheduling constraints and finally maps these groups to processing resources through recovery conscious scheduling. The next chapter is dedicated to tier 1 issues while chapter 4 discusses tier 2 and tier 3 issues.

CHAPTER III

STATE RESTORATION DURING MICRO-RECOVERY

3.1 Introduction

Enabling fine grained recovery can be challenging, especially in legacy systems, and to perform micro-recovery, the following issues must be addressed:

- ***Evaluating recovery success:*** What are the failures that can effectively and efficiently be recovered from, using micro-recovery?
- ***Determining recovery actions:*** What are the recovery strategies and recovery actions that must be performed in order to restore the system from an error state to an error-free state?
- ***Identifying dependencies:*** Given the large number of dynamic dependencies possible in a highly concurrent system, what is the scope of fine-granularity recovery?
- ***Enhancing recovery success and efficiency:*** How can we enhance the system to facilitate better recovery success and efficiency?

In this Chapter, we address the first three questions, focusing on the challenges of tracking and restoring system state during micro-recovery, evaluating the possibility of recovery success and determining recovery actions based on system state. The fourth challenge, i.e. ensuring resource availability during failure recovery and improving recovery efficiency and the probability of recovery success is addressed in the next Chapter (Chapter 3).

We make two unique contributions in terms of effective state restoration during micro-recovery. First, by analyzing the system state space, we identify the set of

events and system states that affect state restoration from the perspective of micro-recovery. We introduce the concepts of *Restoration levels* and *Recovery points* to capture failure and recovery context and describe how to flexibly evaluate the possibility of recovery success. Based on the restoration levels and recovery points, we introduce *Resource Recovery Protocol (RRP)* and *State Recovery Protocol (SRP)*, which provide rules to guide state restoration.

Our second contribution is Log(Lock), a practical and lightweight architecture to track dependencies and perform state restoration in complex, legacy software systems. Log(Lock) passively logs system state changes to help identify dependencies between multiple threads in a concurrent environment. Utilizing this record of state changes and resource ownership, Log(Lock) provides the developer with the failure context necessary to perform micro-recovery. Recovery points and their associated recovery handlers are specified by the developer. Log(Lock) is responsible for tracking dependencies and computing restoration levels at runtime.

We have implemented and evaluated Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient (<10% impact on performance) and effective (reduces a four second downtime to only a 35% performance impact lasting six seconds). In summary, micro-recovery with Log(Lock) presents a promising approach to improving storage software robustness and overall storage system availability.

3.2 Log(Lock): Design Overview

The key challenges in performing micro-recovery are identifying dependencies based on failure and recovery context, determining recovery actions and restoring the system to a consistent state after a failure. In this section we outline the technical challenges for systematic state restoration during micro-recovery and present an overview of the Log(Lock) architecture for state restoration. Using examples, we highlight the

unique characteristics of storage software recovery in terms of state consistency and state restoration efficiency. Finally, we briefly describe the system architecture of Log(Lock).

3.2.1 Technical Challenges

With software recovery, the actions that achieve state restoration depend on the actions of the failed thread and its interactions with state and shared resources.

Threads in the system interact in two fundamental ways: (1) reading/writing shared data and (2) acquiring and releasing resources from/to a common pool. Threads also interact with the outside world through actions such as positioning a disk head or sending a response to an I/O. Often these actions cannot be rolled back and are referred to as *outside world processes (OWP)* [62]. In such a system, state restoration and micro-recovery must consider the sequence and interleaving of the actions of concurrent threads that gives rise to the following conflicts:

- **Dirty Reads (Write-Read Conflict):** Data written by the failed thread has already been consumed by another thread.
- **Lost Updates (Write-Write Conflict):** Rolling back the failed thread may cause the updates of other threads to be overwritten or lost.
- **Unrepeatable Reads (Read-Write Conflict):** The value of the shared state variable required by the failed thread has already been overwritten.
- **Resource Ownership :** The failed thread may continue to be in the possession of resources from a shared pool or may be holding a lock resulting in resource leaks or starvation issues.

The above taxonomy is derived from that used to describe concurrency control concepts in transaction processing systems [73]. For a given failure, the set of recovery actions that need to be performed to return the system to a consistent state may vary depending upon the failure and the occurrence of one or more of the above conflicts.

Note that for application state, the intention is not to deterministically replay the events before the failure, or recover the application state to exactly as it was at the instant of failure. Rather, the goal is to restore the system to an error-free state. In fact, the recovery strategy may itself explicitly rely on non-determinism to remove transient failures. For example, Rx [114] demonstrates an interesting approach to recovery by retrying operations in a modified environment using checkpointed system states for rollbacks.

Recall that checkpointing techniques are mostly targeted at long-running applications [62] such as scientific workloads [63], or applications where the memory footprint and the system performance requirements can tolerate the overhead imposed by checkpointing [114, 91]. Given the highly concurrent nature of controllers, both quiescing the system in order to take the checkpoint, as well as logging the tasks in order to re-execute work beyond the checkpoint is expensive in terms of time and space - especially since system state includes large amounts of metadata and cached data. Next, communication with OWPs such as hosts and media cannot be rolled back and hence invalidate checkpoints. Finally, due to the complexity of the code, not all failures will be amenable to micro-recovery, making checkpointing too heavy weight.

System state restoration and conflict serialization is also of interest to transactional systems [100]. Transactional databases use schemes like strict 2-phase locking (2PL) to guarantee conflict serializability [45]. However, such techniques can increase the length of critical sections (i.e. durations of locks) and are inefficient for storage controllers that execute in a highly concurrent environment. Moreover, we show in Section 3.2.2 that, recovery actions are determined based on both the context and semantics of failure and a "one size fits all" serializability, while simplifying recovery procedures, can constrain the recovery process.

```
R1:  /* Increment number of Users */
     lockWrite( &numActiveUsersLock);
     numActiveUsers ++;
     unlockWrite( &numActiveUsersLock);
     ...
     ...
    /* Decrement number of Users */
    lockWrite( &numActiveUsersLock);
    numActiveUsers --;
    unlockWrite( &numActiveUsersLock);
```

```
R2: /* Start background tasks if no users active */
     lockRead ( &numActiveUsersLock);
     if ( numActiveUsers == 0 ) {
      Start performing background tasks.
     }
     unlockRead( &numActiveUsersLock);
```

Figure 7: Example 1: Lost Update Conflict

3.2.2 Examples

We present three real examples from a storage controller software. We demonstrate how the semantics and success of fine-grained recovery are determined by failure context and the interactions of threads.

Figure 7 shows two code snippets: R1 increments the number of active users before performing work and in R2, a background job is triggered when there are no active users in the system. When a panic (user defined or system failure/exception) occurs during the execution of region R1, then assume that the micro-recovery strategy is to reattempt execution of region R1. The recovery action must ensure clean relinquishing of resources such as the lock *numActiveUsersLock*. It is important to ensure that the system state is consistent since corruption of the counter can either cause the

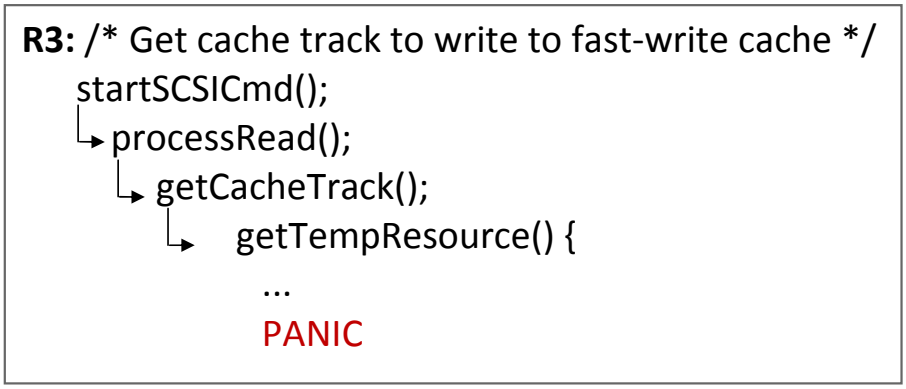


Figure 8: Example 2: Resource Ownership Conflict

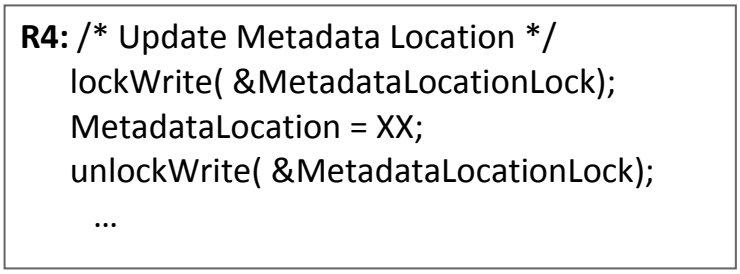


Figure 9: Example 3: Dirty Read Conflict

background jobs to never be triggered or to be triggered in the presence of active users. In Example-1, the system can tolerate dirty reads or unrepeatable reads of the *numActiveUsers* count but must ensure that no updates are lost. On the other hand, if the failure was caused during the execution of region R2, an idempotent background task that is not critical, the recovery strategy may be to just abort the current execution of the background task. However, recovery must ensure that the lock *numActiveUsersLock* has been released.

Figure 8 shows the processing of a write command. In the event of encountering a failure, state restoration must ensure that temporary resources obtained from a shared pool are freed correctly in order to avoid resource leaks or starvation. It may also require that certain cache tracks are checked for consistency, depending upon the point of failure. However, for a resource such as a buffer or empty cache track

obtained from a shared pool for exclusive use, dirty reads, unrepeatable reads and lost updates can be tolerated.

Figure 9 shows a thread that updates a global variable indicating the metadata location, such as for checkpoint activity. In the event of a failure caused due to a failed location, the thread may have the opportunity to modify the location without notifying other threads in the system or causing inconsistency, provided there have been no dirty reads. However, in the event of a dirty read, the system may have to resort to recovery at a higher level.

These examples highlight the fact that consistency requirements for state restoration vary with failure context. For example, in the case of a counter generating unique numbers, the only requirement may be that modifications are monotonous. For a shared resource, the state remains consistent as long as there are no resource leaks that could eventually lead to starvation and system unavailability. Unlike a transactional system, where similar problems are addressed, the semantics of the state and failure may render certain types of conflicts irrelevant from the perspective of system availability and fault tolerance. This emphasizes the need for a flexible state restoration architecture that is also lightweight and efficient, there by allowing the system to sustain high performance.

3.2.3 System Architecture

The Log(Lock) architecture provides support for state restoration during micro-recovery. To achieve this goal, Log(Lock) tracks resources and state dependencies relevant to a thread that has incorporated recovery handlers for micro-recovery.

Figure 10 presents an overview of our system architecture and describes the roles played by the Log(Lock) execution model and restoration protocols. The figure shows a system with concurrently executing threads where the thread depicted by a solid line incorporates micro-recovery mechanisms. In order to facilitate micro-recovery, the

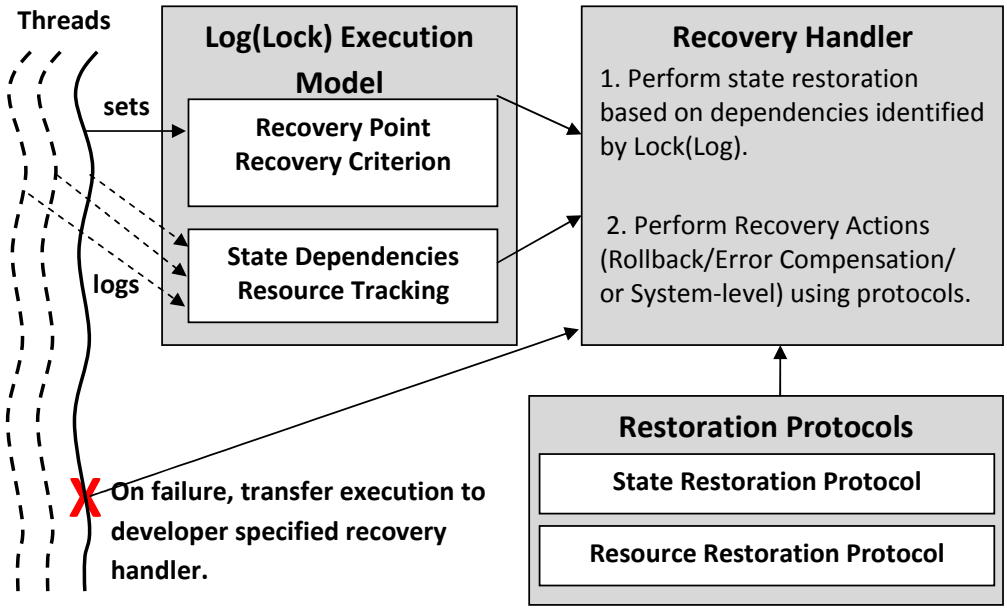


Figure 10: Log(Lock) Architecture Overview

thread sets recovery points during execution, where each recovery point is associated with a recovery criterion. The recovery criterion specifies the conditions that must be satisfied by the failure context in order to use the recovery point as a starting point for recovery. Using the Log(Lock) architecture, the thread (depicted by a solid line) enabled with micro-recovery mechanisms indicates state and resources that are relevant to recovery. Log(Lock) then begins logging all relevant changes and dependencies, based on the actions of both this thread and other concurrent threads (depicted by dotted lines).

In the event of a failure, control transfers to a developer-specified recovery handler. The handler performs state restoration actions by utilizing the resource tracking and state dependency information provided by the Log(Lock) execution model, in consultation with the restoration protocols. It also decides on an appropriate recovery strategy such as rollback, error compensation or system-level recovery. The implementation of the Log(Lock) dependency tracking component must ensure efficiency during normal operation while the recovery protocols ensure consistency of state

Table 1: Valid States for Thread T_i

Notation	Description
T_iS	T_i initial state
T_iR	T_i holds a read lock
T_iW	T_i holds an exclusive write lock
T_iU	T_i has released the lock
T_iF	T_i is in failed state
T_iA	T_i acquired a resource
T_iRe	T_i released a resource
T_iE	T_i performed an externally visible action

restoration during failure recovery.

In the next two sections, we first describe the concepts of 'restoration levels' and 'recovery points' and present the restoration protocols. Then, we present the Log(Lock) execution model and illustrate application of the protocols through example scenarios.

3.3 State Space Exploration

In this section, we model failure scenarios and recovery contexts using a state space analysis approach. Our approach is based on the intuition that in a concurrent system, global state and shared system resources are often protected by locks or similar primitives.

This section is divided into two parts. In the first part, we model system events, state transitions and interleaving of concurrent threads and demonstrate the discrete state space and recovery scenarios. We introduce the concepts of *Restoration Level* and *Recovery Criterion*, that help match a failure context to a recovery strategy. In the second part, we systematically identify the set of recovery strategies that can be applied to each failure scenario and present two protocols for state restoration. The **Resource Recovery Protocol (RRP)** defines the steps to handle resource ownership conditions and the **State Recovery Protocol (SRP)** sets forth the rules to perform state restoration.

3.3.1 Modeling Thread Dependencies

Let $\mathcal{T} = \{T_i | 1 \leq i \leq n\}$ define a system with n concurrent threads. Let $\mathcal{X}_i(t)$ denote the sequence of states of thread T_i up to time t. The schedule $\mathcal{S}(t)$ for the system $\mathcal{T}$ at time t is the interleaving of the sequence of actions in $\mathcal{X}_i(t)$ for each thread T_i. Let v denote a globally shared structure protected by a lock. Table 1 shows the list of valid states for a thread.

The system implements micro-recovery at a thread granularity. Any failure that cannot be handled by micro-recovery is resolved using a system-level recovery mechanism (e.g. software reboots).

The state space for system execution consists of all legitimate schedules $\mathcal{S}(t)$. System states that represent the failed state of one of the executing threads are relevant from the perspective of micro-recovery. To simplify the subsequent discussion, we apply the following rules to reduce the state space:

- We consider the interactions between only two threads T_1 and T_2.
- We only consider system states where the last state of thread T_1 is T_1F.
- Only T_1 encounters a failure. Failures of thread T_2 are symmetric and can be treated similarly.
- Read or write actions performed by T_2 before any such actions by T_1 are ignored.
- We assume that the system can recover from only a single failure. Failure during recovery results in system-level failure recovery.
- The "end" event is equivalent to a commit or externally visible action that cannot be rolled back.

From the perspective of state restoration for micro-recovery, the occurrences of the following patterns in the schedule $\mathcal{S}(t)$ are of interest and relevant to the selection of a recovery strategy by thread T_1. Let $\rightarrow$ denote the "happened before" relation [93].

- **Dirty Read (DR):** $T_1W \rightarrow T_2R \rightarrow T_1F$.
- **Lost Update (LU):** $T_1W \rightarrow T_2W \rightarrow T_1F$.
- **Unrepeatable Read (UR):** $T_1R \rightarrow T_2W \rightarrow T_1F$.
- **Residual Resources (RR):** $(T_1R \rightarrow T_1F) \wedge (T_1U \nrightarrow T_1F)$ or $(T_1W \rightarrow T_1F) \wedge (T_1U \nrightarrow T_1F)$ or $(T_1A \rightarrow T_1F) \wedge (T_1Re \nrightarrow T_1F)$.
- **Committed Dependency (CD):** $T_1W \rightarrow T_2R \rightarrow T_2E \rightarrow T_1F$ or $T_1W \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$ or $T_1R \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$.

To determine the right strategy for recovery, it is important to determine which of the above conflicts have occurred and are relevant to recovery.

Restoration Level: The restoration level $\mathcal{R}_i(t)$ of a thread T_i at instant t, is a 5-tuple $\langle DR, LU, UR, RR, CD \rangle$ indicating the occurrence of dirty reads, lost updates, unrepeatable reads, residual resources and committed dependencies in $\mathcal{S}(t)$.

Recovery Point: A recovery point p_i in thread T_i represents an execution point to which control is transferred at the end of a recovery procedure. A default recovery point defined for all threads is the initial system state.

Recovery Criterion: Each recovery point p_i is associated with a recovery criterion $\mathcal{C}_i$ which is a 4-tuple $\langle DR, LU, UR, RR \rangle$ that represents the set of criteria for dirty reads, lost updates, unrepeatable reads and residual resources, that the system state should satisfy before recovery can be attempted using p_i. For the default recovery point, all elements of the recovery criterion are defined as "don't care".

CD does not figure in the recovery criterion since this information is used only to choose between alternate recovery strategies in the recovery handler. We discuss the use of CD conditions during recovery in the state recovery protocol in Section 3.3.2. In our current design, recovery points and their associated recovery handlers are identified by developers and are associated to an execution context. When a thread leaves

a context, the associated recovery points go out of scope. Within a single execution context, multiple recovery points may be defined, any of which could potentially be used during recovery. Then the appropriate recovery point for the current failure scenario is chosen by the logic in the recovery handler. In the developer-specified recovery handler, the feasibility and correctness of restoring the failed system state using a recovery point, is determined using the resource and state recovery protocols described next. Once the valid recovery points have been identified from the available choices, the selection of an appropriate recovery point and recovery strategy may be a decision depending upon factors such as the amount of resources available for recovery and the time required to complete recovery.

3.3.2 Restoration Protocols

In this discussion, we consider the following possible recovery strategies: (1) Rollback; (2) Roll-forward style recovery or error compensation; (3) System-level recovery [113]. Of these the rollback and error compensation strategies may be applied to the failed thread only (single-thread recovery) or to multiple threads including the failed thread (multi-thread recovery). The following protocols are based on the assumption that committed dependencies cannot be rolled-back.

Resource Recovery Protocol (RRP): System state can be restored to recovery point p_i only if $\mathcal{R}_i(t)$ meets $\mathcal{C}_i$ on the RR criterion. Otherwise, the thread must first attempt to release or acquire resources to meet the criterion.

The state recovery protocol (SRP) specifies the recovery strategies applicable for different failure and recovery contexts. The rationale behind the SRP rules is that an occurrence of DR, LU or UR events imply that an interaction with other concurrent threads in the system have occurred. When the restoration level does not meet the recovery criterion and interactions with other threads have occurred, then single thread recovery is no longer sufficient. Next, the success of multi-thread recovery

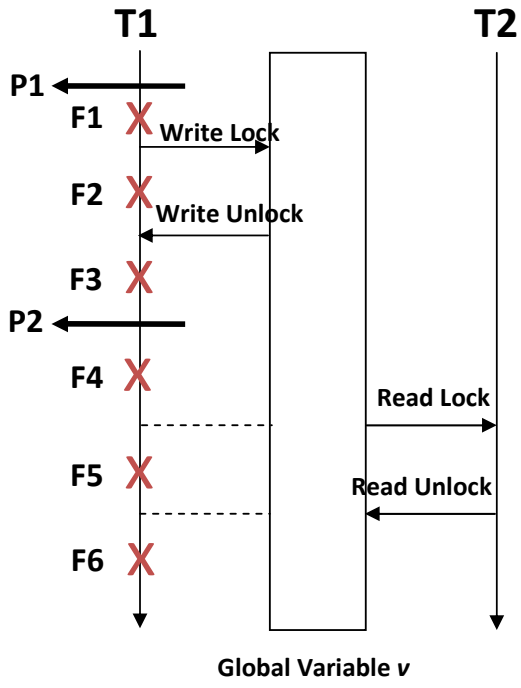


Figure 11: State Recovery with Dirty Reads

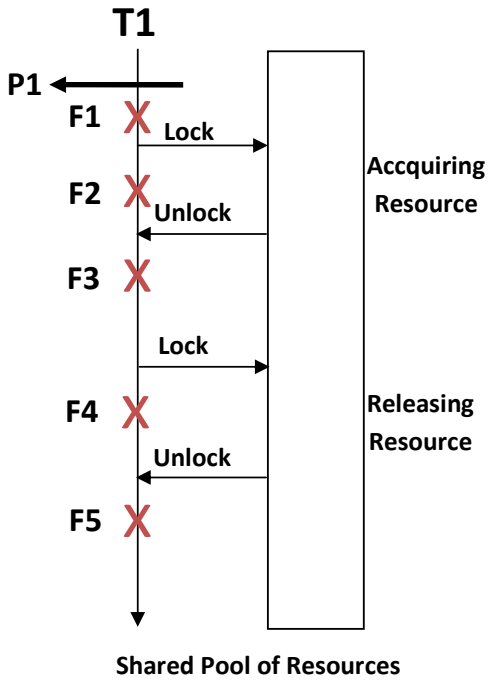


Figure 12: Resource Recovery

depends on the occurrence of an externally visible action and whether the dependency has already been committed. Concretely, the rules of state recovery are:

State Recovery Protocol (SRP): 1. To perform single-thread recovery and restore state to recovery point p_i, $\mathcal{R}_i(t)$ should meet $\mathcal{C}_i$ on every element of $\mathcal{C}_i$.
2. If $\mathcal{R}_i(t)$ does not meet $\mathcal{C}_i$ on DR, LU, UR conditions and CD occurs in $\mathcal{S}(t)$, then only error compensation or system-level recovery can be attempted.
3. If $\mathcal{R}_i(t)$ does not meet $\mathcal{C}_i$ on DR, LU, UR conditions and CD has **not** been observed in $\mathcal{S}(t)$, then only multi-thread rollback, error compensation or system-level recovery is possible.

Figure 11 and 12 show example scenarios with the schedule of execution of T_1 and T_2 and the timing of the write, read and lock actions of the threads over the shared variable $\boldsymbol{v}$ in 11 and over a shared pool of resources in 12. P1 and P2 represent two recovery points and F1-F6 represents possible positions of failures during the thread

execution. In both figures, recovery point P1 is defined for the entire duration of thread execution and in Figure 11 recovery point P2 is defined only in the case of failures F4, F5 and F6. Assume that in the case of Figure 11 the recovery criterion for both P1 and P2 forbid dirty reads. Then in accordance with RRP and SRP, besides system-level recovery, the choice of recovery strategies for each of the failures in Figure 11 are as follows: (a) F1 and F3: Rollback to P1; (b) F2: Release lock and Rollback to P1; (c) F4: Rollback to P1 or P2; (d) F5 or F6: Neither P1 or P2 due to dirty read.

Assume that the only recovery criterion for P1 in Figure 12 is that T1 should not own any resources acquired after the recovery point. Then besides system-level recovery, the choice of recovery strategies for each of the failures in Figure 12 are as follows: (a) F1 and F5: Rollback to P1; (b) F2: Release lock, free resource and Rollback to R1; (c) F3 and F4: Free resource and Rollback to R1.

3.4 Log(Lock) Execution Model

In this section, we present a concrete execution model of Log(Lock), that utilizes the state space analysis presented in the previous section. We show how to decide recovery strategies and how restoration levels can be tracked practically. Although the discussion in this research focuses on a thread-level recovery granularity, the Log(Lock) architecture can easily be extended to a more coarse granularity of micro-recovery such as at a task or component level.

In a complex legacy system such as a storage controller, not all failures can be handled efficiently through fine-grained recovery - either because the failure and recovery code may be too complex, or system-level recovery may be a more effective recovery technique, or simply because there may be insufficient development and testing resources. Therefore, our approach first involves identifying candidates for fine-grained recovery based on the analysis of failure logs and the software itself. The executing

instance of each candidate is known as a **recoverable thread**. Recall that, for each recoverable thread multiple recovery points and associated recovery criterion may be defined. In the event of a failure, control is transferred to the recovery handler (Section 3.2.3).

3.4.1 Tracking State Changes

Log(Lock) is based on the intuition that all shared state and resources are protected by locks or similar synchronization primitives. Tracking lock/unlock calls can therefore guide the understanding of system state changes and provide the information required to identify the restoration level at the instant of failure. At the same time, by tracking these calls on resources and applying the resource recovery protocol, we can prevent deadlocks or resource starvation issues. In order to compute restoration levels and perform system state restoration, Log(Lock) maintains the following:

Undo Logs: Undo logs are local logs maintained by each recoverable thread primarily for the following purposes: (1) Track the sequence of state changes within a single thread; (2) Track the creation of recovery points and (3)Track resource ownership. In general, the Undo logs can be used to encode any information required by a thread's recovery handler. In our implementation, information to be added to the Undo logs are explicitly specified by the developer. Other possible implementations are using regular checkpoints or copy on write techniques that can maintain Undo logs transparently.

Change Track Logs: In order to track conflicts between concurrent threads, Log(Lock) maintains Change Track Logs for each lock. The Change Track Log is used to: (1) Track concurrent changes to shared structures and (2) Track commit actions.

Both the Undo Log and Change Track Logs are maintained only in main memory and are verified for integrity using checksums. In our implementation, the change

track log is implemented as a hashtable indexed using the pointer to the lock as key. Unlike database logs or checkpoints for state restoration, these logs do not need to be flushed to stable storage. If a failure crashes the system causing it to lose or corrupt the logs, then we must perform a system-level restart to restore the system to a consistent, functional state and no longer require the software's state restoration logs from before the failure.

Log(Lock) provides four basic primitives to a recoverable thread:

- *startTracking(lock)*: Start tracking changes to the structure protected by *lock*.
- *stopTracking(lock)*: Stop tracking changes to the structure protected by *lock*.
- *getRestorationLevel(lock)*: Compute the restoration level for the structure protected by *lock*.
- *getResourceOwnership(lock)*: Get ownership information (including lock ownership) for the structure protected by *lock*.

All the above primitives are explicitly inserted into the code by the developer. The *startTracking* call is used to trigger change tracking for shared state and resources protected by the *lock* parameter. These accesses are identified by trapping lock/unlock calls. When the recoverable thread determines that the logs for a particular structure are no longer required, it explicitly issues a *stopTracking* call. In the event of a failure, the system transfers control to the designated recovery handler. The recovery handler can utilize the *getRestorationLevel* and *getResourceOwnership* primitives to determine the current restoration level and resource ownership and then invoke recovery procedures appropriately. The restoration level is determined by examining the undo and change track logs.

3.4.2 Recovery Using Restoration Protocols

The goal of our state restoration approach is to return the system to a correct, functional and known state by performing localized recovery and state restoration actions.

```
/* Recovery Criterion for R1: No residual resources */
        Owner = getResourceOwnership(&numActiveUsersLock);
/* Acquire ownership in write mode for consistent recovery*/
        if( Owner == ReadMode) {
                unlockRead(&numActiveUsersLock);
                lockWrite(&numActiveUsersLock);
        } else if(!Owner)
                lockWrite(&numActiveUsersLock);
        level = getRestorationLevel(&numActiveUsersLock);

        if ( level indicates dirty reads or lost updates ) {
                /* Indicates write completed */
                numActiveUsers -- ;
        } else {
    /* No other operations or write may not have completed */
                Replace old value using the Undo log;
        }
        unlockWrite( &numActiveUsersLock);
/* State restore complete. Jump to new execution point */
Jump to R1;
```

Figure 13: State Restoration Using Log(Lock)

The recovery actions are targeted at only a small subset of the threads in the system and a small region of the total system state that has been identified as affected by failure-recovery. Figure 13 shows pseudo code for state restoration using the restoration protocols and the Log(Lock) architecture for the scenario shown in Figure 7. Assume that, the recovery criterion associated with recovery point R1 specifies that resources (*numActiveUsersLock*) acquired after the recovery point should be released and does not care about occurrences of DR, LU or UR events. As shown in the Figure 13, the *getResourceOwnership* primitive is used to determine ownership of the *numActiveUsersLock* resource. Then, if the restoration level indicates that a DR or LU event has occurred, that would imply that the thread has successfully completed

incrementing *numActiveUsers* in the first place. Then in order to rollback the failed thread execution correctly to recovery point R1 without losing the work done by other threads, a matching decrement operation would need to be performed. If however the change track logs indicate that no other thread has consumed data written by the failed thread, it could imply that the failed thread either did not complete its increment operation or was the last thread to update the value of *numActiveUsers*. In that case, the recoverable thread could use its undo log to undo its changes, if any. The developer of this recovery handler is expected to have used the Undo log interfaces to store the old value prior to modification. Once state restoration is complete, execution is transferred to recovery point R1.

Similarly, in the case of the example in Figure 8, assume that the recovery criterion only specifies the constraint on releasing the temporary resource acquired after the recovery point. Therefore, the *getResourceOwnership* primitive is used to obtain the current ownership status of the temporary resource. If the resource is held by the thread, in order to rollback to recovery point R3, the resource must be cleanly relinquished. The pseudo code for this example and the next is not shown due to lack of space.

In the case of the failure scenario shown in Figure 9, the recovery criterion for recovery point R4 would be that no resources acquired after the recovery point (such as lock *MetadataLocationLock*) should be held by the thread and that no DR or LU events should have occurred. If the restoration level indicates that no other thread has already consumed this value (i.e., no DR or LU events have occurred), then the changes of the failed thread can be undone safely by replacing with the values in the Undo log. However, if the value is likely to have been consumed by another thread (i.e. DR or LU occurred), then the restoration level does not meet the recovery criterion for R4. So, in accordance with SRP, the error cannot be handled using single-thread recovery. Depending upon the support for multi-thread recovery (provided the CD

event has not occurred) recovery may require rollbacks of multiple threads. If however, CD has occurred, then system-level recovery or error-compensation is performed.

3.4.3 Implementation Details

We now discuss some implementation issues involved in the purging algorithm, the decision on recovery success, and the lock granularity, which are critical components in the Log(Lock) execution model.

Purging Algorithm : Undo logs go out of scope i.e., can be purged when a recoverable thread completes execution. Similarly, change track logs for a lock are purged when the recoverable thread issues a *stopTracking* call. However, unlike undo logs, change track logs cannot be purged immediately since these centralized logs may be shared by multiple recoverable threads. In that case, the log entries corresponding to the purging thread are only marked for purging and are actually purged when the last recoverable thread using the log issues a *stopTracking* call on that lock.

Ensuring Recovery Success: Multi-thread recovery i.e., applying state restoration and recovery to more than one thread, can typically handle more failure scenarios compared to single-thread recovery. However, multi-thread recovery is complex to implement. Moreover, multi-thread recovery may result in a domino effect [116] (also referred to as cascading aborts) potentially resulting in unavailability of resources and unbounded recovery time[129].

A simpler and more effective technique would be to limit recovery to a single thread and ensure recovery success through other mechanisms such as dependency tracking and scheduling. Recovery conscious scheduling [129] describes an approach where dependencies between concurrent threads are identified and dependent threads serialized. This approach can help limit the number of concurrent dependent threads and increase single-thread recovery success.

Lock Granularity: Another aspect of the system that affects recovery success

is the granularity of locks - i.e., the coupling between a synchronization primitive such as a lock or mutex and the structure it protects. For example, consider a large shared state protected by a single lock that synchronizes access to disjoint portions of the structure. Such a design could adversely affect both the performance and recovery success of the system since it could artificially increase the number of dependencies between threads. Besides rewriting code, another possible workaround would be to explicitly specify the sub-structure protected by the lock in the *startTracking* call and log both value and access in the change track logs. Shared resource pools are often protected by such coarse-grained locks. However, the lock granularity may not be a problem in the case of resources since recovery handlers are mostly concerned only about ownership of resources.

3.5 Experiments

We have implemented the Log(Lock) architecture for system state restoration and micro-recovery on an industry standard, high-performance storage controller and applied Log(Lock) to a variety of state and resource locks. In this section we present our evaluation of Log(Lock) with respect to performance, failure recovery and scalability. We first describe our experimental setup and evaluation metrics. Then we present our experimentation methodology and results.

We identified state and resource instances that are changed or accessed rapidly through the observation periods, based on instrumenting the system (Table 2). We also identified representative failure scenarios by analyzing bug reports, failure logs and code. Using these scenarios as candidates for micro-recovery and state restoration, we evaluate Log(Lock) efficiency and effectiveness. In summary, our results show that:

- The Log(Lock) architecture imposes negligible overhead and sustains high performance ($< 10\%$ impact) under a variety of workloads, even while tracking rapidly

Table 2: State and Resource Access over a 75 minute run with varying workloads

Lock	Contention CPU Cycles	Contention Counter	Number of locks	% contention	Locks/IO
Fiber channel	2654991	578	137196747	4.21293E-06	10.33500111
IO state	219969	76	90122610	8.43296E-07	6.788916609
Resource pool	608103	100	63482290	1.57524E-06	4.782107098
Resource pool state	124965	52	30040757	1.73098E-06	2.262963691
Throttle timer	79848	11	113316	9.7E-05	0.00853607

changing state (nearly 15K times/second) for significant durations.

- We observe an extremely high rate of recovery success (>99%), i.e., percentage of time restoration levels meet recovery criterion. This high rate of recovery success makes it evident that micro-recovery with Log(Lock) can be a promising approach to system recovery from transient failures.
- The Log(Lock) approach exhibits significant improvement in availability, replacing a four second downtime without micro-recovery with only a 35% performance impact lasting six seconds with Log(Lock).

3.5.1 Experimental Setup

We implemented the Log(Lock)-based state restoration architecture in an enterprise-class high performance, highly concurrent embedded storage controller. The system consists of a 4-way processor complex (4 3.00 GHz Xeon 5160 processors with 12 GB memory running IBM MCP Linux) running the controller software over a simulated backend. The controller implements persistent memory (non-volatile storage) for write caching. Simulating the backend allows flexibility in terms of experimenting with different configurations such as read/write latencies and error injection. The back end configuration varied between 50-250 LUNS of 100GB each with read and write latencies of the disk set to 20 ms. The host functionality was performed from a different system (2 1.133 GHz Pentium III processor with 1 GB memory, RHLinux

9) connected to the storage complex through a high-bandwidth (2 GB) fiber channel interconnect.

Our workload was generated using a randomized synthetic workload generator which took as inputs the following parameters: read/write ratio, block size and queue depth (i.e. maximum number of outstanding requests from the host). The experiments presented in this work utilized three distinct read/write ratios: 100% writes, 50%-50% mix of reads and writes and 100% reads. Block size was set to 4 KB and queue depth varied between 16 and 256.

3.5.2 Metrics

Our experiments evaluate efficiency and effectiveness of the Log(Lock) architecture. Efficiency and effectiveness depend on the following parameters: (1) rate of access to shared state or resources and (2) duration of a recoverable thread. Increasing each of these parameters results in a corresponding increase in the log size and logging overhead. Likewise, increase in each of these parameters increases the probability of conflicts.

Efficiency refers to the impact of Log(Lock) on system performance. To measure performance, we utilize two metrics: *throughput* (IOs per second or IOps) and *latency* (seconds/IO).

Effectiveness refers to the ability of the state restoration architecture to reduce the recovery time and positively impact the availability of the system. It refers to the probability of recovery success with the Log(Lock) architecture and the impact on system recovery time.

Effectiveness is measured using the following metrics: (1) *recovery success*, i.e. the percentage of time the restoration level meets the recovery criterion for single thread recovery, and (2) *recovery time*, i.e. the time required to restore the system to a consistent state after encountering a failure. While our Log(Lock) approach can

also be applied to multi-thread recovery, as described in Section 3.4.3, multi-thread recovery can be costly in terms of coding effort, resource consumption and recovery time. Instead, we assume that a technique such as recovery conscious scheduling [129] can help reduce the need for multi-thread recovery and improve the success of single thread recovery.

3.5.3 Methodology

In order to study the efficiency and effectiveness of Log(Lock), we first identify state and resource instances in the software for tracking. We instrumented the system to identify top locks in terms of access and contention. Table 2 shows the top five locks in the system in terms of number of accesses and contention. The table shows the semantics of the lock (i.e. the state or resource protected), the number of CPU cycles lost to contention, number of occurrences of contention ($>$ 2000 CPU cycles), number of accesses to the lock and the average number of lock acquisitions per IO. Frequently acquired locks are indicative of state that is accessed or modified often. For example, Table 2 shows that the fiber channel lock is accessed nearly 10 times per IO, indicating that this is a good candidate for evaluating the efficiency Log(Lock). Contention, while indicative of longer durations of holding locks, also shows a higher probability of accesses by concurrent threads. As Table 2 shows, the percentage of accesses resulting in lock contention is low as a result of the highly concurrent design of the controller. Thus, for short durations of tracking we expect high recovery success.

To evaluate effectiveness, we first measure the recovery success for the candidates identified from Table 2. We measure recovery success across locks with different rates of access and varying duration of tracking. To evaluate the impact on recovery time, we identify candidates for state restoration based on analysis of failure logs, defects and the software itself.

We present evaluation of the efficiency of our Log(Lock) architecture as compared

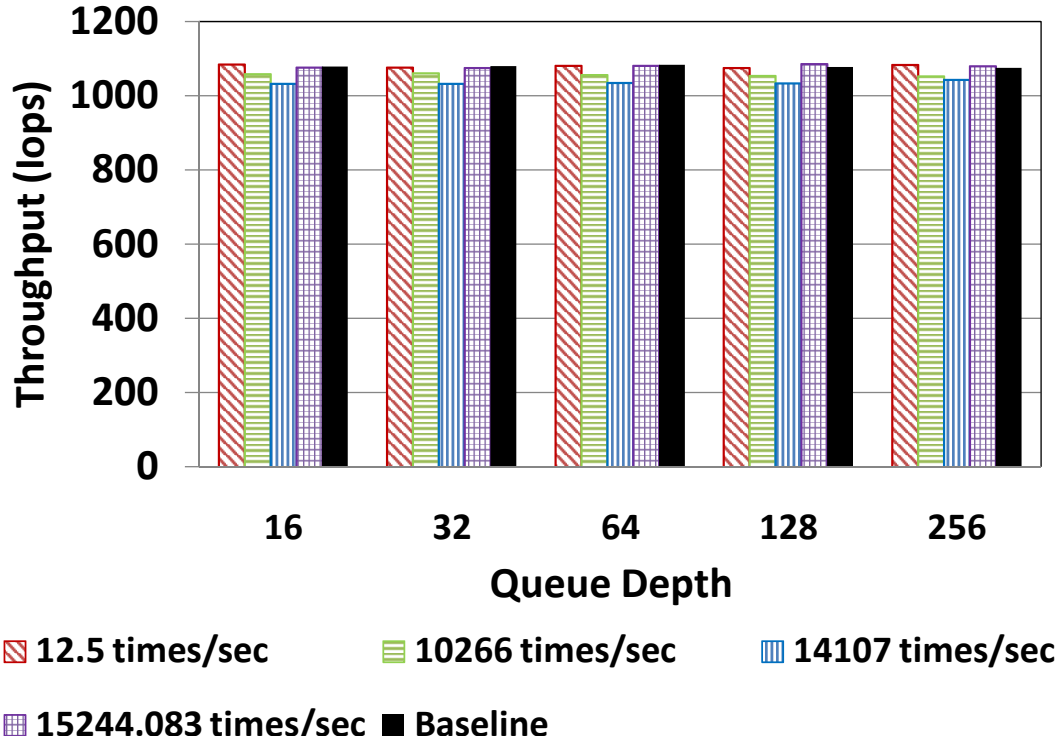


Figure 14: Rate vs Throughput (100% Writes)

to the original system, henceforth referred to as *baseline*. The baseline implementation does not perform state restoration or fine-grained recovery. Instead, it uses a highly efficient system level recovery mechanism (SLR) that checks all persistent system structures such as non-volatile data in the write cache for consistency, reinitializes software state and redrives lost tasks. Note that no hardware reboot is involved.

An alternative approach to Log(Lock) is to implement schemes such as strict 2-phase locking (2PL), commonly used in transactional systems. Essentially, these protocols require locks to be held for the entire duration of a recoverable thread. However, due to the high degree of concurrency in the system and the implementation of lock timeouts, such a scheme when implemented in our storage controller software caused lock timeouts and failed to bring up the system. Therefore, throughout this evaluation section, we primarily use the baseline system for comparison.

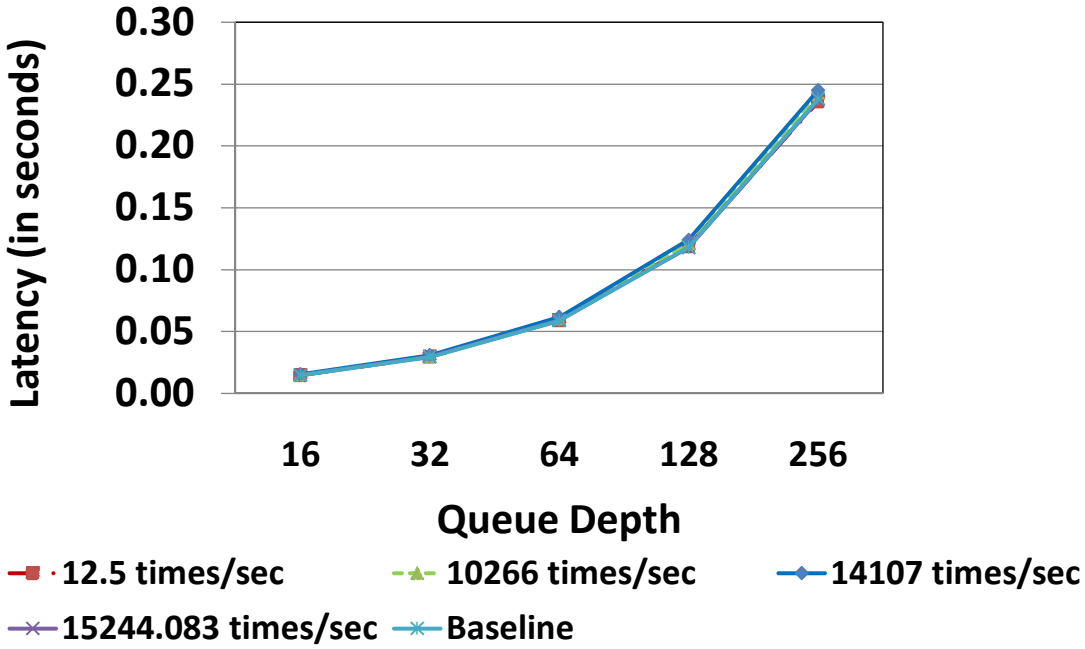


Figure 15: Rate vs Latency (100% Writes)

3.5.4 Efficiency of Log(Lock)

In order to measure efficiency, we compare the performance of the Log(Lock) architecture with the baseline system during failure-free operation.

3.5.4.1 Effect of Frequency of State Change

As described in Section 3.5.2, as the rate of accesses to a state variable or resource being tracked increases, the logging overhead increases. The workloads used for this experiment consisted of 100% write IOs and the data is averaged over a 10 minute run. The queue depth is represented on the x-axis. For this experiment we chose four locks from Table 2, representative of a range of access rates, ranging from 12.5 times/second to 15244 times/second. The duration of tracking was 2600 CPU cycles on average (and standard deviation 265 CPU cycles).

Figure 14 shows the throughput with varying access rates under different queue depths. The numbers show that even for high access rates, the Log(Lock) approach

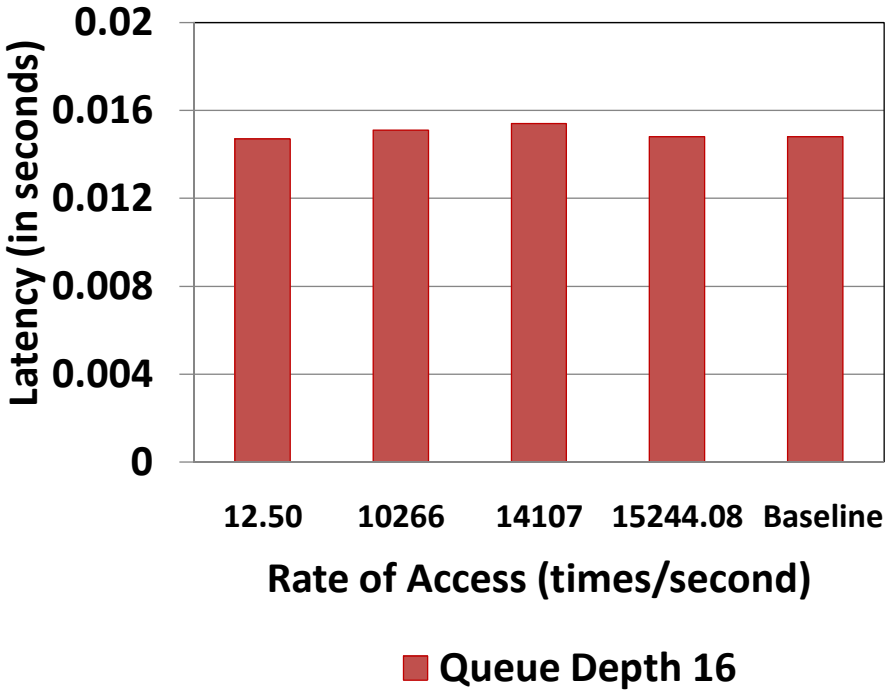


Figure 16: Latency

has negligible impact on performance. The lock with access rate 14107 times/sec (the resource pool lock) was tracked for 2429 CPU cycles and results in a 4.5% drop in throughput. We attribute this to the possibility of nested lock conditions in that particular code path, causing the system to be sensitive to even the small delay introduced by Log(Lock).

Figure 15 shows the variation of latency with queue depth for different access rates. The curves for the various access rates almost completely overlap showing that across configurations, the impact of Log(Lock) on latency, even for high access rates, is negligible. The observation that the latency increases with queue depth is a trend commonly observed in systems and is independent of Log(Lock). Figure 16 zooms into the points for queue depth 16 to give the reader a closer look at the data. As in the case of throughput, latency increases by 4% for the resource pool lock and is attributed to the occurrence of nested lock situations in the code path. The important message from Figures 14 and 15 is that Log(Lock) tracking can sustain

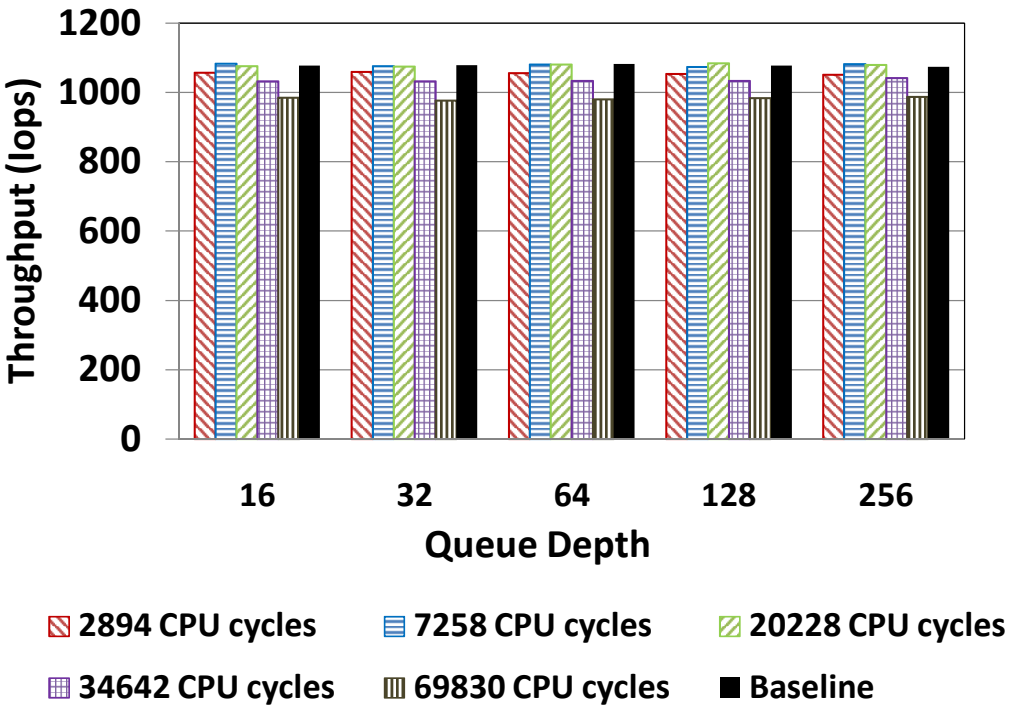


Figure 17: Duration of Tracking vs Throughput (100% Writes)

high performance even while tracking rapidly modified/accessed state or resources.

3.5.4.2 Effect of Duration of Tracking

Figures 17 and Table 3 show the variation of system performance with different durations of tracking. The durations were measured in terms of number of CPU cycles between the *startTracking* and *stopTracking* calls, averaged over a 10 minute run. The independent parameter queue depth is shown on the x-axis. The figures represents the performance for candidate locks from Table 2 that were tracked for different durations ranging from 2894 CPU cycles to 69830 CPU cycles (IO state for 2894 and 69830 CPU cycles, timer, fiber channel and resource pool for 7258, 20228 and 34642 CPU cycles respectively). The numbers were chosen to be representative of a range of tracking durations. Since no functional code was modified, rather than varying the duration of a single lock, different locks were instrumented to obtain this range. The rate of access of each lock varied as shown in Table 5.

Table 3: % Duration of Tracking vs Latency (100% Writes)

Queue Depth	(Duration of tracking in CPU Cycles)				
	2894	7258	20228	34642	69830
	% Increase in latency over baseline				
16	2.03%	0.68%	0.00%	4.05%	9.46%
32	1.69%	0.34%	0.34%	4.39%	10.47%
64	2.72%	0.34%	0.51%	4.76%	10.71%
128	2.54%	0.85%	0.00%	5.08%	9.32%
256	2.10%	0.00%	0.42%	2.94%	8.82%

From Figures 17 and Table 3 we observe that, the performance of the system with Log(Lock) is comparable to the baseline system across various queue depths. For the IO state lock (a lock in the IO path), when the duration of tracking was increased from 2894 CPU cycles to 69830 CPU cycles, the throughput dropped by 8.85% and response time increased by 9.75%. This drop in performance can be attributed to two factors: (1) occurrence of more conflicts with increase in duration of tracking and (2) increased possibility of encountering nested lock conditions, which are sensitive to the delay introduced by tracking. In the case of the resource lock, a tracking duration to 34642 CPU cycles resulted in a drop of only 4%, which is nearly identical to the performance with a tracking duration of only 2429 CPU cycles, as shown in the experiment in Section 3.5.4.1. We conclude that, though the overhead of tracking is a function of both the frequency and duration of tracking, it is more significantly impacted by the semantics of the lock being tracked and the efficiency of the code path involving the lock.

3.5.4.3 Performance with Other Workloads

Table 4 shows the throughput and latency with four other workloads. The figures compare the performance of a system powered by Log(Lock) and the baseline system under varying queue depths for the following workloads: Workload-1 (100% read, disk latency 20ms), Workload-2 (100% read, disk latency 1ms), Workload-3

Table 4: % Overhead (other workloads)

Queue Depth	Workload 1		Workload 2	
	Throughput	Latency	Throughput	Latency
16	0.43%	0.47%	0.08%	∼0.00%
32	0.25%	∼0.00%	0.78%	0.75%
64	0.24%	0.39%	0.13%	∼0.00%
128	0.29%	0.39%	0.79%	0.75%
256	0.25%	0.00%	0.12%	0.19%

(50%Read, disk latency 20ms) and Workload-4 (50% read, disk latency 1ms). Data from tracking the fiber channel lock (15244 times/sec for 20228 CPU cycles each) is shown. Overall, the impact on performance was $< 0.5\%$ in all cases. These results reiterate the observation that the Log(Lock) architecture is lightweight and sustains high performance for a range of workloads.

Examining the object code for our implementation showed that in the event of a lock being tracked, fewer than 200 assembly instructions were added to the code path. Assuming one instruction executes per CPU cycle, even at a frequency of 15244 times/second, on a 3.00 GHz processor, this amounts to a time overhead of less than 1% (assuming that the size of the state being saved to undo logs is small). Also, note that the code for a storage controller by itself is aggressively optimized to sustain high throughput, minimize the duration of locks in the I/O path and avoid nesting of locks to a large extent. Unlike checkpoints, which require a large amount of state to be copied to stable storage, our techniques copy small amounts of relevant state and information in memory only. The combination of all these factors results in the Log(Lock) system being able to sustain high performance despite an extremely high frequency of access to shared state and resources. In conclusion, we believe that the scenarios where performance will be impacted by tracking are when there are multiple levels of nesting with frequently accessed locks, increasing sensitivity to delay introduced by tracking. However, we expect that these situations are uncommon

Table 5: Recovery Success with the 100% Write Workload

Lock	Recovery Criterion	Tracking Calls (times/sec)	#Access (times/sec)	Duration CPU cycles	Recovery Success
Fiber channel	No Residual Resources	3666	15244	20228	100%
IO state	No DR, LU or UR	2500	10266	2894	99.88%
Resource pool	No Residual Resources	10	14107	34642	100%
Resource state	No Residual Resources	5	6675	4806	100%
Throttle timer	No Residual Resources	10	12.59	7258	100%
IO state	No DR, LU or UR	2444	10045	69830	99.38%

in well-designed concurrent systems.

3.5.5 Effectiveness of Log(Lock)

The next set of experiments are focused on evaluating the effectiveness of a micro-recovery framework with Log(Lock) in improving system recovery.

3.5.5.1 Recovery Success

The first metric of effectiveness is recovery success i.e., the percentage of time the restoration level meets the recovery criterion at the end of execution of a recoverable thread. This metric demonstrates the opportunity for micro-recovery in the system and evaluates if the system can effectively utilize Log(Lock)-based state restoration. Table 5 shows the recovery success for locks of varying semantics, rates of access and duration of tracking. For each lock, the recovery criterion, the number of tracking threads per second, the rate of access, duration of tracking and recovery success are shown. The restoration level in each case was obtained by calling the *getRestorationLevel* method before *stopTracking*, and recovery success was computed as the percentage of time the restoration level met the recovery criterion. As Table 5 shows, our storage controller exhibits a high rate of recovery success for a range of locks, even with high rates of access. We conclude that, for failures involving the restoration of these instances of state and resources, fine-grained recovery presents an effective recovery strategy.

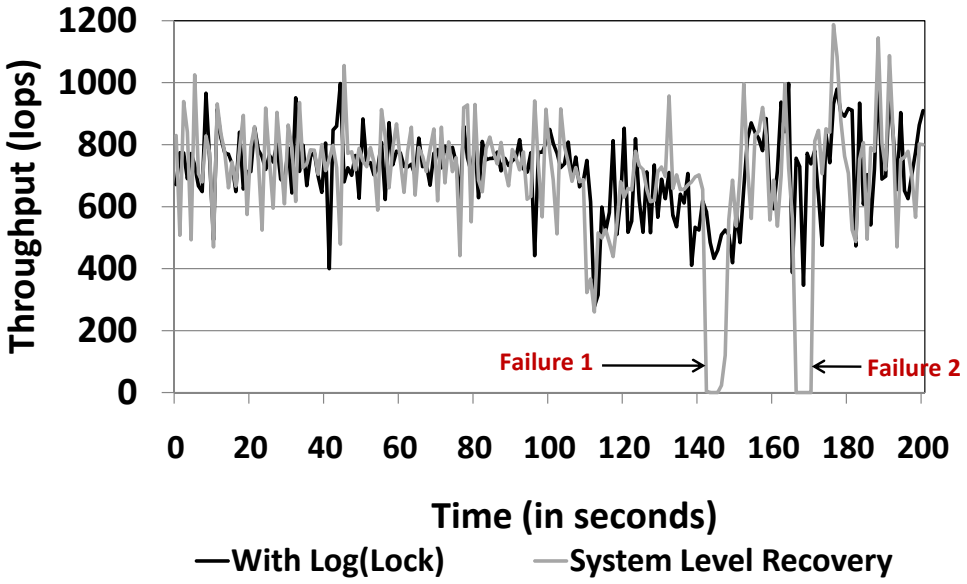


Figure 18: Throughput with Error Injection

3.5.5.2 Recovery Time

To illustrate the impact of Log(Lock)-based micro-recovery on the overall recovery time and availability of the controller software, we injected transient failures that disappeared on retry. The failures required restoration of the IO state to its previous value and a retry of the function. For the Log(Lock) system, the recovery criterion for IO state was set as shown in Table 5. Once the failure was injected, the thread verified if the restoration level at the time of recovery met the recovery criterion, before attempting state restoration and retry. The tracking duration was equivalent to the set up with 69830 CPU cycles.

Figures 18 and 19 show the variation of throughput and latency respectively over time. The points of failure injection are marked in the figures. The throughput and latency shown are for a workload with 100% write IOs, queue depth 64 and disk latency 20 ms. The Log(Lock) architecture is compared to system-level recovery (SLR) in the case of the baseline system. Recall that SLR is implemented entirely in software and involves restarting the controller process and verifying data structures and

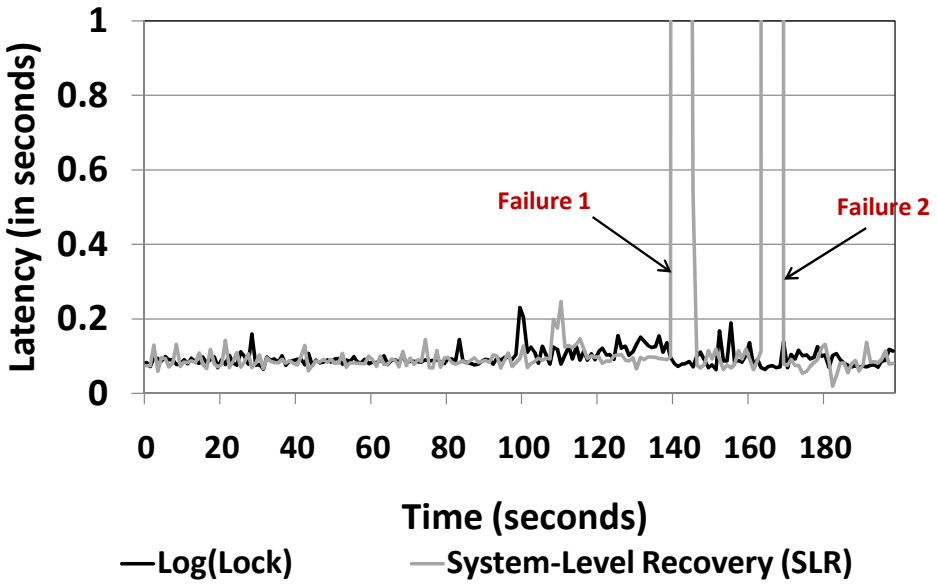


Figure 19: Latency with Error Injection

cache data for consistency before redriving IO transactions. Overall, during failure-free operation, the average throughput and latency respectively with Log(Lock) is $708 IOps$, 0.0946 sec/IO and $710 IOps$, 0.0912 sec/IO for the baseline system.

Log(Lock)-enabled micro-recovery imposes a 35% performance overhead lasting six seconds during recovery. However, system-level recovery results in 4 seconds downtime and it takes an additional 2 seconds to begin sustaining high performance. It is important to remember that as the size of the system and in-memory data structures increase, the recovery time for SLR is bound to increase. This, along with the opportunity for micro-recovery illustrated by the high recovery success shown in the previous experiment, further promote the case for micro-recovery in high performance systems like the storage controller.

3.6 Related Work

In this section we only briefly discuss related work that has not been discussed in previous sections. Our work is largely inspired by previous work in the area of transactional systems, software fault tolerance and storage system availability.

Hardware redundancy and software redundancy [72], rejuvenation [85] or fault isolation approaches such as isolating VMs from the failure of other VMs [115, 91] are complementary to our techniques and are already deployed in our setups. Since these approaches are targeted at handling failures at a different level they focus on a coarser granularity of recovery compared to our techniques.

Application-specific recovery mechanisms such as recovery blocks [116], and exception handling [137] are used in many software systems. However, to the best of our knowledge, fine-grained, localized recovery, in the presence of multiple interacting tasks executing concurrently, has not been well studied in the past both in terms of identifying dynamic dependencies and its impact on performance and availability. Constructs such as try/throw/catch [147] can be used to transfer control to an exception handler and a similar exception model is used by our implementation. However such exception handling constructs alone are insufficient for performing micro-recovery which requires richer failure context information. The goal of the Log(Lock) architecture is to provide this context information and provide the developer with a set of guidelines to decide the precise way in which the system should be restored given the failure context.

Logging of access patterns has been used for deterministic replay [121, 122, 143] and bug detection [96]. However, in micro-recovery, there is no requirement to perform deterministic replay. Also, the purpose of logging access patterns in Log(Lock) is to identify recovery dependencies between concurrent threads.

3.7 Summary

We have presented Log(Lock), a practical and flexible architecture for tracking dynamic dependencies and performing state restoration without rearchitecting legacy code. By exploring system state space, we formally model thread dependencies

based on both state and shared resources, capturing failure contexts through different 'restoration levels'. We develop recovery strategies in the form of restoration protocols based on recovery points and restoration levels. A comprehensive experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient and effective in reducing system recovery time.

CHAPTER IV

RECOVERY-CONSCIOUS SCHEDULING

This chapter addresses issues at the second and third tier of our recovery conscious framework described in Chapter 2. Specifically we address the question of effectively mapping dependent tasks to system resources in order to achieve high recovery efficiency while sustaining good performance. We first introduce the concept of recovery conscious scheduling (RCS) in Section 4.1 and describe recovery conscious scheduling algorithms in Section 4.2. Next, Section 4.3 describes the consideration for mapping recovery scopes identified in tier 1 to recovery groups over which scheduling constraints are imposed. Through prototype and simulation experiments we find that (1) the performance of RCS is critically dependent on the values of recovery-related parameters and (2) RCS promises to enhance storage system availability while keeping the additional overhead and the resulting degradation in performance under control.

4.1 Recovery-Conscious Scheduling

The goal of recovery-conscious scheduling (RCS) is to ensure system availability even during localized recovery. By recovery-consciousness, we mean that the scheduler must assure availability of resources for normal operation even during a localized recovery process. One way to achieve this objective is to intelligently isolate the recovery process by bounding the amount of resources that will be consumed by the recovering tasks.

4.1.1 Performance-Oriented Scheduling

Figure 20 shows a performance-oriented scheduling algorithm that does not take recovery dependencies into consideration while scheduling tasks. The diagram shows

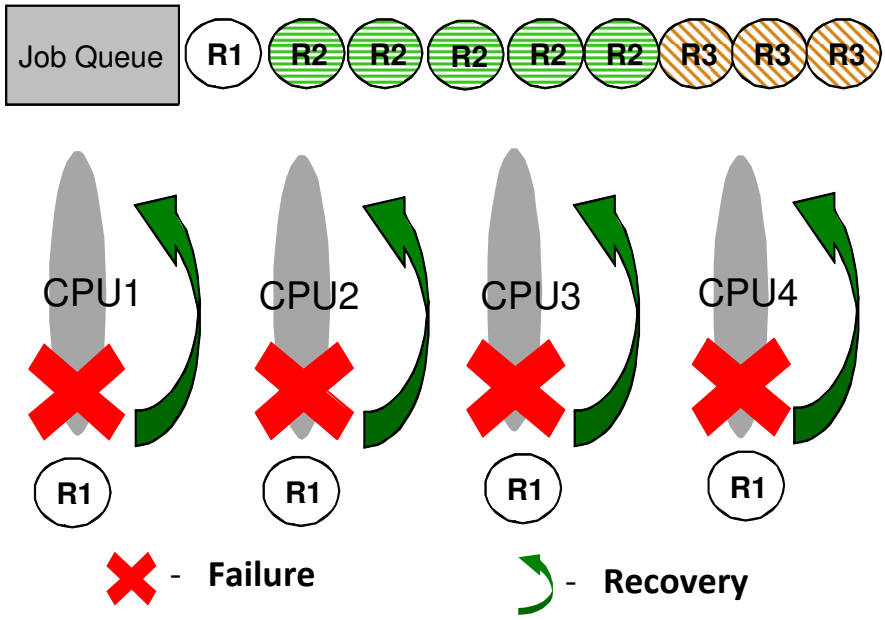


Figure 20: Current Scheduler

a 4-way SMP system where each processor independently schedules tasks from the same job queue. This scheduling algorithm aims at maximizing the throughput and minimizing the response time of user requests, which are internally translated by the system into numerous tasks of three types `R1, R2, R3`. The ovals represent tasks and the same shading scheme is used to denote tasks that are dependent in terms of recoverability. As shown in Figure 20, when all CPU resources are utilized for concurrently executing the tasks that have failure/recovery dependencies, then failure and subsequent recovery can consume all the resources of the system, stalling other tasks that could have proceeded with normal operation. Moreover, continuing to dispatch additional dependent tasks before the localized recovery process can be completed only further aggravates the problem of unavailability.

4.1.2 Recovery Groups and Resource Pools

In order to deal with the problem illustrated in Figure 20, we infuse "recovery consciousness" into the scheduler. Our recovery-conscious scheduler will enforce some

serialization of dependent tasks thereby controlling the extent of a localized recovery operation that may occur at any time. To formally describe recovery conscious scheduling, we first define two important concepts: **recovery groups** and **resource pools**.

Recovery Groups: A recovery group is defined as the unit of a localized recovery operation i.e., the set of tasks that will undergo recovery concurrently. When clean-up procedures are initiated for any task within a recovery group, all other tasks belonging to the same recovery group that are executing concurrently will also initiate appropriate clean-up procedures in order to maintain the system in a consistent state. Recovery groups are formed at the second tier of the framework based on recovery scopes identified in the first tier and additional system considerations such as the performance overhead of tracking and the number of cores in the system. We defer the discussion of mapping of recovery scopes to recovery groups to Section 4.3. By definition, every task belongs to a single recovery group. Thus tasks in the system can be partitioned into multiple disjoint recovery groups.

Resource Pools: The concept of resource pools is used as a method to partition the overall processing resources into smaller independent resource units, called **resource pools**. Although we restrict resource pools in our current work to processors, the concept can be extended to any pool of identical resources such as replicas of metadata or data. Recovery conscious scheduling maps resource pools to recovery groups, thereby confining a recovery operation to the resources available within the resource pool assigned to it.

4.1.3 Mapping of Resource Pools to Recovery-Groups

The recovery-conscious scheduling (RCS) algorithms implement the mapping between recovery groups and resource pools. There are different ways that one can map recovery groups to resource pools. The choice of decision depends on the type of

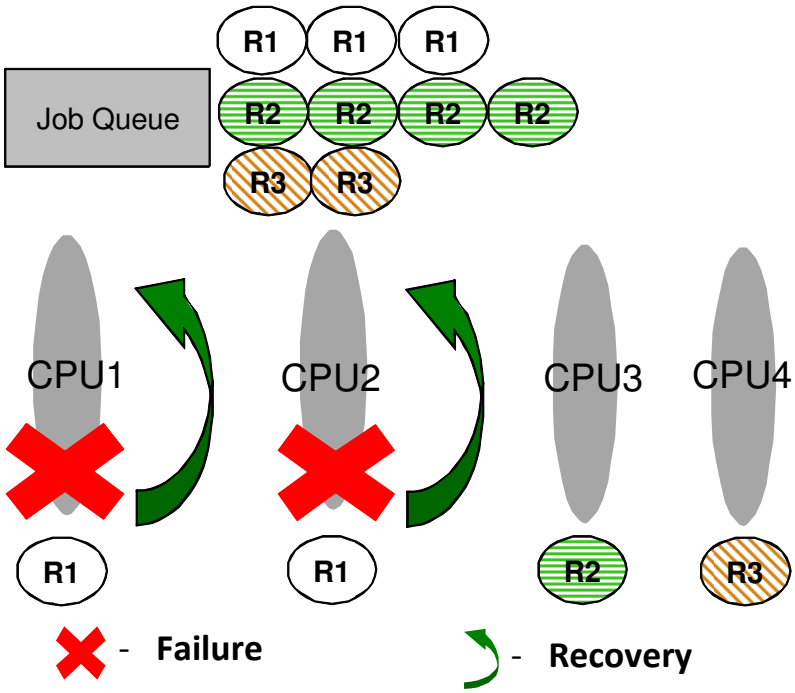


Figure 21: Recovery Oriented Scheduling

trade-offs one would like to make between recovery time and system availability and performance. Static scheduling of resource pools to recovery groups is one end of the spectrum and is only effective in situations where task level dependencies with respect to recoverability are well understood and the workloads of the system is stable. Dynamic scheduling of recovery groups to resource pools represents another end of the spectrum and may better adapt to the changing workload and more effectively utilize resources, but it is more costly in terms of scheduling management. Between the two ends of the spectrum are the partially dynamic scheduling algorithms.

Figure 21 depicts a recovery-conscious scheduler for the same set up as the one used for the performance-oriented scheduler, where tasks are organized into recovery groups − R1 (shaded as fill), R2 (horizontal lines) and R3 (downward diagonal). The processing resources (four CPUs in this example) are organized into three resource pools such that recovery group R1 is mapped to a pool consisting of two processors and recovery groups R2 and R3 are each mapped to a pool consisting of one processor.

In case of a failure within group R1, the recovering tasks are now restricted to two of the available four processors so that the other two processors remain available for normal operation. Additionally, the scheduler suspends further dispatching of tasks belonging to group R1 until the localized recovery process completes. This example highlights two aspects of a recovery-conscious scheduler: **proactive** and **reactive**.

Proactive RCS comes into play during normal operation and enhances availability by enforcing some degree of serialization of dependent tasks. The goal of proactive scheduling is to reduce the impact of a failure by trying to bound the number of outstanding tasks per recovery group. Then in the event of a failure within any recovery group, the number of tasks belonging to that recovery group that are currently executing and need to undergo recovery are also controlled. By limiting the extent of a recovery process, proactive scheduling can help the system recover sooner, and at the same time, it controls the amount of resources dedicated to the recovery process. Proactive RCS thereby ensures resource availability to normal operation even during a localized recovery process.

The reactive aspect of recovery conscious scheduling takes over *after a failure has occurred*. When localized recovery is in progress, reactive RCS suspends the dispatch of tasks belonging to the group undergoing recovery until the recovery completes. This ensures quick completion of recovery by preventing transitive expansion of the recovery scope and avoiding deadlocks.

4.1.4 System Considerations

The deployment of recovery conscious scheduling in practice requires the design and implementation of the scheduler to meet the stringent performance requirements of the storage system, sustaining the desired high throughput and low response time. Put differently, recovery-conscious scheduling should offer comparable efficiency in throughput and latency as those provided by performance oriented scheduling.

```
while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue)
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue)
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue)
end while
```

Figure 22: Qos-based scheduling

```
while true do
  repeat
    repeat
      ScanDispatch(HighPriorityQueue for ρ1)
    until HighPriorityLoopCount
    ScanDispatch(MediumPriorityQueue for ρ1)
  until MediumPriorityLoopCount
  ScanDispatch(LowPriorityQueue for ρ1)
end while
```

Figure 23: Recovery conscious scheduling

We outline below some factors that must be taken into consideration while comparing recovery conscious scheduling with performance oriented scheduling in a multi-core/SMP environment.

Note that our scheduling algorithms are concerned with partitioning resources between tasks belonging to different "components" of the same system which adds a second orthogonal level to the scheduling problem. We continue to respect the QoS or priority considerations specified by the designer at the level of user requests. For example, Figure 22 shows an existing QoS based scheduler using high, medium and low priority queues. Figure 23 shows how recovery-conscious scheduling used by a pool ρ_1 dispatches jobs based on both priority and recovery-consciousness (by picking jobs only from the recovery groups assigned to it).

We use *good-path* and *bad-path* performance as the two main metrics for comparison of the recovery-conscious schedulers with performance oriented schedulers. By 'good-path' performance we mean the performance of the system during normal operation. We use the term 'bad-path' performance to refer to the performance of the system under localized failure/recovery.

Both good path and bad path performance can be measured using end-to-end performance metrics such as throughput and response time. In addition, we can also measure the performance of a scheduler from system-level factors, including cpu utilization, number of tasks dispatched over time, queue lengths, the overall utilization of other resources such as memory, and the ability to meet service level agreements and QoS specifications.

4.2 Classification of RCS Algorithms

We classify recovery conscious scheduling (RCS) algorithms based on the method in which resource pools are distributed across recovery groups. As discussed in the previous section, we categorize recovery-conscious scheduling algorithms into three classes: static, partially dynamic, and fully dynamic. This classification represents varying degrees of trade-offs between fault isolation and performance, ranging from static mappings which emphasize recoverability over performance, to different ways of balancing between recoverability and performance, to a completely dynamic mapping of resources to recovery groups, which maximizes the utilization of resources while trying to meet recovery constraints.

In order to provide a better understanding of the design philosophy of our recovery-conscious scheduling, we devise a *running example scenario* that is used to illustrate the design of all three classes of RCS algorithms. This running example has five resource pools: $\rho_1, \rho_2, \rho_3, \rho_4$ and ρ_5 and four recovery groups: $\gamma_1, \gamma_2, \gamma_3$ and γ_4. We use σ_i to denote the *recoverability constraint* for the recovery group γ_i. Constraint

Recovery Groups	γ_1	γ_2	γ_3	γ_4
% of Workload	40%	20%	20%	20%
Recoverability constraints (σ_i)	2	1	1	1

Table 6: Recovery constraints

σ_i specifies the upper limit on the amount of resources (processors in this case) that can be dedicated to the recovery group γ_i ($1 \leq i \leq 4$ in our running example). Since we are concerned with processing resources in this work, it also indicates the number of tasks belonging to a recovery group that can be dispatched concurrently. The recoverability constraint σ_i is determined based on both the recovery group workload i.e., the number of tasks dispatched, and the observed task-level recovery time. Although recoverability constraints are specified from the availability standpoint, they must take performance requirements into consideration in order to be acceptable. Recoverability constraints are primarily used for proactive RCS.

For ease of exposition we assume that all resource pools are of equal size (1 processor each). Table 6 shows the workload distribution between the recovery groups and the recoverability constraint per group, where two processors are assigned to the recovery group γ_1 and one processor is assigned to each of the remaining three groups.

In contrast to the scenario in Table 6 where no resource pools are shared by two or more recovery groups, when more than one recovery group is mapped to a resource pool the scheduler must ensure that the dispatching scheme does not result in starvation. By avoiding starvation, it ensures that the functional interactions between the components are not disrupted. For example in our implementation we used a simple round-robin scheme for each scheduler to choose the next task from different recovery groups sharing the same resource pool. Other schemes such as those based on queue lengths or task arrival time are also appropriate as long as they avoid starvation.

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	ρ_1, ρ_2	ρ_3	ρ_4	ρ_5

Table 7: Static mapping

4.2.1 Static RCS

Static recovery conscious scheduling algorithms construct static mappings between recovery groups and resource pools. The initial mapping is provided to the system based on the observations of the workload and known recoverability constraints, such as previously observed localized recovery times. The mappings are static in the sense that they do not continuously adapt to changes in resource demands and workload distribution. Table 7 shows a mapping between the pools $\rho_1 \ldots \rho_5$ and the recovery groups $\gamma_1 \ldots \gamma_4$. This mapping assigns resource pools to recovery groups based on the workload distribution and the recoverability constraints given in Table 6. In this mapping recovery group γ_1 is mapped to two pools ρ_1 and ρ_2. Similarly groups γ_2, γ_3 and γ_4 are each assigned a single resource pool. Each processor dispatches work only from its assigned recovery group.

This approach aims at achieving strict recovery isolation. As a result, it loses out on utilization of resources, which in turn impacts both throughput and response time. Although this is a naive approach to performing recovery-conscious scheduling it helps us in understanding issues related to the performance and recoverability trade-off. Note that all our RCS algorithms avoid starvation by using a round-robin scheme to cycle between recovery groups sharing the same resource pool. In systems where the workload is well understood and sparse in terms of resource utilization, static mappings offer a simple means of achieving serialization of recovery dependent tasks.

Implementation Considerations: There are two main data structures that are common to all RCS algorithms: (1) the mapping tables and (2) the job queues. Mapping table implementations keep track of the list of recovery groups assigned to

Recovery Groups	γ_1	γ_2	γ_3	γ_4
Resource Pools	All	All	ρ_4	ρ_5

Table 8: Partial Dynamic RCS: Alternative mapping

each resource pool. They also keep track of groups that are currently undergoing recovery for the purpose of reactive scheduling. In our system we used a simple array-based implementation for mapping tables.

There are a couple of options for implementing job queues. Recall that recovery-consciousness is built on top of the QoS or priority based scheduling. We could use multiple QoS based job queues (for example, high, medium and low priority queues) for each pool or for each group. In our first prototype, we chose the latter option and implemented multiple QoS based job queues for each recovery group for a number of reasons. Firstly, this choice easily fits into the scenario where a single recovery group is assigned to multiple resource pools. Secondly, it offers greater flexibility to modify mappings at runtime. Finally, reactive scheduling (i.e., suspending dispatch of tasks belonging to a group undergoing localized recovery) can be implemented more elegantly as the resource scheduler can simply skip the job queues for the recovering group. Enqueue and dequeue operations on each queue are protected by a lock. An additional advantage of a mapping implemented using multiple independent queues is that it reduces the degree of contention for queue locks.

4.2.2 Partial dynamic RCS

The second class of algorithms are partially dynamic and allow the recovery-conscious scheduler to react (in a constrained fashion though) to sudden spikes or bursty workload of a recovery group.

The main drawback of static RCS is that it results in poor utilization of resources due to the strictly fixed mapping. Partial dynamic RCS attempts to alleviate this problem by using a relatively more flexible mapping of resources to recovery groups,

```
...
repeat
  workFound := false
  for γ_i in current mapping do
    workFound := ScanDispatch(HighQueue for γ_i)
    if workFound then
      break
    end if
  end for
  if !workFound then
    AcquireLease()
    for γ_j in alternative mapping do
      workFound := ScanDispatch(HighQueue for γ_j)
      if leaseExpired() OR workFound then
        break
      end if
    end for
  end if
until HighPriorityLoopCount
//Similarly for Medium and Low Priority tasks
```

Figure 24: Partial Dynamic RCS

allowing groups to utilize spare resources. Partially dynamic RCS algorithms begin with a static mapping. However, when the utilization is low, the system switches to an alternative mapping that redistributes resources across recovery groups.

For example, with the static mapping of Table 7 with changing distribution of workloads, resources allocated to recovery groups γ_3 and γ_4 may be under utilized while groups γ_1 and γ_2 may be swamped with work. Under these circumstances, the system switches to an alternative mapping shown in Table 8. Now groups γ_1 and γ_2 can utilize spare resources across the system even if this may mean potentially violating their recoverability constraints specified in Table 6. Note that γ_3 and γ_4 still obey their recoverability constraints. In summary, partially dynamic mappings allows the flexibility of selectively violating the recoverability constraints when there are spare resources to be utilized, whereas static mappings strictly obey recoverability constraints.

The aim of the partial dynamic mapping is to improve utilization over static schemes by opening up spare resources to recovery groups with heavy workloads. With the above example although there is a danger of a single recovery group (for e.g., γ_1) running concurrently across all resource pools, note that this is highly unlikely if other groups have any tasks enqueued for dispatching. There are multiple combinatorial possibilities in designing alternative mappings for partially-dynamic schemes. The choice of which components should continue to stay within their recoverability bounds is to be made by the system designer using prior information about individual component vulnerabilities to failures.

Implementation Considerations: There are two implementation considerations that are specific to the partially dynamic scheduling schemes: (1) the mechanism to switch between initial schedule and an alternative schedule, and (2) the mapping of recovery group tasks to the shared resource pools.

We use a lease expiry methodology to flexibly switch between alternative mappings. Note that the pool schedulers switch to the alternative mapping based on the resource utilization of the current pool. With the partially dynamic scheme, the alternative mappings are acquired under a lease, which upon expiry causes the scheduler to switch back to the original schedule. The lease-timer is set based on observed component workload trends (such as duration of a burst or spike) and the cost of switching. For example, since our implementation had a low cost of switching between mappings, we set the lease-timer to a single dispatch cycle. Figure 24 shows the pseudo-code for a partial dynamic scheduling scheme using a lease expiry methodology. For the sake of simplicity we do not show the tracking method (round-robin) used to avoid starvation in the scheduler.

Recall from the implementation considerations for the static mapping case that we chose to implement job queues on a per recovery group basis. This allowed for easy switching between the current and alternative mapping which only involves consulting

a different mapping table. Task enqueue operations are unaffected by the switching between mappings.

4.2.3 Dynamic RCS

Dynamic recovery-conscious scheduling algorithms assign recovery groups to resource pools at runtime. In dynamic RCS, tasks are still organized into recovery groups with recoverability bounds specified for each group. However, all resource pools are mapped to all recovery groups. The schedulers cycle through all groups giving preference to groups that are still within their recoverability bounds, i.e., occupying fewer resources than specified by the bound. If no such group is found, then tasks are dispatched while trying to minimize the resource consumption by any individual recovery group.

This class of algorithms aim at maximizing utilization of resources at the cost of selectively violating the recoverability constraints. Note that all recovery-conscious algorithms are still designed to perform reactive scheduling, i.e., suspend the dispatching of tasks whose group is currently undergoing localized recovery. The aspect that differentiates the various mapping schemes is the proactive handling of tasks to improve system availability. The dynamic scheme can be thought of as trying to use load balancing among recovery groups in order to achieve both recovery isolation and good resource utilization.

Implementation Considerations: A key implementation consideration specific to dynamic RCS is the problem of keeping track of the number of outstanding tasks belonging to each recovery group. We maintain this information in a per-processor data structure that keeps track of the current job.

Recall that implementing job queues on the per recovery-group basis helps us implement dynamic mappings efficiently and flexibly. One of the critical optimizations for dynamic RCS algorithms involves understanding and mitigating the scheduling

overhead imposed by the dynamic dequeuing process. In on going work we are conducting experiments with different setups to characterize this overhead. However our results show that even with the additional scheduling cost dynamic RCS schemes perform well both under good-path and bad-path conditions.

4.3 Mapping Recovery Scopes to Recovery Groups

The number of recovery groups in the system and the constraints on these recovery groups are critical factors in determining the system recovery time and thus the fault resiliency of the storage system (a system is resilient if it can continue to operate when a failure occurs and if it can recover from such failures quickly). A large number of recovery groups allows fine-grained dispatching of work and thus the opportunity of improved recovery performance through higher level use of multiprocessing in the multi-core processor. Depending on how tasks are assigned to recovery groups, the performance during normal operation may also be impacted. In general, increasing the number of recovery groups beyond a system-dependent threshold, may cause scheduling overhead that may outweigh the benefit of decreased lock contention. In order to effectively enforce recoverability constraints, we must map scopes appropriately to groups. The simple approach is to make each recovery scope a recovery group. Given the uneven distribution of tasks over recovery scopes, this may result in higher scheduling overhead as the scheduler polls the large number of recovery groups for work and most groups have no pending work. It may also offer little benefit in terms of shorter recovery time. In this section we develop mathematical models that capture the way in which various system parameter values affect recovery performance. Our analysis focuses on the degree of multiprocessing, scheduling discipline, failure and recovery rates, and workload characteristics.

4.3.1 Impact of Recovery Groups on System Resilience

The number of outstanding tasks belonging to a single recovery group and hence the degree of serialization has a direct bearing on the time-to-recovery of the system. For example, in the worst case where all tasks running at the time of failure belong to the same recovery group, massive system-wide recovery will have to be initiated. Intuitively, the recovery time increases with increasing size of the system and with decreasing number of recovery groups.

Based on the definition of recovery groups, we assume that when a task t belonging to the k^{th} recovery scope fails, all tasks belonging to the scope that are executing concurrently with the failed task t need to undergo recovery.

Let λ_k represent the failure rate and μ_k represent the repair rate for failures in the k^{th} recovery scope. The number of processors or cores in the system is represented by variable m and let $\alpha_k(i)$ represent that probability that i outstanding tasks belonging to the k^{th} recovery scope are executing concurrently at the time of failure.

We assume that the recovery process executes serially even for concurrently executing threads in order to restore the system to a consistent state. As a result, the time to complete system recovery is a product of the number of recovering processes and the individual task recovery time. Then the mean time to complete system recovery is given by:

$$\mu = \alpha_k(1) \times \frac{1}{\mu_k} + \alpha_k(2) \times \frac{2}{\mu_k} + \ldots + \alpha_k(m) \times \frac{m}{\mu_k}$$

Let γ_k represent the probability that a task belongs to recovery scope k. Then using the Poisson approximation for the binomial probability mass function, the probability that there are i outstanding tasks belonging to the k^{th} recovery scope is given by:

$$\alpha_k(i) = b(i; m, \gamma_k) = \frac{e^{-\gamma_k m}(\gamma_k m)^i}{i!}$$

With performance-oriented scheduling (POS), there is no notion of bounding the

recovery process. Interdependent tasks belonging to the same recovery scope can potentially be executing on all processors. As a result up to m dependent tasks may be executing concurrently at the time of failure. Under these circumstances the system mean-time-to-recovery (MTTR) for POS given that the failure occurred in the k^{th} recovery group denoted by $MTTR_{POS}|k$ is:

$$MTTR_{POS}|_k = \sum_{i=1}^{m} \frac{e^{-\gamma_k m}(\gamma_k m)^i}{i!} \times \frac{i}{\mu_k}$$

On the other hand, RCS enforces constraints on recovery groups there by ensuring some degree of serialization of dependent tasks. Let us assume that the constraint on the maximum number of concurrent tasks of the recovery group containing the k^{th} recovery scope is given by c_k. Then the system mean-time-to-recovery (MTTR) for RCS given that the failure occurred in the k^{th} recovery group denoted by $MTTR_{RCS}|k$ is:

$$MTTR_{RCS}|_k = \sum_{i=1}^{c_k} \frac{e^{-\gamma_k c_k}(\gamma_k c_k)^i}{i!} \times \frac{i}{\mu_k}$$

However, with dynamic RCS, a more flexible mapping of resources to recovery groups is employed in order to reduce resource idling and improve utilization. Under this scheme in the event that there are spare idle resources even after all tasks have been dispatched according to recoverability constraints, keeping in mind the high-performance requirements of the system, the constraints are selectively violated. Let the number of active recovery groups in the system be denoted by R. Let c_k be the constraint specified on the maximum number of concurrent tasks for the group containing the k^{th} recovery scope. Without loss of generality we assume that there are idle resources only when $\sum_{i=1}^{R} c_i < m$. For the sake of simplicity let us assume that the available spare resources $m - \sum_{i=1}^{R} c_i$ is allocated evenly amongst all groups. Then in the worst case violation of a constraint c_k, denoted as $\overline{c_k}$ is given by:

$$\overline{c_k} = c_k + \lceil \frac{(m - \sum_{i=1}^{R} c_i)}{R} \rceil$$

Thus, the system recovery time with dynamic RCS is obtained by replacing the constraint c_k by $\overline{c_k}$ in the expression for system recovery time for RCS ($MTTR_{RCS}|k$).

Clearly, the system availability under POS is affected by the failure rate λ_k, the repair rate μ_k for failures in the k^{th} recovery scope, the number m of processors or cores in the system, and the probability γ_k that a task belongs to recovery scope k. In contrast, with RCS, availability is also influenced by additional parameters such as the number R of active recovery groups in the system and the constraint c_k on the maximum number of concurrent tasks of the group containing the kth recovery scope.

4.3.2 Impact of RCS Queues on System Performance

In this section we present analysis that shows the impact of recovery groups on the system performance and based on these results we describe criteria for the selection of number of recovery groups for efficient scheduling. Each recovery group is mapped to a single scheduler queue and the serialization constraint imposed on the group applies to all scopes that are mapped to the group.

While evaluating system performance, we must take into consideration both the good-path (i.e. normal operation) and bad-path (during failure recovery) performance. Good path performance is primarily impacted by the efficiency of the scheduler. On the other hand, bad-path performance will be impacted by the extent of failure and recovery (i.e. the degree of serialization) and the availability of resources for normal operation during local recovery.

Variation of service rate with RCS queues : We model the variation of service rate with the number of queues as a hypoexponential distribution with 2 phases where the first phase describes the scenario where the service rate increases with the number of queues due to reduced lock contention. The second phase models the scenario where the increase in the number of queues causes the service rate to

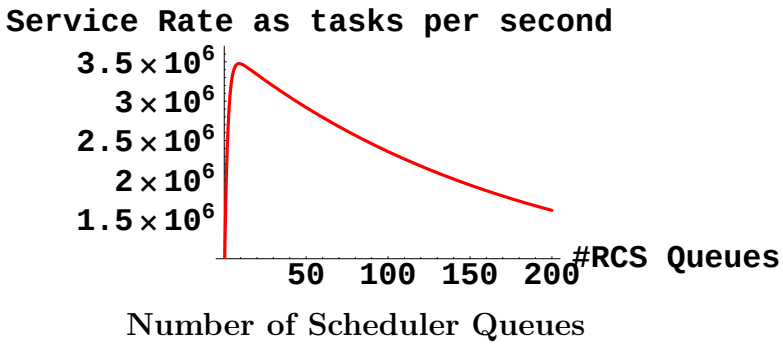


Figure 25: Variation of Service Rate

drop due to the additional scheduling overhead. Figure 25 shows an example of this model for variation of service rate with the number of queues.

In order to study the impact of recovery-consciousness on the performance of the system, we model both POS and RCS with varying system size and during good-path and bad-path operation. In order to model utilization, response time and throughput we adopt the models for M/M/m queuing systems [150].

Consider a system where tasks arrive as a Poisson process with rate λ_a and service times for all cores are independent, identically distributed random variables. Let the mean service rate as a function of the number of scheduler queues (groups in the case of RCS), for performance oriented scheduling be denoted by μ_{pos} and for recovery conscious scheduling be denoted by μ_{rcs}. We assume that the service times include the time required to dequeue tasks from the job queue(s) and iterate through queues (for RCS). Let m denote the total number of cores in the system.

Good-path Performance : During good-path operation, all system resources are available and storage controller performance is limited only be scheduler efficiency. Accordingly, the average number of jobs, N, in the system is given by:

$$E[N] = m\rho + \rho\frac{(m\rho)^m}{m!}\frac{p_0}{(1-\rho)^2}$$

where p_0, the steady state probability that there are no jobs in the system is given

by:

$$p_0 = \left[\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{(1-\rho)} \right]^{-1}$$

For POS, the value ρ, the traffic intensity, is given by, $\rho_{pos} = \frac{\lambda_a}{m\mu_{pos}}$ and that for RCS is given by $\rho_{rcs} = \frac{\lambda_a}{m\mu_{rcs}}$. $E_{POS}[N]$ and $E_{RCS}[N]$ are obtained by substituting ρ by ρ_{pos} and ρ_{rcs} respectively in the expressions for $E[N]$ and p_0. In each case, based on Little's formula [145] the average response time for performance -oriented scheduling ($E_{POS}[R]$) and RCS ($E_{RCS}[R]$) is given by:

$$E_{POS}[R] = \frac{E_{POS}[N]}{\lambda_a} \text{ and } E_{RCS}[R] = \frac{E_{RCS}[N]}{\lambda_a}$$

Assuming that our system utilizes a non-preemptive model where individual tasks complete execution within the service time allocated to them on system cores, the system throughput T can be modeled as follows:

$$E_{POS}[T] = \mu_{pos} U_0^{pos} \text{ and } E_{RCS}[T] = \mu_{rcs} U_0^{rcs}$$

where U_0 the utilization of the system is given by $U_0 = 1 - p_0$ and the values for utilization with POS (U_0^{pos}) and RCS (U_0^{rcs}) are obtained by substituting appropriate values for p_0.

Bad-path Performance : In order to model system performance during bad-path operation we assume that the amount of system resources consumed by the recovery process is proportional to the extent (i.e. the number of outstanding tasks undergoing recovery) of the recovery process.

As described in the Section 4.3.1, with POS, the extent of the recovery process is unbounded and can potentially span all the available cores in the system. As with the analysis of system availability, assume that a task t belonging to the k^{th} recovery scope encounters a failure causing in all executing tasks belonging to the k^{th} recovery group to under go recovery. Let f_k^{pos} and f_k^{rcs} denote the extent of the failure-recovery

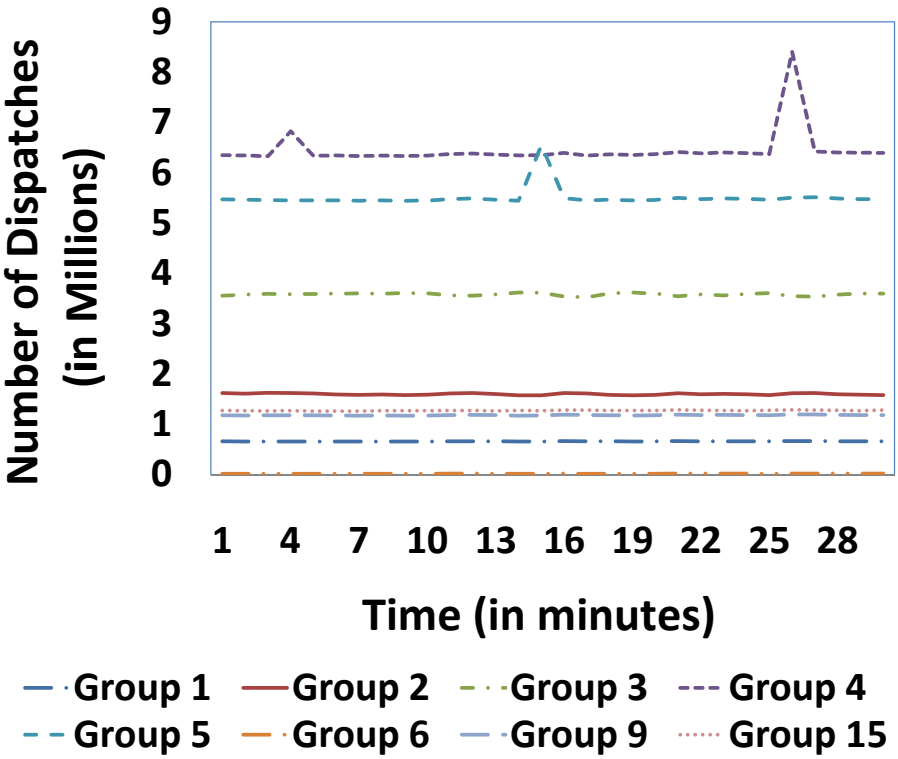


Figure 26: Cache-Standard

for POS and RCS respectively. Let, $\overline{m}_{pos}$ and $\overline{m}_{rcs}$ denote the expected number of cores available for normal operation during failure recovery. Then, as explained in Section 4.3.1

$$\overline{m}_{pos} = m - f_k^{pos} = m - \sum_{i=0}^{m} \frac{e^{-\gamma_k m}(\gamma_k m)^i}{i!} \times i$$

$$\overline{m}_{rcs} = m - f_k^{rcs} = m - \bar{c}_k$$

Then the expected response time and throughput during bad-path: $E'_{POS}[R]$, $E'_{POS}[T]$ and $E'_{RCS}[R]$, $E'_{RCS}[T]$ for POS and RCS respectively can be computed by substituting m in the original expressions with $\overline{m}_{pos}$ and $\overline{m}_{rcs}$ respectively.

4.4 Experiments

In this section we present results from experiments conducted using both simulations and a prototype. The experimental results illustrate the complex dynamics between the various factors that affect performance and recovery efficiency. Our results provide

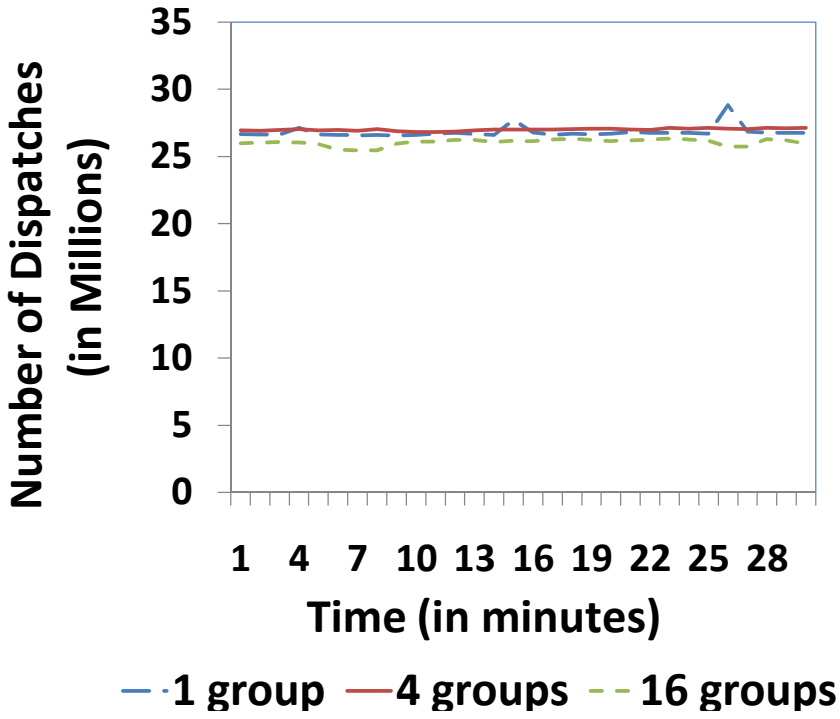


Figure 27: Efficiency vs Recovery groups

valuable insights into the implications and trade-offs associated with an implementation of fine-grained recovery and the effectiveness of our proposed framework. We have implemented our recovery-conscious scheduling algorithms on an industry-standard enterprise storage system with architecture similar to that discussed in Chapter 2. Our implementation involved no changes to the functional architecture. Our results show that dynamic RCS can match performance oriented scheduling under good path conditions while significantly improving performance under failure recovery.

4.4.1 Workload

We use the z/OS Cache-Standard workload [22, 92] to evaluate our algorithms. The z/OS Cache-standard workload is considered comparable to typical online transaction processing in a z/OS environment. The workload has a read/write ratio of 3, read hit ratio of 0.735, destage rate of 11.6% and a 4K average transfer size. The setup for

the cache-standard workload was CPU-bound. Figure 26 shows the number of tasks dispatched per-recovery group under the workload over 30 minutes. Group 4 has the highest task workload (∼6.5M tasks/min) followed by group 5 (∼ 5M/min). Eight of the groups which have nearly negligible workload are not visible in the graph. We use this workload to measure throughput and response times. While measuring cpu utilization we only count time actually spent in task execution and do not include time spent acquiring queue locks, dequeuing jobs or polling for work.

4.4.2 Methodology

We use our workload to measure throughput and response times in our prototype experiments and scheduler efficiency (as measured by the number of task dispatches per unit time) in the simulation experiments. For the prototype experiments we identified 16 component-based recovery scopes. Each recovery scope corresponded to a functional component such as a host adapter, device manager or cache manager.

We measure the effectiveness of RCS against traditional performance-oriented scheduling (POS). POS, either with a single, global queue or multiple load-balanced queues, does not include recovery-dependency in its criteria for resource allocation.

In order to understand the impact on system performance when localized recovery is underway, we inject faults into the workload. We choose a candidate task belonging to recovery scope 5 and introduce faults at a fixed rate. The time required for recovery is specified by the recovery rate. During localized recovery, all tasks belonging to the same recovery scope that are currently executing in the system and that are dispatched during the recovery process also experience a delay for the duration of the recovery time. For example, in our implementation, a recovery time of 20 ms and a failure rate of 1 in every 10K dispatches, for tasks belonging to component 5, introduces an overhead of 5% to aggregate execution time per minute of component 5 execution on average. The recoverability constraint for dynamic RCS was set to 1.

Note that in the case of dynamic RCS, the constraint would selectively be violated only if no task satisfying the constraint was found.

4.4.3 Prototype Experimental Setup

We prototyped our approach to fine-grained recovery by modifying the firmware of a commercial enterprise-class storage system (sensitive information is left out). The storage system consists of a storage controller with two 8-way server processor complexes, memory for I/O caching, persistent memory (NVS) for write caching, multiple fiber channel protocol (FCP), Fiber Connectivity (FICON*) or Enterprise System Connection (ESCON*) adapters connected by a redundant high bandwidth (2 Gbyte) interconnect, fiber channel disk drives, and management consoles. The system is designed to achieve both response time and throughput objectives. The embedded storage controller software is similar to the model presented in Chapter 2. The system has a number of interacting components which dispatch a large number of short running tasks. The system is designed to optimize both response time and throughput. The basic strategy employed to support continuous availability is the use of redundancy and highly reliable components.

The embedded storage controller software is similar to the model presented in Chapter 2. The software is also highly-reliable with provisions for quick recovery(under ∼6 seconds) at the system-level. The system has a number of interacting components which dispatch a large number of short running tasks. For the prototype experiments presented in this thesis we identify 16 recovery scopes based on component-based explicit recovery dependency specifications. However, some recovery groups may perform no work in certain workloads possibly due to features being turned off. We chose a pool size of 1 CPU which resulted in 8 pools of equal size. The system already implements high, medium and low priority job queues. Our recovery-conscious scheduling implementation therefore uses three priority based queues per

recovery group. For the partially dynamic case, based on the workload we have identified two candidates for strict isolation - groups 4 and 5. For the static mapping case each recovery group is mapped to resource pools proportional to its ratio of the total task workload.

Our framework was implemented in the storage-controller firmware micro-code. In our implementation, the recovery-conscious scheduler alone was implemented in approximately 1000 lines of code. Task-level recovery can be implemented incrementally for each failure situation that we intend to handle. Currently, our implementation specifies system-level recovery as the default action, except for cases for which task-level recovery has been implemented. A naive coding and the design effort for task level recovery would be directly proportional to the number of "panics" or failures in the code that are intended to be handled using our framework. In general, the coding effort for a single recovery action is small and is estimated to be around a few tens of lines of code (using semicolons as the definition of lines of code) per recovery action on average.

4.4.4 Prototype Experimental Results

We compare RCS and performance oriented scheduling algorithms using good-path (i.e. normal condition) and bad-path (under failure recovery) performance.

4.4.4.1 Effect of additional job queues

We first performed some benchmarking experiments to understand the effect of additional job queues on the efficiency of the scheduler. Using the cache-standard workload, we measured the aggregate number of dispatches per minute with varying number of recovery groups - 16, 4 and 1 (which is identical to performance-oriented scheduling) to measure scheduler efficiency with dynamic RCS. The four and one recovery group cases were implemented by collapsing multiple groups into a single larger group. Recall that each recovery group results in three job queues for high, medium

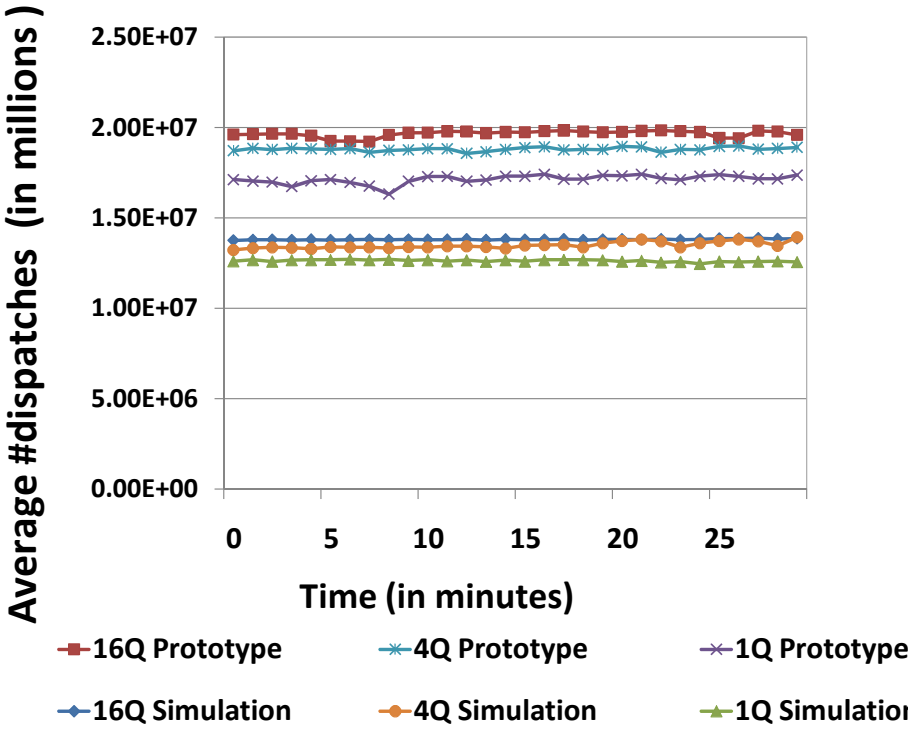


Figure 28: Impact of ♯Recovery Groups

and low priority jobs. Figure 27 shows the aggregate number of tasks dispatched per minute with 1, 4 and 16 recovery groups. As the figure shows the number of dispatches are almost identical in the three cases (+/- 2%). Although more job queues imply having to cycle through more queue locks while dispatching work, increasing the number of job queues reduces contention for queue locks both when enqueuing and dequeuing tasks. For most of the experiments in this chapter we choose a configuration with 16 recovery groups.

Figure 28 shows the average number of task dispatches per minute over 30 minutes with varying number of recovery groups under the Cache-standard workload. The figure also shows the scheduler performance for the same configuration using the simulation. As the figure shows, the number of dispatches initially increases (although modestly) with the increase in the number of groups. For instance, when the number of groups increase from 1 to 16, the number of dispatches increase by nearly 13% (9% in the simulation) and from 1 to 4, the number of dispatches increases by 10% (8%

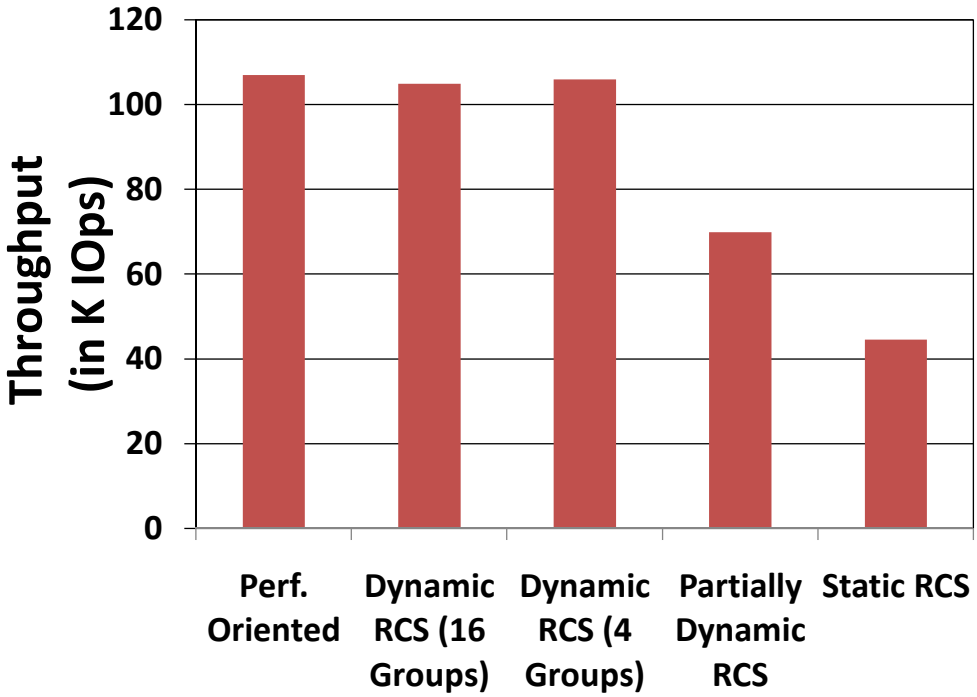


Figure 29: Good path throughput

in the simulation). This experiment was used to validate the simulator and establish the preferred number of recovery groups as 16 for further experimentation with the prototype.

4.4.4.2 Good-path Performance

Recovery-conscious scheduling can be an acceptable solution only if it is able to meet the stringent performance requirements of a high-end system. In this experiment we compare the good-path (i.e. under normal operation) performance of our RCS algorithms with the existing performance-oriented scheduler.

Figure 29 shows the good-path throughput for the performance-oriented and recovery-conscious scheduling algorithms. The average throughput for the dynamic RCS case with 16 groups (105 KIOps) and 4 groups (106 KIOps) was close to that for the performance-oriented scheduler (107 KIOps). On the other hand, with partially

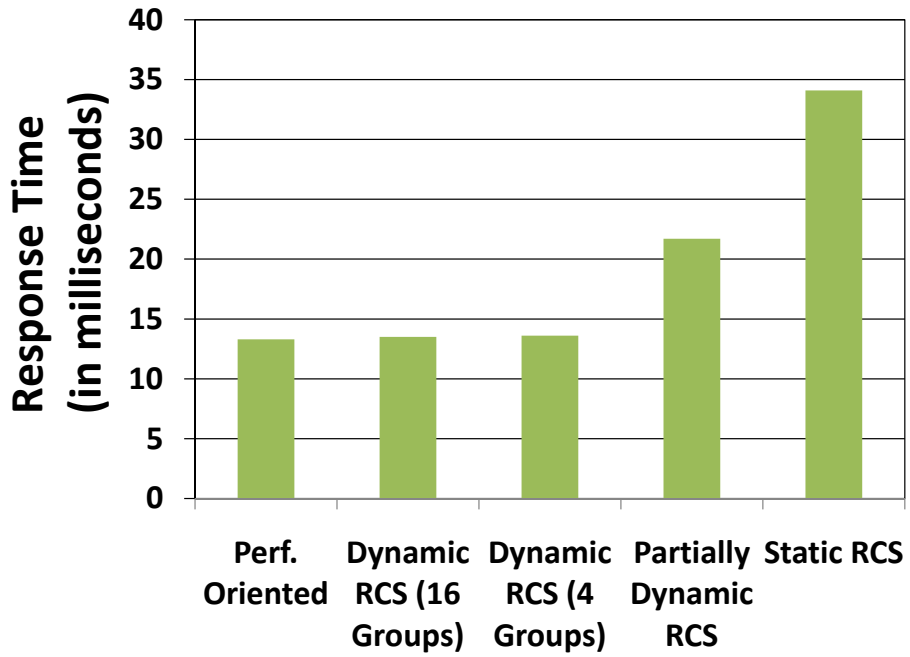


Figure 30: Good path latency

dynamic RCS, the system throughput drops by nearly 34% (∼ 69.9 KIOps), and with static RCS by nearly 58% (∼ 44.6 KIOps) compared to performance oriented scheduling.

Figure 30 compares the response time with different RCS schemes and performance-oriented scheduling. Again, the average response-time for the dynamic RCS case with 16 recovery groups (13.5 ms) and 4 recovery groups (13.6 ms) is close to the performance-oriented case (13.3 ms). However, with the partially-dynamic RCS scheduling, the response time increases by nearly 63% (21.7 ms) and by 156% with static RCS (34.1ms).

Both the throughput and response time numbers can be explained using the next chart, Figure 31. The radar plot shows the relationship between throughput, response time and cpu-utilization for each of the cases. As the figure shows, the cpu utilization has dropped by about 19% with partially dynamic RCS and by 63% with static

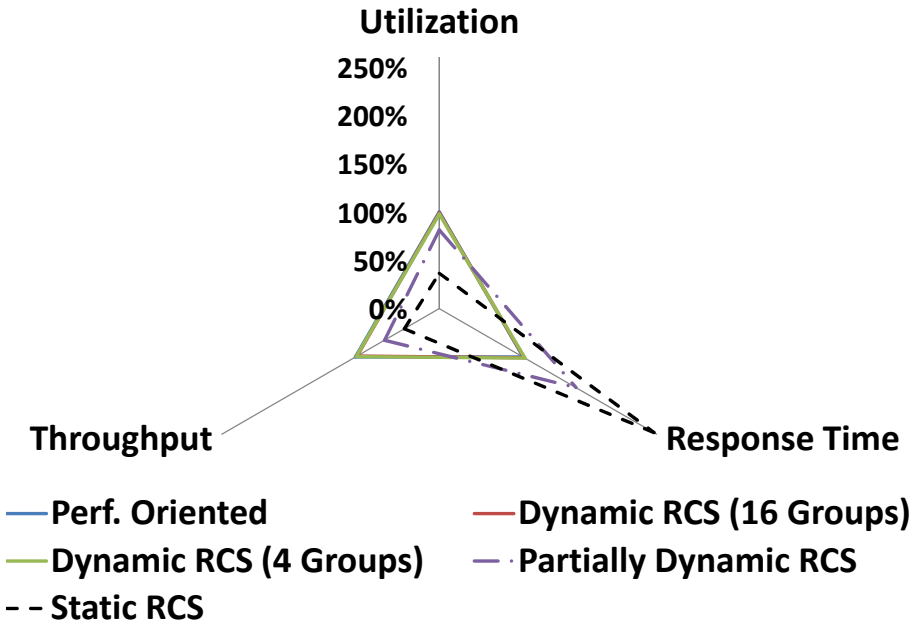


Figure 31: CPU utilization

RCS. The reduction in cpu utilization eventually translates to reduced throughput and increased response time in a cpu-bound workload intensive environment. These numbers seem to indicate that in such an environment schemes that reduce the utilization can result in significant degradation of the overall performance. However note that the normal operating range of many customers may be only around 6-7000 IOps [138]. If that be the case, then partially-dynamic schemes can more than meet the system requirement even while ensuring some recovery isolation.

4.4.4.3 Bad-path Performance

Next, we compare RCS algorithms with performance-oriented scheduling under bad-path or failure conditions. In order to understand the impact on system throughput and response time when localized recovery is underway, we inject faults into the cache standard workload. We choose a candidate task belonging to recovery group 5 and

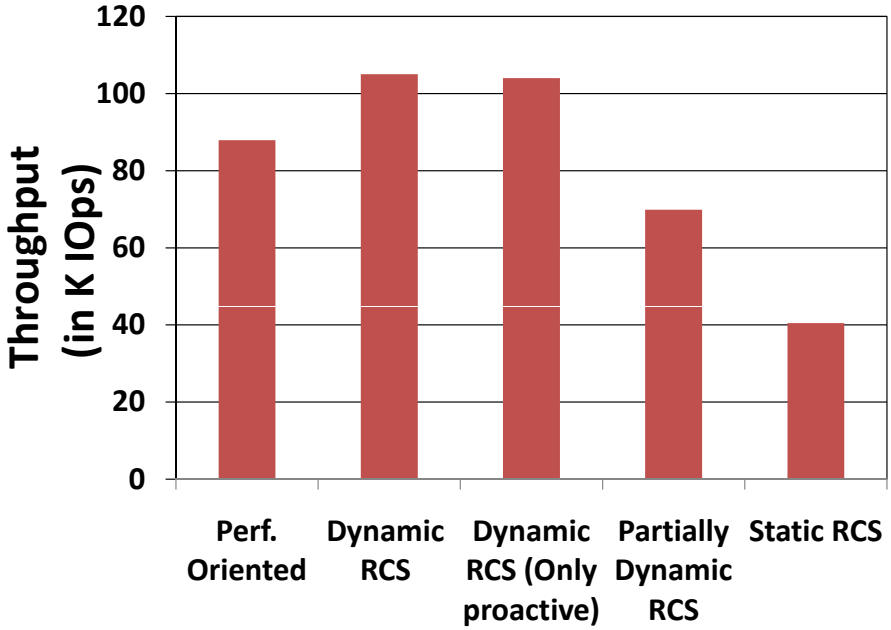


Figure 32: Bad path throughput

introduce faults at a fixed rate (1 for each 10000 dispatches). Recovery was emulated and recovery from each fault was set to take approximately 20 ms. On an average this introduces an overhead of 5% to aggregate execution time per minute of the task. During localized recovery, all tasks belonging to the same recovery group that are currently executing in the system and that are dispatched during the recovery process also experience a recovery time of 20 ms each. We measured performance (throughput and latency) averaged over a 30 minute run.

In the case of recovery-conscious scheduling algorithms, reactive scheduling kicks in when any group is undergoing recovery. Under those circumstances, tasks belonging to that recovery group already under execution are allowed to finish, but further dispatch from that group is suspended until recovery completes.

Figure 32 shows the average system throughput with fault injection. The average throughput using only performance oriented scheduling (87.8 KIOps) drops by nearly

16.3% when compared to dynamic RCS (105 KIOps) that also uses reactive policies. On the other hand dynamic RCS continues to deliver the same throughput as under normal conditions. Note that this is still not the worst case for performance oriented scheduling. In the worst case, all resources may be held up by the recovering tasks resulting in actual service outage and the problem would only worsen with increasing localized recovery time and system size.

The figure also compares proactive and reactive policies in dynamic RCS. The results show that with only proactive scheduling we are able to sustain a throughput (104 KIOps) which is just $\sim$ 1% less than that using both proactive and reactive policies (105 KIOps).

The graph also compares partially dynamic RCS (69.9 KIOps) and static RCS (40.4 KIOps). While these schemes are able to sustain almost the same throughput as they do under good path, overall, the performance of these schemes results in 20% and 54% drop in throughput respectively compared to performance oriented scheduling.

Figure 33 compares the latency under bad-path code with different scheduling schemes. Compared to dynamic RCS (13.5 ms), performance oriented scheduling (16.6 ms) results in a 22.9% increase in response time. At the same time, even without reactive scheduling, dynamic RCS (13.6 ms with only proactive) increases response time by only 0.7%. Again, partially dynamic RCS (21.7ms) and static RCS (37.1 ms) result in latency close to their good path performance but which is still too high when compared to dynamic RCS.

We performed experiments with other configurations of dynamic, partially dynamic and static schemes and using other workloads too. However due to space constraints we only present key findings from those experiments. In particular we used a disk-bound internal workload (and hence low cpu utilization of about $\sim$25%) to study the effect of our scheduling algorithms under a sparse workload. We used

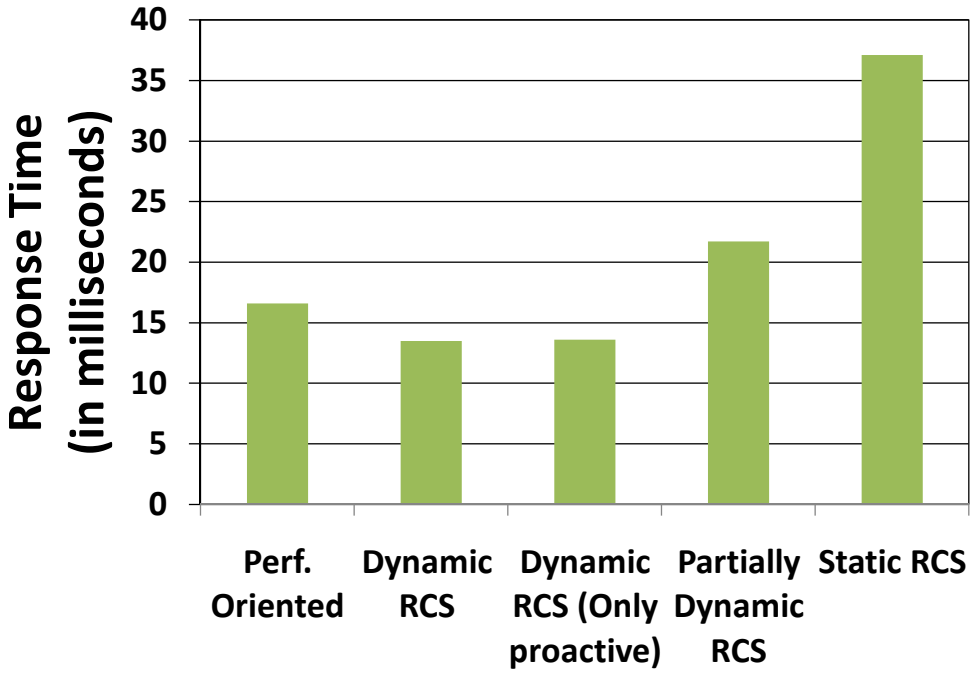


Figure 33: Bad path latency

the number of task dispatches as a metric of scheduler efficiency. The fault injection mechanism was similar to the cache-standard workload, however due to the workload being sparse, we introduced an overhead of only 0.3% to the aggregate execution time of the faulty recovery group. Our results showed that dynamic RCS was able to achieve as many dispatches as performance oriented scheduling under good path operation and increase the number of dispatches by 0.7% under bad-path execution. With partial dynamic RCS dispatches dropped by 20% during good path operation and by only 3.9% during bad path operation compared to performance oriented scheduling. The same static mapping used in the cache standard workload when run in this new environment resulted in the system not coming up. While this may be due to setup issues, it is also likely that insufficient resources were available to the platform tasks during start-up. We are investigating further on a more appropriate static mapping for this environment.

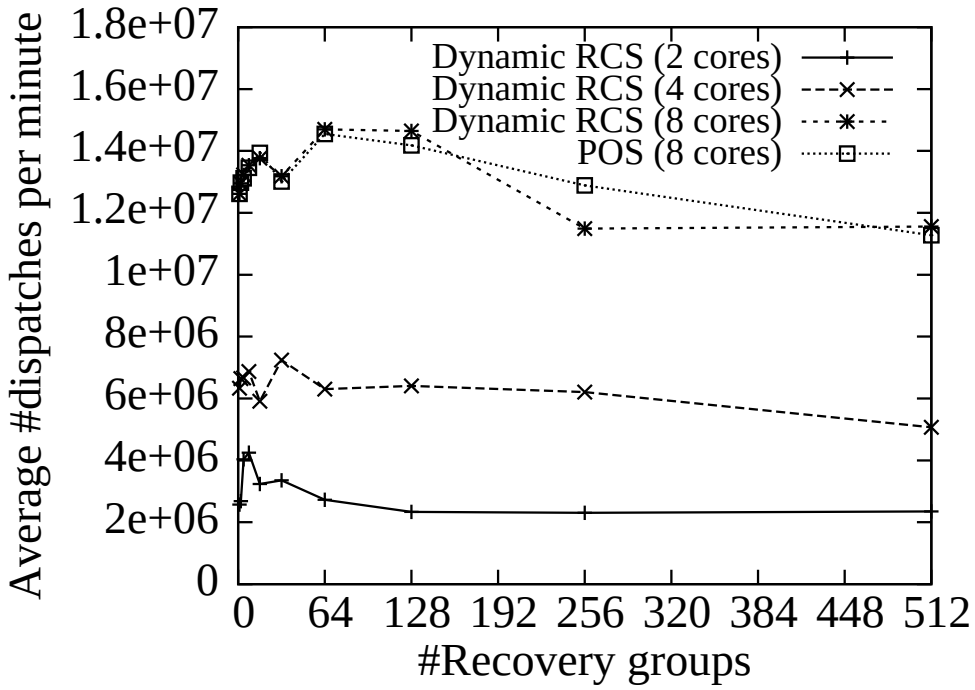


Figure 34: Variation with ♯ Groups (or Queues)

4.4.5 Simulation Experiments Setup

Our simulation studies allowed us to experiment with different system configurations, failure scenarios, recovery parameters and scheduling strategies and various combinations of these factors. The simulator written in C allows configuration of system specifications (such as number of processors and scheduling policy), recovery strategies (proactive/reactive, recovery scope specification) and fault injection parameters (failure rate, failure type, recovery rate). The simulator is driven by an externally provided workload trace specifying individual task descriptions, lock acquisition patterns, task arrival times and execution times. For the simulation experiments described next, we utilized traces of the cache-standard workload described next. Note that, the simulator models the performance behavior of the storage controller but

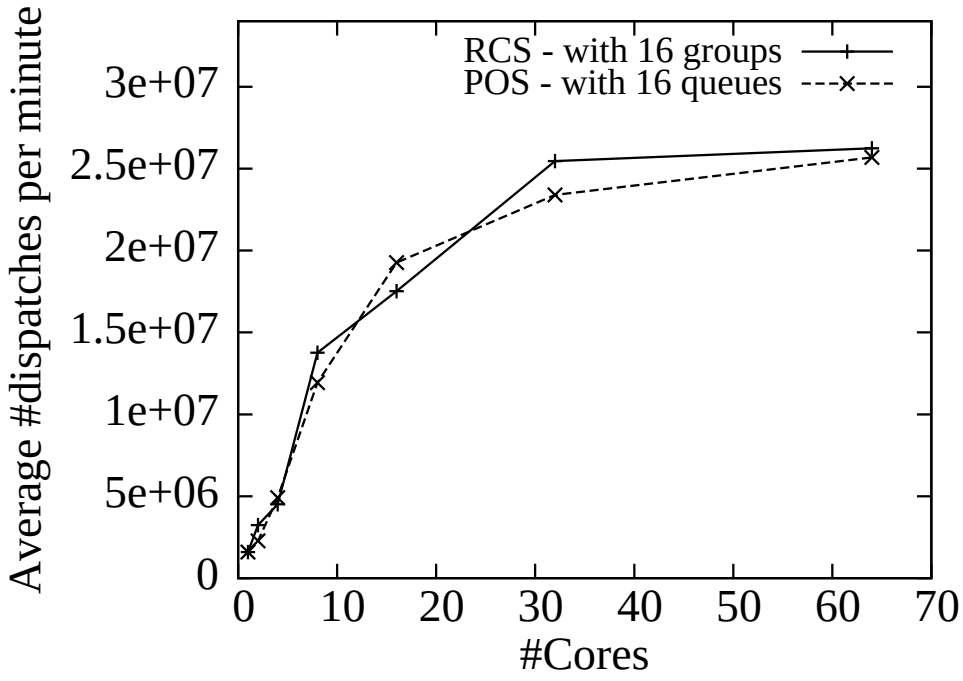


Figure 35: Variation with ♯ Cores (16 Groups or Queues)

does not actually execute the underlying tasks.

4.4.6 Simulation Experimental Results

Based on our simulation results we present guidelines for determining the recovery scopes, the recovery groups, and the mapping of recovery scopes to recovery groups for use in RCS. We show that by selecting appropriate values for the recovery-sensitive system parameters it may be possible to speed up the recovery of storage controllers and achieve good performance at the same time. In summary, the conclusions from our simulation results are:

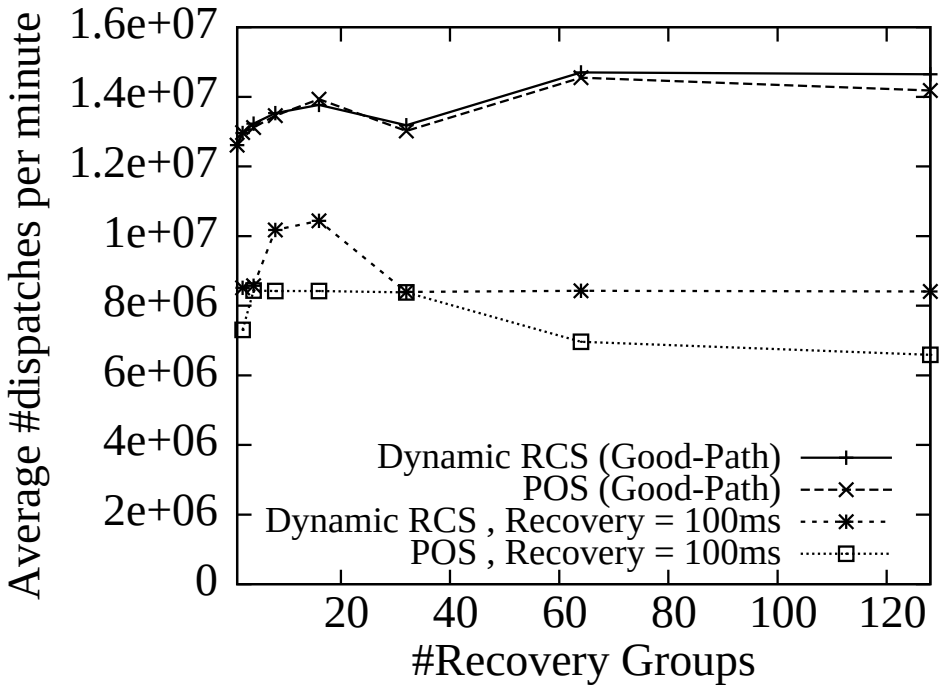


Figure 36: Comparison with Bad-path performance

- The higher the level of multi-threading (number of cores in the processor) the higher the number of recovery groups for effective recovery. Thus, as the multi-threading capability increases, it is beneficial to track finer-granularity recovery scopes through more recovery groups.

- Under operating conditions, in which failure rates (mean time between failures) and recovery rates (mean time to recovery) make it very unlikely that the recovery mechanism has to deal with more than one error at the time, the number of recovery groups does not depend on those rates.

- When mapping recovery scopes to recovery groups it is beneficial to distribute the workload as evenly as possible among the groups.

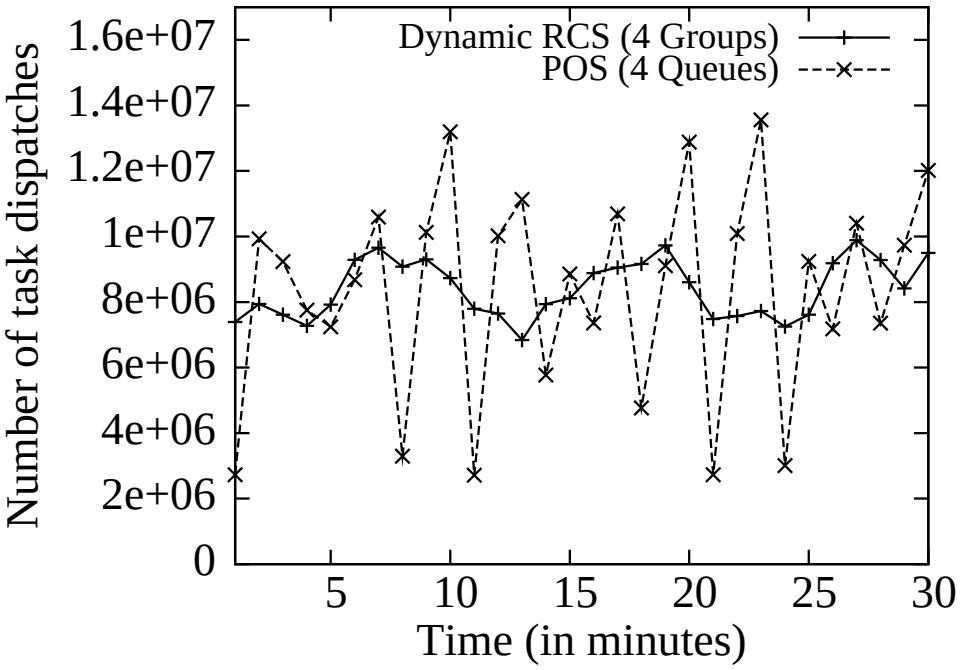


Figure 37: Bad-path performance: 4 queues

- Even if the number of recovery groups and the mapping of recovery scopes to these groups are not optimal, the performance during recovery of the storage system with RCS surpasses the performance of the system with the standard (performance-oriented) scheduling.

4.4.6.1 Effect of Fine-Grained Recovery on System Performance

In order to infuse recovery-consciousness into the allocation of resources, we need to keep track of recovery-dependencies. However, when recovery dependencies are tracked at a fine granularity, the overhead of managing a large number of recovery scopes may induce a severe performance penalty. Therefore, we need to first understand the performance impact of tracking fine-grained recovery scopes. For this set of

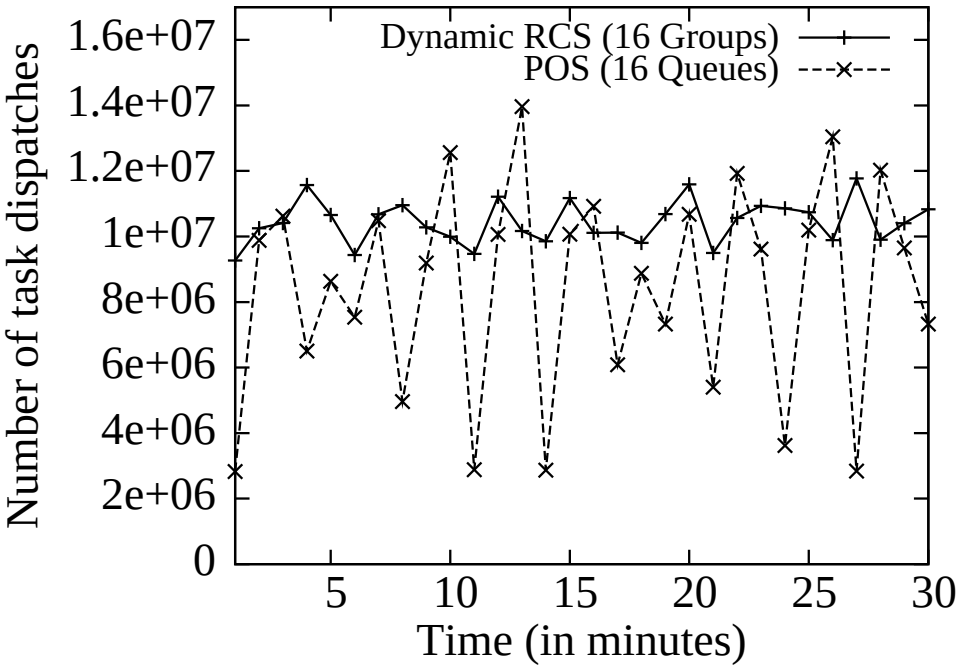


Figure 38: Bad-path performance: 16 queues

experiments, recovery dependency tracking and resource allocation is done through a 1-1 mapping of recovery scopes to recovery groups. Using this mapping, we study the performance impact of fine-grained recovery under different system sizes, and scheduling policies. We measure scheduler performance during normal operation and failure recovery using the number of task dispatches per unit time as a metric.

Figure 34 shows the average number of dispatches per minute during normal operation with varying number of recovery groups. The plot shows the curves for 2, 4 and 8 cores with the dynamic RCS scheduling scheme and that of performance-oriented scheduling with 8 cores. In the case of POS, the workload is uniformly distributed between the queues. Recall that each recovery group is managed using a separate scheduler queue. With RCS, in all three cases, the number of dispatches

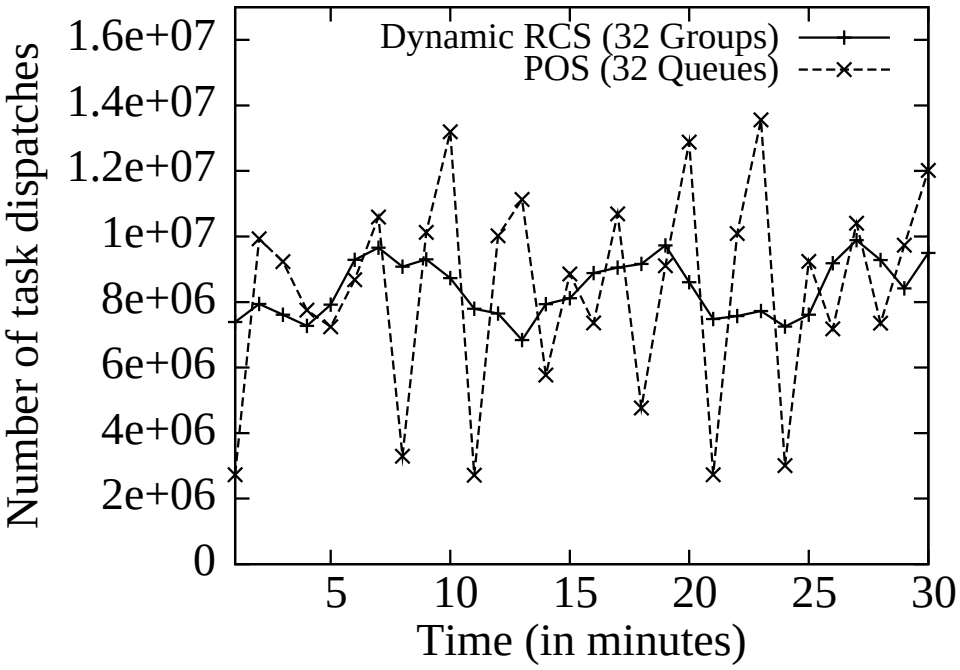


Figure 39: Bad-path performance: 32 queues

initially increase (as much as 16%, 14% and 65% in the case of 8, 4 and 2 cores respectively) and then decreases (as much as 21%, 30% and 45% in the case of 8, 4 and 2 cores respectively). The high performance peak is achieved with 64 groups in the case of 8 cores, 32 groups with 4 cores and 8 groups with 2 cores.

- This shows that the decision on the best choice of number of recovery groups depends on the system size.

Next, although the scheduler initially benefits from the increased concurrency afforded by additional scheduling queues, as the number of queues increases, due to the uneven distribution of workload between recovery groups, scheduling efficiency decreases.

- Depending upon system size, beyond a certain granularity, recovery-consciousness

and keeping track of fine-grained recovery scopes may degrade system performance.

On the other hand, while performance oriented scheduling also exhibits decreasing efficiency with a large number of queues, the degradation in scheduler performance is more graceful due to the uniform distribution of workload.

- The choice of number of recovery groups and mapping of tasks to recovery groups should take into consideration workload distribution between the groups and try to achieve load-balancing.

In order to emphasize the importance of right choice of number of recovery groups on performance, we next compare scheduler performance under dynamic RCS and a load-balanced performance-oriented scheduler with varying system size. We use 16 recovery groups for the dynamic RCS scheduler and 16 queues, with uniform workload distribution for the performance oriented scheduler. Figure 35 shows the average number of dispatches per minute in both cases with varying system size. With this hand-picked choice of number of recovery groups, we see that the system can achieve performance that is close to a performance-oriented architecture, even while tracking recovery dependencies across varying system sizes.

4.4.6.2 Effect of Fine-Grained Recovery on System Availability

The benefit from tracking recovery dependencies is realized during failure recovery. Figure 36 compares scheduler performance during normal operation with that during failure recovery for a system with 8 cores. By availability, we refer to service availability and also the ability of the service to meet performance expectations during failure-recovery. We measure this using scheduler performance during failure-recovery. Failure was emulated by injecting faults into a chosen component at the rate of once in every 10K dispatches of the tasks belonging to that component.

First, the graph shows that, during failure-recovery, for low number of recovery groups, i.e. a coarse granularity of recovery tracking, the benefit from recovery consciousness is low - although still higher than the performance-oriented case. However, at the right granularity, recovery consciousness can make a significant improvement in scheduler performance. In this case, at a group size of 16, recovery-consciousness can effect a 23% improvement in scheduler performance.

Next, consider the group sizes 4 and 32 where POS almost matches the performance of RCS. Figure 37, 38 and 39 represent the number of dispatches per minute over a duration of 30 minutes. The graphs show that even at group sizes of 4 and 32 where POS matches RCS in average number of task dispatches per minute, POS results in serious fluctuations of scheduler performance. At some instances, the number of dispatches with POS drops to as low as 65% of that with RCS. Recall that, POS distributes workload equally amongst all processors without considering recovery dependencies. Therefore, during failure, many tasks dependent on the failing task may be executing concurrently. As a result, in spite of fine-grained recovery, the entire recovery process takes longer, resulting in a drop in performance due to unavailability of resources for normally operating tasks.

- We can argue that even with some inaccuracy in the selection of number of recovery groups, there is a conclusive advantage over performance oriented scheduling during failure recovery by being able to track recovery dependencies.

- Also, in spite of implementing fine-grained recovery, it is crucial to track recovery-dependencies to improve performance during failure recovery.

Figure 40 shows the variation in system recovery time by varying individual task recovery time, the number of cores, and the distribution of tasks between groups. The figure is generated based on the model for $MTTR_{POS}$ and $MTTR_{RCS}$ described in Section 4.3.1. The lower surface (in red) depicts the recovery time variation for

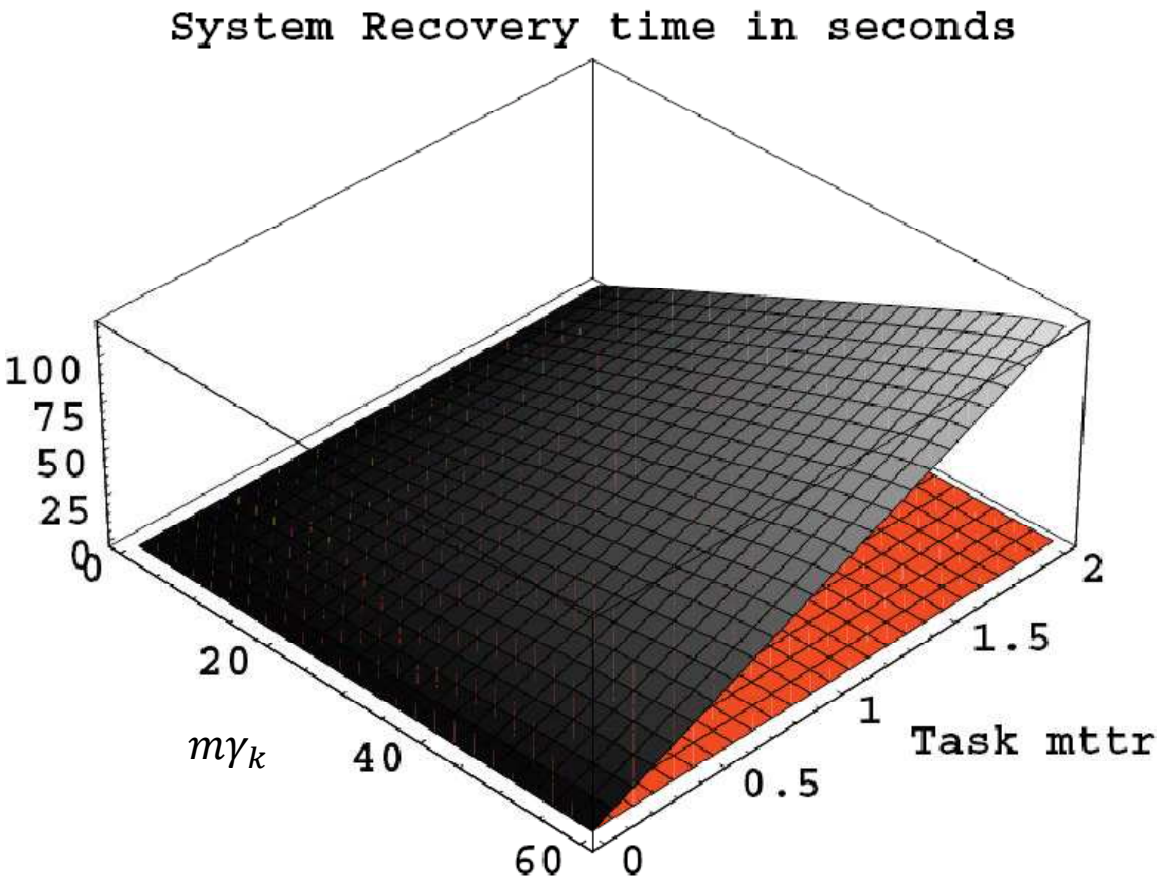


Figure 40: System MTTR

RCS and the upper surface (in gray) depicts the recovery time variation under POS. The x-axis represents the variable $m\gamma_k$ where m represents the number of cores in the system and γ_k represents the probability that a task belongs to the failing recovery group k. Intuitively the x-axis can be thought of as the number of cores per recovery group. The y-axis represents individual task recovery time in seconds and the z-axis represents the total system recovery time in seconds. The constraint for RCS is set as $c_k = 10$. As the graph shows, for POS, the system recovery time increases rapidly with increasing task recovery time and $m\gamma_k$. The extent of recovery may increase either due to increase in system size or due to a large proportion of tasks belonging to the failing recovery group. On the other hand, with RCS, the recoverability constraint ensures that the system recovery time remains low by restricting the number of cores

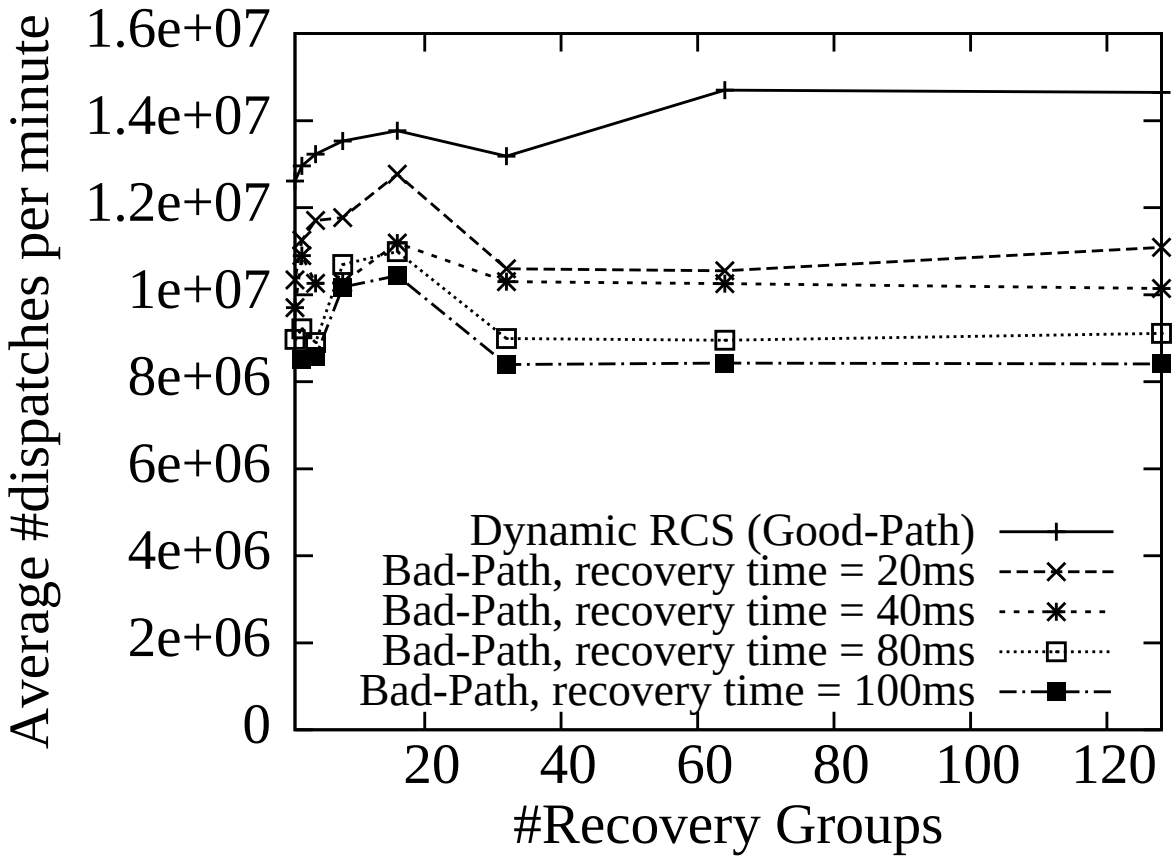


Figure 41: Variation with Recovery Rate

assigned per recovery group.

4.4.6.3 Sensitivity to Recovery and Failure Rate

Figure 41 shows the variation of scheduler performance for different recovery rates for tasks belonging to component 5. The failure rate was fixed at 1 in every 10K dispatches of tasks belonging to component 5. The figures tells us that the choice of number of recovery groups is nearly independent of the recovery rate, since if 'x' number of recovery groups is a better choice than 'y' for a certain recovery rate, it is

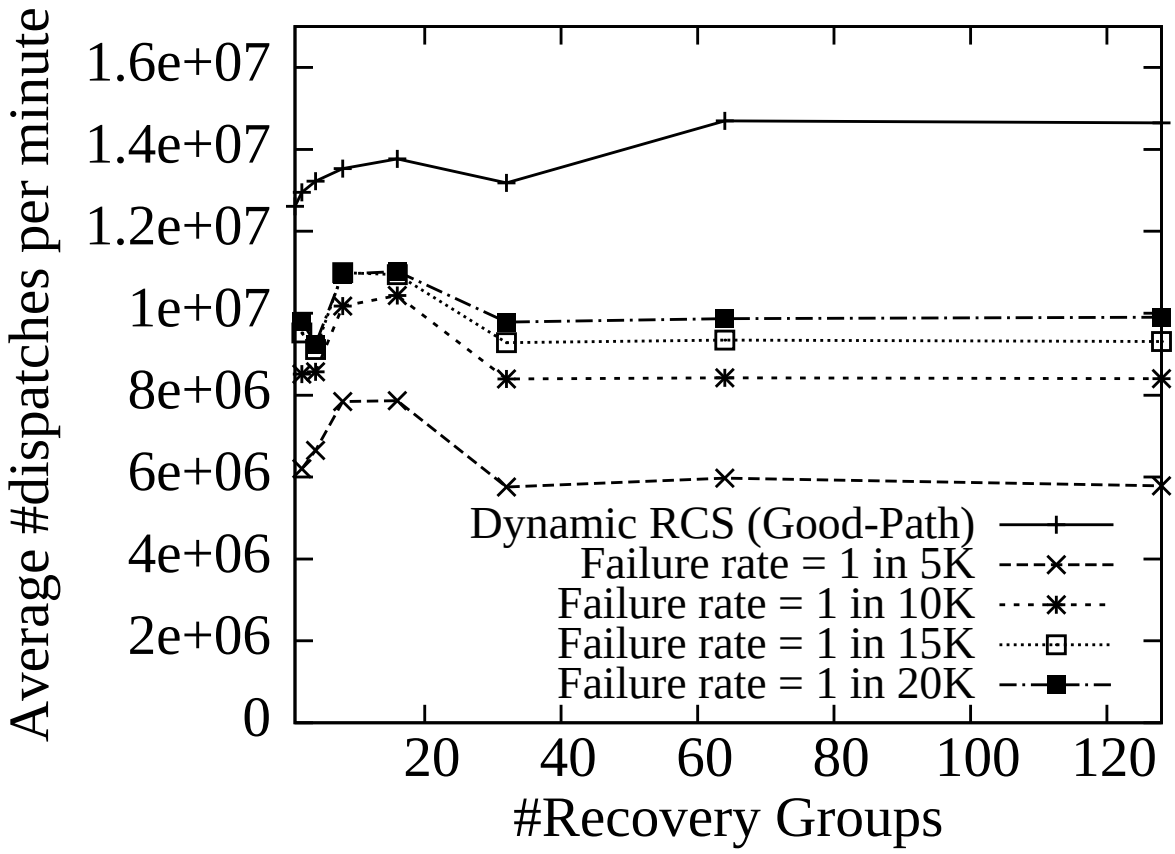


Figure 42: Variation with Failure Rate

almost true for all other recovery rates also.

Figure 42 shows the variation of scheduler performance with different failure rates. The recovery time for a single failed task was set to 100 ms and failure injected into tasks belonging to component 5. As with the case of recovery rate, the figure shows that the choice of number of recovery groups is nearly independent of failure rate.

4.5 Discussion

The fact that recovery-conscious scheduling requires minimal change to the software allows for it to be easily incorporated even in legacy systems.

Dynamic RCS can match good path performance of performance oriented scheduling and at the same time significantly improve performance under localized recovery. Even for the small 5% recovery overhead introduced by us, we could witness a 16.3% improvement in throughput and a 22.9% improvement in response time with dynamic RCS. Moreover, the qualitative benefits of RCS in enhancing availability and ensuring that localized recovery is scalable with system size makes it an interesting possibility as systems are moving toward more parallel architectures. Our experiments with various scheduling schemes have given us some insights into the overhead costs such as lock spin times imposed by RCS algorithms. In ongoing work we are continuing to characterize and investigate further optimizations to RCS schemes.

Our results also seem to indicate that for small localized recovery time and system sizes, proactive policies i.e. mapping resource pools to recovery groups, can deliver the advantage of recovery-consciousness. However as system size increases or localized recovery time increases, we believe that the actual benefits of reactive policies such as suspending dispatch from groups undergoing recovery may become more pronounced. In ongoing research we are experimenting with larger setups and longer localized recovery times.

Static and partial dynamic RCS schemes are limited by their poor resource utilization in workload intensive environments. Hence we do not recommend these schemes in an environment where the system is expected to run at maximum throughput. However, the tighter qualitative control that these schemes offer may make them, especially partially dynamic RCS, more desirable in less intensive environments where there is a possibility to over-provision resources, or when the workload is well understood. Besides in environments where it 'pays' to isolate some components of the

system from the rest such mappings may be useful. We are continuing research on optimizing these algorithms and understanding properties that would prescribe the use of such static or partially dynamic schemes.

While our experiments provide some insights into the selection of parameters such as recovery groups, clearly these decisions are largely impacted by the nature of the software. Below, we present certain guidelines for the selection of these parameters which must be validated for the particular instance of software and system configuration. A possible procedure to perform this validation is to evaluate the impact of various parameters using simulation based studies and workload traces as shown in our paper [134]. The number of recovery scopes in the system is a characteristic of the software and the dependencies between tasks. Once the granularity of recovery has been identified, and the dependency information has been specified (with explicit dependencies being specified initially and the system identifying implicit dependencies over certain duration of observation), the recovery scopes are specified. During runtime, tasks are enqueued based on the recovery scope that has been identified for the task. The scheduler efficiency now depends on the number of recovery groups that need to be iterated through at runtime. This choice of recovery groups, depends on the degree of multiprocessing (number of cores) and the mapping of recovery scopes to recovery groups depends on the distribution of tasks among recovery scopes. Thus the guidelines for selection of recovery-aware parameters can be summarized as follows:

- The optimal number of recovery groups depends on the degree of multiprocessing. However, choosing the number of groups to be more than the number of cores can help improve performance by reducing contention for job queue locks, for example.

- The choice of the number of recovery groups and the mapping of tasks to recovery groups should take into consideration workload distribution between the groups and try to achieve load-balancing and avoid idle cycling of the scheduler

through empty queues looking for work. The information required to perform load-balancing can be acquired by studying the workload for distribution of tasks between recovery scopes and their arrival rates.

- Even with some inaccuracy in the selection of number of recovery groups, there is a conclusive advantage of RCS over POS during failure recovery by being able to track recovery dependencies. This gives the developer some flexibility in choosing the number of recovery groups.

4.6 Related Work

Our work is largely inspired by previous work in the area of software fault tolerance and storage system availability. Techniques for software fault tolerance can be classified into fault treatment and error processing. Fault treatment aims at avoiding the activation of faults through environmental diversity, for example by rebooting the entire system [72, 148], micro-rebooting sub-components of the system [50], through periodic rejuvenation [85, 68] of the software, or by retrying the operation in a different environment [114]. Error processing techniques are primarily checkpointing and recovery techniques [73], application-specific techniques like exception handling [137] and recovery blocks [116] or more recent techniques like failure-oblivious computing [119].

In general our recovery conscious approaches are complementary to the above techniques. However we are faced with several unique challenges in the context of embedded storage software. First, the software being legacy code rules out re-architecting the system. Second, the tight coupling between components makes both micro-reboots and periodic rejuvenation tricky. Rx [114] demonstrates an interesting approach to recovery by retrying operations in a modified environment but it requires checkpointing of the system state in order to allow 'rollbacks'. However given the high volume of requests (tasks) experienced by the embedded storage controller and their

complex operational semantics, such a solution may not be feasible in this setup.

The idea of localized recovery has been exercised by many. Transactional recovery using checkpointing/logging methods is a classic topic in DBMSs [100] and is a successful implementation of fine-grained recovery. In fact application-specific recovery mechanisms such as recovery blocks [116], and exception handling [137] are used in almost every software system. However, few have made an effort on understanding the implications of localized recovery on system availability and performance in a multi-core environment where interacting tasks are executing concurrently. Likewise, the idea of recovery-conscious scheduling is to raise the awareness about localized recovery in the resource scheduling algorithms to ensure that the benefits of localized recovery actually percolate to the level of system availability and performance visible to the user. Although vast amounts of prior work have been dedicated to resource scheduling, to the best of our knowledge, such work has mainly focused on performance [161, 81, 83, 74, 57]. Also much work in the virtualization context has been focused on improving system reliability [115] by isolating VMs from failures at other VMs. In contrast, our development focuses more on improving system availability by distributing resources *within* an embedded storage software system by identifying fine-grained recovery scopes. Compared to earlier work on improving storage system availability at the RAID level [140], we are concerned with the embedded storage software reliability. These techniques are at different levels of the storage system and are complementary.

4.7 Summary

In this chapter we addressed the issues in the second and third tier of our recovery-conscious framework. Our main contributions include the development of recovery-conscious scheduling, a non-intrusive technique to reduce the ripple effect of software failure and improve the availability of the system and guidelines for effective mappings

of dependent tasks to recovery groups over which recovery-conscious scheduling is performed. We presented a suite of RCS algorithms and quantitatively evaluated them against performance oriented scheduling. We focused on developing effective mappings of dependent tasks to processor resources through careful tuning of recovery-sensitive parameters. Through our analysis and experimentation we have shown that through careful tuning of the system configuration and the recovery-sensitive parameters, RCS can significantly improve system performance during failure recovery and thus improve system resiliency to faults while continuing to sustain high performance during normal operation.

CHAPTER V

FAULT-TOLERANT MIDDLEWARE OVERLAYS

5.1 Introduction

With rapid advances in network computing and communication technology and the continued decrease in digital storage cost, the amount of digital information continues to grow at an astonishing pace. Many organizations and enterprises today are facing the challenge of dealing with data storage ranging from terabytes and petabytes to exabytes, zettabytes, yottabytes and beyond. Such trends create continuous high demands for massive storage and storage services. High-availability scale-out storage clusters combine smaller units of storage to provide a scalable and cost-effective storage solution [14, 21, 70]. The scale-out clustered storage architectures allow enterprises to gain significantly in terms of cost, scalability and performance.

Current scale-out storage systems use active replication [124] based middleware to ensure consistent access to shared resources in the absence of centralized control and at the same time provide high throughput and a single system image (SSI). In order to guarantee high-availability to applications, the middleware typically maintains critical application state and check-point information persistently across nodes through active replication. Active replication based models are not only used in clustered storage systems, but are also common in distributed locking services [47] and high-performance computing [64].

However, though the use of symmetric active replication models removes hardware as both single-points-of-control and single-points-of-failure, it causes the storage middleware itself to now become a single-point-of-failure. In some instances, unavailability of a highly-available, active replication based service due to network

outages and software errors can cause as much as 0.0326% unavailability [47, 52]. This is equivalent to three days of downtime over a period of 100 days and intolerable for many applications that demand above 99.99% uptime from storage services. Moreover, the current scale-out storage architecture is also vulnerable to application-induced failures of the middleware, in addition to other issues like application-level non-determinism [141] and middleware bugs themselves. By application-induced middleware failure, we mean that a single cluster application can cause an error which leads to unavailability of the cluster middleware for all other applications concurrently running over the cluster. For example, the highly available Chubby locking service reports that the failure of application developers to take availability into consideration "magnified the consequence of a single failure by a factor of a hundred, both in time and the number of machines affected" [47].

One obvious approach to improving availability and reliability in such systems is to provide application fault isolation and eliminate symmetric clustering middleware as a single-point-of-failure. While application fault isolation can be achieved through partitioning of a single storage cluster into smaller independent clusters [30, 26], without care, one may lose the SSI and the flexibility to access storage from anywhere within the system. The key challenge, therefore, is to provide fault-boundaries while continuing to deliver SSI and flexible accessibility. In this chapter we introduce the notion of hierarchical middleware architectures. We organize critical cluster management services into a hierarchical overlay network, which separates persistent application state from global system control state. This clean separation, on one hand, allows the cluster to maintain SSI by communicating control state to all nodes in the network, and on the other hand, provides fault isolation by replicating application state within only a subset of nodes.

We present baseline availability analysis in flat and hierarchical clusters with varying storage nodes. Our analysis shows two important results. First, we show that

simply increasing the number of hardware nodes in a symmetric system will not improve availability or reliability as long as the middleware is vulnerable to simultaneous failures. Second, we show that by trading some symmetry for better fault isolation, hierarchical overlay storage architectures can significantly improve system availability and reliability.

We conducted a thorough experimental evaluation of the availability and reliability of our proposed solution. Utilizing the relationship between workload and software failure rate [149, 160, 80], we quantitatively show that hierarchical middleware architectures can provide significant improvements in reliability and availability of mass storage systems. For example, organizing a 32 node system into a hierarchical cluster can reduce downtime by nearly 94% and improve Mean-time-to-failure (MTTF) by approximately 16 times.

The rest of this chapter is organized as follows. We first present the problem statement and describe our approach in Section 5.2. We develop probabilistic markov models for analyzing the availability and reliability of the proposed architectures in Section 5.3. In Section 5.4 we present an in-depth experimental evaluation of availability and reliability in both flat and hierarchical architectures. We compare and contrast the two models in Section 5.5, and conclude in Section 5.7 with related work and a summary.

5.2 Overview

Before presenting an overview of the proposed hierarchical overlay architecture, we first describe the conventional approach, its flat cluster architecture, and the potential problems with respect to application-induced failures.

5.2.1 Conventional Approach

High availability scale-out storage clusters are composed of nodes, a network that connects the nodes, and middleware that provides a highly available environment

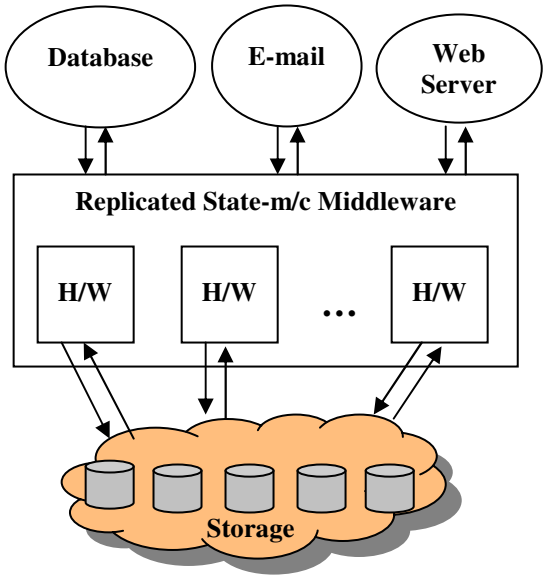


Figure 43: Traditional flat storage cluster.

for applications that operate in the cluster. The back-end storage is shared by all nodes in the cluster. All nodes within the cluster are peers with no single-point-of-control. In this thesis, we will refer to this ubiquitous form of clusters as a **flat** cluster (Figure 43). Examples of such systems exist in both research [30, 155, 64] and commercial [70, 14] settings.

In such systems, the middleware must ensure consistent access to the shared storage and provide a highly available environment to applications in the absence of centralized control. Typically this is achieved through active replication based on replicated state machines (RSMs) [124]. RSMs maintain consistent state on every node by applying an identical ordered set of requests, referred to as **events**, on each instance of the state machine. Since consistent state is maintained on all nodes, applications may failover to any node when the node currently serving the application fails.

Events belong to two categories - global or setup events used for liveness, node joins, departures, resource allocations etc., and application events that are used to

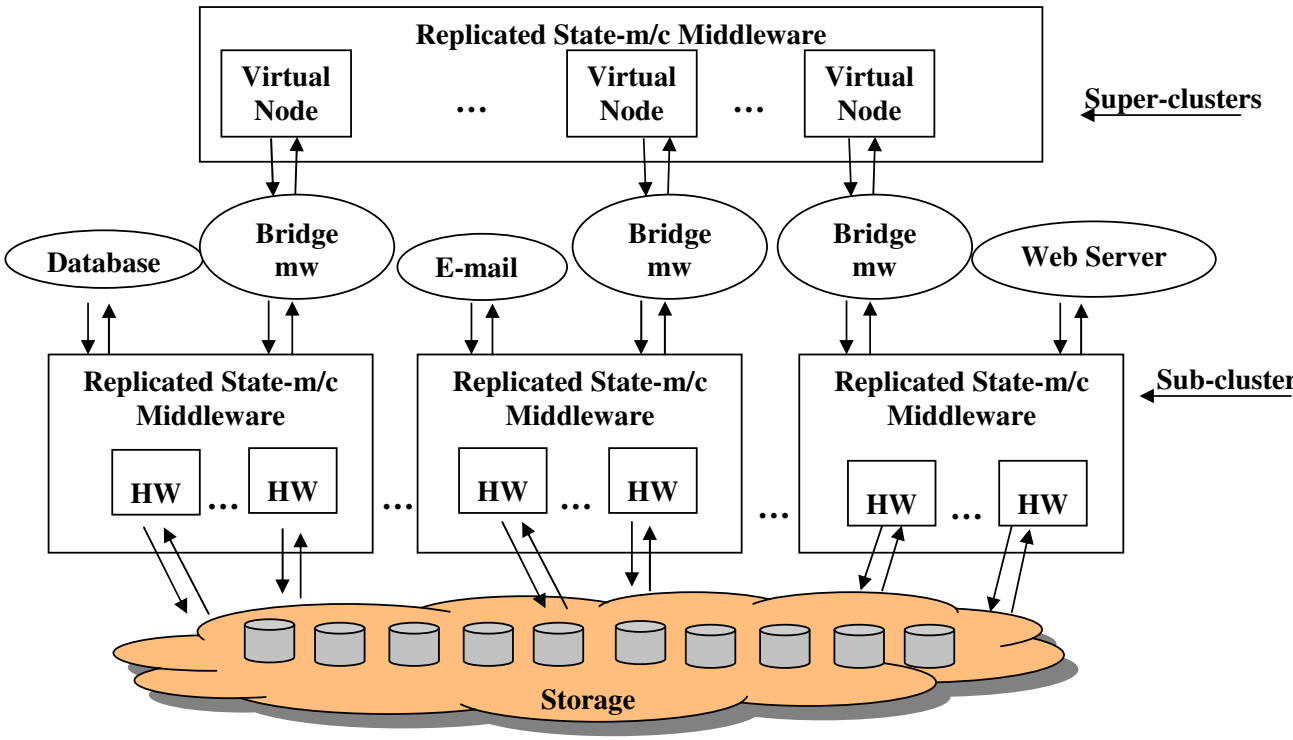


Figure 44: Hierarchical cluster.

maintain application state globally. Typically, middleware generated setup events belong to a (usually well-tested) pre-defined set. On the other hand, application events such as check-points and persistent state are generated by the applications and passed on to the middleware through interfaces provided by the middleware. This potentially opens up the middleware to faulty application generated events which when processed synchronously will cause all instances of the middleware to fail simultaneously, bringing down the whole cluster. The goal of this work is to eliminate or reduce such application-induced dependent middleware failures in scale-out storage systems while maintaining the provision of SSI, scalability and high-availability of scale-out storage systems and services.

5.2.2 Hierarchical Middleware Architectures: Our Approach

We first present an outline of the design of hierarchical middleware architectures. The key idea for improving system availability and reliability is to define application-fault

boundaries by separating global control state from persistent application state. In order to prevent the symmetric middleware from becoming a single-point-of-failure, hierarchical architectures organize cluster management services into an overlay such that global control events are communicated to all nodes and application generated events are processed by only a subset of nodes. Note that, the hierarchy is only used to regulate control messages and does not interfere with the actual data path.

Figure 44 shows a two level hierarchical cluster that utilizes a hierarchical middleware architecture. Clusters at the upper level are called super-clusters and clusters at the lower level are called sub-clusters. Sub-clusters and super-clusters are flat clusters connected together by a bridging middleware. The bridging middleware is a cluster application that runs over a sub-cluster. It is essentially stateless and serves two main purposes: first, it creates a virtual node environment which emulates a hardware node by utilizing underlying resources; second, it provides a channel between sub-clusters and super-clusters for communicating global events. It also serves as a channel for communicating application-specific processing requests from super-clusters to sub-clusters.

Nodes in the super-cluster are called virtual nodes. Each virtual node represents a sub-cluster. Super-clusters provide SSI and manage the communication of global control state across sub-clusters. The construction of such a hierarchical model and the decision on critical hierarchical parameters such as sub-cluster size and number of hierarchical levels will rely on a number of system-supplied parameters, such as known or historical failure rates, lease timers (heartbeats required among the nodes for better response times), load skew per node and load skew per sub-cluster or per application category. Due to space constraints, we delay the discussion on issues regarding hierarchical cluster overlay construction to Section 5.5. Below, we focus more on the availability and reliability properties of our hierarchical organization.

Since the bridging middleware is a cluster application that runs over a sub-cluster,

if the underlying node fails, the bridge application itself would instantaneously failover to another node within the sub-cluster. In the event of either a virtual node dropping out of the super-cluster or the corresponding node dropping out of the underlying sub-cluster, the bridge middleware will instantiate another virtual node over its underlying sub-cluster to participate in the super-cluster. The bridge middleware relies on the underlying sub-clusters to provide actual storage services.

In the hierarchical clustering paradigm, applications may be deployed on a cluster at any level in the hierarchy. Applications that are deployed at the higher levels distribute work to sub-clusters at the lower level according to the current system state. Typically, load balancing and resource management applications are placed at upper levels as they are shared across a number of sub-clusters. While global events are communicated to all nodes in the hierarchical cluster through the bridge middleware and super-clusters, application events are processed only by the nodes within the current sub-cluster in which the application is processed. As a result, the total application-generated middleware workload is partitioned amongst sub-clusters.

Storage cluster applications may have global or local access points for accepting external requests. Global access points will be owned by a virtual node at the top of the hierarchical cluster. External requests received at the global access point are decomposed and propagated along the cluster hierarchy to the appropriate sub-clusters at the lowest level. Local access points are provided for services specific to a part of the hierarchical cluster. This allows some cluster services to continue even if the hierarchical cluster is not fully connected momentarily. We next illustrate the partitioning of middleware workload in a hierarchical cluster.

5.2.3 Example Application: Data Migration

Moving data from one location to another to change its retention cost, availability, or performance is a well known strategy for managing data. With clustered storage

it is possible for data movement to be managed internally to the cluster. The cluster middleware provides the persistent storage to track a data movement operation, which would at least include the source data location, the target data location, and a current copy location. The current copy location is kept to minimize the amount of duplicate data copied after recovery from a cluster transition.

For example, if the current copy location is updated for every 1GB of data moved, moving 300GB of data results in 300 application-generated update events to the cluster middleware. In a flat cluster the middleware workload (number of events) per node for this data movement request would include the setup (control) events in addition to 300 application-generated events. In a hierarchical cluster a data movement operation can be initiated between any two storage nodes in the cluster at the expense of a setup cost. Also since super-clusters distribute work among its underlying sub-clusters, not all sub-clusters perform work per hierarchical cluster request. Thus for a two level hierarchy, while the setup messages would be propagated to and processed by all instances of the middleware, the actual application-generated requests will only be processed in the sub-cluster(s) handling the application.

5.2.4 Example Application: Virtualization

Virtualization tools for storage have been deployed for storage consolidation, flexibility in management, better utilization and availability. With the increase in the processing ability of storage controllers, spare processing capacity can be utilized for data intensive applications [10, 18] which can also exploit the proximity to data, resulting in significant performance improvement. Thus, sub-components of applications such as databases, webserver and management applications can run over the storage controller while the rest of the applications continue to execute at higher layers. Since the virtualization layer is built over the cluster middleware infrastructure, application sub-components can directly exploit the high availability and replication

Availability	Fraction of time an application provides service to its users.
Reliability	Probability of failure-free operation over a specified period of time.
$MTTF$	Mean-time-to-failure
$MTTR$	Mean-time-to-restore
λ_{hw}	Hardware failure rate = $1/MTTF_{hw}$
μ_{hw}	Hardware repair rate = $1/MTTR_{hw}$
λ_{mw}	-Middleware failure rate = $1/MTTF_{mw}$
μ_{mw}	Middleware repair rate = $1/MTTR_{mw}$
λ_{vn}	Virtual node failure rate = $1/MTTF_{vn}$
μ_{vn}	Virtual node repair rate = $1/MTTR_{vn}$
λ_{vn_app}	Bridge middleware failure rate = $1/MTTF_{vn\_app}$
μ_{vn_app}	Bridge middleware repair rate = $1/MTTR_{vn\_app}$

Table 9: List of Terms

services offered by the clustering middleware. In order to improve both availability and performance, applications can maintain critical state persistently and globally through interfaces offered by the middleware. However, this opens up the middleware to possible application-induced failures due to carelessness on the part of application developers or unexpected use of interfaces. Additionally policy constraints might dictate keeping different application sub-components on different storage servers. Thus partitioning of the storage cluster is necessary to ensure not only performance benefits but also fault domains. Hierarchical middleware architectures help retain the benefits of virtualization while guaranteeing availability and fault-isolation.

5.3 Availability and Reliability Analysis

In this section we develop probabilistic markov models [150] for the analysis of availability and reliability of flat and hierarchical clustering architectures. Our models were generated using an automated tool developed by us. System states are represented as states in a Markov chain and failure and repair rates are represented by the transition rates between the states in a Markov chain. Failure and repair are treated as stochastic processes and Markov chains are the standard way to model stochastic

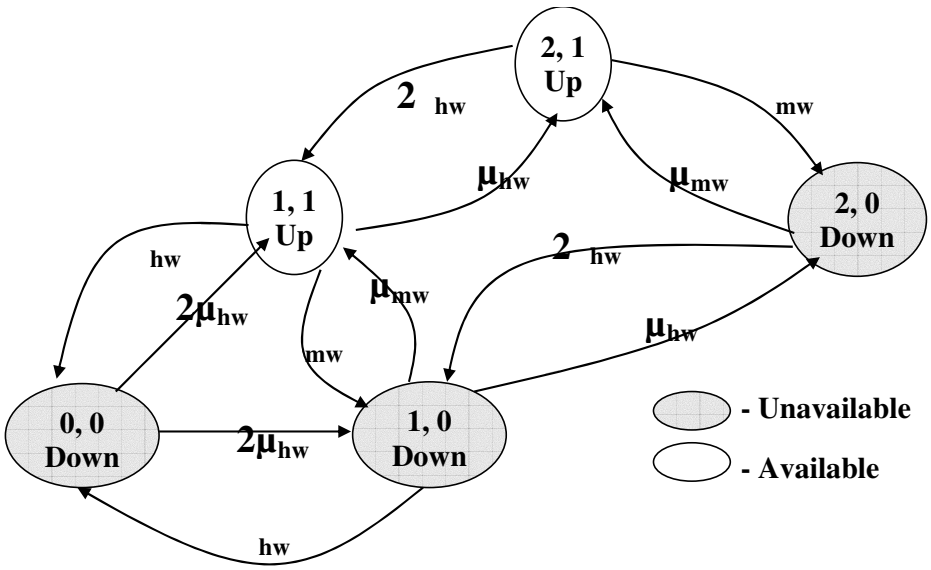


Figure 45: Availability of a 2 node cluster.

processes in order to determine the average probability of being in a particular state. We assume that the time to failure and repair are exponentially distributed.

An application becomes unavailable when failure of the hardware or middleware occurs in a combination such that the application is no longer running and has to be restarted after restoration of the underlying components. We assume that there is sufficient redundancy in network connectivity and that it is not a limiting factor. Each node runs an identical instance of the middleware.

We assume that hardware failures are independent while middleware failures are correlated. Our aim is to concentrate on the effect of correlated middleware failures and study the effectiveness of our proposed hierarchical architectures in eliminating such failures. Therefore, without loss of generality, we do not consider independent middleware failures or correlated hardware failures in this study. Note that independent middleware failures typically will not cause a marked difference in our comparison.

Dependent middleware failures cause all instances of the middleware to crash simultaneously. Middleware repair is also assumed to be simultaneous. Middleware

repair involves restarting of the failed components, synchronization of state and isolation of the failure inducing request (most often, the last event processed by the cluster). Since the middleware already makes critical application state available to all nodes within a single cluster, we assume that failover of an application, when it occurs, is instantaneous.

Table 9 gives a list of important parameters and terms used in the chapter and their definitions, including Mean-time-to-failure (MTTF), mean-time-to-restore (MTTR), failure rate (λ) and repair rate (μ) of hardware, middleware, virtual node, and bridge middleware. Each state in the Markov diagram is given by a pair of values indicating the number of instances of hardware and middleware that are functional, and the status of the application. The transitions between states represent failure and repair rates. We illustrate our modeling methodology on two node flat clusters, sub-clusters and super-clusters. Extending this model to an N node cluster is straightforward.

5.3.1 Availability Modeling

Figure 45 shows the Markov model for availability of a flat two node cluster. Note that, since we consider only dependent failures of the middleware, only a single instance of the middleware is considered to be running on the entire cluster. The system is considered available if at least one instance of the hardware and the middleware are functional.

For the hierarchical system, the model in Figure 45 now represents the Markov model for applications deployed over the sub-clusters at the lowest level. At that level, the middleware instances are run over the actual hardware nodes. At higher levels, middleware instances are run over virtual nodes. In order to simplify our analysis, we consider the same middleware to be running at all levels in the hierarchy. We can represent the availability at a level running over virtual nodes by replacing λ_{hw} and μ_{hw} with λ_{vn} and μ_{vn} respectively in Figure 45. We next describe the reliability

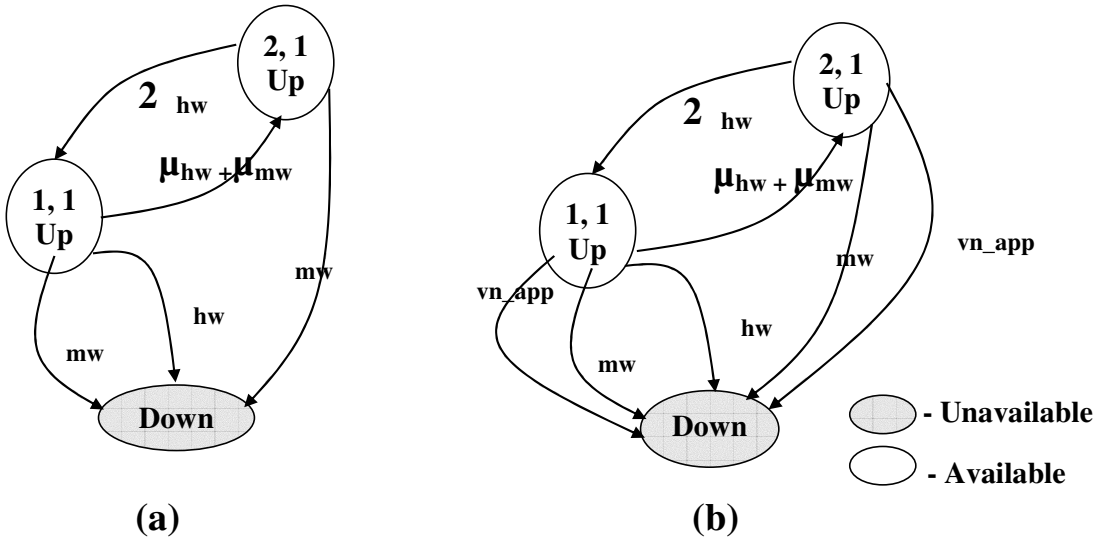


Figure 46: Reliability models: (a) Flat clusters (b) Virtual Node

modeling and computation of virtual node failure rates.

5.3.2 Reliability Modeling

We analyze the reliability of clustering architectures using Markov models with absorbing states [150]. Figure 46(a) shows the Markov model for a two node system. The model for the flat architecture shown in Figure 46(a) also represents the reliability model for the lowest level sub-clusters in the hierarchical architecture. Again, the model can be extended to a sub-cluster of any size. As in the case of availability, we can obtain the Markov model for reliability of virtual node clusters (super-clusters), by replacing hardware failure and repair rates in Figure 46(a) by virtual node failure and repair rates.

The reliability of the virtual node presented by a lowest level cluster that directly runs over the hardware is computed using the model shown in Figure 46(b). The model shows that a virtual node failure may be caused either due to failure of all instances of the underlying hardware, the middleware or the bridge middleware. Restoring a virtual node at a given level involves restoring the underlying hardware and virtual nodes up to that level (for clusters that may be running at a level > 2).

$MTTF_{hw}$	2 years
$MTTR_{hw}$	4 hours
$MTTF_{mw}$	3 years
$MTTR_{mw}$	15 minutes
$MTTF_{vn\_app}$	3 years (expected)
$MTTR_{vn\_app}$	15 minutes (expected)

Table 10: Component failure and repair rates

We approximate virtual node repair rates to be the same as that of the hardware since hardware repair is orders of magnitude slower than that of software.

5.4 Evaluation of Clustering Architectures

In this section we evaluate the flat and hierarchical architectures based on the models presented in the previous section. We use failure and repair rates based on our experiences with real deployments of commercial storage systems. Since actual failure and recovery rates are sensitive information and held closely by vendors, we are unable to disclose the identity of the system and can only state that the numbers are representative of current deployments. Table 10 represents the failure and repair rates for the various components in the system. Note that the middleware failure rate is for the occurrence of correlated failures. Independent middleware failures are expected to occur more frequently [151]. We expect the bridge middleware to have failure and repair rates close to that of the clustering middleware. We validated our availability and reliability models against numbers reported for current deployments. Accordingly, the error in the availability model is less than 0.00004% while that for the reliability model is around 1.5%.

We have developed an automated tool to compose Markov models from specifications and compute availability and reliability. The tool, written in C++, uses the Mathematica package [154] for computations. The experiments reported in this work are based on a system with a cluster size of up to 32 nodes. With the failure and repair rates presented in Table 10, when considering only independent failures of the

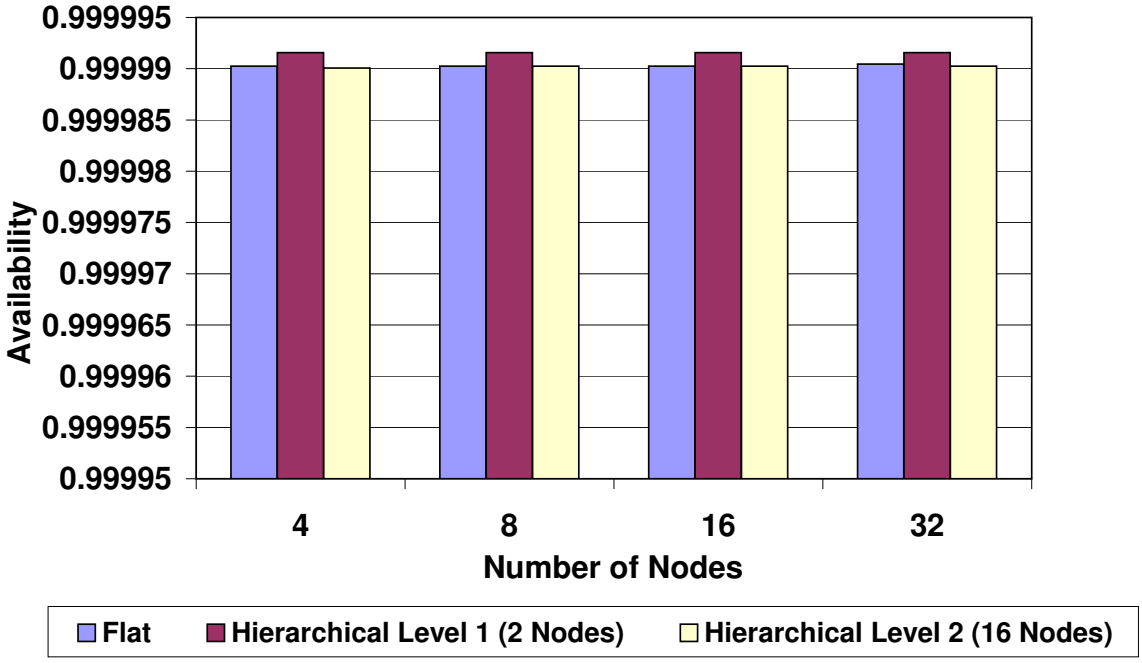


Figure 47: Baseline availability.

components, a 2 node cluster was able to achieve 6 nines of availability. Based on this we choose a sub-cluster size of 2 for our experiments. Results with other sub-cluster sizes and other configurations have been briefly summarized in Section 5.4.3.3.

Our analysis reveals two interesting phenomena. First, although it is common to focus on hardware solutions to availability and reliability, our analysis (Section 5.4.1 and 5.4.2) show that middleware (software) failure rates are a far bigger contributor to unavailability. Second, we show that in the common case where increasing workload leads to more software failures, the hierarchical approach provides significantly higher robustness by isolating applications from middleware failures.

5.4.1 Baseline Availability Analysis

Figure 47 shows the baseline availability in flat and hierarchical clusters with 4-32 storage nodes. In order to present an apples-apples comparison, we compare only 2 level hierarchical organizations of the nodes with the sub-cluster size set to 2 nodes. As the graph shows, availability in the two levels of hierarchical clusters are comparable

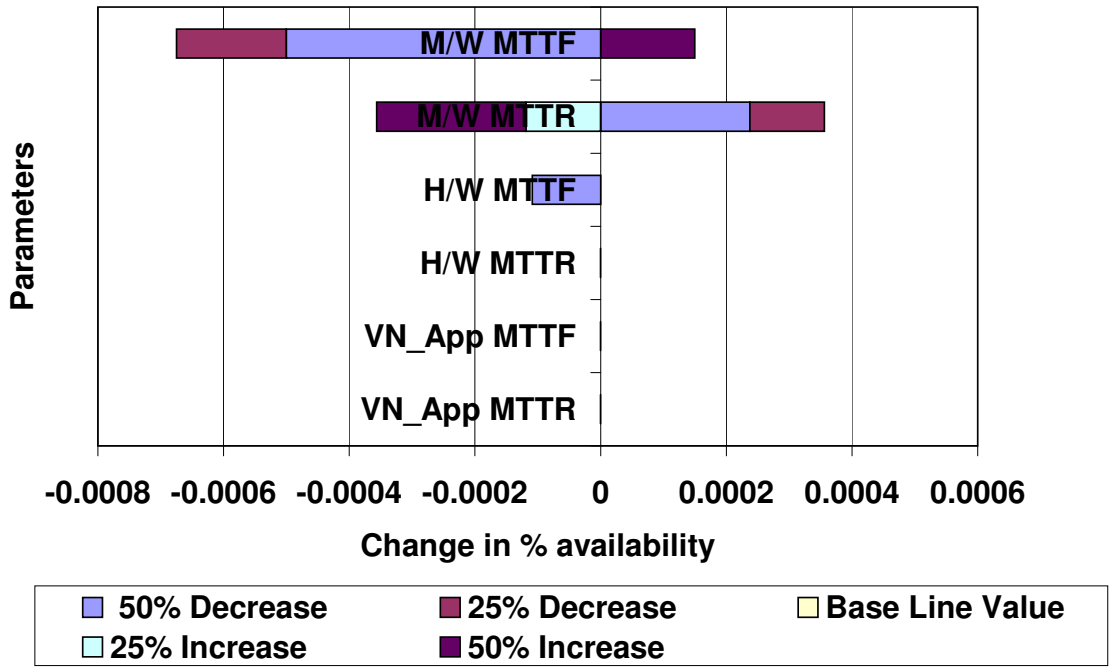


Figure 48: Sensitivity analysis of availability.

to that of the corresponding flat clusters. However, note that applications are now isolated from middleware failures in other sub-clusters. Also note that, just increasing cluster size has little effect on availability.

5.4.1.1 Sensitivity Analysis for Availability

Sensitivity analysis allows us to understand which parameters have the most pronounced effect on availability and reliability. We choose a 32 node system and compare flat and 2 level hierarchical configurations with sub-cluster size set to 2. Using this setup, we evaluate the sensitivity of availability and reliability to the parameters specified in Table 10.

We present the results of our sensitivity analysis using a tornado plot. Each bar represents the variation in availability with changes to a single parameter while all other parameters are fixed at the baseline configurations. The length of the bar is proportional to the sensitivity of the measure to the particular parameter. The parameters are specified along the y-axis and the x-axis shows the change in the

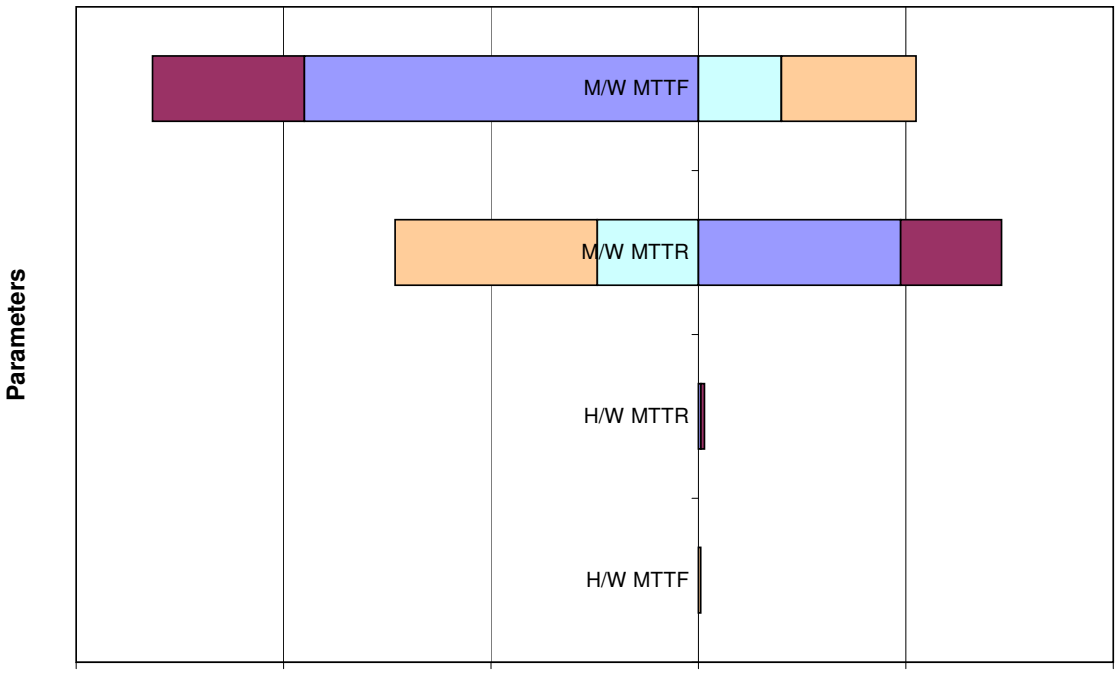


Figure 49: Sensitivity analysis of availability in a flat cluster.

measure (availability in this case).

Figure 48 depicts the sensitivity of system availability to component failure and repair rates in a hierarchical cluster. The plot for a flat cluster is shown in Figure 49 and is similar except for the absence of the bridge middleware. The graphs show that, availability is most sensitive to middleware repair and failure rates as compared to hardware. Also, note that the bridge middleware's failure and repair rates have minimal impact on the availability of a super-cluster. From this analysis we conclude that dependent failures of the middleware continue to be the limiting factor in availability in both architectures. Also introduction of independently failing components (bridge middleware) impacts availability minimally.

5.4.2 Baseline Reliability Analysis

Figure 50 presents the variation in system reliability with increasing cluster size using the setup described in Section 5.4.1. The graph shows that reliability of both sub-clusters and super-clusters in hierarchical architectures are comparable to that of

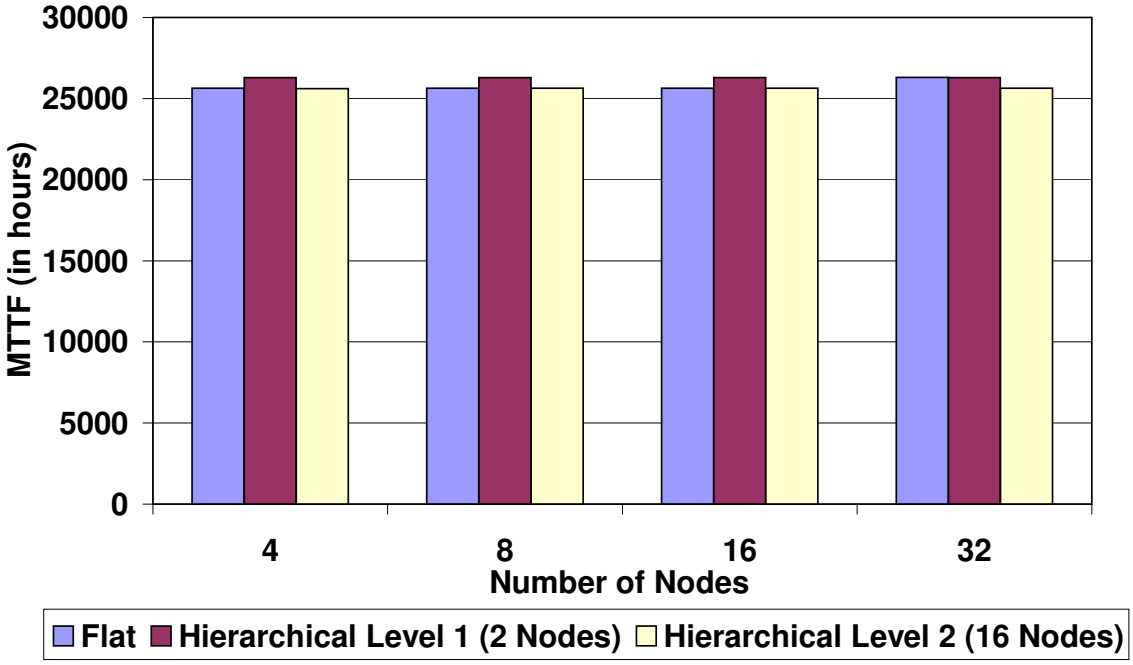


Figure 50: Baseline reliability.

the corresponding flat clusters despite inclusion of additional components (bridge middleware). However, note that sub-clusters in hierarchical architectures are units with independent failure modes unlike the flat architecture. The graph also shows that increasing the number of redundant hardware nodes does not improve reliability as long as the middleware remains a single-point-of-failure.

5.4.2.1 Sensitivity Analysis for Reliability

Figure 52 depicts the sensitivity of reliability in super-clusters using the same set-up as Section 5.4.1.1. The sensitivity of reliability in sub-clusters and flat clusters is depicted in Figure 51. The graphs show that reliability is most sensitive to middleware failure rates. For example, a decrease of MTTF_{mw} by 50% causes as much as a 50% drop in system MTTF. Again, note that bridge middleware failure rates hardly impact reliability when compared to middleware failure rates.

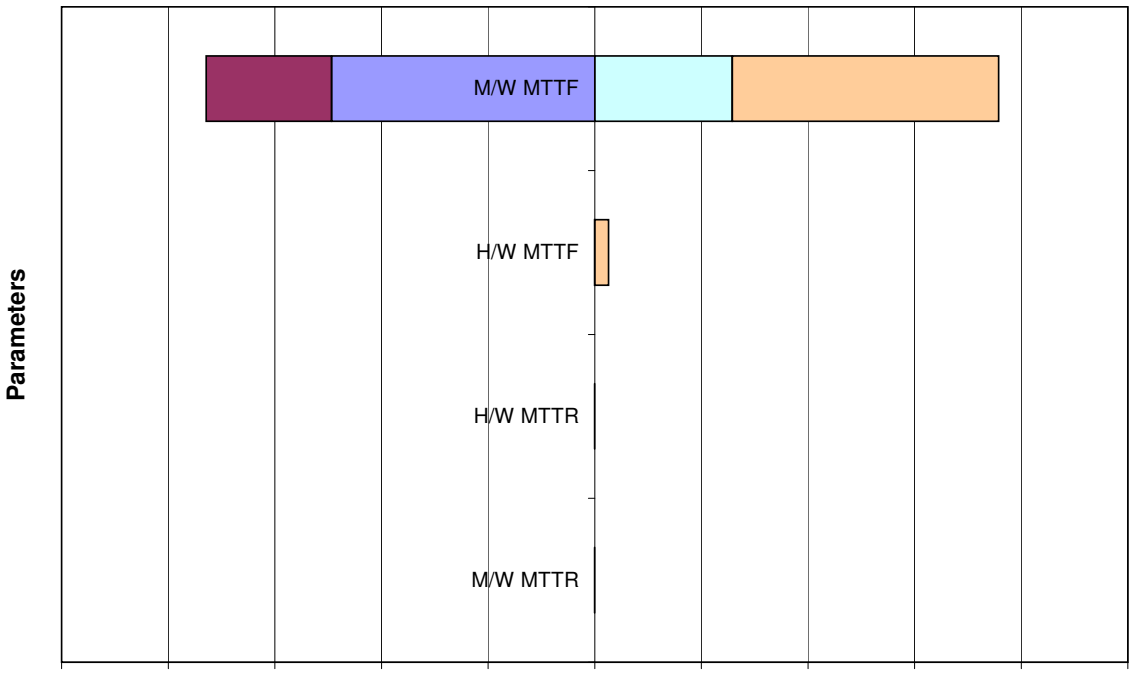


Figure 51: Sensitivity analysis of reliability in a flat cluster.

5.4.3 Modeling the Middleware Workload

Although the merits of defining application fault boundaries can be qualitatively understood, it is still desirable to quantify the availability and reliability of the resulting system. In this section, we use the middleware "workload-failure rate" relationship to characterize the benefits of defining fault-boundaries.

A general formulation of software reliability as a linearly increasing function of number of failures per encountered error (s), error density (d) and the workload (w) is provided in [149, 160]. Based on this formulation, we model the middleware failure rate as a linearly increasing function of middleware workload (number of events per unit time), assuming that the parameters s and d are constant for a single implementation. However, the relationship between software failure rates and the workload may vary depending upon the system, software and workload. Unfortunately, we do not have sufficient data to precisely establish this relationship.

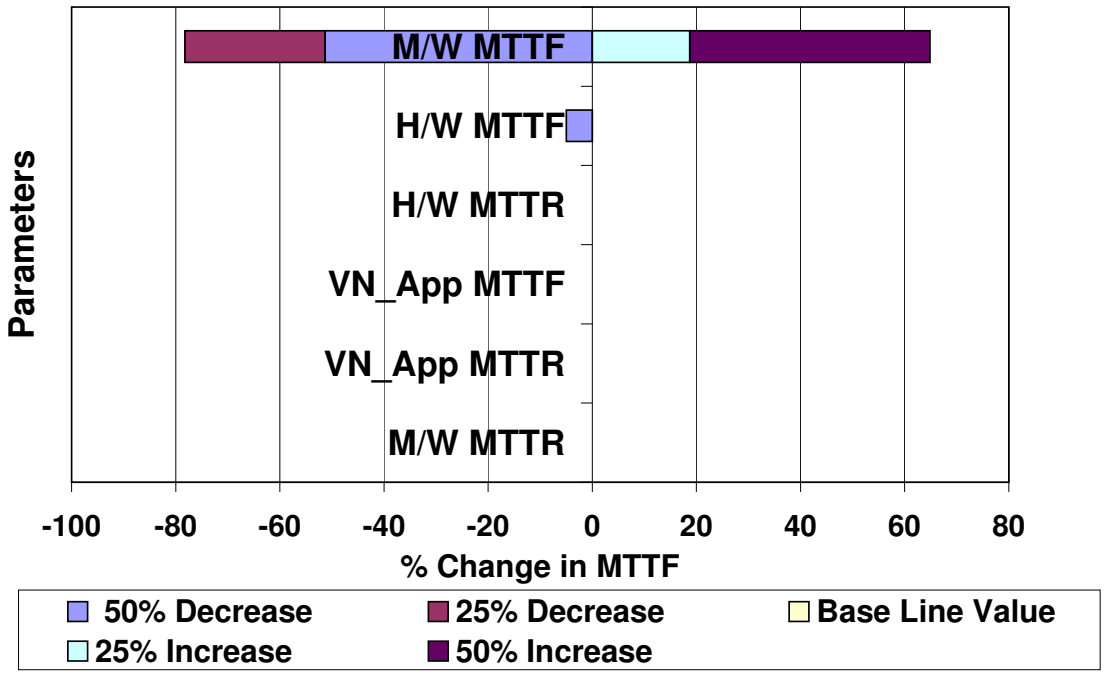


Figure 52: Sensitivity analysis of reliability.

We define the middleware workload under which the baseline failure and repair rates were observed as the standard workload (WL_{std}). We then introduce the notion of a normalized workload (NWL) which is the ratio of the actual observed workload to the standard workload. We consider the average case where workload is uniformly partitioned amongst the sub-clusters. The middleware workload in the super-cluster of virtual nodes consists primarily of global control and setup events. Based on observations in real systems, we expect this class of events to constitute around 5% of the total workload.

In this work we consider a sample of three different increasing functions of the middleware workload (WL): (a) linear, (b) sub-linear (square root) and (c) super-linear (exponential). The analysis can be extended to any function.

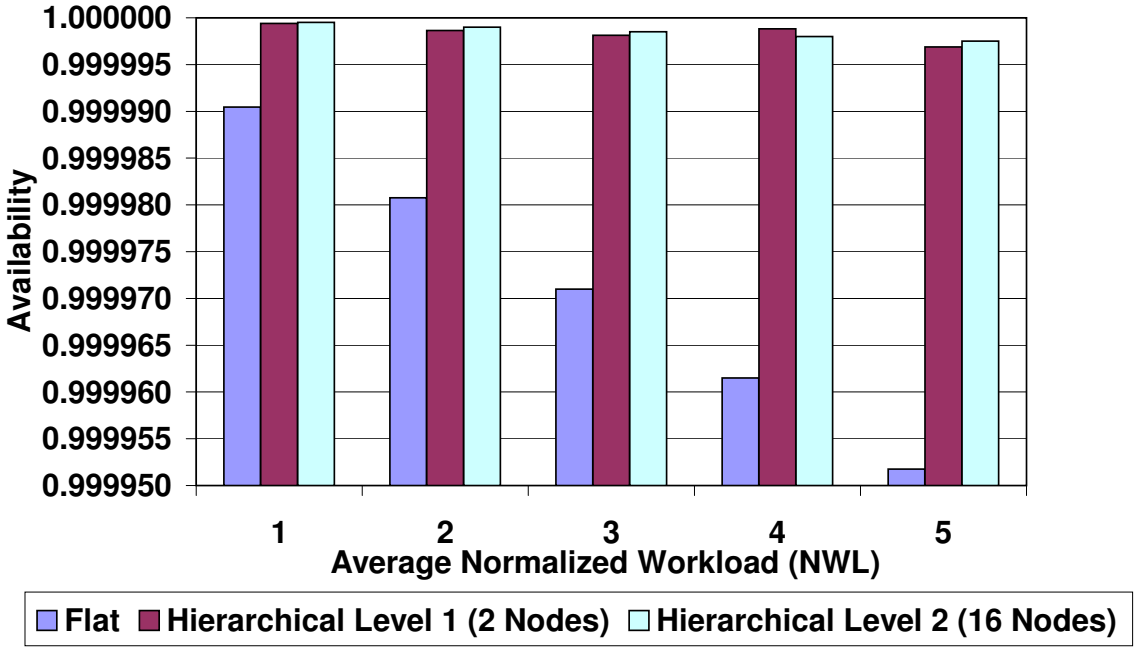


Figure 53: Availability with linear function.

5.4.3.1 Effect of Workload on Availability

Figure 53, 55 and 56 show the variation of availability with increasing workload when the middleware failure rate is a linear, square root and exponential function respectively of the middleware workload. In each of the figures, the x-axis represents the average normalized workload (number of events per unit time). The y-axis represents the fraction of time that an application deployed over the system is available. The bars represent the availability in a flat cluster with 32 nodes and a 2 level hierarchical configuration of the same nodes with sub-cluster size 2.

Linear : Assume $\lambda_{m/w}$ is a linearly increasing function of the middleware workload, given by:

$$\lambda_{m/w} = NWL \times \bar{\lambda}_{m/w} + constant \quad (1)$$

where $\bar{\lambda}_{m/w}$ is the middleware failure rate observed with the standard workload. (Since we assume that no failures are encountered in the absence of workload, the constant becomes zero.) The variation of availability with workload under this linear

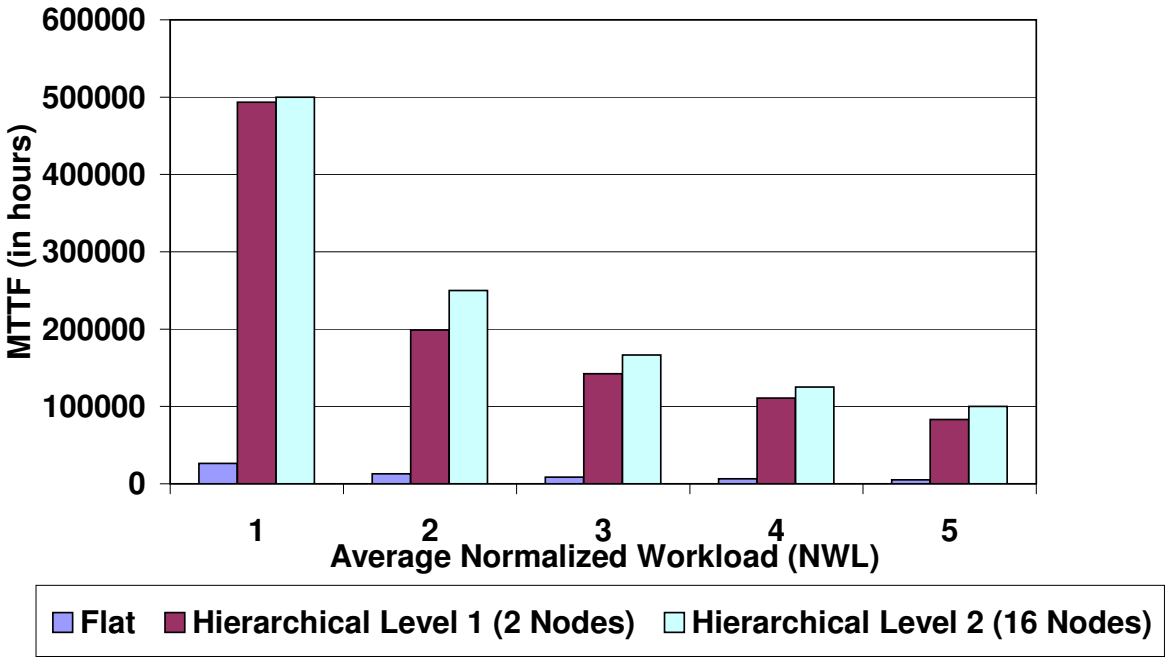


Figure 54: Reliability with linear function.

function is shown in Figure 53. It shows that the hierarchical configuration provides better availability at both the sub-cluster level and the super-cluster level. At both levels downtime is reduced by 94% on average, over the flat clusters. Expressed as downtime minutes, when the workload is 5 times the standard workload, the flat configuration translates to an additional 23 minutes downtime annually, compared to a sub-cluster in the hierarchical configuration.

Sub-linear : Figure 55 shows variation in availability when the middleware failure rate $\lambda_{m/w}$, varies as a sub-linear increasing function of the middleware workload given by:

$$\lambda_{m/w} = \sqrt{NWL} \times \bar{\lambda}_{m/w} \tag{2}$$

Again, the hierarchical configuration provides better availability at both the sub-cluster level and the super-cluster level. At the sub-cluster level downtime is reduced by 81% on average, over the flat clusters. At the super-cluster level, compared to the flat cluster, downtime is reduced by 76%. Expressed as downtime minutes, when

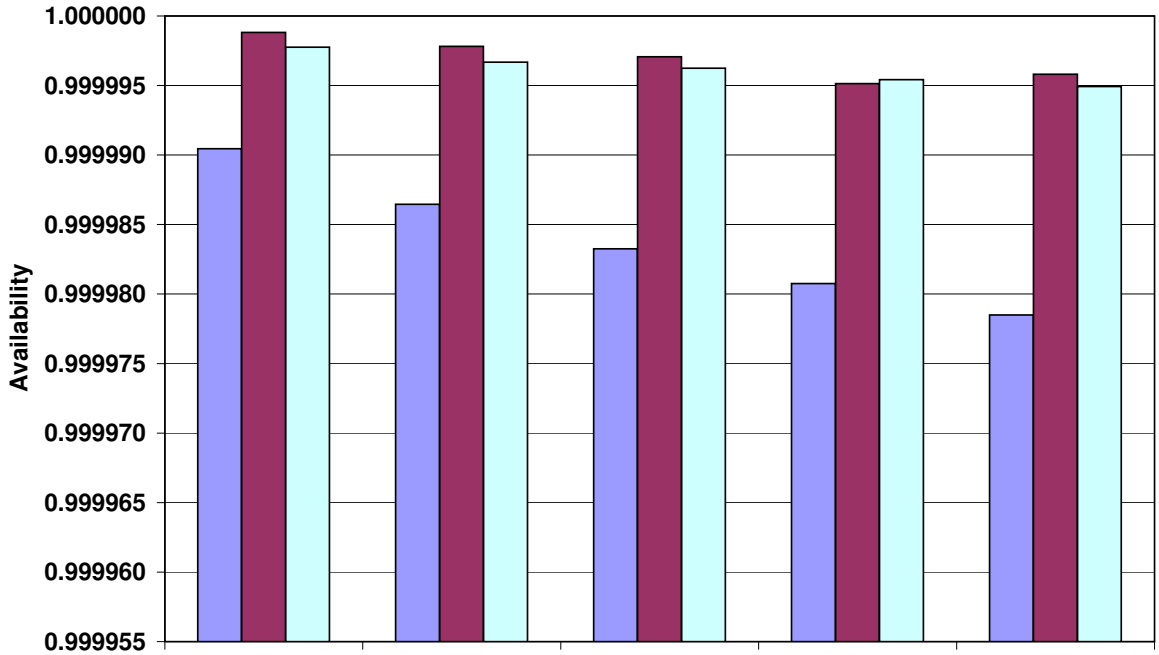


Figure 55: Availability with square-root function.

the workload is 5 times the standard workload, the flat configuration translates to an additional 9 minutes downtime per year, compared to a sub-cluster in the hierarchical configuration.

Super-linear : Figure 56 shows variation in availability when the middleware failure rate $\lambda_{m/w}$, varies as a super-linear increasing function of the middleware workload given by:

$$\lambda_{m/w} = \alpha \exp^{NWL} + c \tag{3}$$

where,

$$\alpha = \frac{\bar{\lambda}_{m/w}}{e-1}$$

and $c = -\alpha$, since failure rate is zero in the absence of workload. At both levels downtime is reduced by 98% on average, over flat clusters. This implies that, under 5 times the standard workload, the flat configuration translates to an additional 433 minutes downtime per year compared to a sub-cluster in the hierarchical configuration.

The above graphs give us an estimate of the expected improvement in availability

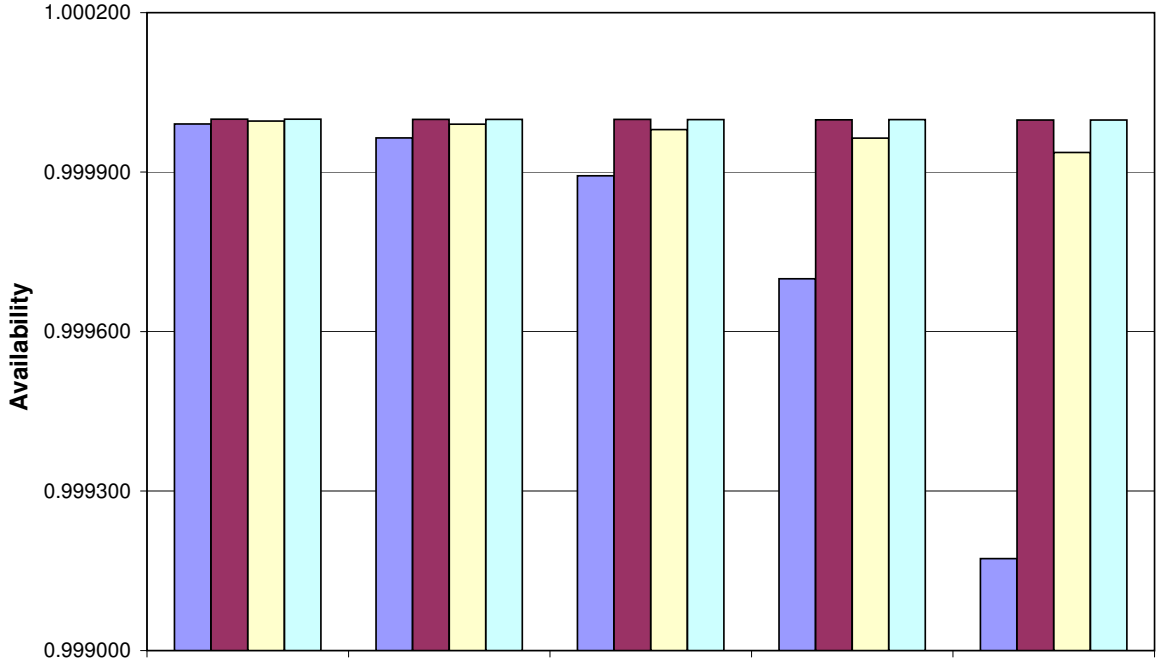


Figure 56: Availability with exponential function.

achieved by using a hierarchical configuration when the operational profile indicates a linear, sub-linear or super-linear dependence of middleware failure rates on middleware workload. The actual expected improvement in availability in a particular system can be estimated based on how the system's workload-failure rate function compares with these functions.

5.4.3.2 Effect of Workload on Reliability

Figure 54, 57 and 58 show the variation of reliability with increasing workload when the middleware failure rate is a linear, square root and exponential function respectively of the middleware workload. In each of the figures, the x-axis represents the average normalized workload (number of events). The y-axis represents the MTTF (in hours) of an application deployed over the system. The bars represent the availability in a flat cluster with 32 nodes and a 2 level hierarchical configuration of the same nodes with sub-cluster size 2.

Linear : Figure 54 represents the variation in reliability when the relationship

between the middleware failure rates and the workload is linear and follows Equation 1. As evident from the figure, all levels of the hierarchical configuration offer better MTTF than the flat architecture. Our results show that in a system with N nodes, compared to flat architectures, hierarchical architectures can improve the MTTF of the sub-cluster by nearly P times, where P is the size of the super-cluster. (In our 32 node system, the super-cluster size is 16, which means we get nearly 16 times improvement in MTTF). At the super-cluster level, the improvement in MTTF is approximately 20 times. Although this may lead us to conclude that setting P equal to N would result in the best reliability, note that this is not true since, under this condition, the limiting factor would be the hardware reliability. In general the guideline for choosing P is that the value should represent the point in system growth where the hardware ceases to be a limiting factor in reliability.

Sub-linear : Figure 57 represents the variation in reliability when the relationship between the middleware failure rates and the workload is sub-linear and follows Equation 2.

The graph shows that with the above relationship, a hierarchical configuration improves MTTF by nearly $\sqrt{P}$ times, where P is the size of the super-cluster. For example, at 5 times the standard workload, the MTTF of the flat cluster is 1.3 years (approx.) while that of the hierarchical cluster is nearly 5.4 years at the sub-cluster level and 5.7 years at the super-cluster level.

Super-linear : Figure 58 represents the variation in reliability when the relationship between the middleware failure rates and workload is super-linear and follows Equation 3. In the case of a super-linear relationship between middleware failure rates and middleware workload, the hierarchical architecture offers orders of magnitude improvement in MTTF over the flat cluster. For example, at 5 times the standard workload the MTTF of the flat cluster is 12 days (approx.) while that of the hierarchical cluster is nearly 14 years at the sub-cluster level and 16 years at the

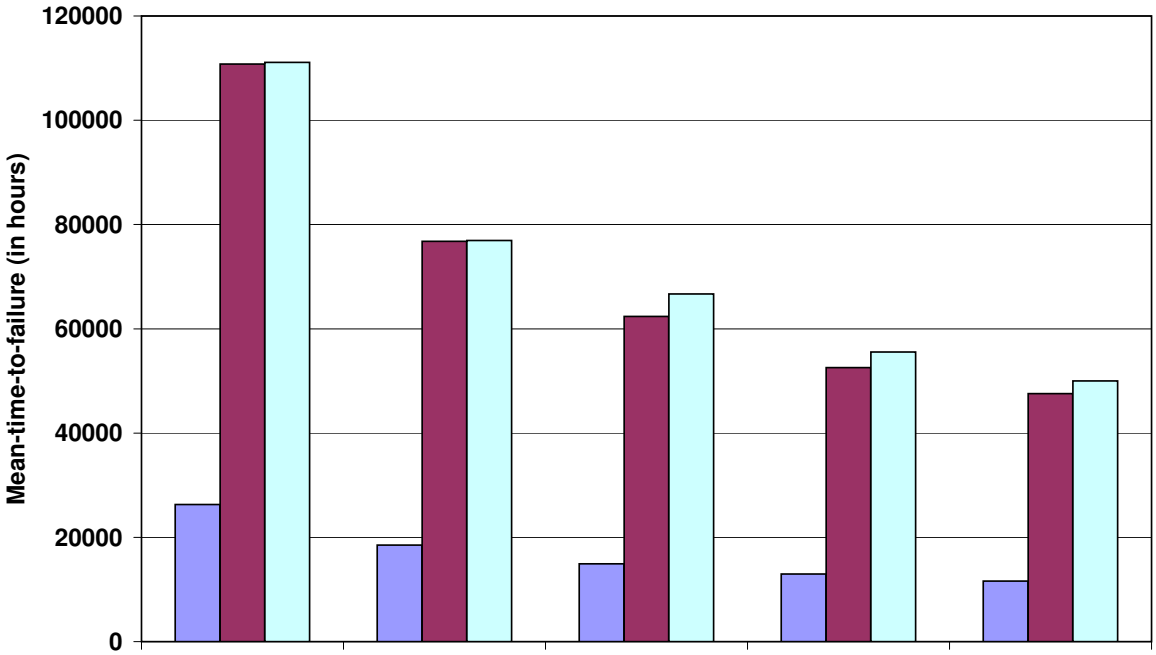


Figure 57: Reliability with square-root function.

super-cluster level.

5.4.3.3 Evaluation with other configurations

We repeated our experiments with other hierarchical set-ups, varying both system size and sub-cluster size. In general, given a N node system, varying the sub-cluster size resulted in a corresponding variation in partitioning of the workload. As shown in Section 5.4.1 and 5.4.2, beyond a certain point the hardware is no longer the limiting factor, availability and reliability vary according to the partitioning of workload and are determined by the super-cluster size, P. Our results from the sensitivity analysis of availability (Section 5.4.1.1) and reliability (Section 5.4.2.1) show that even at upper levels of the hierarchy, the middleware failure and repair rates are the dominating factor in availability and reliability. Likewise, organizing the nodes into hierarchies with more than two levels indicate that the availability and reliability are primarily indicative of only the workload (and hence the middleware failure rates) at those levels. Even under a 'workload-failure rate' independence assumption, the availability

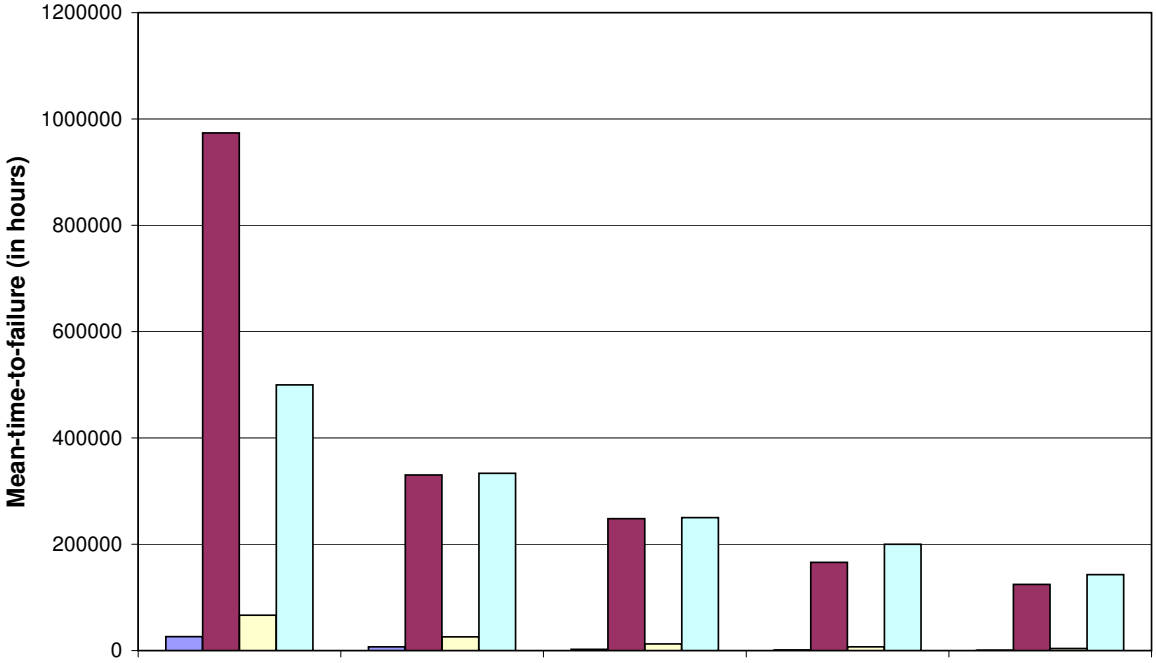


Figure 58: Reliability with exponential function.

and reliability at higher levels vary only marginally compared to sub-clusters.

A number of parameters can determine the selection of a particular hierarchical structure. Obviously, hardware and middleware failure and repair rates and specified availability and reliability targets are important factors for determining the appropriate sub-cluster sizes. For example, the availability (Figure 47) and reliability (Figure 50) of the system can help determine minimum sub-cluster sizes. In general the guideline for choosing sub-cluster sizes is that the size should represent the point in system growth where the hardware reliability ceases to be a limiting factor.

5.5 Comparison of Clustering Architectures

The hierarchical architecture trades-off system symmetry for application fault boundaries. As a consequence of this, while application-induced middleware faults are contained within sub-clusters, additional overhead in terms of global event communication and structural complexity are incurred. The disadvantage of decreased symmetry

is that although applications can be deployed anywhere within the hierarchical cluster initially, failover is possible only within the sub-cluster resulting in possible load skews. In order to deal with this, techniques for dynamic workload based migrations of application across sub-clusters are required. We leave such directions to future work.

While the virtual node selection made by the bridge middleware is susceptible to dependent failures, the inter-cluster communication segments are expected to have independent failure modes. In our experience, these primitives tend to be well tested and unlike application-induced events, are not a major-source of failures.

Flat clusters that utilize symmetric middleware architectures have limited scalability. The middleware maintains tight lease timers for better response times in detecting failed nodes. The lease-time must increase with the number of nodes in the system and cannot be lesser than the worst round trip time between any two nodes in the cluster. Thus, in flat clusters, failure response times worsen with geographic spread. Hierarchical clusters on the other hand are a natural way to combine remote nodes. For example, the local clusters can form one level of hierarchy and remote clusters may maintain consistent state at another level. The hierarchy allows the flexibility of using different lease timers at different levels and thereby improves response times for applications running over sub-clusters.

5.6 Related Work

A large body of existing research studies availability and reliability from the view point of storage arrays [107, 152] and controllers [118, 79]. Our work emphasizes the fact that in large scale-out systems, middleware (software) reliability is often the key determinant in system availability and reliability. Our approach proposes to provide graceful isolation of application-induced middleware failures, while retaining SSI, by organizing cluster storage services into a hierarchical overlay. Note that, our approach

is different from resilient overlay networks such as RON [35], which are designed to improve connectivity in Internet environments.

While fault isolation can be achieved through partitioning of a single cluster into smaller independent clusters [30, 26], without care, SSI and the flexibility to access storage from anywhere within the system will be lost in the process. Our goal is to provide fault-boundaries while continuing to provide these features. Although techniques like software rejuvenation [151] through rolling restart can reset software state, such techniques cannot deal with application-induced failures which would be encountered irrespective of the age of the software. Alternatively techniques like N-version programming [37] can be utilized to decouple middleware instance failures. However, this technique may be expensive and different implementations based on the same specification may still have common bugs [84].

5.7 Summary

This chapter has presented our investigation on the availability and reliability of a fault-tolerant middleware architecture that uses a pre-determined hierarchical structure. Determining application placements over the hierarchical structure in order to minimize cross-cluster traffic and issues relating to dynamic creation and maintenance of the hierarchical structure are topics of ongoing research.

CHAPTER VI

DATA AVAILABILITY THROUGH OPERATOR REUSE

6.1 Introduction

Modern enterprise applications [104, 2, 12], business and scientific collaborations across wide area networks [19], and large-scale distributed sensor networks [156, 97] are placing growing demands on high performance distributed data stream management services to provide capabilities beyond basic data transport such as wide area data storage [1] and basic in-network processing of continuous stream service requests [12]. Over the past decade, an increasing number of streaming applications are applying 'in-network' and 'in-flight' data manipulation to data streaming services designed for enterprise systems [20], financial management [11], scientific computing [48], and situation monitoring applications [66, 95]. One challenge of 'in-network' processing [29, 109, 156] is how to best utilize these geographically distributed resources to carry out end-user tasks while reducing the bandwidth usage and minimizing delay in service response time [60, 28], especially considering the dynamic and distributed nature of these applications and the variations in their underlying execution environments.

In this thesis we address this challenge by exploiting reuse opportunities in large scale distributed stream processing systems, focusing on the class of stream data manipulations represented as long-running continuous query services. It is observed that stream queries are typically processed by a selection of collaborative nodes and often share similar stream filters (such as stream selection or stream projection filters). The ability to reuse existing operators during query service deployment, especially for long running query services, is critical to the performance and scalability of a

distributed stream processing system.

We argue that by taking advantage of opportunities to reuse the same distributed operators for processing multiple and different concurrent query service requests and by intelligently consolidating operator computation across multiple query service deployment processes, we can reduce the total cost of query service deployment and minimize duplicate in-network processing. The technical challenges of reuse in streaming systems include dealing with large and time-varying workloads, and dynamically exploiting both the similarities between query service requests and the ability for runtime application of network knowledge. We believe that an effective reuse approach can provide high performance and high scalability for distributed streaming systems and intelligently capitalizing on both network locality awareness and operator similarity awareness are critical for making an effective reuse decision.

In this chapter we present the design and evaluation of a reuse-conscious distributed stream query processing system, called StreamReuse. We develop a suite of reuse-conscious stream query grouping techniques that dynamically find cost-effective reuse opportunities based on multiple performance factors, such as network locality, data rates, and operator lifetime. StreamReusescalably performs runtime query optimization by dynamically grouping queries based on reuse possibilities identified at runtime. We identify a three step process for generating reuse-conscious query service deployment plans. The first step is operator similarity based reuse. StreamReuse not only groups queries with the same operators but also provides capabilities to take into account containments and overlaps between queries in order to utilize reuse opportunities in queries that are partially similar, such as sharing some filtering operations though with different predicates. We develop a set of query relaxation techniques to identify operator similarity based reuse opportunities. The second step is reuse refinement by combining operator similarity and network locality to enhance the effectiveness of in-network reuse. We aim at locating and evaluating

different reuse opportunities possible at different network locations. A reuse lattice is devised to encode both operator similarity and network locality using a uniform data structure. Another feature of the reuse lattice is its ability to assist in fast identification of reuse opportunities from a large space of operators. With the reuse lattice we can efficiently generate an optimized query grouping plan that capitalizes on those 'relaxed operators' satisfying both operator similarity and network locality requirements. Our evaluation of reuse opportunities utilizes a cost model that assists us in making reuse decisions that are cost-effective. The third step is to effectively modify and migrate existing plans at runtime without compromising the correctness of query results. We develop techniques to perform 'relaxations' at runtime and to allow modifications and seamless migration of existing query services to new plans. Existing deployments continue to operate without pausing during runtime relaxation.

We conduct a detailed experimental evaluation of the StreamReuse approach with both simulations and a prototype. In particular, we examine two popular approaches used for optimizing distributed queries in the database community. The first approach is based on the construction of distributed query graphs. However, it is known that distributed query graphs cannot be statically analyzed [28, 55, 88, 78, 156] or optimized due to dynamic arrivals and departures of query services, and due to difficulty in obtaining accurate *a priori* knowledge of workload. The other popular approach is to devise a dynamic query optimization solution that considers re-planning of all query processing schedules in the system upon the arrival or departure of each individual query request. Obviously this approach suffers from inordinately large computational overheads [27]. Our experimental results confirm that the StreamReuse approach outperforms existing approaches under different workloads by reducing network and computational resource usage, and offers an order of magnitude improvement in the throughput of our stream query service processing system.

6.2 StreamReuse System Overview

Our techniques for dynamic system-level operator reuse are implemented in the StreamReuse sub-system within a distributed stream query processing system [90]. This section presents some motivating examples and an overview of the StreamReuse system architecture.

6.2.1 Motivating Examples

Exploiting operator level reuse for enhancing query service performance is important for a wide variety of systems and applications. Examples include query services that are long running and operating over a distributed collection of data centers. For example, most of the service requests in airline computer reservation systems or in enterprise operational information systems are long running and often perform pre-caching over distributed data repositories for business or scientific collaborations. The specific motivating example used in our research is derived from the airline industry based on our collaboration with Delta Air Lines [104].

An enterprise operational system (OIS) is a large scale distributed system that provides continuous support for a company or organization's daily operations. The OIS run by Delta Air Lines provides the company with up-to-date information about all of their flight operations including crews, passengers, weather and baggages. Delta's OIS combines three different types of functionality: continuous data capture for information such as flight and passenger status; continuous status updates to a range of end-systems including overhead displays, gate agent PCs and large enterprise databases; and responses to client requests which arrive in the form of queries. In order to respond to these query services, data streams from multiple sources need to be joined based on the flight, location or time attribute, using techniques such as a symmetric hash join.

Consider the following simple example. Assume that Delta's OIS is operating

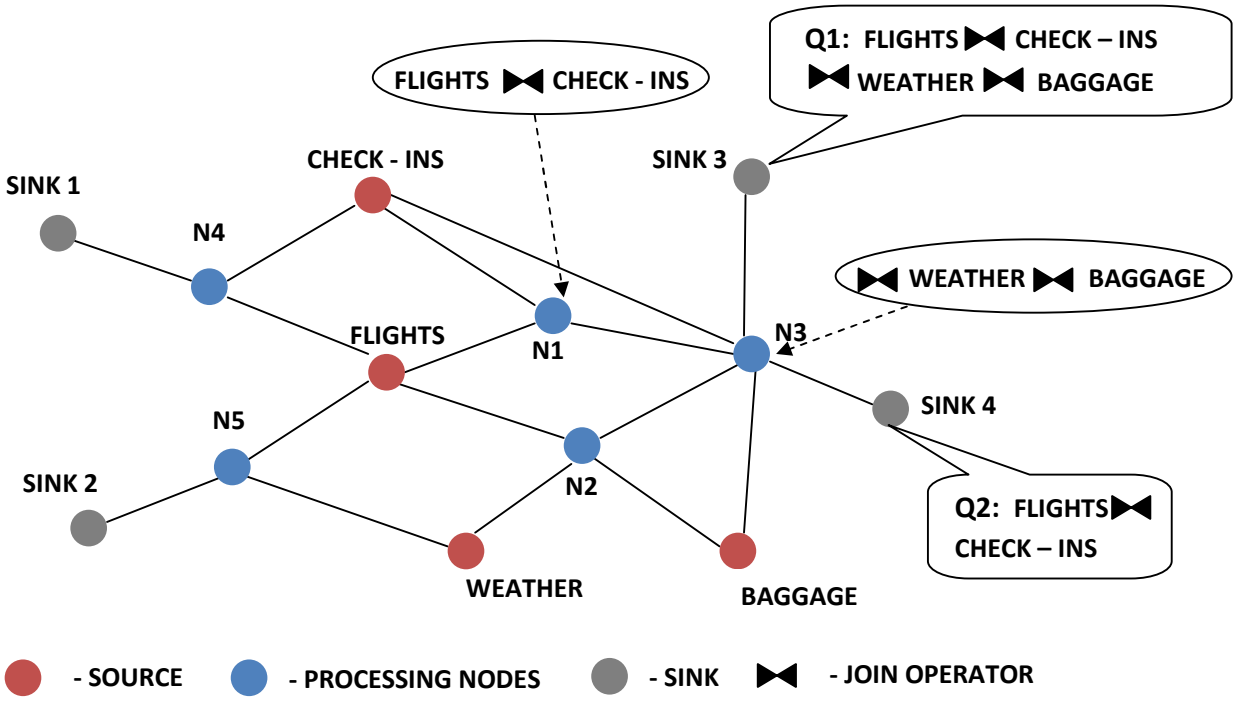


Figure 59: An example network N

over the small network N shown in Figure 59. Let WEATHER, FLIGHTS, CHECK-INS and BAGGAGE represent sources of data-streams of the same name and nodes N1-N5 be available for in-network processing. Each line in the diagram represents a physical network link. Also assume that we can estimate the expected data-rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data or measured by special purpose nodes deployed specifically to gather data statistics.

Assume that the following SQL-like query request Q1 needs to be streamed to a terminal overhead display SINK3 and the results will be updated every 1 minute.

```
Q1: SELECT FLIGHTS.NUM, FLIGHTS.GATE, BAGGAGE.AREA,
      CHECK-INS.STATUS, WEATHER.FORECAST
      FROM FLIGHTS [RANGE 5 MIN], WEATHER [RANGE 5 MIN], CHECK-INS [RANGE 1 MIN], BAGGAGE
      [RANGE 1 MIN]
      WHERE FLIGHTS.DEST = WEATHER.CITY
```

```
AND FLIGHTS.NUM = CHECK-INS.FLIGHT
AND FLIGHTS.NUM = BAGGAGE.FLIGHT
AND FLIGHTS.TERMINAL = ''TERMINAL A''
AND FLIGHTS.CARRIER_CODE = ''DL'';
```

One way to deploy the service request Q1 is to apply the filter conditions and the project operators for the various attributes at the source. The join operator FLIGHTS⋈CHECK-INS is placed at node N1 and join with WEATHER and BAGGAGE at N3. All join operators are evaluated every minute.

Assume that a new ad-hoc service request in form of query Q2 is posed by an airline manager in order to determine whether any low-capacity flights can be canceled and customers shifted to a partner airline's flight. Let us assume that the results need to be refreshed every 5 minutes.

Q2:
```
SELECT FLIGHTS.NUM, CHECK-INS.STATUS,
CHECK-INS.VACANT_SEATS
FROM FLIGHTS [RANGE 5 MIN], CHECK-INS [RANGE 5 MIN]
WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
AND FLIGHTS.CARRIER_CODE IN (''DL'',''CO'');
```

Clearly, Q1 and Q2 share a join filter operator "`FLIGHTS.NUM = CHECK-INS.FLIGHT`" at N1. Firstly, depending upon the sink for Q2, we may decide to either reuse the existing join operator at node N1 or redeploy a new join operator. For example, if Q2 arrives at SINK4 it may be beneficial to reuse the operator but if it arrives at SINK1 we may prefer to deploy a new join operator. Secondly, in order to be able to reuse the join operator FLIGHTS⋈CHECK-INS, we would have to completely remove some filter conditions (on attribute TERMINAL) before the join, relax some conditions (on attribute CARRIER_CODE) and place the original conditions after the join. Thirdly, this would imply that we would have to project some additional columns (attribute TERMINAL and VACANT_SEATS in this case). Also, we must now expand the window size for the CHECK-INS stream at the FLIGHTS⋈CHECK-INS operator to 5 minutes, but only forward

CHECK-INS data within a one minute window to query Q1. Additionally, we must filter updates based on timestamp such that results of query Q2 are streamed only every 5 minutes. Moreover, these reuse opportunities need to be identified and evaluated at run-time and should be implemented without disrupting existing queries, causing inconsistent information, or incurring excessive delays.

Several attributes of the example presented in this section are important to our research. First, when query services are not known *a priori*, reuse opportunities that exploit operator level similarity across multiple query services need to be identified at runtime. Second, the benefit from operator-level reuse depends on network locality. When the operator level similarity occurs at two or more network nodes that are far apart from one another, the potential communication delay in the network may take away the benefit of this reuse opportunity. Thus an effective reuse-conscious scheduling should take into account of both operator-level computation similarity and network locality. Finally, reusing an operator would often imply that existing operators and existing service deployments may need to be modified at runtime. Thus an efficient runtime migration of deployment plans is critical.

6.2.2 StreamReuse System Architecture

This section provides an architectural and implementation overview of the STREAM-REUSE system that has been implemented on top of IFLOW [90], a distributed data stream processing system, developed at Georgia Tech. A scalable virtual hierarchical network partition structure is used to collect and maintain information about the nodes and operators currently in use. This virtual hierarchy is composed of a network of nodes, grouped into regions based on the notion of network locality. In each region, one node is designated as the coordinator or **planner node** for the region. This coordinator node represents the region at the next level in the virtual hierarchical structure and collects and maintains network and operator information for all

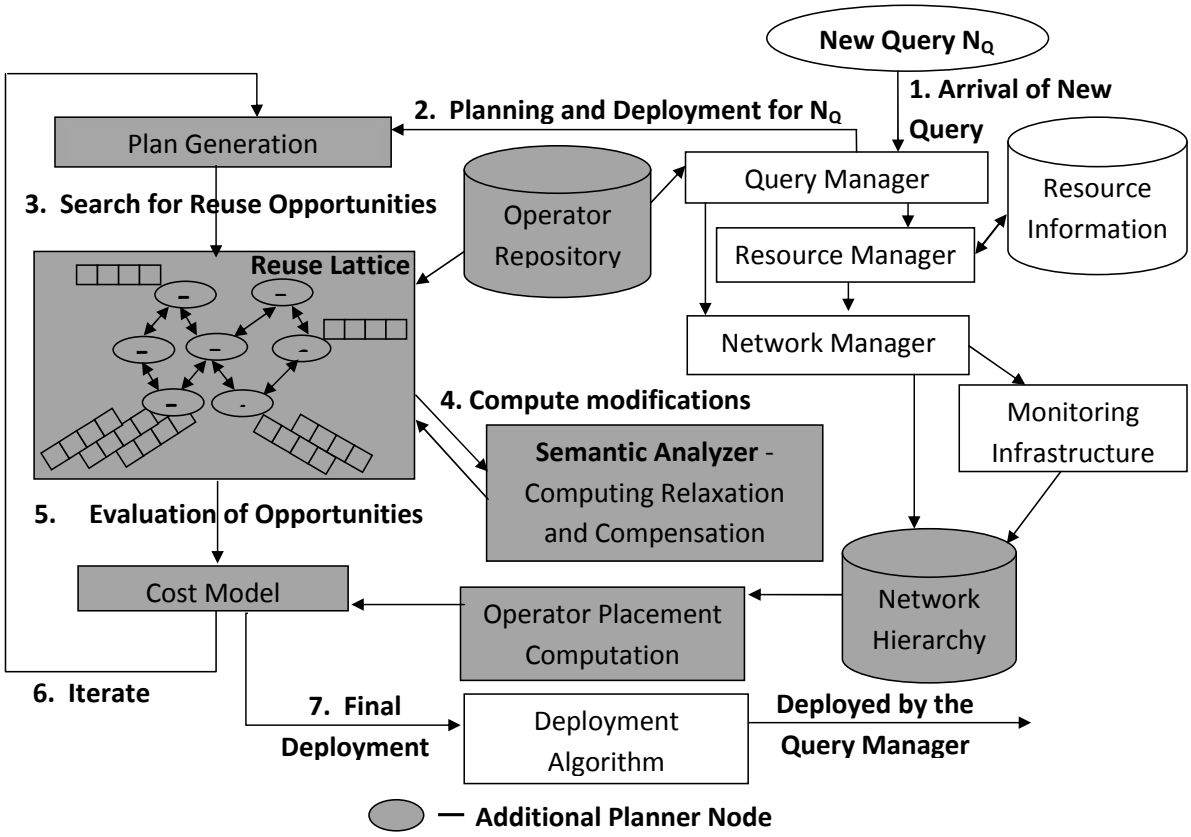


Figure 60: System Design

nodes and links within its region. This process of partitioning nodes into regions and coordinator selection continues until an acceptable approximation of the underlying network has been created. At each level, the coordinator consolidates operator and network information from all underlying nodes/regions. Readers who are interested in details on the hierarchy creation and maintenance may refer to [90].

Figure 60 shows a schematic representation of the StreamReuse system architecture. Each node in the system consists of three main components: (1) a query manager for maintaining information about stream queries and operators currently deployed at the node, (2) a resource manager for managing local processing resources and maintaining information about sources originating at that node, and (3) a network manager and associated monitoring infrastructure for managing and maintaining

information about network resources. The network information is collected by the network manager and maintained in the network hierarchy data structure at coordinator nodes, and is utilized for operator placement computation by the cost model.

Local planning and placement decisions of operators within a region are performed by the representative planner nodes. In summary, our deployment algorithm, the **Top-Down algorithm** described in [133], works as follows: the query arrives at the top-most level in the virtual hierarchy and the planner node at that level performs query planning and approximate region-level placement decisions of operators. The concrete operators determined at this level are passed on to coordinators of respective regions for placement within their region. Operators are thus passed down to coordinators at the next lower level in the hierarchy until they are mapped to actual physical nodes. The planning process is triggered when a new query service request arrives in the system or when network topology changes, such as new nodes join or existing nodes fail or depart.

The key contribution of this work is the reuse-conscious query planning process at the coordinator nodes, henceforth referred to as the **planner node**. Figure 60 shows the architecture of the planner node with the additional components indicated by shading in the diagram. In addition to network, resource and query manager components, the planner node also maintains (1) a plan generator, (2) a reuse lattice that organizes the information in the operator repository into an efficient search-optimized structure, (3) a semantic analyzer, (4) a cost model, (5) network information organized based on the virtual hierarchical structure, and (6) a component to compute operator placement. Of these components, the reuse lattice, the semantic analyzer, and the cost model provide the functionality required for identifying and evaluating reuse opportunities. The plan generator is similar to those found in traditional database engines for computing combinatorial operator ordering possibilities. The

network information component and the operator placement and deployment algorithms have been summarized in the previous paragraphs and have been discussed in detail in [133].

In this work, we concentrate on aspects of the system that are specifically designed to promote efficient identification and utilization of reuse opportunities in a system where multiple similar distributed continual queries are running simultaneously. In particular:

- **Semantic Analyzer:** utilizes operator semantics to identify existing operators that can be reused in the computation of the new query request. For each such identified opportunity, the semantic analyzer computes 'relaxations', i.e. modifications to the existing operators, in order to allow the operator to be reused and 'compensations', i.e. new operators that need to be introduced at runtime to ensure that existing query computations are consistent and are not disrupted. (Section 6.3)
- **Reuse Lattice:** congregates information from the operator repository and the network hierarchy into a single structure that allows efficient search and identification of reuse opportunities based on both operator similarity and network locality. (Section 6.4)
- **Cost Model:** combines network costs from the operator placement computation model based on network information and data-flow rates from semantic information to compute a cost-measure for each candidate reuse opportunity/deployment. Using this information along with query lifetimes, the planner node chooses efficient deployments for queries while taking into account reuse opportunities that increase the overall efficiency of the system. (Section 6.5.1)

6.2.3 Basic Concepts and Notations

We now briefly discuss some background concepts and notations used throughout the chapter. Throughout the discussion, continuous queries are specified using SQL semantics. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ represent the set of sources in a distributed stream processing system G. Let $\Psi(t)$ represent the set of queries, each associated with a lifetime, executing over the system at time t. The query workload results in a set of operators being deployed across the system. A single operator may be shared by multiple queries. Let $\Theta(t) = \{\theta_1, \theta_2, \ldots, \theta_m\}$ represent the set of operators (including sources) executing in the system at time t. Each operator is the source for the stream computed by its underlying query.

An operator θ in the distributed stream processing system is uniquely identified by the following:

- $\mathcal{V}$, the definition of an operator which declaratively represents its output stream by an equivalent query,
- $\mathcal{N}$, the network location which is the identity of the physical node over which the operator is executing, and
- $\mathcal{T}$, the lifetime which is the duration of time for which the operator persists in the system.

The operator definition is in the form a single level 'select-project-join-group by' (SPJG) query with a FROM clause consisting only of base streams sources (i.e, not other operators or derived streams). The definition includes window (RANGE and SLIDE i.e. the window size and frequency of computation of results respectively) specifications for each of the input streams. The lifetime of an operator is the maximum over the lifetimes of all queries that share that operator. Note that two physical operators with identical definitions may appear at different network locations with different lifetimes.

The next section characterizes the semantic criteria for reuse candidate selection

and describes how the semantic analyzer computes modifications to existing operators in order to reuse them. The reuse lattice data structure, our cost model and the actual process of implementing operator reuse and plan migration at runtime are presented in subsequent sections.

6.3 Identifying Reuse Candidates

The aim is to identify two kinds of reuse opportunities:

- Containment or Exact matches: we identify reuse possibilities where existing operators and their results can readily be used to evaluate a new query.
- Overlap: we explore opportunities where even existing operators that cannot be directly reused to compute a query can be modified in order to induce reuse.

Section 6.3.1 identifies the basic conditions that must be satisfied by any operator θ_i in order to be reused in the evaluation of query Q. The main operator relaxation steps that identify the operators for modifications and compute relaxations and additional compensations are detailed in Section 6.3.2.

6.3.1 Base Conditions

In order to rewrite a newly arriving query Q in terms of an existing operator θ_i (with view definition $\mathcal{V}_i$), the following conditions referred to as '**base conditions**' should be satisfied.

1. Source table condition : Let S^Q be the set of source streams referenced by the stream query. Let S^θ represent the set of source streams referenced by $\mathcal{V}_i$. Then $S^Q = S^\theta$ must be satisfied. Here '=' represents set equivalence. If $\mathcal{V}_i$ refers to additional source streams, it can still be used to rewrite Q provided the joins with the additional source streams are cardinality preserving. However, we may not be able to guarantee this when stream characteristics and rates are constantly varying. Thus we do not consider operators with additional source streams for rewriting.

2. Join predicate conditions : Let J^Q be the set of predicates that represent the join operations in the where clause of Q. Similarly, let J^θ represent the join predicates in $\mathcal{V}_i$. Then, set equivalence $J^Q = J^\theta$ must be satisfied in order to be able to rewrite the query in terms of the operator.

3. Two-column predicates : A two-column predicate is of the form $S_k.C_a$ OP $S_n.C_b$ where OP is one of $\{<, \leq, >, \geq, \neq\}$, S_k,S_n represent stream sources and C_a,C_b represent attributes. Let T^Q be the set of two-column predicates in the query. Let T^θ be the set of two-column predicates specified in $\mathcal{V}_i$. Then, set containment $T^\theta \subseteq T^Q$ must hold; i.e., every two-column predicate in the operator θ must also appear in the query Q.

4. Grouping column condition : Let G^Q be the group-by columns in the stream query. Let G^θ be the group-by columns in $\mathcal{V}_i$. Then $G^Q \subseteq G^\theta$ must hold i.e., every group-by column in the query must also appear in the operator θ. If the group-by condition in the operator includes additional columns, then the query can be computed by performing further aggregations on the operator results using techniques such as those described in [78].

If the above conditions are satisfied, then the operator θ_i can be used for rewriting the query Q. The conditions are adapted from the context of rewriting queries in terms of materialized views. While rewriting queries in terms of materialized views, certain additional conditions such as selection predicates and project columns are also considered since materialized views cannot be modified at runtime. However, in our case, it is possible to modify existing operators. Moreover, stream queries have window specifications which must also be taken into consideration during rewriting. The following discussion outlines the semantic considerations for identifying runtime reuse opportunities of existing operators.

6.3.2 Relaxation and Compensation

By exploiting query semantics, it is possible to modify existing operators to facilitate sharing of operators between multiple queries. According to our definition of relaxation, the query and original operator must be obtainable from the relaxed operator by imposing just selection, projection and temporal filter operators. We restrict relaxation in this way since we must ensure that the existing operator can be relaxed at runtime without disrupting queries that may already be executing over the original deployed operator. Hence our focus is primarily on relaxing join operators.

We consider four kinds of relaxations of existing operators in the system: (1) relaxation of selection predicates, (2) relaxation of project operators (3) relaxation of operator lifetime, and (4) relaxation of window specification. Depending upon the query and the existing operator, one or more of these relaxations may be applied. During relaxation, an existing operator is modified into a 'relaxed' operator and compensation operators. Compensation operators are introduced to ensure the consistency of results of existing queries and are also used to rewrite the new and existing queries in terms of the relaxed operator.

While relaxing an operator, the condition being relaxed may be local, meaning that it is applied by the current operator, or it may already be embedded into the input stream by some operator at an upstream node. If the condition is local, then relaxation only involves modifying the current operator. If the condition being relaxed occurs at some upstream operator, the relaxation operation will cascade and result in relaxations of multiple upstream operators. The actual implementation of the migration from the existing plan to the new plan that uses the relaxed operator is described in Section 6.5. We first introduce some preliminary definitions and the notion of minimal relaxation.

Since stream data is potentially unbounded, queries typically have a window specification. In CQL [36] window size is specified using a RANGE clause r and the frequency

of computation of the operator is specified with an optional SLIDE clause s. For ease of exposition, we describe the process of relaxing windows for windows where a single range and slide specification applies to all input streams in the operator. However, note that, our techniques are applicable to any general window specification. First, we define temporal operators for the operator θ_i and query Q: r^{θ}, r^{Q}, referred to as range filters filter tuples based on RANGE specifications and operators s^{Q}, s^{θ}, referred to as slide filters, filter tuples based on SLIDE specifications.

Operator Containment: An operator (or query) θ_k is said to be contained in another operator θ_i, denoted by $\theta_k \sqsubseteq \theta_i$, iff for every tuple t_k produced by θ_k $\exists$ a unique tuple t_i produced by θ_i such that $t_k \sqsubseteq t_i$. (A tuple t_k is said to be contained within another tuple t_i iff every attribute in t_k also appears in t_i.)

Containment relationship with windows : Given operators θ_i and θ_j with identical SPJG definitions and (RANGE, SLIDE) window specifications $(r_i,\ s_i)$ and (r_j, s_j) respectively, the containment relation $\sqsubseteq$ imposes a partial ordering on the set of operators and is defined as follows: $\theta_i \sqsubseteq \theta_j$ iff $r_i < r_j$ and $s_i \ mod \ s_j \equiv 0$.

Relaxation: Let θ represent an already deployed operator and Q represent a new query. Then, $\overline{\theta}$ is called a 'relaxation' of θ under Q, if

1. $\theta \sqsubseteq \overline{\theta}$ and
2. $Q \sqsubseteq \overline{\theta}$ and
3. $\exists \sigma^{\theta}, \Pi^{\theta}, r^{\theta}, s^{\theta}$ such that $r^{\theta}(s^{\theta}(\Pi^{\theta}(\sigma^{\theta}(\overline{\theta})))) \equiv \theta$ and
4. $\exists \sigma^{Q}, \Pi^{Q}, r^{Q}, s^{Q}$ such that $r^{Q}(s^{Q}(\Pi^{Q}(\sigma^{Q}(\overline{\theta})))) \equiv Q$.

Compensation Operators : Selection operators $\sigma^{\theta}, \sigma^{Q}$, projection operators Π^{θ}, Π^{Q} and temporal filter operators $r^{\theta}, r^{Q}, s^{Q}, s^{\theta}$ are referred to as compensation operators and $\overline{\theta}$ as a 'relaxed operator'.

Minimal relaxation: A relaxation $\overline{\theta}$ of θ under Q is called a minimal relaxation if $\forall$ relaxations θ_i of θ under Q, $\overline{\theta} \sqsubseteq \theta_i$.

Lemma 6.3.1. *Given query Q and operator θ, if $Q \sqsubseteq \theta$, then θ is the only minimally relaxed operator.*

Proof. From the definition of relaxation and operator containment, we can easily see that θ represents a relaxation of itself under Q. Since all other relaxations must contain θ, by definition, θ is a minimal relaxation under Q. □

Given that an operator θ satisfies the base conditions with a query Q, a minimal relaxation of the operator is computed using the following steps.

1. Relaxing selection predicates : An operator can be relaxed by modifying the selection predicates and there by imposing a less restrictive filter condition. We relax the operator θ such that the new relaxed operator $\overline{\theta}$ is a minimal cover of θ and Q. We compute a minimally relaxed operator from θ by analyzing the predicates of both θ and Q. $\overline{\theta}$ retains predicates that are common to both Q and θ. For predicates that involve different conditions over the same attribute, $\overline{\theta}$ includes a predicate that is a disjunction of the two predicates. In some cases this disjunction can be further simplified. For example, if Q contains a predicate of the form $S.a > constant1$ and θ contains a predicate of the form $S.a > constant2$, then $\overline{\theta}$ will use predicate $S.a > min(constant1, constant2)$. Similarly, if Q and θ involve complementary predicates on the same attribute, then the resulting disjunction can be removed from the set of predicates of $\overline{\theta}$ without loss of minimality. Other predicates are omitted from $\overline{\theta}$.

Since selection operators are idempotent, a simple way to compose compensation operators σ^Q (or σ^θ) is to include all predicates that appear in Q (or θ). Although correct, this may lead to redundant condition checking. Therefore, we compose compensation operators σ^Q (or σ^θ) by including only those predicates that have not been included in the relaxed operator.

2. Relaxing projections : Q may require projection of additional columns than those projected by θ. Moreover, columns that are not part of the output column list may be required by the compensation selection operators σ^Q and σ^θ. $\overline{\theta}$ is modified to project, in addition to columns projected by θ, extra columns required by Q, σ^Q and σ^θ. Additionally, if group-by conditions are specified, then the group-by columns required by Q and θ also need to be projected. The compensation projection operators Π^Q and Π^θ are simply those columns in the output list of Q and θ respectively.

3. Relaxing operator lifetimes : The relaxed operator must persist in the system for as long as any of the queries using it still execute in the system. Thus the lifetime of the relaxed operator is set to the maximum of the lifetimes of all the queries using it. The lifetime of compensation operators σ^θ and Π^θ is set to that of the original operator θ. Lifetimes of σ^Q and Π^Q is set to that of Q.

4. Relaxing windows : Our relaxation techniques are primarily aimed at sliding-window join operators. Note that, sharing of a single blocking (aggregation) operator between queries with different range or slide specifications is not feasible. This is because no single input data timestamp can be associated with the output, rendering temporal filtering impossible. However, it is still possible to resort to node-level reuse and share computations between multiple such operators placed on the same node [89].

The range specification, i.e. the window size, for the relaxed operator $\overline{\theta}$ is the larger of r^θ and r^Q. Relaxation of slide specifications are slightly more involved. Let s^θ and s^Q represent the slide specification of the operator θ and the query Q respectively. A naive solution would be set the slide to the GCD(s^θ, s^Q). Thus, if $s^\theta \ mod \ s^Q \equiv 0$ then $s^{\overline{\theta}} = s^Q$; if $s^Q \ mod \ s^\theta \equiv 0$ then $s^{\overline{\theta}} = s^\theta$ and $s^{\overline{\theta}} = 1$ otherwise. However, this could potentially lead to a large amount of unnecessary computations. For example, if s^θ and s^Q are two different large primes, then for most instants

of time, computation of operator $\overline{\theta}$ would be wasted. Instead we specify window slide frequencies as a boolean condition which when evaluated to true results in a window slide. For example, the operator performs its computation at time instant t if the following condition evaluates to true: $((t\ mod\ s^{\theta} \equiv 0) \vee (t\ mod\ s^{Q} \equiv 0))$. In general, systems which support adaptivity have the ability to modify filter and window specifications at runtime [28, 61]. We are able to leverage the dynamic code generation capabilities of our system [61] to modify windows at runtime.

We introduce compensation range filters r^{θ} and r^{Q} that filter tuples from the relaxed operator based on the timestamps associated with the tuples and the range specifications of θ and Q respectively. For example, if θ specifies a range of 10 minutes and Q specifies a range of 8 minutes, then $\overline{\theta}$ is computed over a window of 10 minutes. The compensation operator r^{Q} then filters tuples whose data items have timestamps that fall beyond the 8 minute window. Similarly compensation slide filters s^{θ} and s^{Q} filter tuples from the relaxed operator based on the timestamp of the output from $\overline{\theta}$. For example, if θ specifies a slide of 2 minutes and Q specifies a slide of 1 minute, then $\overline{\theta}$ is computed at a slide of 1 minute. Compensation slide filter s^{θ} only forwards results which appear at 2 minute intervals. Note that in order to perform compensation, we must project per tuple timestamps for each data item participating in the join in addition to the items already being projected.

Sharing an operator between two queries with different ranges and slides can result in a significant increase in the size of intermediate data due to an increase in both window size and frequency of slide. However, this is taken into consideration by our cost function since we include operator output rate while computing network usage per unit time.

5. Cascading relaxations : If relaxing an operator involves the relaxation of conditions that are not local (i.e. not at the current operator itself) but instead are embedded into the input streams by some upstream operator, then we may need

to perform cascading relaxations (and the associated compensations). For each upstream operator that requires relaxation, the process is repeated and relaxed and compensation operators are computed. Cascading terminates when all the conditions being relaxed are local.

6.3.3 Example

The following example explains the relaxation and compensation process for the queries $Q1$ and $Q2$ described in Section 6.2.1. We explain how the FLIGHTS⋈CHECK-INS join operator θ_j deployed for query $Q1$ should be relaxed to be reused in the evaluation of query $Q2$. Originally,

θ_j :SELECT FLIGHTS.NUM, FLIGHTS.GATE, FLIGHTS.DEST, CHECK-INS.STATUS FROM FLIGHTS [RANGE
5 MIN], CHECK-INS [RANGE 1 MIN]
WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
AND FLIGHTS.CARRIER_CODE = ‘‘DL’’
AND FLIGHTS.TERMINAL = ‘‘TERMINAL A’’;

Since the base conditions are satisfied by θ_j under $Q2$, the operator can be reused with the query. However, relaxation is required. We briefly outline the steps to compute the minimally relaxed operator $\overline{\theta_j}$ next.

1. **Relaxing selection predicates :** The selection conditions C of $\overline{\theta_j}$ is given by:

C : FLIGHTS.NUM = CHECK-INS.FLIGHT
AND FLIGHTS.CARRIER_CODE IN (‘‘DL’’,‘‘CO’’)

In this case, compensation selection operators are required only for the existing query, and the conditions C^{θ} in the compensation operator σ^{θ} is given by:

C^{θ} : FLIGHTS.TERMINAL = ‘‘TERMINAL A’’
AND FLIGHTS.CARRIER_CODE = ‘‘DL’’

2. **Relaxing project operators :** The project column list L in $\overline{\theta_j}$ is given by:

L : FLIGHTS.NUM, FLIGHTS.GATE, FLIGHTS.DEST,
FLIGHTS.TERMINAL,FLIGHTS.CARRIER_CODE,
CHECK-INS.STATUS, CHECK-INS.VACANT_SEATS

The compensation projection operators are simply those columns specified by θ_j and $Q2$.

3. **Relaxing lifetimes :** The lifetime of the new operator $\overline{\theta_j}$ will be the maximum of the lifetimes of $Q2$ and $Q1$.

4. **Relaxing windows :** The window range for the CHECK-INS stream in $\overline{\theta_j}$ is set to 5 minutes (maximum of range for Q1 (i.e. 1 minute) and Q2 (i.e. 5 minute)). Similarly, the slide is specified by the following boolean condition: $((t\ mod\ 1 \equiv 0) \vee (t\ mod\ 5 \equiv 0))$, which is simplified to just $(t\ mod\ 1 \equiv 0)$. Range compensation operator τ^{θ} filters out tuples whose data items corresponding to the CHECK-INS stream fall beyond a 1 minute window. Similarly, slide compensation operator Γ^{Q2} only forwards results that appear at a 5 minute interval.

5. **Cascading relaxations :** Since the selection conditions on the FLIGHTS source are actually performed at the source, the conditions in the selection operator at the source will have to be replaced with C. The same applies to the project operators at the sources FLIGHTS and CHECK-INS. Thus, the definition of the minimally relaxed operator $\overline{\theta_j}$ is:

```
θ̄j : SELECT FLIGHTS.NUM, FLIGHTS.GATE, FLIGHTS.DEST,
     FLIGHTS.TERMINAL,FLIGHTS.CARRIER_CODE,
     CHECK-INS.STATUS, CHECK-INS.VACANT_SEATS
     FROM FLIGHTS FL, CHECK-INS CI
     WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
     AND FLIGHTS.CARRIER_CODE IN (‘‘DL’’,‘‘CO’’);
```

6.4 Searching for Reuse Candidates using Reuse Lattice

The ‘Reuse Lattice’ data structure combines information from the operator repository and the network hierarchy into a single structure that allows efficient search and identification of reuse opportunities based on both operator similarity and network locality. This section answers the following question: how do we efficiently search for

reuse opportunities?

The reuse lattice uses the operator definition $\mathcal{V}_i$ to encode containment. Since operator definitions are similar to view definitions in traditional databases, this allows us to leverage the large body of existing work in rewriting queries using materialized views. Particularly, we adapt the filter tree index structure described in [71] to efficiently maintain containment relationships between operators. Section 6.4.1 describes the adaptation of the structure to the context of continuous queries to create the reuse lattice that handles window specifications and relaxations. Section 6.4.2 describes techniques that extend the structure to incorporate network locality. The reuse lattice supports the following search operations: (1) **Network locality :** Search for candidate operators only within specified regions, (2) **Reuse without modification :** Search for operators that can be reused without modifications, and (3)**Reuse with modification :** Searches for operators that can be reused with modifications. An exhaustive search would search all network regions for all operators including relaxation opportunities.

6.4.1 Encoding Operator Containment

The reuse lattice adapts a restricted filter tree structure [71] to the context of a distributed stream processing system. Given a query, the filter tree structure can be used to quickly narrow down the list of candidate operators in the system that will give rise to valid rewrites. The filter tree is a multiway search tree where all leaves are at the same level. A single node in this structure represents a collection of operators. Different partitioning conditions are applied to the nodes at each level to further partition the operators into multiple smaller disjoint nodes at the next level. For example, at the top-most level, operators are partitioned into disjoint subsets based on source streams (specified in the FROM clause of the operator definition). Each disjoint subset is represented at that level by a single node in the filter tree. A

different partitioning condition is applied at each subsequent level. For example, we partition nodes into disjoint subsets based on join predicates at the next level, based on two column predicates at the third level and group-by predicates at the fourth level. At this point, all the base conditions have been accounted for. We further partition each node based on each of the relaxable conditions, viz. selection predicates, project column list and window specifications. These last three levels in the filter tree are only used when searching for reuse opportunities that do not require modifications. The key at each level is determined by the partitioning condition. For example, if the partitioning condition is the set of source streams, then the list of sources specified in the FROM clause of the definitions serves as the key. If the partitioning condition is the window specification, then the (RANGE, SLIDE) specifications serve as the key. Each node in the filter tree is a collection of <key,pointer> pairs that may further be organized into an internal index structure based on containment of keys (determined by the partitioning condition) to further speed-up search within a node. For example, Figure 61 shows a single lattice node at the window specification level, where the keys (which is (RANGE, SLIDE) specifications in this case) are organized into an internal index structure based on containment. For example, the Figure 61 shows that given operators that are identical in all other specifications, an operator with a (RANGE, SLIDE) specification of (5,2) can be derived from an operator with (5,1) specification or an operator with a (6,2) specification. At the lowest level in the lattice, the internal nodes contain pointers to actual operator definitions. Since the Figure 61 indicates a leaf level node (since window specification forms the lowest level in the lattice tree), the internal nodes contain pointers to actual operator definitions.

6.4.2 Encoding Network Location

In order to allow search based on different granularities of network locality, network nodes are organized into 'regions' based on the notion of "nearness in the network".

The organization of network nodes into regions can be based on a clustering algorithm like K-Means that uses inter-node delay as a clustering parameter or a static grouping if the distribution of nodes in the infrastructure is known before hand. Each region is identified by a unique bit-vector of length n, where n is the number of regions. We refer to this bit-vector as a 'Region ID' (RID). The RID for the ith region has the ith bit set to 1 and all other bits set to 0.

The network location indicator (NID) of an operator, is a bit-vector that represents the region(s) to which the operator belongs. The ith bit of the NID is set to 1 only if the node belongs to the ith region. At the lowest level, each internal node contains pointers to all operators with the same key at each level of the lattice. Each internal node in the lattice is again associated with an NID which is the bitwise OR of all the associated operator NIDs. Note that the same operator may appear at multiple network locations causing the operator NID to have more than one bit set to 1.

Figure 61 shows an example lattice node. The figure shows a single leaf level lattice node where the partitioning condition is the window specification. As the figure shows, the lattice node contains a collection of keys, (RANGE, SLIDE) specifications in this case, such as (5,1), (6,2) etc., organized into a containment (see Section 6.3.2) based structure. Since this is a leaf node, each internal node contains a pointer to a set of operators. Note that all operators belonging to one internal node at this level have identical keys at all levels of the lattice. Each operator is associated with a NID corresponding to the regions where the operator resides. For example, the NID of key (5,1) indicates that such an operator is available in regions 1, 2 and 4.

6.4.3 Lattice Operations: Insert, Delete, Search

Inserting a new operator into the lattice involves first applying the partitioning condition to the operator in order to locate the appropriate lattice node to which the operator belongs, followed by a search for the key within the lattice node. If the key

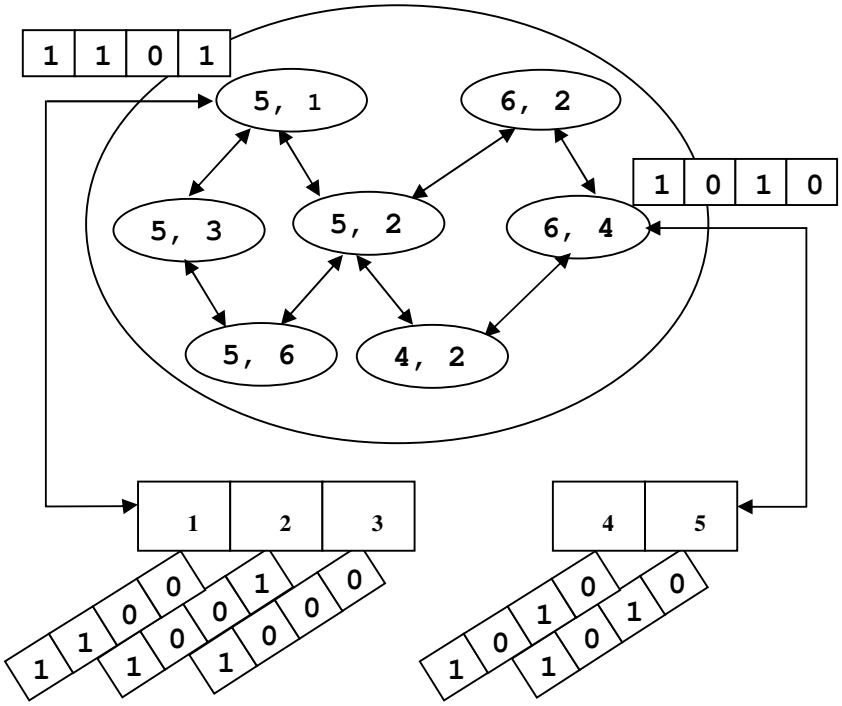


Figure 61: A single leaf lattice node.

is already present at an internal node within the lattice node, then the NID of the internal node is set to the bitwise OR of the existing and the new operator's NIDs. Otherwise a new internal node is created with the key and new operator's NID. Inserting an operator into the lattice may result in the insertion of a new internal node at each level of the lattice. The pseudo code for insertion into a single lattice node at some level i is shown in Figure 62.

In the delete operation, locating the internal lattice node at each level is similar to that during insertion. In order to avoid recomputation of the internal node NIDs for each deletion, we maintain a per-bit counter for each internal node which maintains the number of operators in the region with that key. The per-bit counter is decremented each time an operator with that key belonging to the region is deleted and incremented when an operator is inserted. When the last operator in the region with a given key is deleted, the bit corresponding to the region in the NID is set to

InsertIntoLatticeNode(LatticeNode, i, θ): *Insert operator θ into LatticeNode which is at level i.*

```
1. key := getKeyForLevel(θ,i);
2. List internalNodes := findKey(LatticeNode, key, PCONDN);
/*PCONDN ∈ {EQUAL, SUBSET, SUPERSET} */
3. internalNode := internalNodes[0];
4. if (internalNode is NULL)
5.      internalNode := create new internal node with key;
6.      setNetworkLocation(internalNode,getNetworkLocation(θ));
7.      setMinmalSupersets(internalNode, LatticeNode);
8.      setMaximalSubsets(internalNode, LatticeNode);
9. else setNetworkLocation(internalNode,
getNetworkLocation(θ)|getNetworkLocation(internalNode));
10. end if
11. return internalNode
```

Figure 62: Inserting operator into lattice node

zero. The pseudo-code for deletion is similar to that of insert.

Recall that the lattice supports search based on network location and support for relaxation. Figure 63 shows the pseudo code for searches for exact matches (without modification) within the regions specified as a bit-vector. In order to search for all operators that exactly match an input operator (the view definition of the query), the *SearchNetworkRestrictExact* function is invoked with the *root* as the start node. The AREAID bit-vector has one bit per region and any combination of regions can be searched by using an appropriate bit-vector. Using a bit-vector with all 1s, an exhaustive search can be performed. If relaxations are allowed, the *SearchNetworkRestrictExact* function is modified to stop at the level where all base conditions are satisfied and the set of candidate nodes are selected. The set of operators that can be reused with modifications are then obtained by traversing all the lower level nodes of selected candidate nodes.

SearchNetworkRestrictExact(θ, AREAID, StartNode):
Search for exact matches for θ AREAID beginning at StartNode.

```
1. i := getLevel(StartNode);
2. key := getKeyForLevel(θ,i);
3. LatticeNode := getLatticeNodeByKeyCond(StartNode, key, i);
4. List internalNodes := findKey(LatticeNode, key, EQUAL);
/*PCONDN ∈ {EQUAL, SUBSET, SUPERSET} */
5. if (internalNodes is NULL) return NULL; else
6.   for each internalNode in internalNodes
7.     if(getNetworkLocation(internalNode) & AREAID is 0)
8.         return NULL;
9.     else if level is LEAF
10.          return operators of internalNode within AREAID;
11.        else
12.          SearchNetworkRestrictExact(θ,AREAID, internalNode);
13.        ...
```

Figure 63: Search with network restriction.

6.5 Putting the Pieces Together

Thus far, we have presented key pieces of our StreamReuse subsystem – the semantic analyzer and the reuse lattice data structure. We described how dynamic groupings of queries are identified and how relaxations to existing operators and the corresponding compensations are computed. The question that now arises is: "how do we actually implement these operations at runtime and migrate existing queries to the new plans while ensuring the correctness of results?". This section describes the cost model that is used to evaluate deployment candidates identified from our reuse lattice and the steps involved in implementing dynamic query groupings.

6.5.1 Cost Model

In a distributed data-stream system where communication and processing costs are high and incurred continuously, an optimal query execution plan should ideally try to achieve multiple objectives – a minimum response-time while also incurring minimum communication and processing cost per unit time. However, these objectives may be

conflicting, since it is possible that lower delay paths have higher communication cost, or it may be the case that paths that incur low communication cost can cause a processing overload at some intervening network node. To optimize across such conflicting objectives, we choose the metric of 'network usage' described in [109] to compute cost of deployments. The network usage metric computes the total amount of data in-transit in the network. This metric captures the bandwidth-delay product of a query and trades off the overall application delay and network bandwidth consumption.

Consider a data stream processing system G. Let $\mathcal{L}_\theta$ represent the set of links that are used to transfer input data to the operator θ from immediate predecessor operators or sources (in the case of leaf operators). Then the **instantaneous network usage** $u(G, \theta, t)$ of operator θ at a time t over system G is given by

$$u(G, \theta, t) = \sum_{\ell \in \mathcal{L}_\theta} \lambda(\ell)\mu(\ell)$$

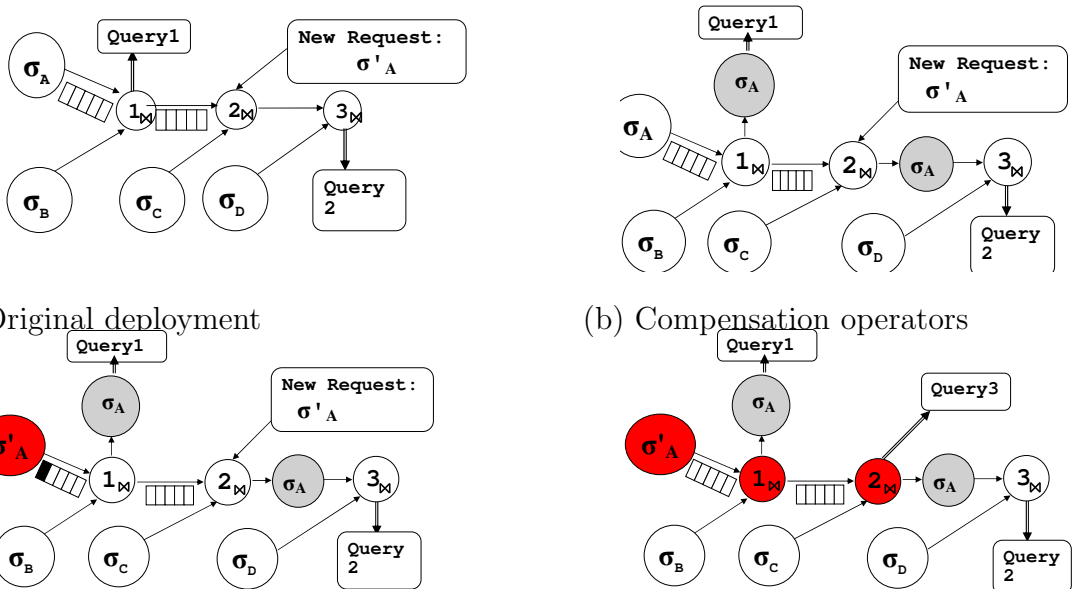
where $\lambda(\ell)$ and $\mu(\ell)$ represents the data-rate and latency respectively of link ℓ. Since each operator θ has an associated lifetime $T(\theta)$, this cost is incurred continuously over the lifetime of the operator. Thus, the total cost including estimated future costs that are incurred due to operator θ over its lifetime $T(\theta)$ is given by

$$\mathcal{U}(G, \theta) = \int_t^{T(\theta)} u(G, \theta, t)$$

Thus, the total system cost (including future estimates) under a set of operators $\Theta(t)$ is given by :

$$\mathcal{C}(G, \Theta(t)) = \sum_{\theta \in \Theta(t)} \mathcal{U}(G, \theta)$$

Taking into consideration the long running nature of the queries, given a new query Q to be deployed over the system G, our goal is to find an efficient query deployment such that the new set of operators $\Theta'(t)$ minimizes the total system cost including estimated future costs. Our objective is therefore to find this query deployment $\Theta'(t)$ from the space Θ of all such deployments such that the total system cost $\mathcal{C}(G, \Theta'(t))$ is given by



(a) Original deployment

(b) Compensation operators

(c) Operator Relaxation

(d) Final deployment

Key : ● - Relaxed operator ● - Compensation operator ■ - Marker Tuple

Figure 64: Runtime Migration Steps

$$\mathcal{C}(G, \Theta'(t)) = \min_{\Theta_i(t) \in \Theta} \sum_{\theta \in \Theta_i(t)} \mathcal{U}(G, \theta)$$

Note that our optimization function takes network locality, data rates and operator lifetimes into consideration. When an operator is reused, costs (network and processing) for the operator are incurred only once. Intuitively, it is in general more beneficial to share operators with long-lived queries than with short-lived queries, since with the former, the benefit from longer durations of sharing may outweigh the temporary difference in cost. The lifetime parameter in the expression essentially captures this intuition. The planner evaluates candidate deployments *including those without operator grouping* using our cost function. Thus queries are grouped only when the cost model indicates that it is beneficial to do so.

The effectiveness of the StreamReuse approach is measured using two key metrics: **instantaneous network usage** and **end-to-end latency**. The instantaneous

network usage under a set of operators $\Theta(t)$, which is the total system resource consumption per unit time is given by the expression

$$c(G, \Theta(t), t) = \sum_{\theta \in \Theta(t)} u(G, \theta, t)$$

In order to measure the response time while taking independent intra-query parallelism into consideration we adopt the end-to-end latency cost model devised in Ganguly et al. [67]. Using this model, the response time for a deployment is computed as the latency along its critical path which is the longest path from the root to a leaf node in the operator tree.

6.5.2 Runtime Plan Migration

Techniques for runtime plan migration exploit the fact that selection, projection and temporal filter operators are idempotent, i.e, the effect of applying the operation multiple times is the same as that of applying it once.

Figure 64 outlines the steps involved in the process of runtime relaxation. The white circles indicate operators already deployed in the system and the arrows indicate the direction of flow of data. Two queries are already being served by the deployment as indicated in the figure. The figure explains how the deployment is relaxed at runtime in order to serve a third request that requires the output of join operator $2_{\bowtie}$ with the selection predicate σ_A modified to σ'_A. For the sake of clarity, we show the runtime relaxation of only selection predicates. The same steps apply to projection operators and temporal (range or slide) filters also.

Figure 64(a) shows the original deployment with the new request for relaxation at operator $2_{\bowtie}$. Upon arrival of the request, the relaxations and compensations involved are computed recursively using the steps described in Section 6.3.2 and the deployment with the least cost is chosen. The first step is to introduce the idempotent compensation operators at the earliest point along the outputs to existing queries from the operators to be relaxed. This step is shown in Figure 64(b) where

the compensation operators for Query1 and Query2 are added as shown in the figure. Adding a compensation operator at runtime involves creation and placement of the compensation operator followed by redirection of the data-flow. Since the compensation operators have no state, plugging them into the deployment is simple.

The second step, as shown in Figure 64(c), is the beginning of the relaxation process. Since relaxation of input conditions (upstream operators) must be performed before relaxation of local conditions, the actual relaxation is first performed by the most upstream operator where the filter conditions are actually applied. Relaxation involves runtime modification of the filter predicates. Most stream systems supporting adaptivity have the facility to modify filter conditions at runtime [28, 61]. In our case, the selections are instantiated as parameterized filters [61], and this reduces the task of filter modification to the task of changing the parameter associated with the filter. When the relaxed operator begins executing, it inserts a special tuple, called the marker tuple, into the input stream. When the marker tuple is received by a downstream operator, it indicates that the relaxation has been performed upstream and all tuples that follow result from the new filter condition. The downstream operator then performs the local relaxations and indicates this to other downstream nodes by again introducing a marker tuple. Compensation operators and operators that do not perform local relaxations absorb the marker tuples and do not propagate them further. Once relaxations have been performed, operators activate output streams that requested the relaxations, as shown in Figure 64(d).

A natural next step to runtime relaxation of operators with query arrival is runtime contraction with query departure. Given the steps for runtime relaxation of queries, runtime contraction is also possible by performing the steps in the reverse order. Runtime contraction is triggered by the departure of a query from the system. Briefly, if one of the many queries sharing an operator leaves the system, the disjunctions in predicates at all upstream vertices introduced by the query can be removed resulting

in more restrictive filters. Again, when more than one operator is involved, cascading contractions should be performed beginning at the most downstream operator used by the query.

6.6 Experimental Evaluation

Experimental evaluation of the StreamReuse approach studies the performance of our techniques with respect to a number of metrics such as resource usage, latency, and planning time. Our experimental evaluation answers the following questions: (1) What is the cost-benefit of reusing operators across continual queries in terms of resource usage? (2) How do these techniques behave under different workloads? (3) What is the impact on the latency of the existing deployments and the throughput of the system? (4) What are the effects of dynamic grouping on query planning time and the time to initial deployment?

Experiments were performed on both simulations and a prototype and were conducted using two very different workloads: an enterprise workload obtained from Delta Air Lines and a synthetically generated RFID workload. Our results show that:

- Our techniques can reduce network usage by as much as 96% when compared to the state-of-art approaches.
- By our dynamic grouping approach, computation costs can be reduced by more than an order of magnitude while the increase in latency and time-to-deployment is negligible.

6.6.1 Workloads

Figure 65 partitions the space of workloads into four quadrants based on the number of sources and queries. Our interest primarily lies in the shaded quadrants that represent a high number of simultaneously executing continuous queries where there is significant opportunity for operator reuse. It should be noted that while our system

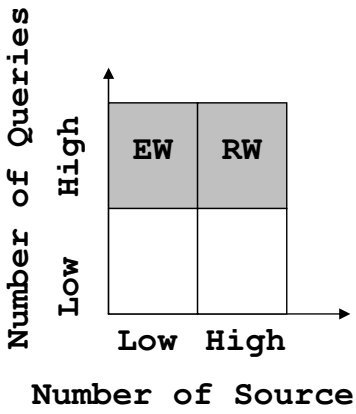


Figure 65: Classification of workloads

Source	Rate	Selectivity
Flights	1500 per 5 min	1 per flight per 5 min
Check-ins	240 per min	2 per flight per min
Baggage	500 per min	4 per flight per min
Weather	450 per 5 min	1 per destination per 5 min
Sales	70 per min	1 per destination per min

Figure 66: Enterprise Workload

Parameter	Value
Number of sources	20
Query size	3 joins (80%), 4 joins (15%), 5 joins (5%)
Number of predicates	1-3
Query duration	1 hour (80%), 6 hours (15%), 12 hours (5%)
Query arrival rate	Poisson $\mu = 30$ *queries/hour*
Range, Slide specifications	U[1-5] Minutes
Total record size	Fixed (100 bytes)

Figure 67: RFID Workload

can handle the other two quadrants, due to the low number of queries, the benefit to be realized from enabling operator reuse is also low. Another factor that determines the performance of our techniques is the number of data-stream sources in the system – a high number of sources may lead to a varied set of operators implying a low possibility of operator reuse, while a relatively low number of sources increases overlap between queries and the possibility of reuse. Accordingly, we model two workloads that are representative of those two quadrants. For example, enterprise information systems (Section 6.2.1) are representative of the quadrant which covers a region with a high number of queries and a low number of sources. Similarly, applications like the ones used for inventory tracking using RFID tags can be categorized into the quadrant with a high number of sources and a high number of queries.

EW: Enterprise-Workload The enterprise workload is a real-world workload consisting of gate-agent, terminal and monitoring queries posed as part of the day-to-day

operations of Delta Air Lines' enterprise operational information system. The query workload is based on the 5 query sources whose characteristics are shown in Table 66. Gate agent queries constitute 80% of the workload and the SLIDE for these queries is set to 1 minute. The queries use the following template.

```
Q3: SELECT FL.GATE, BG.STATUS, CI.STATUS
    FROM FLIGHTS FL [RANGE 5 MIN], BAGGAGE BG [RANGE 1 MIN], CHECK-INS CI [RANGE 1
    MIN]
    WHERE FLIGHTS.NUM = CHECK-INS.FLIGHT
    AND FLIGHTS.NUM = BAGGAGE.FLIGHT
    AND FLIGHTS.NUM = ?;
```

Each gate agent query originates at the gate of departure of a flight and lasts for 2 hours prior to departure of the flight. Terminal queries, which represent 15% of the workload, are longer running queries (12 hours lifetime) and follow the template of query Q1 in Section 6.2.1 and are evaluated every minute. Finally, the last 5% of the workload represent long-running (6 hours) ad-hoc monitoring queries over any combination of the 5 sources. For these queries, we use window ranges and slides that are uniformly distributed between [1-5] minutes for all streams. With nearly 1500 flights a day, we assume that queries arrive with a poisson distribution with $\mu = 60$ *queries/hour*. Each update record was assumed to be of the same size (100 bytes).

RW: RFID-Application Workload The synthetic RFID workload (Table 67) models the quadrant representing systems with a large number of queries over a large numbers of sources resulting in smaller overlaps between queries.

6.6.2 Experimental Setup

Our prototype was built over a distributed stream processing system [90] and used a testbed of 128 Emulab nodes (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9),

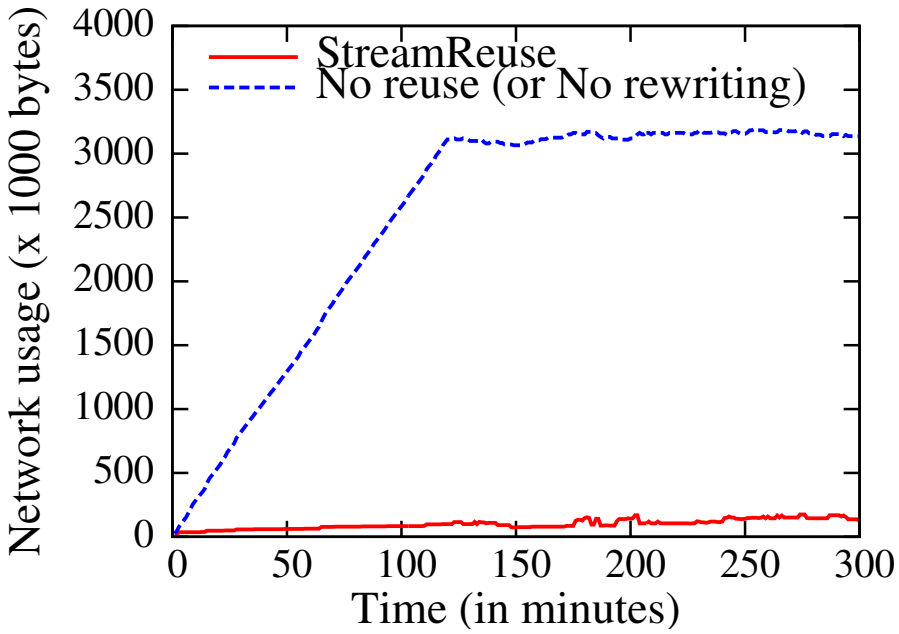


Figure 68: EW: Network usage

organized into a topology that was generated using the standard tool, the GT-ITM internetwork topology generator [159]. Links were 100Mbps and the inter-node delays were set between 1msec and 6msec. The simulation experiments were conducted over transit-stub topology networks generated using GT-ITM. Experiments used a 128 node network, with a standard Internet-style topology: 1 transit (e.g., "backbone") domain of 4 nodes, and 4 "stub" domains (each of 8 nodes) connected to each transit domain node. Link costs (per byte transferred) were assigned such that the links in the stub domains had lower costs than those in the transit domain, corresponding to transmission within an intranet being far cheaper than long-haul links. We used a uniformly random selection of nodes for sink placements.

In order to compute the placement of new operators over the network, we implemented a version of the 'Top-Down' algorithm described in [133]. The algorithm first clusters nodes into regions based on the notion of 'network nearness' and then maintains a virtual hierarchy of regions that helps to scalably compute placements over a large set of nodes with near-optimal approximations. The algorithm exploits

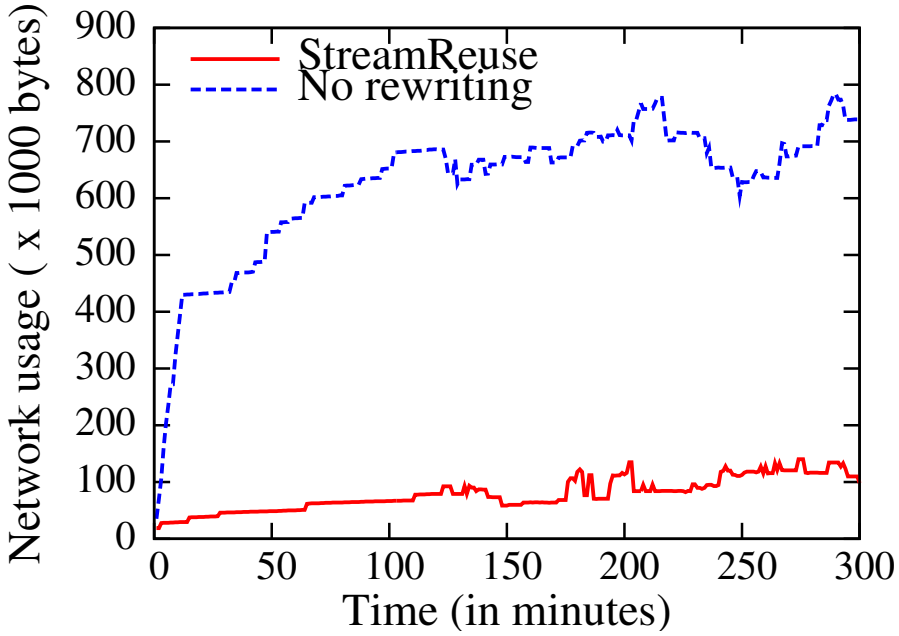


Figure 69: 10% Overlap

the network locality based search of our reuse lattice to perform searches for reuse opportunities within a subset of of regions.

The StreamReuse approach is compared with two other state-of-the-art techniques: (1) no rewriting, reuses existing operators only if their definitions exactly match the requirements and does not perform any rewritings [109, 133] and (2) no reuse [32] does not take into consideration any existing operators while deploying new queries.

In order to study the resource usage of our techniques and compare with other existing approaches the following two concrete metrics are used: the **instantaneous network resource usage** and the **number of operators** in the system which is indicative of the processing resource usage. Recall that the network usage $u(q)$ of a query q represents the total amount of data that is in-transit for a query at any given instant. The total number of operators in the system and the number of join and select operators are indicative of the processing load imposed on the system and again the throughput to which the system can scale. The effect of dynamic operator

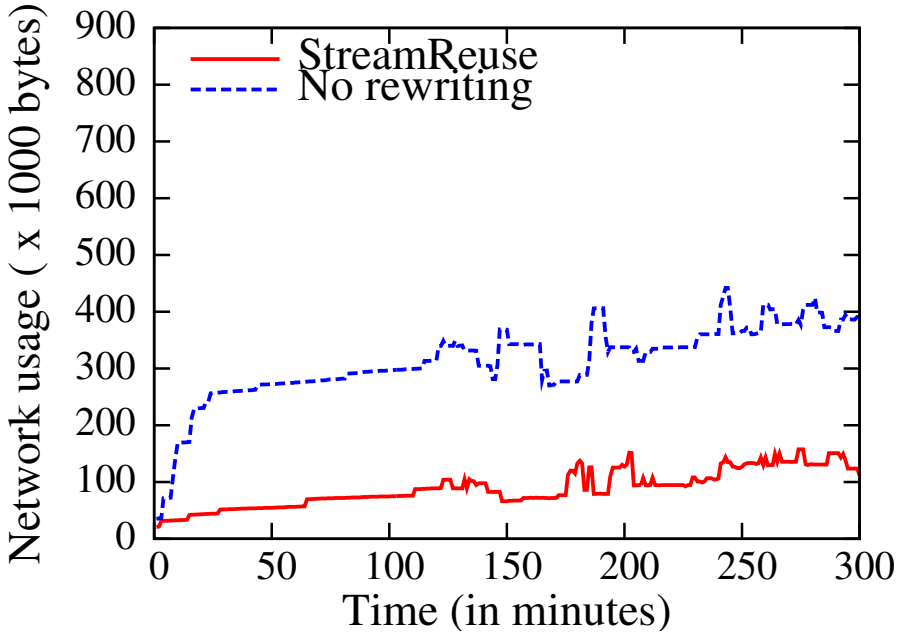


Figure 70: 25% Overlap

grouping on response time of individual deployments is measured using the **end-to-end latency** of deployments [67]. Finally, the **time-to-deployment** is used to evaluate the overhead imposed on the planning process.

6.6.3 Efficiency of Deployments

Figure 68 shows the total network usage per instant of time for a 5 hour duration of system deployment with the EW workload. Gate, terminal and monitoring queries last for 2, 12 and 6 hours respectively. After initial ramp-up, approximately 120 queries execute concurrently.

Under the EW workload each gate query specifies a unique flight number. Similarly, all terminal queries are unique as are the monitoring queries. All selection predicates are placed earlier in the query deployment, as close to the source as possible. In the presence of all unique predicates, the NO REWRITING technique degrades to a NO REUSE technique. As the graph shows, even in the presence of unique selection

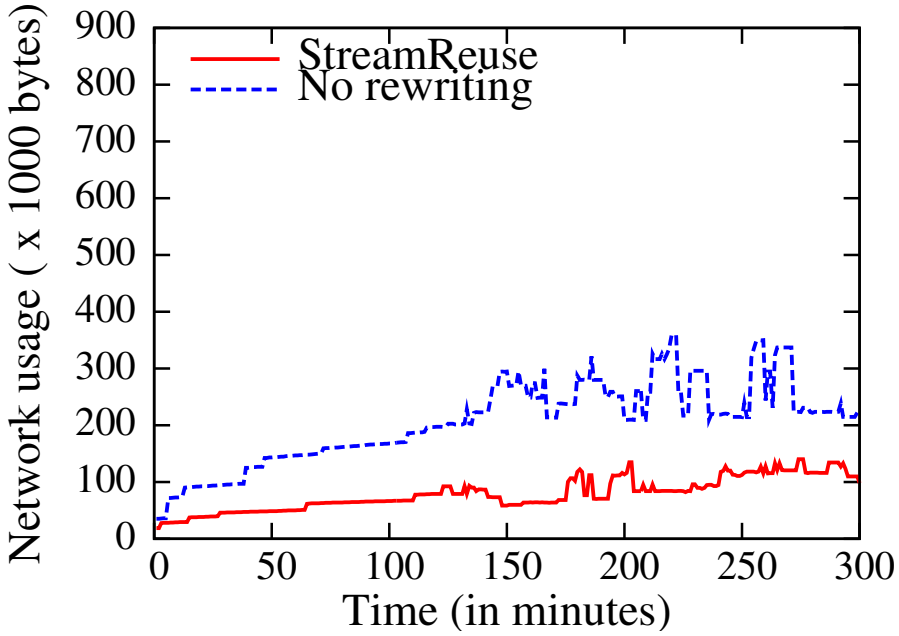


Figure 71: 50% Overlap

predicates, our approach of runtime relaxation and rewriting can reduce network usage by nearly 96% as compared to a NO REUSE/ NO REWRITING approach. This large win can be attributed to the fact that fewer join operators to which inputs need to be streamed are deployed in the system. In the presence of highly selective joins, the small increase in input size to few join operators is negligible compared to streaming inputs to a large number of join operators.

In order to study the effect of STREAMREUSE under varying degrees of overlap between queries and compare with a NO REWRITING approach, controlled variations were induced in the degree of overlap between selection predicates in the EW workload. Figures 69, 70 and 71 show the network usage over 5 hours when the total number of unique selection predicates are only 10, 4 and 2 respectively. The figures show that as the variance between queries increases, NO REWRITING techniques result in more network usage. In fact, by adopting a runtime relaxation based approach, with only 10 unique predicates, network usage can be reduced by as much as 75% compared to NO REWRITING (Figure 69).

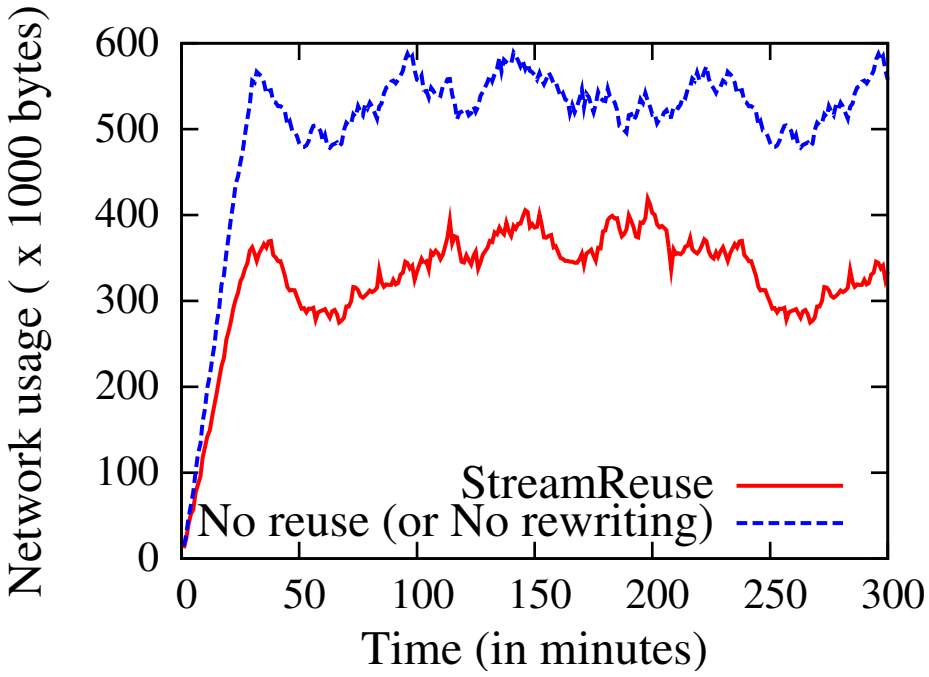


Figure 72: RW: Network usage

Figure 72 shows the total network usage with the RFID workload (RW) over 5 hours. The system executes with approximately 30 concurrently executing queries after initial ramp-up. This workload has more sources than the enterprise workload, and sources specified in queries are chosen from a large range of combinations. In fact, with the RW workload, within any set of concurrently executing queries, only 5 queries had any common joins with other queries. Consequently the number of rewrite/reuse opportunities available are fewer. Again, as in the case of the enterprise workload, since negligible number of queries specify the exact same selection predicates, the NO REWRITING approach degenerates into a NO REUSE approach. The figure shows that a dynamic grouping based approach results in a 28% reduction in network usage even with this workload.

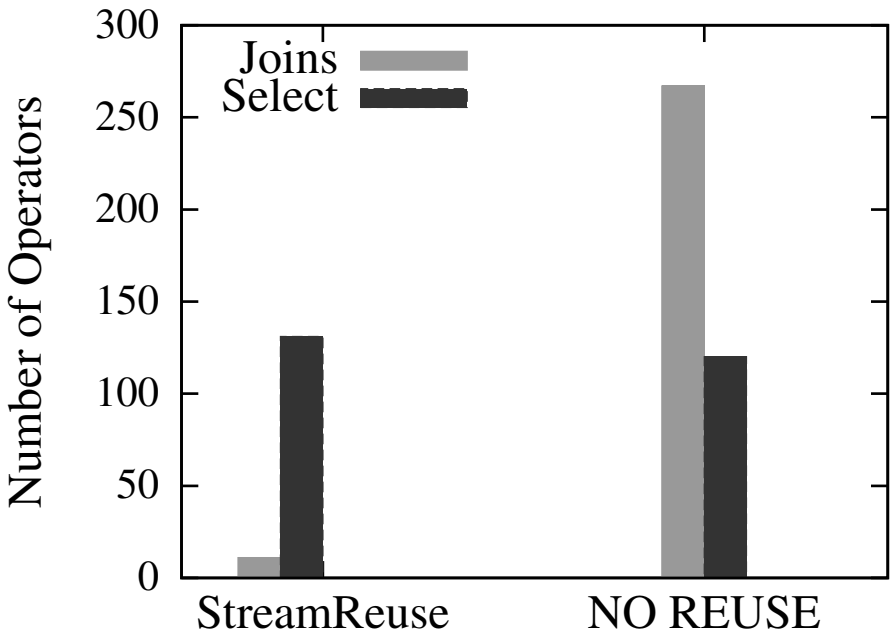


Figure 73: EW: Deployed operators

6.6.4 Evaluation of Computation Costs

Computation costs are evaluated using the average number of operators deployed at any instant of time. Recall that, since all selection predicates are unique in EW, the NO REWRITING approach degenerates into a NO REUSE approach. Figure 73 shows the average number of join, and select operators that are concurrently executing at any given instant of time with STREAMREUSE and the NO REUSE approaches. The figure shows that the STREAMREUSE approach effects a massive decrease in the number of join operators with only a slight increase in the number of selection operators. The increase in selection operators can be attributed to the introduction of additional compensation operators while reusing existing joins. This figure shows that even in the presence of unique queries, by effectively sharing operators between queries through dynamic grouping, the number of join operators can be reduced by an order of magnitude. It is a well known fact that joins are expensive operators and can reduce throughput. By reducing the number of such expensive joins, we expect the

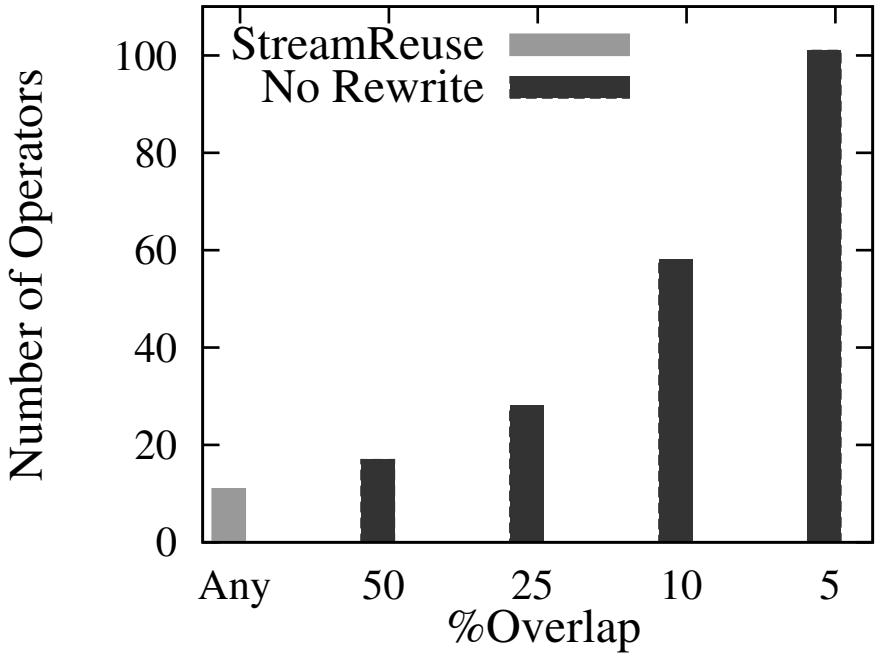


Figure 74: EW: Join operators

throughput to increase significantly.

The EW workload was modified to compare StreamReuse against no rewriting under varying degrees of overlap between queries. Figure 74 shows the number of concurrent join operators with the two approaches in a set-up where the number of unique predicates in the workload is varied. A 50% overlap represents a workload with only 2 unique predicates. Similarly 25%, 10% and 5% overlaps represent 4, 10 and 20 unique predicates respectively. The figure shows that the number of join operators with no rewriting increases rapidly as the number of unique predicates increases. The StreamReuse approach, however, deploys fewer joins (11 joins) even in the presence of a workload where all queries are unique.

Figure 75 shows the number of deployed operators with the RW workload. Briefly, in spite of the low overlap between queries, our techniques resulted in a 28% reduction in join operators over the no reuse approach.

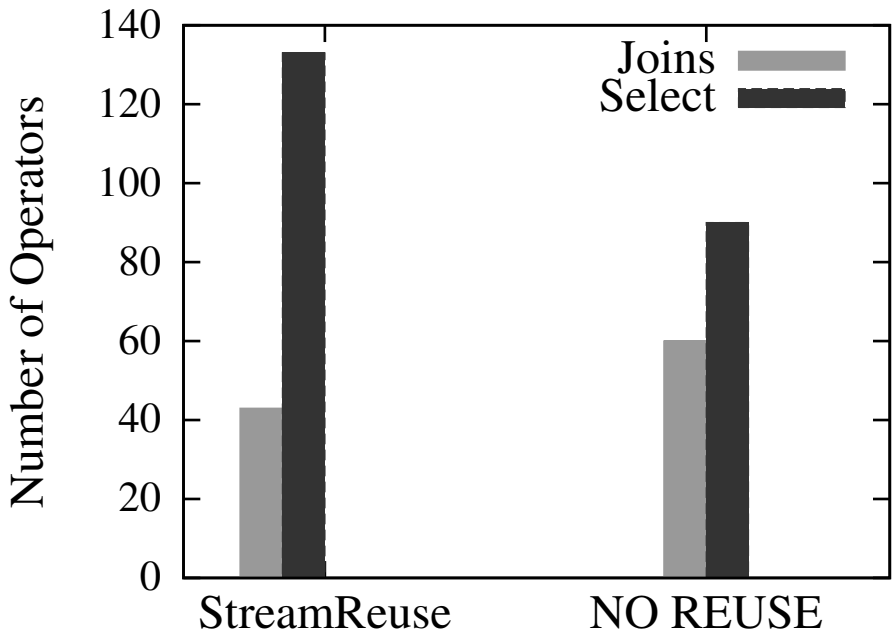


Figure 75: RW: Deployed operators

Workload	Operation Time	Average
All	Total time to deployment	3.5 sec
EW	Plan Computation	8.733 ms
	Rewrite/Lattice search	1.158 ms
RW	Plan Computation	9.262 ms
	Rewrite/Lattice search	0.738 ms

Table 11: Deployment times

6.6.5 Effect of Grouping on Latency

Figure 76 shows the end-to-end latency of 120 queries of the EW workload with StreamReuse and with no reuse/ no rewriting. The figure shows that latencies experienced by the queries with StreamReuse is comparable to that with no reuse or no rewriting. On an average, latency increases by 13 msec with StreamReuse. Figure 77 shows the end-to-end latency of 120 queries of the RW workload with StreamReuse and with no reuse/no rewriting. In this case, the average increase in latency with StreamReuse is only 3.675 msec. Since most applications can tolerate

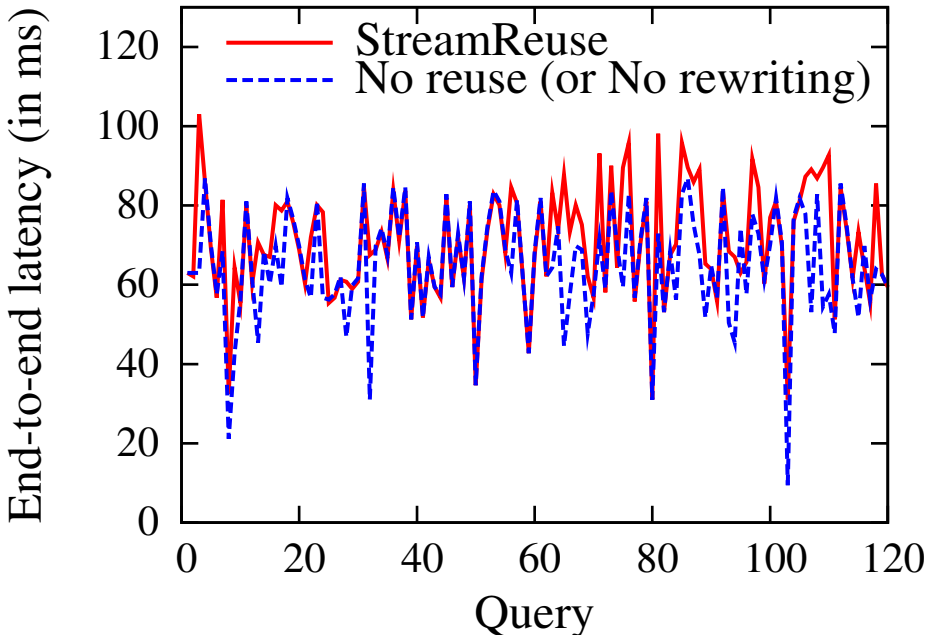


Figure 76: EW: Latency

this slight increase, this is a small price to pay for the large savings in network and computational costs.

6.6.6 Prototype Experiment: Deployment Time

The scalability of the StreamReuse approach was studied by examining the overhead imposed by the planning process (including searching the lattice and computing relaxations) on the total time to deployment. Figure 78 shows the total planning time for the deployment of 300 queries from the EW workload each with an average of 8 operators (including selections and projections). The figure demonstrates: (1)that the increase in planning time is negligible (an average increase of 1 ms) and (2) the increase in planning time with the size of the lattice is near linear.

The times for the different actions performed during deployment are summarized in Figure 11. The total time to deployment is the time taken to contact nodes in the system and deploy the operators. Since the nodes are contacted in parallel, this time

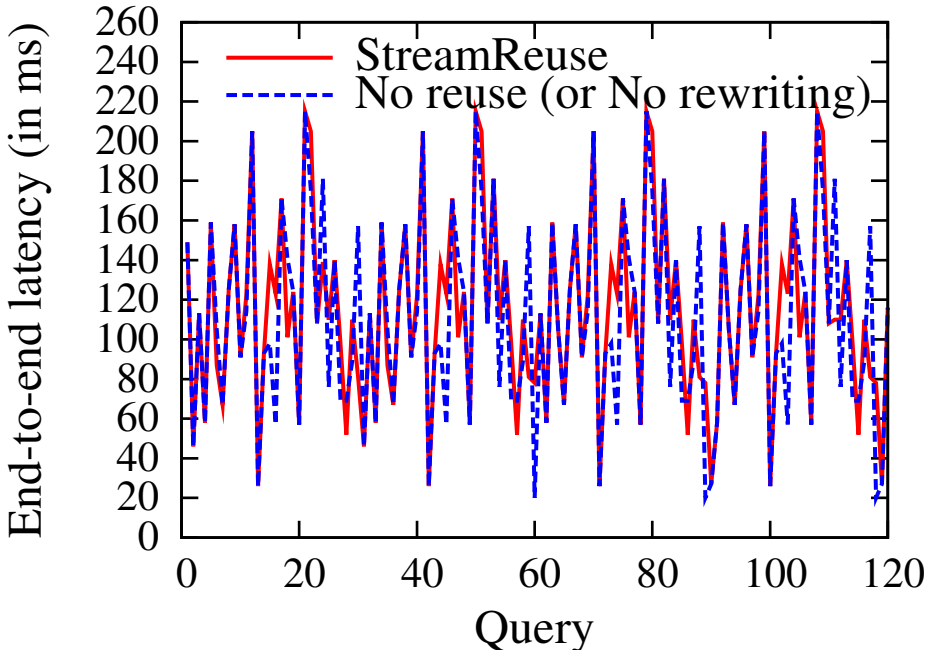


Figure 77: RW: Latency

is nearly independent of the number of operators and is mainly a function of inter-node delays. The plan computation times are common to the NO REUSE, NO REWRITING and runtime rewriting cases. The rewrite and lattice search time is the additional overhead imposed on the planning process by our techniques. Note that, compared to the deployment times and planning times, the rewrite and lattice search time is negligible. Given the large gains in network and computational costs, this leads us to conclude that this one time overhead at the time of deployment is completely justifiable.

6.7 Related Work

A number of data-stream systems such as STREAM [36], Borealis [28], TelegraphCQ [51], NiagaraCQ [55] and System S [34] have been developed to process queries over continuous streams of data. We present a summary of related work pertaining to techniques for the reuse of operators between distributed stream query deployments.

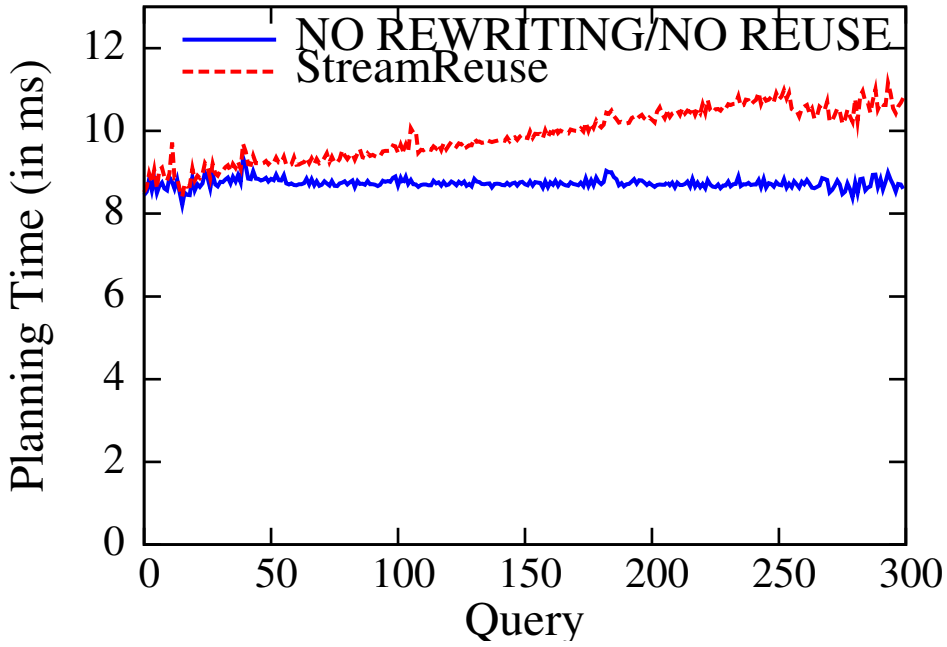


Figure 78: Total Planning Time

Coincidental Reuse (Static): With this approach, a newly arriving query reuses an existing operator only if it exactly matches the query's requirement [32, 109, 133]. Even with a high degree of overlap amongst queries in the workload, this strategy offers only average performance since it is unable to reuse operators unless they exactly match.

Optimistic-deployment (Static): Such systems use 'rules of thumb' such as 'pull-up selections' to improve reusability [55, 88] or assume that the workload is already known [156, 78]. The approach may be effective when the workload is known *a priori*, but is unable to handle a dynamic workload since the system must pay the price of increased intermediate data size even for operators that are never reused.

Runtime Recomputation (Dynamic): In this approach, with each arrival or departure of a single query, a portion of the query network [28] is replanned. However, operator groupings are performed by the system administrator and are not dynamic.

System level functionalities that allow runtime modification of operator parameters have been implemented in systems such as Borealis [28]. Our techniques utilize such system-level functionality, along with semantic knowledge, to perform dynamic grouping of queries and runtime migration of query plans. Our work also builds on query rewriting techniques that have been widely studied in the context of materialized views [54, 71]. While operator definitions in our context are similar to view definitions, our problem is complicated by the need to consider network locality, operator similarity, windows and runtime modifications in addition to containment.

6.8 Summary

We have described StreamReuse, a reuse-conscious distributed stream query processing system for scaling distributed stream query services. The main idea behind the design of StreamReuse is three folds. First, we exploit the operator similarity of multiple concurrent stream query services, aiming at enhancing the performance and scalability of query services by minimizing the amount of duplicate processing in the system. Second, we refine operator level reuse opportunities by taking into account of the network locality of these operators to ensure that only the highly profitable reuse opportunities are capitalized on by StreamReuse. Third but not the least, we introduce the notion of 'relaxations' and the 'reuse lattice' data structure to encode reuse opportunities through operator semantics and network locality awareness, and to enable the runtime identification of reusable operators that exhibit both operator level similarity and network locality similarity. A unique characteristics of StreamReuse is its three step reuse opportunity discovery and deployment process: (i) operator-similarity based query relaxation, (ii) network locality-based reuse refinement, and (iii) seamless runtime reuse plan migration. We evaluate the StreamReuse approach by conducting experiments over a range of different workloads. The experimental results show that our reuse techniques can reduce resource

consumption and computational costs by more than an order of magnitude compared to the existing approaches.

CHAPTER VII

NETWORK-AWARE OPERATOR REUSE

7.1 Introduction

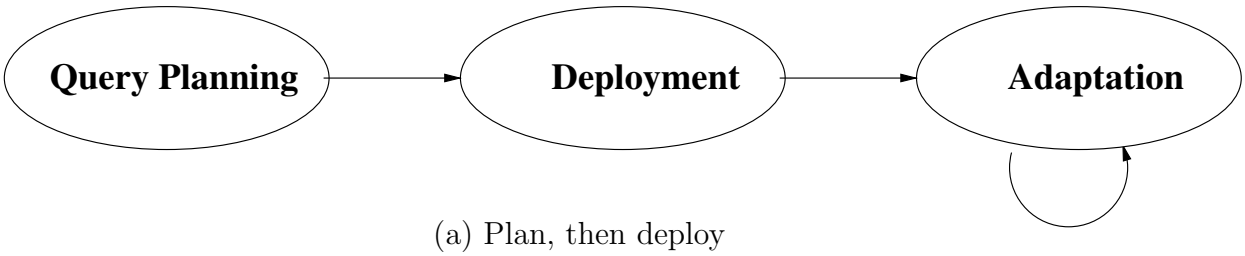
Many data stream delivery and dissemination systems today produce stream data at multiple, geographically distributed locations. It is often too expensive to stream all of the data to a centralized query processor, both because of the high communication costs, and the high and yet continuously changing processing load at the central server. Therefore, in order to ensure efficiency and scalability, these naturally distributed applications adopt a distributed processing paradigm.

Distributed data streams systems are distinguished by a number of characteristics. First, a network of computing nodes with heterogeneous bandwidth and computing resources together serves as a distributed data stream delivery system. Second, data streams originate from multiple sources and are disseminated to multiple receivers. Third, multiple continuous stream queries are executing simultaneously on the stream delivery network with different input and output rates. Instead of shipping all data streams to a single node and processing all the stream queries in a centralized server, many have shown that performing distributed processing of stream queries using techniques such as in-network processing [158, 99, 32] and filtering at the source [105] minimizes the communication overhead on the system and helps spread processing load, significantly improving performance.

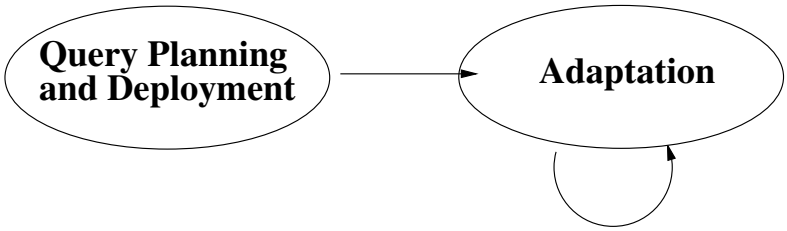
Given that data streams are typically produced from multiple disparate nodes, stream queries naturally consist of many operators (filters, joins etc.) on multiple data streams of interest. We can think of a data stream query as a continual query

being "deployed" in the network, with data streams flowing between operators associated with distributed physical streaming nodes, which may either be sensor nodes or the relay nodes in a data stream delivery network. The conventional approach to stream query processing used in many existing distributed data stream management systems [28, 136] consists of three consecutive phases: query planning, query deployment, and query adaptation. The system constructs a query plan (e.g., the stream query processing should follow a specified join ordering) at compile time and deploys this plan at runtime to improve performance. Figure 7.1(a) gives a sketch of this approach. A fundamental problem with this static optimization approach is its inability to respond to the unexpected data and resource changes occurring at runtime. For example, the join order chosen at compile time may require intermediate results to be transported to another network node over a long distance, even though there exists an alternate join order that is more efficient. Similarly, a predefined join order may involve a transfer or a processing of an intermediate result to a node that is currently unavailable, thus causing the query to halt even though an alternate join order exists and is available. Furthermore, given that each query plan is computed at compile time independently and once for all, the pre-defined join order from one query plan may prevent us from reusing the results of an already deployed join from another query at runtime. This limits the scope of the adaptation which aims at exploiting runtime environment properties to further optimize the efficiency of distributed stream query deliveries.

Bearing these issues in mind, we propose a distributed stream query optimization framework that considers the query plan and the deployment simultaneously (Figure 7.1(b)). Our framework consists of the system architecture for integrating distributed stream query planning and query plan deployment and a suite of techniques for performing query planning in conjunction with deployment planning. One of the key ideas in our framework is to use hierarchical network partitions to scalably



(a) Plan, then deploy



(b) Our approach

Figure 79: Approaches

exploit various opportunities for operator level reuse in the processing of multiple stream queries. Figure 80 compares the approach of integrating planning and deployment through operator reuse with two existing "Plan, then deploy" approaches − the Relaxation algorithm [109] and an optimal deployment through exhaustive search. The graph shows the total communication cost (the total data transferred along each link times the link cost) incurred by 100 queries over 5 stream sources each, on a 64-node network. The figure shows that significant ($> 50\%$) cost savings can be achieved by combining the planning and deployment phases.

It is well known that, as the size of the network grows, the number of possible plan and deployment combinations can grow exponentially. The cost of considering all possibilities exhaustively is prohibitive. Consider Figure 80. With a network of 64 nodes, combining query plans and plan deployments simultaneously required us to examine nearly 3.02×10^9 plans for a single query over 5 streams. Clearly, a key technical challenge for effectively combining query planning and plan deployment is to reduce the search space in the presence of large networks and a large number of query operators.

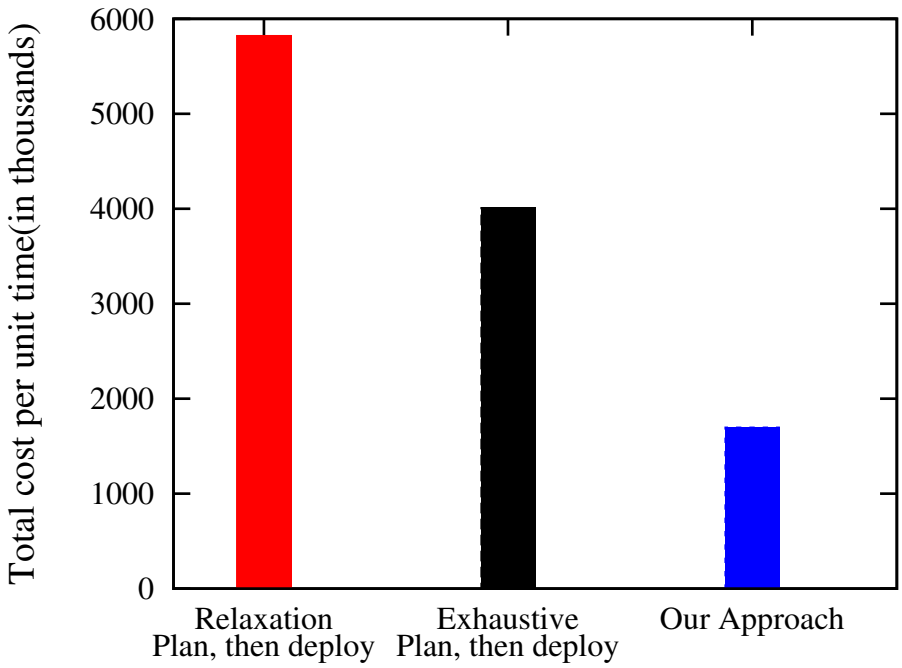


Figure 80: Comparison with typical approaches

One idea we explore in this work is to address this challenge by using hierarchical network partitions as a heuristic, aiming at trading some optimality for a much smaller search space. We organize the network of physical nodes into a virtual hierarchy and utilize this hierarchy along with "**stream advertisements**" to guide query planning and deployment. We develop three alternative algorithms to facilitate operator reuse through hierarchical network partitions. In the **Top-Down** algorithm, the query starts at the top of the hierarchy, and is recursively planned by progressively partitioning the query and assigning sub-queries to progressively smaller portions of the network. In the **Bottom-Up** algorithm, the query starts at the bottom of the hierarchy, and is propagated up the hierarchy, such that portions of the query are progressively planned and deployed. While both algorithms choose efficient deployments by exploring only a small fraction of the search space, the Top-Down algorithm is more effective in limiting the sub-optimality of the solutions while the Bottom-Up approach is more effective in reducing the search space, and thereby the time to deployment.

We further develop a heuristic based hybrid algorithms that combines the strengths of both the Top-Down and Bottom-Up algorithms - the **Net Present Cost (NPC)** algorithm. The NPC algorithm is a probabilistic algorithm that guides the planning process based on cost estimates of choosing a join order locally or delaying the decision to the next level. We have implemented our algorithms using IFLOW [90], a distributed data stream system. In this chapter we also present formal analysis and experiments to show that our algorithms can compute efficient deployments, and at the same time, reduce the search space by orders of magnitude compared to an exhaustive search, even using dynamic programming. For example, experimentally, the Top-Down algorithm on average was able to achieve solutions that were sub-optimal by only 10% while considering less than 1% of the search space.

The remainder of this chapter is organized as follows. We formally describe the distributed stream query optimization problem and give an overview of our distributed optimization framework in Section 7.2. We present the Top-Down, Bottom-Up and NPC algorithms and a rigorous analysis of their effectiveness in Section 7.3. Our experimental evaluation of the proposed solutions are reported in Section 7.4. The chapter ends with a discussion on the related work and a summary.

7.2 System Overview

Many modern enterprise applications [104, 2, 12], scientific collaborations across wide area networks [110, 19], and large-scale distributed sensor systems [156, 97] are placing growing demands on distributed streaming systems to provide capabilities beyond basic data transport such as wide area data storage [1] and continuous and opportunistic processing [12]. An increasing number of streaming applications are applying 'in-network' and 'in-flight' data manipulation to data streaming systems designed for enterprise systems [20], financial management [11], scientific computing [56, 110, 48], and situation monitoring applications [56, 66, 95].

The specific motivating example that we present in this work is based on enterprise-level data streaming systems such as the Operational Information System (OIS) [104] employed by our collaborators, Delta Air Lines. An OIS is a large scale distributed system that provides continuous support for a company or organization's daily operations. The OIS run by Delta Air Lines provides the company with up-to-date information about all of their flight operations, including crews, passengers, weather and baggage. Delta's OIS combines three different types of functionality: continuous data capture, for information like crew dispositions, passengers and flight locations; continuous status updates, for systems ranging from low-end devices like overhead displays to PCs used by gate agents and even large enterprise databases; and responses to client requests which arrive in the form of queries.

In such a system multiple continuous queries may be executing simultaneously and hundreds of nodes, distributed across multiple geographic locations are available for processing. In order to answer these queries data streams from multiple sources need to be joined based on the flight or time attribute, perhaps using something like a symmetric hash join. We next use a small example network and sample queries to illustrate the optimizations opportunities that may be available in such a setup.

7.2.1 Motivating Application Scenario

Let us assume Delta's OIS to be operating over the small network N shown in Figure 81. Let WEATHER, FLIGHTS and CHECK-INS represent sources of data-streams of the same name and nodes $N1 - N5$ be available for in-network processing. Each line in the diagram represents a physical network link. Also assume that we can estimate the expected data-rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data or measured by special purpose nodes deployed specifically to gather data statistics.

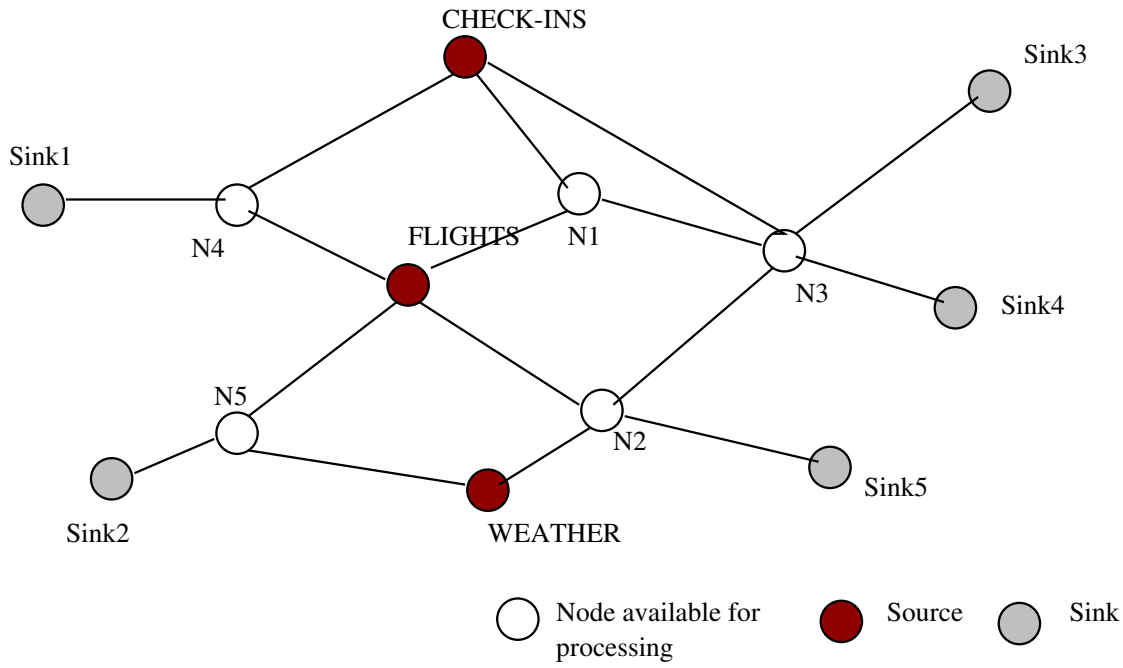


Figure 81: An example network N

Assume that the following query Q1 is to be streamed to a terminal overhead display $Sink4$. Q1 displays flight, weather and check-in information for flights departing in the next 12 hours.

```
Q1: SELECT FL.STATUS, WR.FORECAST, CI.STATUS
    FROM FLIGHTS FL, WEATHER WR, CHECK-INS CI WHERE FL.DEPARTING='ATLANTA'
    AND FL.DESTN = WR.CITY AND FL.NUM = CI.FLNUM AND FL.DP-TIME - CURRENT_TIME < 12:00:00
```

Network-aware join ordering: Based purely on the size of intermediate results, we may normally choose the join order (FLIGHTS⋈WEATHER)⋈CHECK-INS. Then we would deploy the join FLIGHTS⋈WEATHER at node $N2$, and the join with stream CHECK-INS at node $N3$. However, node $N2$ may be overloaded, or the link FLIGHTS→$N2$ may be congested. In this case, the network conditions dictate that a more efficient join ordering is (FLIGHTS⋈CHECK-INS)⋈WEATHER, with FLIGHTS⋈CHECK-INS deployed at $N1$, and the join with WEATHER at $N3$.

Now, consider situations where we may be able to reuse an already deployed

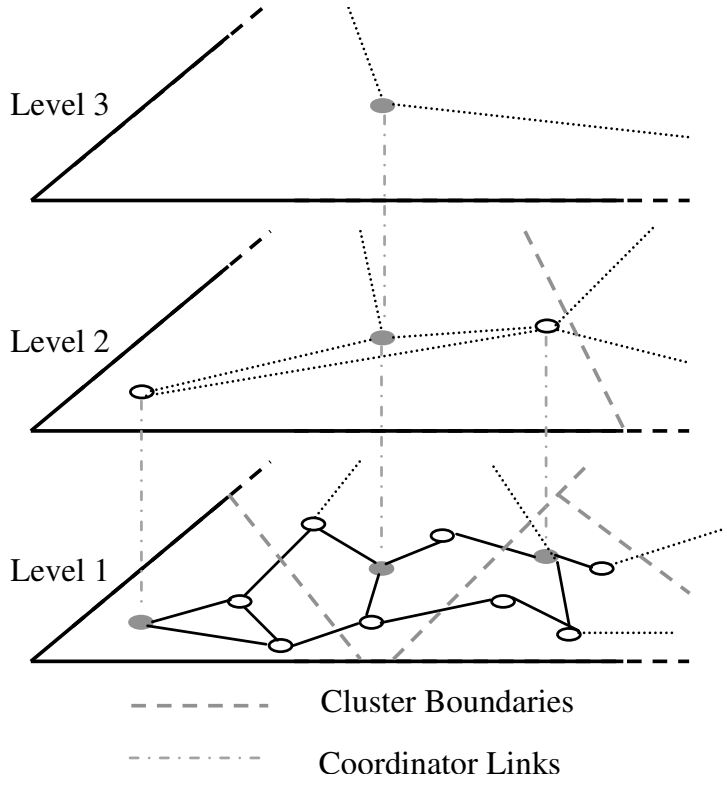


Figure 82: Hierarchical network clusters

operator. This will reduce network usage (since the base data only needs to be streamed once) and processing (since the join only needs to be computed once). Imagine that query Q2 has already been deployed:

Q2: `SELECT FL.STATUS, CI.STATUS FROM FLIGHTS FL, CHECK-INS CI`
`WHERE FL.DEPARTING='ATLANTA' AND FL.NUM = CI.FLNUM AND FL.DP-TIME - CURRENT_TIME < 12:00:00`

with the join FLIGHTS⋈CHECK-INS deployed at $N1$. Assume that the sink for the query Q2 is located at node $Sink3$.

Operator Reuse: Although the optimal operator ordering in terms of the size of intermediate results for query Q1 may be (FLIGHTS⋈WEATHER)⋈CHECK-INS, in order to reuse the already deployed operator FLIGHTS⋈CHECK-INS, we must pick the alternate join ordering (FLIGHTS⋈CHECK-INS)⋈WEATHER. Note that, reuse may require additional columns to be projected. In contrast, if the sinks for the two

queries are far apart (say, at opposite ends of the network), we may decide not to reuse Q2's join; instead, we would duplicate the FLIGHTS⋈CHECK-INS operator at different network nodes, or use a different join-ordering. Thus, having knowledge of already deployed queries influences our query planning.

These examples show that the network conditions and already deployed operators must often be considered when choosing a query plan and deployment in order to achieve the highest performance.

7.2.2 System Definition

We now formally describe the components of our distributed data stream system. Let $N(V_n, E_n)$ represent a physical network of nodes where vertices V_n represent the set of actual physical nodes and the network connections between the nodes are represented by the set of edges E_n. Let Q represent a single continuous query and let $P^Q = \{p_1^Q, \ldots, p_m^Q\}$ represent the set of all relational algebra query trees (e.g. operator orderings) for query Q. The *deployment* of a query tree p_j^Q over the network N is defined as a mapping $M(p_j^Q, N)$ that assigns each operator in p_j^Q to a network node $v_{nk} \in V_n$.

Since network costs are a primary concern in wide-area stream processing systems, to illustrate our techniques, we choose a formulation that tries to minimize the communication cost incurred per unit time by the deployed query plan. We use the metric of 'network usage' [109] to compute costs of query deployments. The network usage metric computes the total amount of data in-transit in the network at a given instant. This metric captures the bandwidth-delay product of a query and trades off the overall application delay and network bandwidth consumption. We define a cost function $Cost(M(p_i^Q, N))$ that estimates the total network usage per unit time for the deployment $M(p_i^Q, N)$. Using network usage as the cost function, for a deployment $M(p_i^Q, N)$ the cost is given by $\sum_{l \in M(p_i^Q, N)} \lambda(l) \times latency(l)$ where λ(l) represents the

data rate over the physical link l. We present a further discussion on the choice of the cost metric in Section 7.5.

7.2.3 Optimization Problem

Our problem definition addresses the continual query equivalent of 'select-project-join' queries that involve simple selection, projection and join operations on one or more data streams. The focus of this work is on join-ordering and the initial placement of operators. Note that it may be possible to modify existing deployments to get a better solution. However, such modifications require us to consider the cost of re-configurations and deal with translation of state as well. We leave such possibilities for the future. We assume stream joins are performed using standard techniques (e.g. doubly-pipelined operators and windows if necessary). We assume that potentially, *any* operator can be deployed at *any* node in the system. Given a query, there could possibly be multiple execution plans that the system could follow to produce results. We assume that all such plans produce equivalent results.

Query-Optimization Problem: Given a query Q to be deployed over a network N, and a (possibly empty) set of existing query deployments $D = \{D_1,\ldots,D_n\}$, find a query tree $\{p_i^Q\}$ and a deployment $M(p_i^Q, N)$ for Q such that $Cost(M(p_i^Q, N))$ is minimum over all possible query trees and deployments.

7.3 Query Optimization Algorithms

In order to choose an optimal execution plan, traditional query optimizers typically perform an exhaustive search of the solution space using dynamic programming, estimating the cost of each plan using pre-computed statistics. Lemma 1 shows the size of the exhaustive search space for the query optimization problem in distributed data stream systems.

Lemma 1. *Let $\mathcal{Q}$ be a query over K (> 1) sources to be deployed on a network with*

N nodes. Then the size of the solution space of an exhaustive search is given by:

$$\mathcal{O}_{\texttt{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}$$

Proof. We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle \mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_K \rangle$. The search space is given by all plans (permutations of join-orders) and all possible placements of each plan. The number of query re-writings i.e. an enumeration of both linear and bushy joins of K streams is given by:

$$\binom{K}{2} \times \binom{K-1}{2} \times \ldots \times \binom{2}{2} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right)$$

The number of network placements of the joins in a query with K streams in a network of size N is given by $N^{(K-1)}$. Thus, the exhaustive search space $\mathcal{O}_{exhaustive}$ given by:

$$\mathcal{O}_{\texttt{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}$$

□

As shown in the Lemma 1, the search space increases exponentially with an increase in the query size. Certainly, in a system with thousands of nodes such an exhaustive search even with dynamic programming would be infeasible. We now present our optimization infrastructure and heuristics for finding good plans and deployments while avoiding the cost of exhaustive search. Note that in the case of distributed query optimization, dynamic programming does not result in any pruning of the search space without loss of optimality since the query optimization problem in distributed data stream systems does not exhibit the property of optimal substructure [86].

7.3.1 Optimization infrastructure

In this section we describe the key components of our optimization infrastructure - *hierarchical network partitions* that guide our planning heuristics and *stream advertisements* that facilitate operator reuse. We can tune the hierarchy to trade-off between search space size and sub-optimality by adjusting the max_{cs} parameter, which

is the maximum number of nodes allowed per network partition. This trade-off is complex, and is analyzed in detail in our discussion of the Top-Down (Section 7.3.2) and Bottom-Up (Section 7.3.3) algorithms.

7.3.1.1 ***Hierarchical Network Clusters***

We organize physical network nodes into a virtual clustering hierarchy, by clustering nodes based on link costs which represents the cost of transmitting a unit amount of data across the link. We refer to this clustering parameter as ***inter-node/cluster traversal cost***. Nodes that are close to each other in the sense of this clustering parameter are allocated to the same cluster. We allow no more than max_{cs} nodes per cluster.

Clusters are formed into a hierarchy. At the lowest level, i.e. *Level 1*, the physical nodes are organized into clusters of max_{cs} or fewer nodes. Each node within a cluster is aware of the inter-node traversal cost between every pair of nodes in the cluster. A single node from each cluster is then selected as the *coordinator* node for that cluster and promoted to the next level, *Level 2*. There may be a set of nodes in a cluster, each of which qualifies to be a representative coordinator node as long as they do not modify the ordering of Euclidean distances between the clusters. Nodes in *Level 2* are again clustered according to average inter-node traversal cost, with the cluster size again limited by max_{cs}. This process of clustering and coordinator selection continues until *Level N* where we have just a single cluster. An example hierarchy is shown in Figure 82.

As a result of our clustering approach we can determine the upper bounds on the cost approximation at each level, which is described in the following theorem.

Theorem 1. *Let d_i be the maximum intra-cluster traversal cost at level i in the network hierarchy and $c_{act}(v_{nj}, v_{nk})$ be the actual traversal cost between the network nodes v_{nj} and v_{nk}. Then the estimated cost between network nodes v_{nj} and v_{nk}*

at any level l, represented as $c^l_{est}(v_{nj}, v_{nk})$, is related to the actual cost as follows: $c_{act}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=1}^{i<l} 2d_i$

Proof. At a particular level l the cost of traversal between nodes v_{nj} and v_{nk} is given by the inter-node traversal cost between the nodes representing them at that level. However, each node will be resolved to some node in the underlying cluster at level $l-1$. Inter-node traversal costs at this level are bounded by the value d_{l-1}. Therefore, nodes at level $l-1$ will be at most d_{l-1} distance away from the node representing them at level l. Thus the inter-node traversal costs between nodes v_{nj} and v_{nk} at level $l-1$ is given by: $c^{l-1}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + 2d_{l-1}$. Similarly,

$$c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^{l-1}_{est}(v_{nj}, v_{nk}) + 2d_{l-2}$$

$$\Rightarrow c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=l-2}^{i<l} 2d_i$$

This process continues down the hierarchy. At level 1, the estimated cost is the same as the actual traversal cost and thus is at most $\sum_{i=1}^{i<l} 2d_i$ less than the actual cost. □

7.3.1.2 Discussion

The hierarchical organization is created and maintained as follows. When a node joins the infrastructure, it contacts an existing node that forwards the join request to its coordinator. The request is propagated up the hierarchy and the top level coordinator assigns it to the top level node that is closest to the new node. This top level node passes the request down to its child that is closest to the new node. The child repeats the process, which continues until the node is assigned to a bottom level cluster. Note that similar organization strategies appear in other domains such as hierarchies for internet routing [101] and for data aggregation in sensor networks [44]. However, to the best of our knowledge we are the first to use such hierarchical approximations and clustering techniques for distributed continual query optimization. The virtual hierarchy is robust enough to adapt as necessary. It can handle both node joins and

departures at runtime. Failure of coordinator nodes can be handled by maintaining active back-ups of the coordinator node within each cluster. However, the issue of fault tolerance is beyond the scope of this work.

In situations where nodes are distributed such that it is easy to find clusters meeting the clustering condition of *inter-cluster distances* >> *intra-cluster distances*, the planning decisions are likely to be less sensitive to the selection of coordinator nodes. However, situations where nodes in the entire system either are all widely distributed or are all close to one another in terms of network cost, may result in loosely-defined clusters, which further impact the quality of coordinator nodes selected. Such situations are relatively rare. Also in the worst case, it is possible to choose appropriate values for max_{cs} in order to improve accuracy of the planning process. Also, note that since the hierarchy is only a virtual structure and since query deployment times (Section 7.4.6) are in the order of seconds, when it is known *a priori* that the node distribution in the network might possibly result in loosely-defined clusters, it may be beneficial to compare planning decisions across multiple hierarchical structures with different values of max_{cs}.

7.3.1.3 ***Stream Advertisements***

Stream Advertisements are used by nodes in the network to advertise the stream sources available at that node. A node may advertise two kinds of stream sources - *base stream sources* and *derived stream sources*. We observe that each sink and deployed operator is a new stream source for the data computed by its underlying query or sub-query. We refer to these stream sources as derived stream sources and the original stream sources as base stream sources. As a result of the advertisement of derived stream sources, nodes are now aware of operators that are readily available at multiple locations in the network and can be reused with no additional cost involved for transporting input data. The stream advertisements are aggregated by the

coordinator nodes and propagated up the hierarchy. Thus the coordinator node at each level is aware of all the stream sources available in its underlying cluster. Advertisements of derived stream sources are key to operator reuse in our algorithms. The advertisements are one-time messages exchanged only at the initial time of operator instantiation and deployment.

7.3.2 The Top-Down Algorithm

The *Top-Down* algorithm bounds sub-optimality by making deployment decisions using bounded approximations of the underlying network; specifically, each coordinator's estimate of the distance between its cluster and other clusters. The algorithm works as follows: The query $\mathcal{Q}$ is submitted as input to the top level (say level t) coordinator. The coordinator exhaustively constructs the possible query trees for the query, and then for each such tree constructs a set of all possible node assignments within its current cluster. The cost for each assignment is calculated and the assignment with least cost is chosen. An assignment of operators to nodes partitions the query into a number of views, each allocated to a single node at level t. Each node is then responsible for instantiating such a view using sources (base or derived) available within its underlying cluster. The allocated views act as the queries that are again deployed in a similar manner at level $t-1$, with all possible assignments within the cluster being evaluated exhaustively and the one with the least cost being chosen. This process continues until level 1, which is the level at which all the physical nodes reside, and operators are assigned to actual physical nodes. Since each level has fewer nodes and operators are progressively partitioned and assigned to different cluster coordinators, the search space is still much smaller compared to a global exhaustive search (even using DP). Whenever a coordinator is exhaustively mapping a portion of the query, it considers both base and derived streams available locally. Thus, operator reuse is automatically considered in the planning process. In particular, if the

coordinator calculates that reuse would result in the best plan, derived streams are used; otherwise, operators are duplicated.

7.3.2.1 Bounding Search Space with the Top-Down Algorithm

In a network of N nodes that is organized into a clustering hierarchy, for a query $\mathcal{Q}$ over K (> 1) sources the search space depends on the clustering parameter $\texttt{max}_{cs}$ and the resulting height $\texttt{h}(\approx log_{max_{cs}} N)$ of the hierarchy. We define the following:

$$\beta = h(\frac{max_{cs}}{N})^{K-1} \tag{4}$$

In Theorem 2 we prove that β represents the upper bound on the ratio of the search space of the Top-Down algorithm to that of the exhaustive search. Note that as the ratio $\frac{max_{cs}}{N}$ decreases linearly, β decreases exponentially. When $max_{cs} << N$, β is orders of magnitude less than 1 and thus, the Top-Down algorithm is orders of magnitude cheaper than exhaustive search. For example, for a query over 4 streams on a network with 1000 nodes, with a max_{cs} value of 100, $\beta \approx 0.0015$.

Theorem 2. *Let $\mathcal{Q}$ be a query over K (> 1) sources to be deployed on a network with N nodes. Let the clustering parameter used to organize the network into a hierarchical cluster be max_{cs} and let the height of such a hierarchical cluster be h. If $\mathcal{O}_{top-down}$ represents the size of the solution space for the top-down algorithm, then $\mathcal{O}_{top-down} \leq \beta \mathcal{O}_{exhaustive}$*

Proof. The worst case search space of the Top-Down algorithm results when all query tree nodes (sources, operators and sink) appear in the same cluster. As in the case of Theorem 4 we compute this search space by considering all possible query trees and all possible placements of operators within a single cluster at each level.

We are given a network with $\texttt{N}$ nodes, and a query $\mathcal{Q}$ over K streams $\langle \texttt{S}_1, \texttt{S}_2, \ldots \texttt{S}_K \rangle$. At the top level t, we have K streams. At any level the search space is given by all

plans (permutations of join-orders) and all possible placements of each plan. Therefore, the search space $\mathcal{O}_t$ at level t is given by:

$$\mathcal{O}_t = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)}$$

In the worst case the coordinator at each level may assign all streams to a single partition thereby causing the search space to be the same at all levels. Thus, $\mathcal{O}_{top-down}$ is given by

$$\mathcal{O}_{top\text{-}down} \leq h \times \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)} \quad (5)$$

Thus from Equation 5 and Lemma 1 we have $\mathcal{O}_{top-down} \leq \beta \mathcal{O}_{exhaustive}$. □

7.3.2.2 Sub-Optimality in the Top-Down Algorithm

The Top-Down algorithm works by propagating a query down the network hierarchy, described in Section 7.3.1.1. Given a query $\mathcal{Q}$, at each level a coordinator chooses a query plan and a deployment with the least cost for the sub-query assigned to it. As the network approximations increase at higher levels of the hierarchy (refer Theorem 1), it follows that the maximum approximation is incurred at the top most level of the hierarchy. Therefore the Top-Down algorithm is most sub-optimal when all the edges of the query plan are deployed at the top-most level. The following theorem establishes the bounds on sub-optimality of the top-down algorithm as compared to an optimal deployment.

Theorem 3. *A query $\mathcal{Q}$ deployed using the Top-Down algorithm over a network N is no more than $\sum_{e_k \in E^Q}(\sum_{i=1}^{i<h} 2d_i) \times s_k$ sub-optimal compared to the optimal deployment of query $\mathcal{Q}$ over the same network N, where h is the number of levels in the network hierarchy of N, E^Q represents the set of edges of the tree chosen for query $\mathcal{Q}$, d_i is the maximum intra-cluster traversal cost at level i and s_k is the stream rate for the k^{th} edge e_k.*

Proof. The maximum sub-optimality of the Top-Down algorithm occurs only when all the edges of the tree chosen for $\mathcal{Q}$ are mapped to the top-most level, i.e. no two nodes (operators or sources or sinks) lie in the same underlying cluster. The proof then follows directly from Theorem 1. □

The point of this proof is to establish a relationship between the hierarchical cluster structure and the sub-optimality of the resulting solution. The intra-cluster traversal cost increases with the levels of the hierarchy since the hierarchical structure provides a more approximate representation of the network at higher levels. The height of the hierarchical structure in turn, is determined by the max_{cs} parameter and the density of node distributions in the network. The proof shows that the sub-optimality can increase with increasing number of levels in the hierarchy and decreasing cluster density. This proof, along with Theorem 2, can help decide an optimization hierarchy that offers a desirable trade-off between search space and optimality for a given network. We present empirical results that corroborate these theorems in Section 7.4.

7.3.3 The Bottom-Up Algorithm

We now describe the *Bottom-Up* algorithm which propagates queries up the hierarchy, progressively constructing complete query execution plans. Unlike the Top-Down approach, the Bottom-Up algorithm does not provide a good bound on the sub-optimality of the solution. However, in return, the Bottom-Up approach is usually able to further reduce the search space compared to the Top-Down algorithm. Thus, in situations where quick planning is needed, the Bottom-Up algorithm may be appropriate, perhaps to be replaced later with a Top-Down deployment.

Queries are registered at their sink. When a new query $\mathcal{Q}$ over base stream sources arrives at a sink at *Level 1*, the sink informs its coordinator at *Level 2*. The coordinator rewrites the query $\mathcal{Q}$ as $\mathcal{Q}'$ with respect to two views - $V^{\mathcal{Q}}_{local}$ and $V^{\mathcal{Q}}_{remote}$ where $V^{\mathcal{Q}}_{local}$ is composed of base and derived sources available locally within the cluster and $V^{\mathcal{Q}}_{remote}$

is composed of base sources not available locally. The coordinator deploys V^{Q}_{local} within the current cluster, and then advertises V^{Q}_{local} as a derived stream at the next level. The above rewriting causes any joins between local streams to be deployed within the current cluster, leaving the joins of local streams with remote streams or joins between remote streams to be deployed further up in the hierarchy. The coordinator then requests $\mathcal{Q}'$ from its next level coordinator. This process continues up the hierarchy, with the query $\mathcal{Q}'$ progressively decomposed into locally available views and remote views and the re-written query being requested from the current cluster's coordinator. The coordinator performs an exhaustive search only within its underlying cluster to determine an optimal execution plan for V^{Q}_{local}. The search space is limited to a single network partition and the local sub-query.

Operator reuse is taken into consideration by coordinators by taking into account all possible constructions of V^{Q}_{local} that utilize derived sources within the cluster. When using a derived stream source, communication costs for transporting input data to the node that is the source of the derived stream, and processing costs for computing the result of the operator are incurred only once. Note that if it is cheaper to duplicate operators rather than reuse existing ones, the coordinator will do so.

For example, assume that query $\mathcal{Q}1$ described in Section 7.2.1 arrives at a node which belongs to a cluster where sources FLIGHTS and CHECK-INS are available locally. The Bottom-Up algorithm proceeds as follows. $\mathcal{Q}1$ is partitioned into local and remote views V^{Q}_{local} and V^{Q}_{remote} respectively, where V^{Q}_{local} consists of sources FLIGHTS and CHECK-INS and V^{Q}_{remote} consists of source WEATHER as shown below.

V^{Q}_{local}: `SELECT FLIGHTS.STATUS, CHECK-INS.STATUS, FLIGHTS.DESTN FROM FLIGHTS, CHECK-INS`
`WHERE FLIGHTS.DEPARTING='ATLANTA' AND FLIGHTS.NUM = CHECK-INS.FLNUM`
`AND FLIGHTS.DP-TIME - CURRENT_TIME < 12:00:00`

V^{Q}_{remote}: `SELECT WEATHER.FORECAST, WEATHER.CITY FROM WEATHER`

Query $\mathcal{Q}1$ is then rewritten (with V^{Q}_{remote} expanded) as $\mathcal{Q}1$' given by:

Q1': SELECT FLIGHTS.STATUS, WEATHER.FORECAST, CHECK-INS.STATUS FROM $V^{\mathcal{Q}}_{local}$, WEATHER WHERE FLIGHTS.DESTN = WEATHER.CITY

$V^{\mathcal{Q}}_{local}$ is deployed within the current cluster using any locally available derived streams, if required. Thus the join between sources FLIGHTS and CHECK-INS is deployed locally within the cluster. $V^{\mathcal{Q}}_{local}$ is then advertised as a derived stream and query $\mathcal{Q}1$' is propagated to the next level for deployment at some higher level cluster. This process continues up the hierarchy until all sources for $\mathcal{Q}1$' are found locally in some cluster. At that point, all operators are placed at appropriate representative nodes and passed down the hierarchy for placement on an actual physical node.

7.3.3.1 Bounding Search Space with the Bottom-Up Algorithm

Recall our definition of β in Section 7.3.2.1. We now show in Theorem 4 that β also represents the the upper bound on the ratio of the search space of the Bottom-Up algorithm to that of the exhaustive search. Although the worst case bounds are the same for the two algorithms, in Section 7.4.1 we show experimentally that the Bottom-Up algorithm examines a smaller search space in the average case. As before, when $max_{cs} << N$, β is orders of magnitude less than 1. Thus, the search space of the Bottom-Up algorithm is orders of magnitude less than the exhaustive search space.

Theorem 4. *Let $\mathcal{O}_{bottom-up}$ represent the size of the solution space for the bottom-up algorithm. Then, $\mathcal{O}_{bottom-up} \leq \beta\mathcal{O}_{exhaustive}$*

Proof. We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle S_1, S_2, \ldots S_K \rangle$. Let σ_i represent the number of streams, for query Q, requested by a node at level $i-1$ and available within the partition of a single coordinator at level i. Also, $\sigma_1 + \ldots + \sigma_h = K$. Let α_i represent the actual number of streams to be considered at level i. At the level where $V^{Q_i}_{remote}=\phi$, $\alpha_i=\sigma_i$. At all other levels $\alpha_i=\sigma_i+1$ to take into consideration the presence of the remote stream $V^{Q_i}_{remote}$. Thus, $\alpha_1 + \ldots + \alpha_h \leq K + h$. At any level the search space is given by all plans (permutations of join-orders) and

all possible placements of each plan. Thus the search space $\mathcal{O}_i$ at level i with α_i streams is given by:

$$\mathcal{O}_i \leq \left(\frac{\alpha_i! \times (\alpha_i - 1)!}{2^{\alpha_i - 1}} \right) \times (max_{cs})^{(\alpha_i - 1)}$$

Thus the total search space in the Bottom-Up algorithm, $\mathcal{O}_{bottom-up}$ for a query Q is:

$$\mathcal{O}_{bottom-up} \leq \sum_{i=1}^{i \leq h} \mathcal{O}_i \tag{6}$$

Since $\forall$ i, $\alpha_i \leq K$, and not all $\alpha_i = K$ (since the query is totally composed of only K streams and streams found at each level are different), we have

$$\mathcal{O}_{bottom-up} \leq \sum_{i=1}^{i \leq h} \left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)} \leq$$

$$\left(\frac{K! \times (K-1)!}{2^{K-1}} \right) \times (max_{cs})^{(K-1)} \times (h)$$

Thus, from Lemma 1 and the above equation we have: $\mathcal{O}_{bottom-up} \leq \beta \mathcal{O}_{exhaustive}$. □

7.3.3.2 Sub-Optimality in the Bottom-Up Algorithm

The Bottom-Up algorithm partitions queries into locally and remotely available views as the result of which all local sources are now represented as a single source deployed at the coordinator. This results in a pruning of the plan search space since only join orderings between streams available within a single cluster are considered. While the Bottom-Up algorithm can find optimal join orderings among local sources, the resulting overall execution plan may be sub-optimal. As an example, consider a high volume stream S_r that is in a remote cluster, and which we want to join with two low volume, local streams S_1 and S_2. An overall optimal plan might be to perform a selective join between S_r and S_1 in the remote cluster, and then stream the resulting (low-volume) intermediate results to the local cluster for joining with S_2. The Bottom-Up algorithm will not consider this plan. However, note that the Bottom-Up algorithm may instead stream the results of $S_1 \bowtie S_2$ to the remote cluster for joining with S_r.

In the worst case the resulting deployment may be arbitrarily bad making it impossible to bound the sub-optimality of the algorithm. However, note that the situations under which this algorithm performs badly can be well characterized: it performs badly when streams available remotely have significantly higher data rates than those available close to the sink. In order to overcome this limitation, we next present heuristic-based hybrid algorithms that aim at improving the planning process of the Bottom-Up algorithm while retaining the advantage of a small search space.

7.3.4 The NPC Algorithm

In this section we introduce a heuristic based hybrid algorithm that combines the advantages of reduced search space from the Bottom-Up algorithm and improved query planning from the Top-Down algorithm. We present a heuristic based hybrid algorithm - the *Net Present Cost (NPC)* algorithm. The NPC algorithm is a probabilistic algorithm that uses cost estimates of local and delayed query planning decisions to guide the planning process. A decision to choose a join order at the current level may result in a penalty if a poor join order is chosen. On the other hand, delaying the planning process to the next level will result in wasted planning time and also a possible increase in cost due to coarser approximations. In the NPC algorithm, the query planning process is delayed to the next level in the hierarchy only when the cost of making a decision at the current level exceeds the estimated cost of delaying the decision to the next level in the hierarchy. We next describe the computation of local and delayed cost estimates.

Let V^{Q}_{local} and V^{Q}_{remote} represent the sub-queries composed of sources available locally and remotely respectively. In the worst case, if a poor join order is chosen as the result of making a local decision, it may result in a high-volume remote query that needs to be streamed to the current cluster for joining with the local query. Let $\mathcal{C}_l$ denote the cost incurred immediately within a cluster at level l by making a join

ordering decision locally at level l. We use the term 'net present cost', Γ_l, to indicate all present and future costs that may be incurred as the result of making a local join ordering decision at the level l. Then the estimated net present cost Γ_l is computed as follows:

$$\Gamma_l = \mathcal{C}_l + \sum_{i=l+1}^{h} p_i \times \lambda(\mathtt{V}^{\mathtt{Q}}_{remote}) \times d_i$$

where p_i represents the probability of finding $\mathtt{V}^{\mathtt{Q}}_{remote}$ at the level i, $\lambda(\mathtt{V}^{\mathtt{Q}}_{remote})$ represents the data rate of stream $\mathtt{V}^{\mathtt{Q}}_{remote}$, h represents the height of the hierarchy and d_i represents the maximum intra-cluster traversal cost at level i. The probability of finding the remote query at a particular level is computed based on the fraction of network visible at that level assuming sources are likely to appear anywhere within the network. In this expression, the first term represents the present cost and the second term represents all future costs likely to be incurred at higher levels in the hierarchy. Note that Γ_l is a conservative estimate of the future costs aimed mainly at penalizing query partitioning where the join order results in a high-volume remote stream that needs to be transported across longer distances.

The cost of delaying the decision to the next level, i.e. cost Ω_l at level l is computed as follows:

$$\Omega_l = \sum_{i=l+1}^{h} p_i \times \lambda(\mathtt{Q}) \times d_i$$

In order to compute Ω_l we compute the expected future costs of delaying the query partitioning decision to the next level. The NPC algorithm then performs query partitioning at the current level l if $\Omega_l \geq \Gamma_l$. Unlike the other algorithms, the NPC algorithm requires knowledge of the hierarchical structure in terms of height, number of nodes in a cluster and maximum intra-cluster traversal costs at each level. It also requires knowledge of join selectivities.

Since the NPC algorithm attempts to avoid poor join orders, it is expected to perform better than the Bottom-Up algorithm. However, since it continues to make

query partitioning decisions based only on efficiency of join orders, oblivious to the availability of reuse opportunities, it is expected to produce less efficient deployments as compared to the Top-Down algorithm. The NPC algorithm performs query partitioning when it perceives that partitioning the query at some level is beneficial. As a result it is more effective in reducing the search space compared to the Top-Down algorithm.

7.4 Experiments

We present both simulation based experiments and prototype experiments conducted on Emulab [7] using IFLOW [90]. Our experiments focus on (1) the effect of the max_{cs} clustering parameter on the trade-off between sub-optimality and search space (2) the effectiveness of our algorithms as compared to existing approaches (3) and the efficiency of our algorithms compared to an optimal solution computed through an exhaustive search. Our experiments show that our algorithms result in acceptable sub-optimality: the Top-Down algorithm is sub-optimal by only 10% and the Bottom-Up algorithm by 34% while exploring less than 1% of the total search space. At the same time, our algorithms clearly outperform existing approaches. For example, the Bottom-Up algorithm reduces cost by nearly 25% when compared to the *In-network* [32] algorithm while exploring only a small fraction of the search space. Also, the NPC algorithm allows us to further fine tune the trade-off between search space and sub-optimality and help us achieve plans that were close to the Top-Down algorithm in optimality and Bottom-Up algorithm in search space.

7.4.1 Experimental Setup

Our simulation experiments were conducted over transit-stub topology networks generated using the standard tool, the GT-ITM internetwork topology generator [159]. Most experiments were conducted using a 128 node network, with a standard Internet-style topology: 1 transit (e.g. "backbone") domain of 4 nodes, and 4 "stub" domains

(each of 8 nodes) connected to each transit domain node. Link costs (per byte transferred) were assigned such that the links in the stub domains had lower costs than those in the transit domain, corresponding to transmission within an intranet being far cheaper than long-haul links. As described in Section 7.2.2, we adopt the 'network usage' metric [109] to compute costs of query deployments. Recall that, the network usage $u(q)$ of a query q represents the total amount of data that is in-transit for a query at any given instant.

As described in Section 7.3.1.1 our network is organized into a virtual clustering hierarchy based on link costs which represent the cost of transmitting a unit amount of data across the link. We used the K-Means [82] clustering in order to create the clustering hierarchy.

7.4.1.1 Workloads

We evaluate our approaches using two different workloads: a synthetic workload generated using a random workload generator and a real enterprise workload based on the enterprise operational information system used by Delta Airlines [104, 90]. We used a synthetic workload so that we could experiment with a large variety of stream rates, query complexities, and operator selectivities. Our synthetic workload was generated using a uniformly random workload generator. The workload generator generated stream rates, selectivities and source placements for a specified number of streams according to a uniform distribution. It also generated queries with the number of joins per query varying within a specified range (2-5 joins per query) with random sink placements.

The enterprise workload is a real-world workload consisting of gate-agent, terminal and monitoring queries posed as part of the day-to-day operations of Delta Air Lines' enterprise operational information system. The query workload is based on the 5 query sources whose characteristics are shown in Table 83. Each update record

was assumed to be of the same size (100 bytes). Each query definition includes window (RANGE and SLIDE i.e. the window size and frequency of computation of results respectively) specifications for each of the input streams.

Gate agent queries, which originate at the gate of departure of a flight, constitute 80% of the workload and the SLIDE for these queries is set to 1 minute. The queries use the following template.

Q1: SELECT FL.GATE, BG.STATUS, CI.STATUS
FROM FLIGHTS FL [RANGE 5 MIN], BAGGAGE BG [RANGE 1 MIN], CHECK-INS CI [RANGE 1 MIN]
WHERE FL.NUM = CI.FLIGHT AND FL.NUM = BG.FLIGHT AND FL.NUM = ?;

Terminal queries follow the template of query Q2 given below and represent 15% of the workload. The results of these queries, evaluated every minute are streamed to overhead terminal displays.

Q2: SELECT FL.NUM, FL.GATE, BG.AREA, CI.STATUS, WR.FORECAST
FROM FLIGHTS FL [RANGE 5 MIN], WEATHER WR [RANGE 5 MIN], CHECK-INS CI [RANGE 1 MIN], BAGGAGE BG [RANGE 1 MIN] WHERE FL.DEST = WR.CITY AND FL.NUM = CI.FLIGHT AND FL.NUM = BG.FLIGHT AND FL.TERMINAL = ? AND FL.CARRIER_CODE = ‘‘DL’’;

Finally, the last 5% of the workload represent long-running ad-hoc monitoring queries over any combination of the 5 sources. For these queries, window ranges and slides are uniformly distributed between [1-5] minutes for all streams. Note that each gate agent, terminal and monitoring query may have unique selection predicates. In the current work, our focus is on join ordering and discovering join reuse opportunities. In our simulation experiments, sharing join operators between queries with different selection criteria over the input stream is implemented by modifying selection predicates at runtime. In our prototype implementation, the selections are instantiated as parameterized filters [61], and this reduces the task of selection predicate modification

Source	Rate	Selectivity
Flights	1500/5 min	1/flight/5 min
Check-ins	240/min	2/flight/min
Baggage	500/min	4/flight/min
Weather	450/5 min	1/dest/5 min
Sales	70/min	1/dest/min

Figure 83: Enterprise Workload

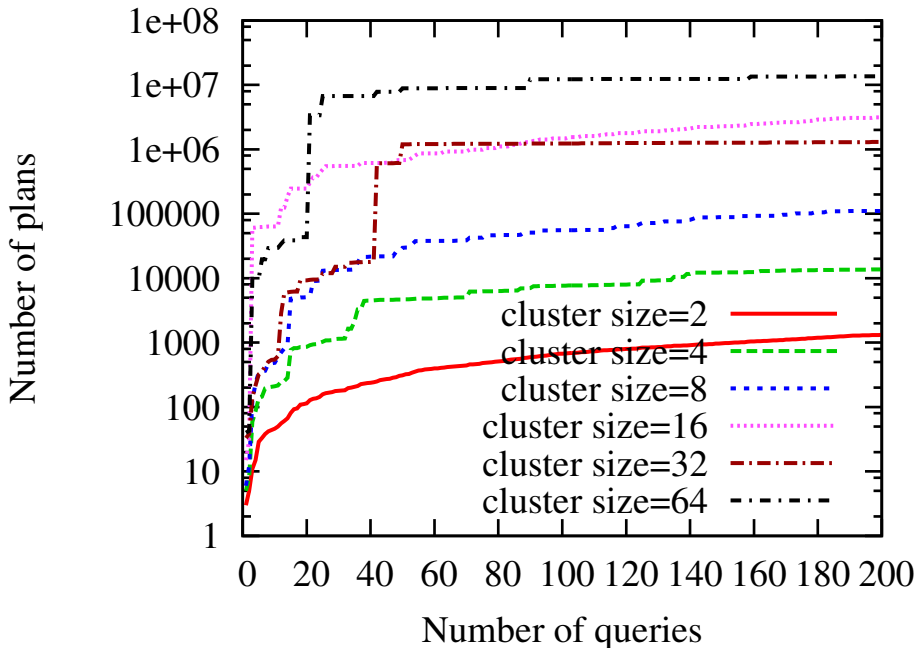


Figure 84: Bottom-Up: Plans

to the task of changing the parameter associated with the filter. While evaluating the benefit of reusing a join operator, the cost of projecting additional columns and relaxing the filter specifications upstream are also taken into consideration and operators are reused only when the resulting cost is less than that of deploying a new operator. The synthetic workload is used to study the trade-off between sub-optimality and search space in our algorithms. We use the enterprise workload consisting of 300 queries to compare the performance of our algorithms with existing techniques in a realistic setup.

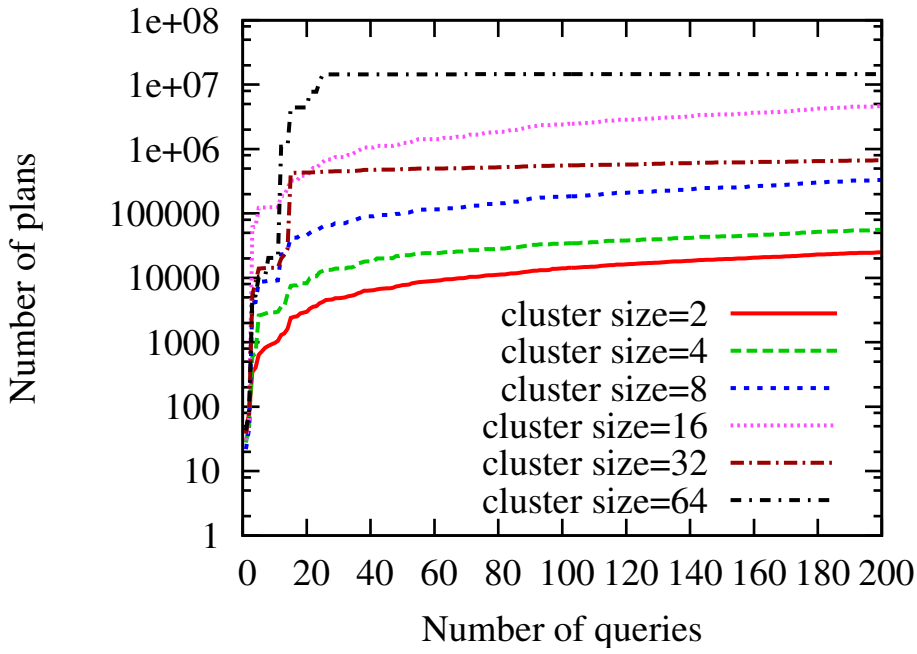


Figure 85: Top-Down: Plans

7.4.2 Tuning Cluster Size: Trade-off between Sub-Optimality and Search Space

An exhaustive search of all possible query plans and all possible placement of operators may not be feasible as network size increases. For example, an exhaustive search on a 128 node network for the deployment of a single query over 5 stream sources required enumeration of approximately 4.83×10^{10} plans that took nearly 3 hours to complete. In this section we demonstrate how the max_{cs} parameter can be used to tune the trade-off between the sub-optimality of the heuristic and minimizing the search space. The experiments were conducted using the synthetic workload described in Section 7.4.1.

7.4.2.1 Effect of Cluster Size on Search Space

In this experiment we studied the effect of the cluster size parameter max_{cs} on the search space with the Bottom-Up and Top-Down algorithms. Figure 84 and Figure 85 depict the cumulative number of plans examined on a log scale, with varying max_{cs}

for the Bottom-Up and Top-Down algorithms respectively. As the figure shows, the number of plans increases as max_{cs} increases.

Interestingly, we notice that in both algorithms, a max_{cs} value of 32 results in a smaller search space than a value of 16. In both cases, the hierarchy had the same number of levels, but in the case of $max_{cs} = 32$, the upper level clusters were smaller (since the lower level clusters were larger.) Since many sources are found remotely, most of the planning is done at the upper levels, and having small cluster sizes at those levels results in a smaller search space overall.

In contrast, a max_{cs} value of 64 resulted in the maximum search space, nearly an order of magnitude larger than $max_{cs} = 16$. This is a straightforward effect of the increased probability of finding sources in a larger cluster. For example a query over 4 streams, with all streams found within a 57 node *Level 1* cluster, considers as many as 3.3×10^6 deployments.

In general, the Bottom-Up algorithm considers on an average 67% fewer plans than the Top-Down algorithm. The exception to this rule occurs when the virtual hierarchy structure is an unbalanced structure with very few nodes at the top, as in the case of a $max_{cs} = 64$ (3 top-level nodes) and $max_{cs} = 32$ (5 top-level nodes). In these cases, due to the small cluster size at the level where the query is partitioned the Top-Down algorithm has a smaller search space than the Bottom-Up algorithm.

7.4.2.2 Effect of Cluster Size on Cost

Figure 86 shows the cumulative deployed cost per unit time of queries deployed incrementally using the Bottom-Up algorithm for different values of the max_{cs} parameter. It can be noticed that cost decreases as the max_{cs} value is increased. For example, a max_{cs} value of 64 results in a 21% decrease in cost compared to a max_{cs} value of 8. With smaller cluster sizes, the number of levels in the hierarchy increases. As a result, more deployments are computed at higher levels resulting in greater approximations.

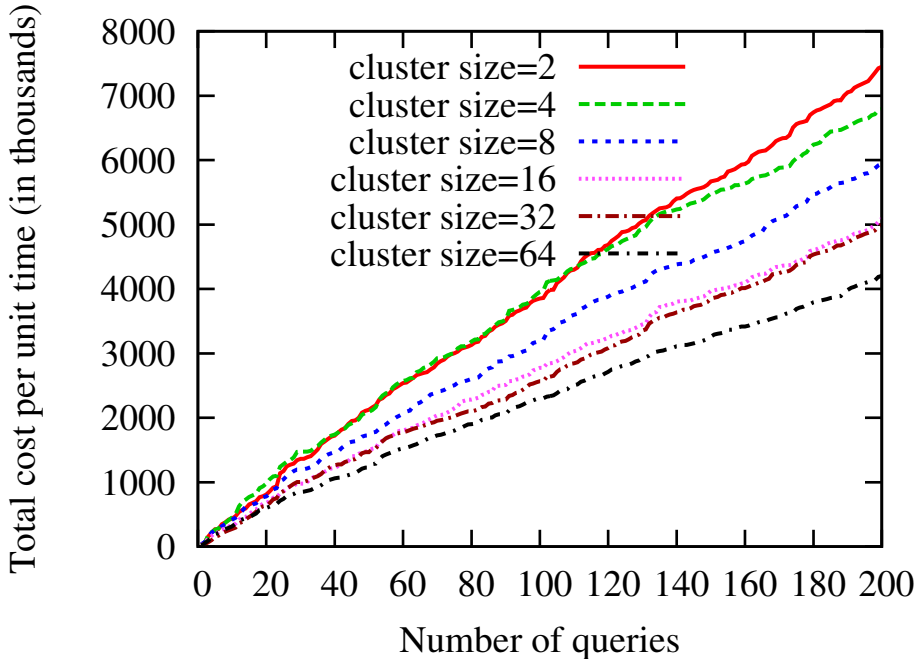


Figure 86: Bottom-Up: Cost

To summarize, in terms of sub-optimality, fewer levels and more nodes per level is best. In terms of search space, fewer nodes per level is best. A useful guideline for choosing max_{cs} for the Bottom-Up algorithm is:

- Choose the largest value of max_{cs} that results in a search space (Theorem 4) that is acceptable.

Figure 87 shows the effect of the cluster size parameter max_{cs} on the cost in the Top-Down algorithm. Note that large values of max_{cs} (> 4) result in deployed costs that are close to each other. The Top-Down algorithm considers all possible operator orderings at the top-most level (regardless of max_{cs}). This results in a good and mostly 'similar' choice of operator ordering for a range of max_{cs} values. However, if max_{cs} is too small, there are many levels in the hierarchy and each level adds more inaccuracy to the approximation. Hence, a useful guideline for the Top-Down algorithm is:

- Choose the smallest value of max_{cs} that is large enough so that the height of the

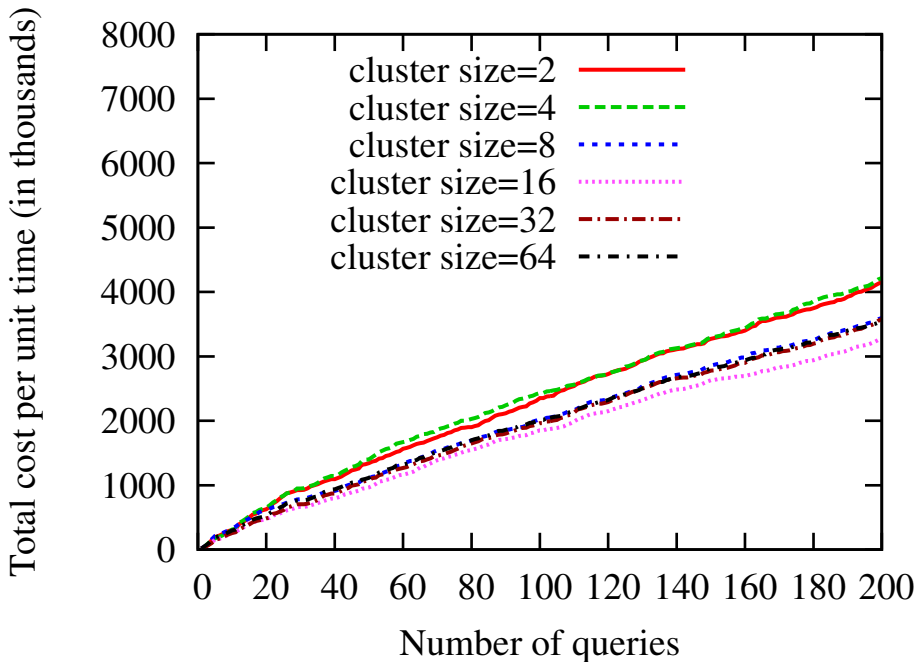


Figure 87: Top-Down: Cost

hierarchy results in reasonable sub-optimality (based on Theorem 3).

7.4.3 Efficiency of NPC Algorithm

The NPC algorithm allows us to further fine tune the trade-off between search space and sub-optimality. Figure 88 shows the cost with the NPC algorithm with $max_{cs} = 16$ as compared with the Top-Down and Bottom-Up algorithms. We choose to present the graph for this value of max_{cs} since this is the largest value that results in a balanced virtual hierarchical structure with almost full clusters at each level representing the standard behaviors of the Top-Down and Bottom-Up algorithms. As the figure shows, the NPC algorithm results in plans that are sub-optimal by only 1% compared to the Top-Down algorithm. At the same time, as Figure 89 shows, the NPC algorithm explores 14% fewer plans than the Top-Down algorithm. Note that, for each value of max_{cs} where the Top-Down algorithm explored fewer plans than the Bottom-Up algorithm, the NPC algorithm explored only as many plans at

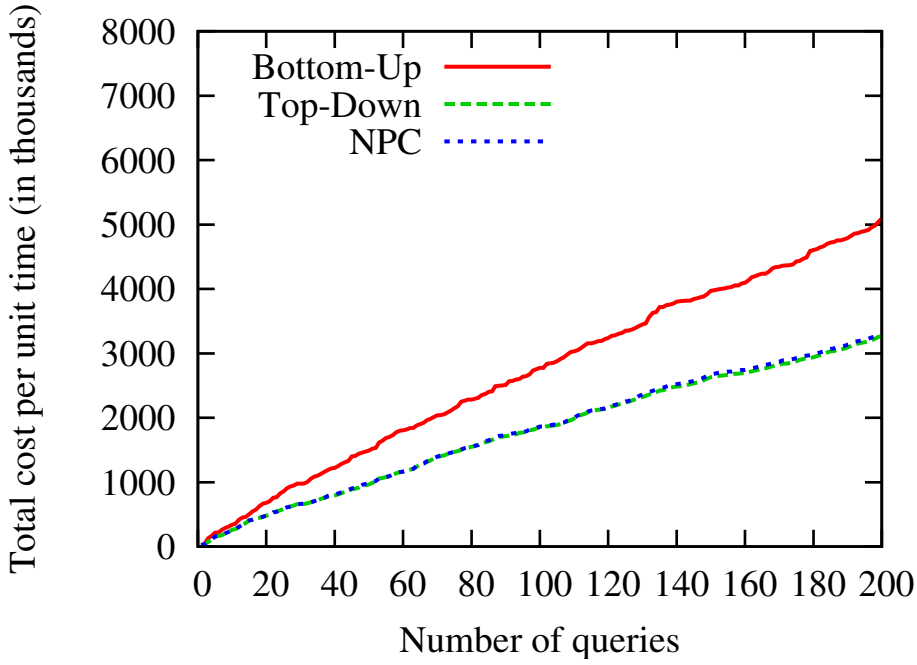


Figure 88: NPC algorithm: Cost

the Top-Down algorithm while still resulting in solutions that were close to the Top-Down algorithm in optimality. While the NPC algorithm avoids poor join orders, it performs less efficiently than the Top-Down algorithm since it is unable to take into account reuse opportunities that may appear at the upper-levels in the hierarchy while deciding on join orders.

7.4.4 Comparison with existing approaches

In this experiment we compare our Top-Down and Bottom-Up approaches with existing approaches - the Relaxation algorithm [109] and *In-network* [32], a network-aware query processing algorithm. Both Relaxation and In-network are phased deployment approaches that first plan and then deploy (see Figure 7.1(a)). Operator reuse was implemented through stream-advertisements. The communication cost of advertisements was negligible compared to the data streams themselves.

Figure 90 and 91 show the cumulative cost of deployments computed using the

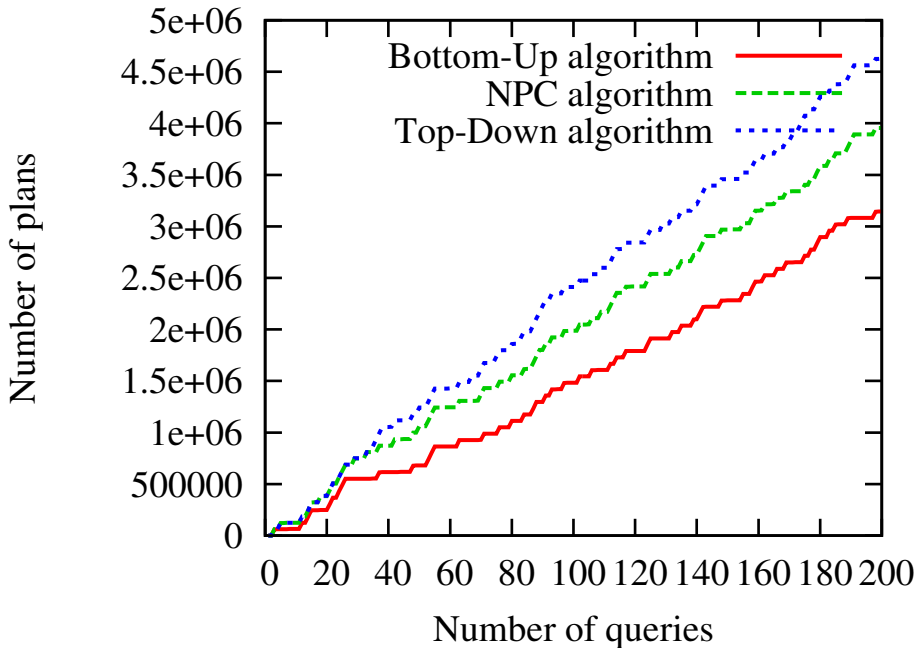


Figure 89: NPC Algorithm: Plans

Top-Down, Bottom-Up and NPC algorithms as compared with the Relaxation and In-network algorithms, using the synthetic and enterprise workload respectively. The graphs also shows the costs of optimal deployments computed using an exhaustive search. Operator reuse was taken into consideration for all algorithms. We used a 3-dimensional cost space for the Relaxation algorithm and considered a virtual hierarchy with max_{cs} 32 for the Top-Down, Bottom-Up and NPC algorithms. We chose this value of max_{cs} based on the above guideline for the Bottom-Up algorithm; and we used the same value for the Top-Down and NPC algorithms to provide an apples-to-apples comparison. In order to correspond with this max_{cs} value, we divided the network into 5 *zones* for the In-network algorithm.

Figure 90 shows that, under the synthetic workload, when compared to the In-network algorithm, the Top-Down algorithm can provide nearly 40% additional cost savings per unit time, and the Bottom-Up algorithm, savings of 27%. Also, note

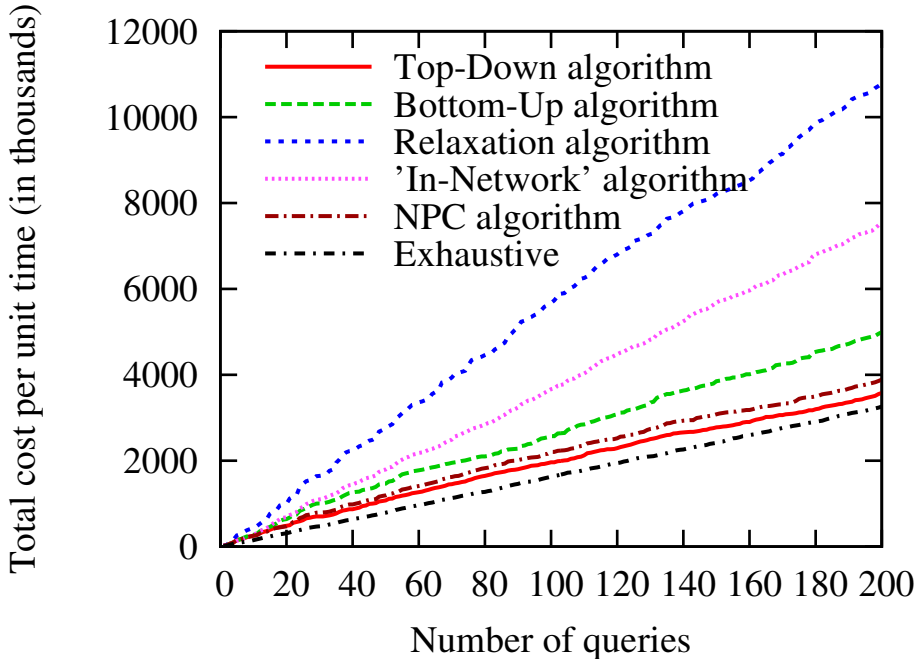


Figure 90: Comparison with existing approaches

that, the search space of the In-network algorithm was nearly 70% that of the Top-Down algorithm and 200% that of the Bottom-Up algorithm. When compared to the Relaxation algorithm, the Top-Down algorithm reduces cost by nearly 59% and the Bottom-Up algorithm by nearly 49%. The search space of the Relaxation algorithm is not directly comparable with that of the Top-Down and Bottom-Up algorithms, due to the variable number of iterations that may be performed for each step of the Relaxation algorithm. In our experiment, the 3-dimensional cost space [59] was calculated using 4000 iterations and we used as many iterations for the Relaxation algorithm and the running time was comparable to that of the Bottom-Up algorithm.

Figure 91 represents the network usage of the deployment algorithms under the enterprise workload and Figure 92 shows the cumulative number of operators deployed after 300 queries under the same workload. Note that, the y-axis of Figure 91 uses a log scale. The graph shows that compared to the In-network algorithm, Top-Down, Bottom-Up and NPC algorithms result in approximately 42%, 25% and 41% cost

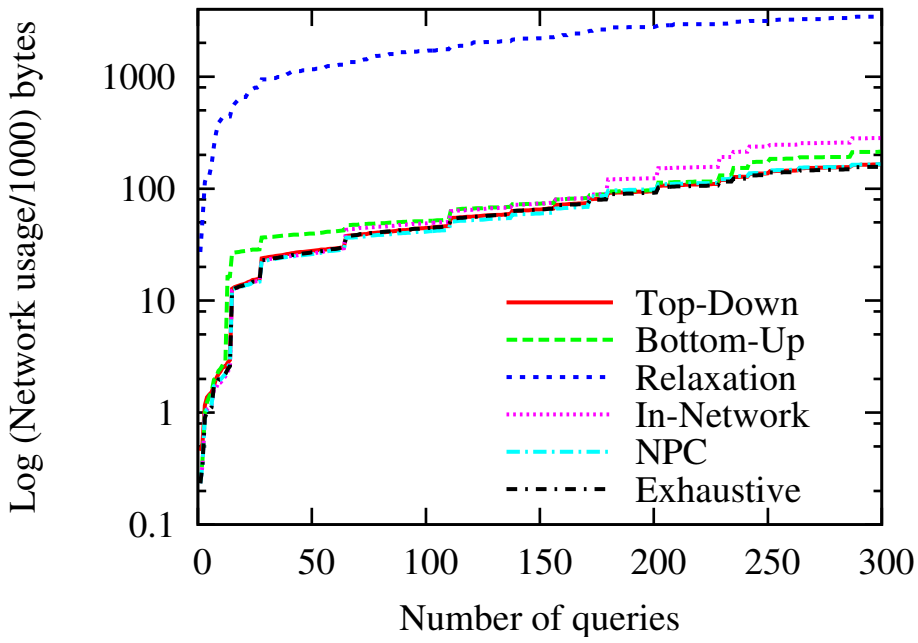


Figure 91: Enterprise Workload: Cost

savings respectively. However, note that, the In-network algorithm examined only 3% fewer plans compared to Top-Down and 170% more plans than Bottom-Up with this workload. When compared to the Relaxation algorithm, Top-Down, Bottom-Up and NPC algorithms result in approximately 96%, 93% and 95% cost savings respectively.

Figure 92 shows the total number of deployed operators. It is a well known fact that join operators are expensive and reduce throughput. Figure 92 shows that algorithms using our framework resulted in better utilization of system resources with the Top-Down, Bottom-Up and NPC algorithms utilizing nearly 81%, 73% and 48% fewer join operators compared to the phased-deployment algorithms (In-network and relaxation). The increase in the number of selection filters with our algorithms result from telescoping filter placements (i.e. placing a less restrictive filter first to allow a join operator to be reused, followed by more restrictive filters) to facilitate join reuse. Although the number of select operators have increased, these operators are stateless

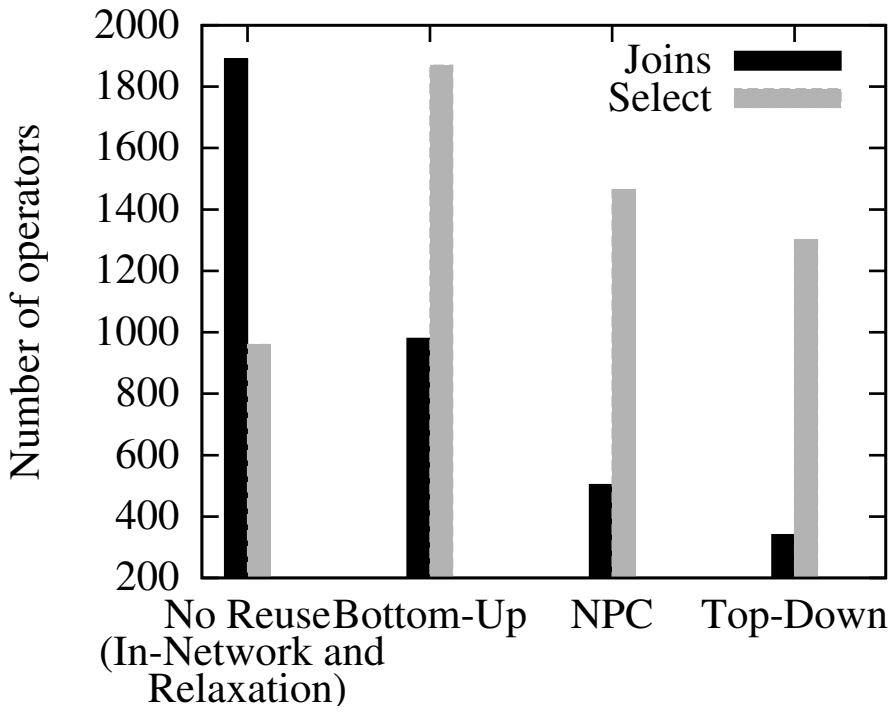


Figure 92: Enterprise Workload: # Operators

and require fewer processing resources. By reducing the number of join operators, we expect that the system throughput will increase significantly.

Figure 90 and Figure 91 also allows us to compare the deployed costs of our algorithms with the optimal solution computed using DP under the two workloads. Figure 90 shows that the Top-Down algorithm performs better than the Bottom-Up algorithm by nearly 19% and when compared to the optimal, the Bottom-Up algorithm, performs sub-optimally by 34% and the Top-Down algorithm by only 10%. The NPC algorithm performs sub-optimally by only 18%. Similarly, with the enterprise workload, as Figure 91 shows, the sub-optimality of the Top-Down algorithm is only 5%, while that of the Bottom-Up algorithm is 36%. On the other hand, the performance of the NPC algorithm is close to that of the Top-Down algorithm, with a sub-optimality of only 6%.

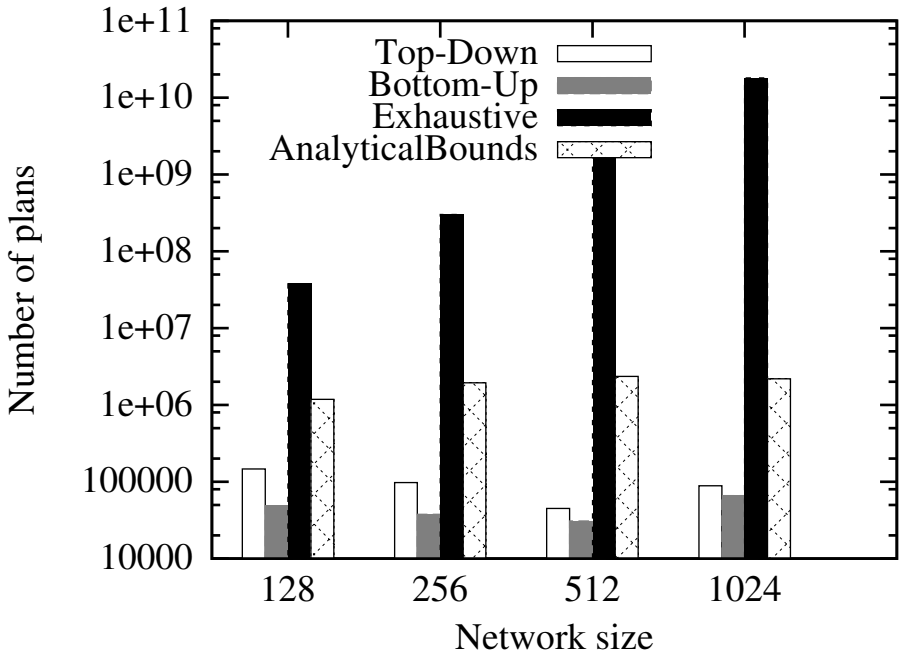


Figure 93: Scalability with Network Size

7.4.5 Scalability with Network Size

In this experiment we study the scalability of the algorithms with respect to the number of deployments considered as network size increases. We generated a workload of 100 queries using 10 stream sources with each query performing joins over 4 streams. We measured the average number of deployments considered over 4 different transit-stub topologies of different sizes generated using GT-ITM. Again, sinks were placed at random nodes in the network. Figure 93 shows the deployments considered for a single query with Bottom-Up and Top-Down algorithms with max_{cs} 32 and exhaustive search. The figure also shows how the average case (experimental) compares with the worst case (theoretical) analytical bounds. Again, the value of max_{cs} was set to 32 to produce the largest feasible search space. (An exhaustive search on a 128 node network for the deployment of a single query took nearly 3 hours to complete on our system.) Note that the increase in $\mathcal{O}_{exhaustive}$ is offset by the decrease in β such that the worst case bounds are nearly identical across the different networks. Note that

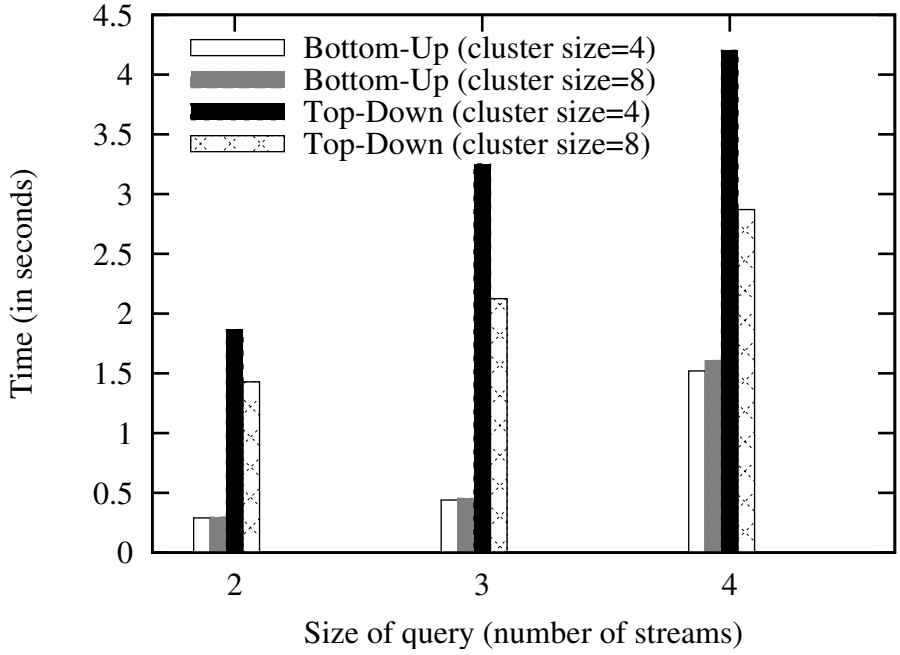


Figure 94: Query deployment time

the y-axis has a log scale.

The values for exhaustive search were calculated using Lemma 1 and the analytical bounds using Theorems 2 and 4. Clearly, performing exhaustive searches in such systems is infeasible. Both the Top-Down and Bottom-Up algorithms decrease the search space by at least 99%. We also see that the search space per query with Bottom-Up is nearly 45% less than that of Top-Down. This can be attributed to the early splitting of queries between levels in the Bottom-Up algorithm resulting in fewer operators being considered for placement at each level. Meanwhile, the Top-Down algorithm must consider all operator deployments at all levels in the hierarchy.

Although the search space of Top-Down and Bottom-Up algorithms seems to first decrease with network size and then increase, note that this is only a particular characteristic of our sample networks. For example, clustering using max_{cs} 32 resulted in an average lowest level (i.e. *Level 1*) cluster size of 26 with a 128-node network, and 15 with a 510-node network. Thus the search space for a 510-node network is less

than that of the 128 node network. Note that the search space, while being limited by the max_{cs} parameter, is affected by the average cluster size too, which depends on the particular network topology.

7.4.6 Deployment Time

We conducted prototype experiments on Emulab using IFLOW [90], our implementation of the distributed data stream system which supports hierarchies and advertisements as described earlier. The testbed on Emulab consisted of 32 nodes (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9), organized into a topology that was again generated with GT-ITM. Links were 100Mbps and the inter-node delays were set between 1msec and 6msec. The workload for the following experiments consisted of 25 queries over 8 stream sources and sinks distributed across the system. The number of joins per query varied from 1 to 3.

The experiment conducted on Emulab was aimed at measuring the time to deployment of a query over the system when using our algorithms. Figure 94 shows the average deployment time in seconds for different query sizes. We observe that the deployment times of the Bottom-Up algorithm is almost 70% less than that of the Top-Down algorithm. This can be attributed to two factors: (1) the smaller search space in the Bottom-Up algorithm, and (2) the fact that the Top-Down algorithm must always traverse the entire depth of the network hierarchy. We also observe that the deployment time of the Top-Down algorithm decreases with increasing max_{cs} value. With lower max_{cs}, there are more hierarchy levels to be traversed, resulting in higher deployment times. Our experiment allows us to conclude that our algorithms can greatly reduce the search space for the query deployment problem while offering efficient deployments with acceptable sub-optimality.

7.5 Related Work and Discussion

Distributed query optimization has received a great deal of attention from researchers since the 1980s [86]. Classic efforts in this area include R* [153] and Distributed INGRES [146]. Both the R* algorithm and the Distributed INGRES algorithm execute at a *master* site. Since our system may consist of thousands of nodes, it is infeasible to maintain all network information at a single node or perform exhaustive searches for an optimal deployment.

A number of data-stream systems including SQL-based systems such as STREAM [39], dQUOB [110], TelegraphCQ [51] and NiagaraCQ [55], as well as general purpose systems such as GATES [56], Borealis [28] and IFLOW [90] have been developed to process queries over continuous streams of data. Centralized stream processing systems have explored use of techniques like common sub-expression elimination [110] and commutative ordering of operators [40, 55] to enable operator reuse. However, our problem is complicated by the need to consider an operator's network location while computing plans that can take advantage of reuse. At the other extreme, novel systems like Eddies [38] have also used a tuple-by-tuple routing approach to adaptively decide the execution plan of a query.

The paradigm of in-network query processing has been used earlier in sensor networks [99, 158] and also in scientific data flows [56] and large scale visualizations [110] with data manipulations sometimes pushed to the source for efficiency. The use of this technique in stream based systems to only decide operator placement when the query tree is already known is described in [32, 109]. The network-aware algorithms in [32] firstly perform phased deployments which we have shown to be sub-optimal. Secondly, they do not address the important question of how the query should be divided and assigned to different portions of the network. Clearly, as seen from our experiments on varying cluster sizes, this decision can impact the efficiency of the resulting deployments. Also, no analysis is provided on the impact of the number

of zones and the placement heuristics on the computational complexity of the algorithms.

The Relaxation algorithm [109] is a novel heuristic for operator placements in distributed stream processing systems. However, the approach does not take into consideration planning and deployment simultaneously resulting in increased sub-optimality, both due to lost reuse opportunities and the subsequent approximate placement decisions. Optimal placement for a single query on a sensor network is considered in [144]. However, we consider the more generic problem of determining both operator ordering and placements. Moreover, both our algorithms are able to bound the search space and the Top-Down algorithm is also able to bound the sub-optimality.

In our current design, we consider that communication overhead, especially the amount of data transmitted, is the dominating cost in a distributed data stream system for continuous streaming applications, in comparison to the amount of local processing at each node. Although our cluster hierarchy creation using the *network usage* metric does not explicitly take into account the differences in computing power of individual nodes, it does incorporate the effect of imbalance between computing capacity and communication capacity of a node in the process of creating our cluster hierarchy using the delay parameter in the *network usage* metric [109].

In a distributed data-stream system where communication and processing costs are not only high but also changing continuously, an optimal query execution plan should ideally try to achieve multiple objectives at the same time, such as minimum response-time, minimum communication and processing cost per unit time. However, these objectives are often conflicting, since it is possible that lower delay paths have higher communication cost, or paths that incur low communication cost can cause a processing overload at some intervening network node. Epstein et.al [21] have developed an approach to optimize across such conflicting objectives, where the

optimization criteria is some 'application dependent' cost function expressed in terms of objectives, such as communication cost and response time at each node; the latter is often dependent on the node's processing and buffering capacity and memory utilization. Note that, our framework and the query optimization algorithms presented in this thesis are capable of incorporating existing 'application dependent' cost functions to find an efficient query execution plan, both in terms of cluster coordinator selection and distributed query planning.

An alternative approach to deal with a system of nodes with heterogeneous processing capacities would be to design a more complex task scheduler for the coordinator node, which allows the coordinator node that maps the query operators to the actual physical node to additionally take into consideration available processing capacities.

7.6 Summary

We have described a distributed stream query optimization framework that integrates query planning and deployment through hierarchical network partitions. Our framework consists of two key components: a hierarchical clustering of network nodes that allows network approximations and stream advertisements that enable operator reuse. We described three alternative algorithms – *Top-Down*, *Bottom-Up* and *Hybrid*, which exploit different ways of using hierarchical network partitions for operator level reuse and search space reduction. We show that although Top-Down and Bottom-Up algorithms can both choose efficient deployments while exploring only a small fraction of the search space, the Top-Down algorithm is more effective in limiting the sub-optimality of the solutions, while the Bottom-Up approach is more effective in reducing the search space and the time-to-deployment. The hybrid algorithm *NPC*, find efficient execution plans while examining a small search space, allowing us to further tune the trade-off between search space and algorithm sub-optimality. We show

through both experimental and analytical results that our algorithms are efficient and scalable at costs comparable to optimal while exploring much fewer plans.

CHAPTER VIII

CONCLUSION

The unprecedented growth of the amount of digital information in almost every sphere of life, the increasing reliance on anytime, anywhere access to this information and the high cost and repercussions of downtime are placing ever increasing availability demands on storage systems. Given the fact that these demands are achieved (1) by rapidly adapting existing legacy software to new hardware architectures (2) with development processes that are often concurrent with quality assurance processes and (3) in systems which span thousands of nodes managing hundreds of terabytes of data, software and hardware failures are expected to be the norm. However, existing failure recovery mechanisms are insufficient to handle the scale and complexity of these systems while achieving availability and service quality expectations.

In this dissertation we focused on the issues of firmware and middleware availability in storage systems that have rarely been addressed. The firmware and middleware layers of the storage system have grown tremendously in terms of both complexity and functionality over the past several years. With trends such as consolidation for easier management, virtualization and application offloading, such systems are rapidly adapting to new hardware and system architectures. However despite traditional high availability mechanisms such as hardware redundancy, decentralization and process pairs that are already in place, such systems still face challenges meeting the high availability expectations of today's enterprise applications and users. System level recovery procedures like reboots and failover to hot standbys may soon become a barrier to achieving high availability due to the coarse granularity of such processes and since the recovery time using these measures will increase as systems continue to grow in

terms of number of cores and persistent in-memory data. Next middleware scale and functionality that exposes the software through application programmer interfaces will increase the susceptibility of middleware to application induced failures and also massive simultaneous failures. Finally, modern specialized storage systems built on the principles of massive scalability, decentralization and autonomy specifically to serve the needs of niche applications such as web search, mining and stream processing call for high data availability techniques that look beyond traditional caching and replication mechanisms.

Toward addressing these challenges we have presented:

1. A recovery conscious framework and a suite of techniques to improve the fault resiliency and recovery efficiency of storage controller firmware over multi-core architectures through micro recovery that can be retrofitted into existing legacy software.

2. A fault tolerant middleware architecture that utilizes a hierarchical overlay to separate application state from control state in order to provide fault isolation from application induced failures and middleware bugs without loss of functionality while continuing to provide a SSI.

3. We present STREAMREUSE a reuse-conscious store forward network of storage nodes that utilize dynamic grouping techniques and runtime modification of operators in order to allow similar operators to be shared between multiple concurrent queries. We present algorithms to scalably identify these operators from a large search space while considering multiple factors such as network location, semantics and operator lifetimes.

Below we summarize the contributions of this dissertation research chapter-wise.

Chapter 2 presented our recovery conscious framework which divides the task of retrofitting fine-grained recovery into highly concurrent legacy storage software into

three stages. The stages progressively identify recovery dependencies and strategies, organize dependent tasks into groups for enforcing scheduling constraints and finally maps these groups to processing resources through recovery conscious scheduling.

Chapter 3 presented Log(Lock), a practical and flexible architecture for tracking dynamic dependencies and performing state restoration without rearchitecting legacy code. A comprehensive experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient and effective in reducing system recovery time.

Chapter 4 addressed the issues in the second and third tier of our recovery-conscious framework. Our main contributions include (1) the development of recovery-conscious scheduling, a non-intrusive technique to reduce the ripple effect of software failure and improve the availability of the system and (2) guidelines for effective mappings of dependent tasks to recovery groups over which recovery-conscious scheduling is performed. Through our analysis and experimentation we have shown that through careful tuning of the system configuration and the recovery-sensitive parameters, RCS can significantly improve system performance during failure recovery and thus improve system resiliency to faults while continuing to sustain high performance during normal operation.

Chapter 5 presented our fault tolerant architectures for scale-out storage middleware. Our highly available architectures utilize hierarchical overlays to provide fault isolation while continuing to deliver existing functionality and a single-system-image.

Chapter 6 described StreamReuse, a reuse-conscious network of storage nodes for distributed stream query processing systems. A unique characteristics of StreamReuse is its three step reuse opportunity discovery and deployment process: (i) operator-similarity based query relaxation, (ii) network locality-based reuse refinement, and (iii) seamless runtime reuse plan migration. Experimental evaluation of the StreamReuse approach shows that our reuse techniques can reduce resource consumption and computational costs by more than an order of magnitude compared

to the existing approaches.

Chapter 7 described an optimization framework that integrates query planning and deployment through hierarchical network partitions to perform network-aware operator reuse. Our framework consists of two key components: a hierarchical clustering of network nodes that allows network approximations and stream advertisements that enable operator reuse. We described three alternative algorithms – *Top-Down*, *Bottom-Up* and *Hybrid*, which exploit different ways of using hierarchical network partitions for operator level reuse and search space reduction. We show through both experimental and analytical results that our algorithms are efficient and scalable at costs comparable to optimal while exploring much fewer plans.

8.1 Future Work

Our research continues along a number of directions which we detail next. First, in order to adopt our recovery-conscious framework for large software systems, a significant challenge is to identify efficient recovery scopes. In ongoing work we are working on developing more generic techniques that would assist in improving the efficiency and accuracy of our recovery scope classification. The design and concepts of our recovery-conscious framework are more broadly applicable to software systems beyond storage systems. One open direction of research is to identify techniques for identifying recovery dependencies in other high performance software systems.

Next, the Log(Lock) architecture can easily be extended to a more coarse granularity of micro-recovery such as at a task or component level. However, a limitation of our approach is that although the protocols and the Log(Lock) architecture provide guidelines and information required for micro-recovery, programmer intervention is still required to define the recovery actions. One of our ongoing efforts is to improve this aspect by providing the programmer with better abstractions and hints to simplify and error-proof recovery. We are also interested in deploying and evaluating the

Log(Lock) approach in other high performance systems both to observe performance and also to get more insights in term of effectiveness of state restoration. Another line of effort is to extend the Log(Lock) capability to support longer durations of tracking, for example, across multiple threads.

With respect to the hierarchical middleware architectures, determining application placements over the hierarchical structure in order to minimize cross-cluster traffic, for example by placing communicating applications within the same cluster whenever possible, is another interesting direction of research. In ongoing work we are investigating how the hierarchical structure can be maintained dynamically, how the structure can be modified at run-time based on existing conditions and how applications can be moved between sub-clusters at run-time. and issues relating to dynamic creation and maintenance of the hierarchical structure are topics of ongoing research.

Our research on StreamReuse continues along a number of dimensions. Currently, we are investigating issues pertaining to the migration of stateful operators and the scalable techniques for distribution of the planning process and the reuse lattice. We are also interested in extending the reuse semantics from SQL like operator similarity to other messaging based service description and query languages, such as WSDL [6], SOAP [5], and BPL [53]. We believe that the design framework of the StreamReuse is sufficiently general to be applicable to exploit operator level reuse opportunities inherent in different types of service description languages [106, 157]. Finally, we are interested in examining other types of system parameters and understanding how they may impact on the effectiveness of operator level reuse.

Even with pluggable mechanisms like RCS it is necessary to emphasize that high-availability should still be a design concern and not an after-thought. Failure recovery is best thought of during the design stage itself and should not be construed as a mechanism that can be patched onto existing software. However, in a situation where we are already dealing with legacy software, like in the case of a storage controller,

we think of implementing micro-recovery more as completing the design, rather than using a patch fix. From our discussions with microcode developers we can think of a few recommendations: It is easier to implement micro-recovery in a system where there is clear separation of components and few well-defined interfaces for communicating between these components. For instance, using a client-server style separation between components and including failure handling capabilities may be a good strategy. A server should be able to handle client failures and vice-versa. Again, failure recovery and recovery handling should be thought of while writing good-path code. We hope our framework would encourage developers to incorporate additional error handling and anticipate more error scenarios and that our scheduling schemes would aid in scaling efficient error handling with system size.

Virtualization is emerging as a popular paradigm across the industry. Data-centers and enterprises are rapidly moving towards embracing virtualization in order to maximize utilization and improve quality of service, more so given that new multi-core architectures naturally call for virtualized solutions. However, unlike earlier generations of hardware, considerable burden has now shifted over to the software developers. The question now becomes how should applications be designed/ deployed in a virtualized environment in order to provide the best availability to the end-users of the service? Different components have different characteristics in terms of statefulness and the time to recovery with restart mechanisms. Currently four different recovery mechanism exists which are best suited for different component styles. (1) Microreboot - Best suited for components with no state and requiring very little initialization and hence have fast times to restart; (2) Passive stand-by - Stateless components with high initialization (like caches). (3) Active stand-by - Stateful components requiring high amount of initialization. (e.g., write-back caches). (4) Software rejuvenation (with checkpoints) - can be applied to almost all components. Instead of the application developer having to implement all these techniques, one approach is to implement

these at the virtual machine and then spread the components across virtual machines, each configured to use an appropriate method for fault-tolerance. The open research question is to study the feasibility of these techniques, the resulting reliability and the performance impact.

REFERENCES

[1] "Akamai," *http://akamai.com/.*

[2] "Amazon ec2. amazon elastic computing cloud," *aws.amazon.com/ec2.*

[3] "Amazon s3 availability event: July 20, 2008," *http://status.aws.amazon.com/s3-20080720.html.*

[4] "Amazon simple storage service (amazon s3)," *http://aws.amazon.com/s3/.*

[5] "Box, d., et al.: Simple object access protocol (soap), version 1.1. w3c note, w3c, may 2000. http://www.w3.org/tr/ soap/.,"

[6] "Christensen, e., et al.: Web services description language (wsdl) 1.1. w3c note, w3c, march 2001. http://www.w3.org/tr/2001/note-wsdl-20010315,"

[7] "Emulab network testbed," *http://www.emulab.net/.*

[8] "Health insurance portability and accountability act (hipaa)," *104th Congress, United States of America Public Law 104-191.*

[9] "How much information?," *http://www.sims.berkeley. edu/projects/how-much-info/.*

[10] "Hp TagmaStore," *http://www.hds.com/go/virtualization/.*

[11] "IBM Unveils Enterprise Stream Processing System," *http://www.hpcwire.com/hpc/1623603.html.*

[12] "Ibm websphere," *http://www-306.ibm.com/software/websphere/.*

[13] "Industry data retention regulations," *http://www.veritas. com/van/articles/4435.jsp.*

[14] "Isilon systems," *http://www.isilon.com.*

[15] "Personal communication with lawrence chiu and paul muench, ibm almaden research center, 2007,"

[16] "Remote data backups," *http://www.remotedatabackup.com.*

[17] "Storage networking industry association," `http://www.snia.org`.

[18] "Storage systems projects: Tiburon," `http://www.almaden.ibm.com/StorageSystems/projects/tiburon`.

[19] "Terascale supernova initiative," *http://www.phy.ornl.gov/tsi/,2005.*

[20] "TIBCO," *http://www.tibco.com/.*

[21] "Veritas Foundation Suite," *http://www.veritas.com/us/products/foundation.*

[22] "Ibm z/architecture principles of operation," *SA22-7832, IBM Corporation*, 2001.

[23] "Data clinic. hard disk failure.," *http://www.dataclinic.co.uk/harddisk-failures.htm*, 2004.

[24] "The data clinic. hard disk failure.," *http://www.dataclinic.co.uk/harddisk-failures.htm*, 2004.

[25] "Cdp buyers guide," *Available at http://www.snia.org/tech activities/dmf/docs/CDP Buyers Guide 20050822.pdf*, 2005.

[26] "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006.

[27] Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S., "Aurora: a new model and architecture for data stream management," *VLDB Journal*, 2003.

[28] Abadi, D. J. and others, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.

[29] Abrams, Z. and Liu, J., "Greedy is good: On service tree placement for in-network stream processing," in *ICDCS*, 2006.

[30] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P., "FARSITE: federated, available, and reliable storage for an incompletely trusted environment," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, 2002.

[31] Aguilera, M. K., Keeton, K., Merchant, A., Muniswamy-Reddy, K.-K., and Uysal, M., "Improving recoverability in multi-tier storage systems," in *DSN*, 2007.

[32] Ahmad, Y. and Cetintemel, U., "Network-aware query processing for stream-based applications," in *VLDB*, 2004.

[33] Alvarez, G. A., Burkhard, W. A., and Cristian, F., "Tolerating multiple failures in raid architectures with optimal storage and uniform declustering," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 62–72, 1997.

[34] Amini, L., Jain, N., Sehgal, A., Silber, J., and Verscheure, O., "Adaptive control of extreme-scale stream processing systems," in *ICDCS*, 2006.

[35] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R., "Resilient overlay networks," in *SOSP*, 2001.

[36] Arasu, A., Babu, S., and Widom, J., "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, 2006.

[37] Avizienis, A., "The N-version approach to fault-tolerant software.," *IEEE Trans. Software Eng.*, vol. 11, no. 12, pp. 1491–1501, 1985.

[38] Avnur, R. and Hellerstein, J. M., "Eddies: continuously adaptive query processing," in *SIGMOD*, 2000.

[39] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J., "Models and issues in data stream systems.," in *PODS*, 2002.

[40] Babu, S., Motwani, R., Munagala, K., Nishizawa, I., and Widom, J., "Adaptive ordering of pipelined stream filters," in *SIGMOD*, 2004.

[41] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J., "An analysis of latent sector errors in disk drives," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 289–300, 2007.

[42] Bairavasundaram, L. N., Goodson, G. R., Schroeder, B., Arpaci-Dusseau, A. C., and Arpaci-Dussea, R. H., "An analysis of data corruption in the storage stack," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2008.

[43] Baker, M., Shah, M., Rosenthal, D. S. H., Roussopoulos, M., Maniatis, P., Giuli, T., and Bungale, P., "A fresh look at the reliability of long-term digital storage," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, (New York, NY, USA), pp. 221–234, ACM, 2006.

[44] Beaver, J. and Sharaf, M. A., "Location-aware routing for data aggregation for sensor networks," in *Geo Sensor Networks Workshop*, 2003.

[45] Bernstein, P. A., Hadzilacos, V., and Goodman, N., *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[46] Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., and Voelker, G. M., "Total recall: system support for automated availability management," in *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2004.

[47] Burrows, M., "The chubby lock service for loosely-coupled distributed systems," *OSDI*, 2006.

[48] Cai, Z., Kumar, V., and Schwan, K., "Iq-paths: Self-regulating data streams across network overlays," in *HPDC*, 2006.

[49] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A., "Microreboot–a technique for cheap recovery," *OSDI*, 2004.

[50] Candea, G., Cutler, J., and Fox, A., "Improving availability with recursive microreboots: a soft-state system case study," *Perform. Eval.*, vol. 56, no. 1-4, pp. 213–248, 2004.

[51] Chandrasekaran, S. and others, "TelegraphCQ: Continuous dataflow processing for an uncertain world.," in *CIDR*, 2003.

[52] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A distributed storage system for structured data," *OSDI*, 2006.

[53] Charfi, A. and Mezini, M., "Hybrid web service composition: business processes meet business rules," in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pp. 30–38, 2004.

[54] Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K., "Optimizing queries with materialized views," in *ICDE*, 1995.

[55] Chen, J., DeWitt, D. J., and Naughton, J. F., "Design and evaluation of alternative selection placement strategies in optimizing continuous queries.," in *ICDE*, 2002.

[56] Chen, L., Reddy, K., and Agrawal, G., "GATES: A grid-based middleware for processing distributed data streams," in *HPDC*, 2004.

[57] Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C., and Wilkerson, C., "Scheduling threads for constructive cache sharing on cmps," in *SPAA*, (New York, NY, USA), pp. 105–115, ACM Press, 2007.

[58] Clarke, E. M. and Wing, J. M., "Formal methods: state of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, 1996.

[59] Cox, R., Dabek, F., Kaashoek, F., Li, J., and Morris, R., "Practical, distributed network coordinates," in *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, ACM SIGCOMM, November 2003.

[60] Datta, A., Stoica, I., and Franklin, M., "Lagover: Latency gradated overlays," in *ICDCS*, 2007.

[61] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "Event services for high performance computing," in *HPDC*, 2000.

[62] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B., "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[63] Elnozahy, E. N. and Plank, J. S., "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 2, pp. 97–108, 2004.

[64] Engelmann, C., Scott, S. L., Leangsuksun, C., and He, X., "Symmetric active/active high availability for high-performance computing system services," *Journal of Computers (JCP)*, vol. 1, no. 8, pp. 43–54, 2006.

[65] Fagan, M., "Design and code inspections to reduce errors in program development," pp. 575–607, 2002.

[66] Ganguly, S., Garofalakis, M., Rastogi, R., and Sabnani, K., "Streaming algorithms for robust, real-time detection of ddos attacks," in *ICDCS*, 2007.

[67] Ganguly, S., Goel, A., and Silberschatz, A., "Efficient and accurate cost models for parallel query optimization (extended abstract)," in *PODS*, 1996.

[68] Garg, S., Puliafito, A., Telek, M., and Trivedi, K., "On the analysis of software rejuvenation policies," in *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, 1997.

[69] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[70] Glider, J. S., Fuente, C. F., and Scales, W. J., "The software architecture of a SAN storage control system," *IBM System Journal*, vol. 42, no. 2, pp. 232–249, 2003.

[71] Goldstein, J. and Larson, P.-A., "Optimizing queries using materialized views: a practical, scalable solution," in *SIGMOD*, 2001.

[72] Gray, J., "Why do computers stop and what can be done about it?," in *Symposium on Reliability in Distributed Software and Database Systems*, pp. 3–12, 1986.

[73] Gray, J. and Reuter, A., *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, October 1992.

[74] Gulati, A., Merchant, A., and Varman, P. J., "pclock: an arrival curve based approach for qos guarantees in shared storage systems," in *SIGMETRICS*, (New York, NY, USA), pp. 13–24, ACM Press, 2007.

[75] Hartung, M., "IBM totalstorage enterprise storage server: A designer's view," *IBM Syst. J.*, vol. 42, no. 2, pp. 383–396, 2003.

[76] Hildrum, K., Douglis, F., Wolf, J. L., Yu, P. S., Fleischer, L., and Katta, A., "Storage optimization for large-scale distributed stream-processing systems," *Trans. Storage*, vol. 3, no. 4, pp. 1–28, 2008.

[77] HP, "HSG80 array controller software,"

[78] Huebsch, R., Garofalakis, M., Hellerstein, J. M., and Stoica, I., "Sharing aggregate computation for distributed queries," in *SIGMOD*, 2007.

[79] Hunter, S. W. and Smith, W. E., "Availability modeling and analysis of a two node cluster," in *5th Intl. Conference on Information Systems, Analysis and Synthesis*, 1999.

[80] Iyer, R. K., Rossetti, D. J., and Hsueh, M. C., "Measurement and modeling of computer reliability as affected by system activity," *ACM Trans. Comput. Syst.*, vol. 4, no. 3, 1986.

[81] Iyer, S. and Druschel, P., "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o," in *Symposium on Operating Systems Principles*, pp. 117–130, 2001.

[82] Jain, A. K. and Dubes, R. C., *Algorithms for clustering data*. NJ, USA: Prentice-Hall, Inc., 1988.

[83] Karlsson, M., Karamanolis, C., and Zhu, X., "Triage: Performance differentiation for storage systems using adaptive control," *Trans. Storage*, vol. 1, no. 4, pp. 457–480, 2005.

[84] Knight, J. C. and Leveson, N. G., "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 96–109, 1986.

[85] Kolettis, N. and Fulton, N. D., "Software rejuvenation: Analysis, module and applications," in *FTCS*, 1995.

[86] Kossmann, D., "The state of the art in distributed query processing," *ACM Comput. Surv.*, 2000.

[87] Kotla, R., Alvisi, L., and Dahlin, M., "Safestore: a durable and practical storage system," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2007.

[88] Krishnamurthy, S., Franklin, M. J., Hellerstein, J. M., and Jacobson, G., "The case for precision sharing.," in *VLDB*, 2004.

[89] Krishnamurthy, S., Wu, C., and Franklin, M., "On-the-fly sharing for streamed aggregation," in *SIGMOD*, 2006.

[90] Kumar, V. and others, "Implementing diverse messaging models with self-managing properties using IFLOW.," in *IEEE International Conference on Autonomic Computing*, 2006.

[91] Laadan, O. and Nieh, J., "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2007.

[92] LaFrese, L., "Ibm totalstorage enterprise storage server model 800 new features in lic level 2.3.0 (performance white paper)," *ESS Performance Evaluation, IBM Corporation*, 2003.

[93] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, 1978.

[94] Lee, I. and Iyer, R. K., "Software dependability in the tandem guardian system," *IEEE Trans. Softw. Eng.*, vol. 21, no. 5, pp. 455–467, 1995.

[95] Li, X. and others, "Mind: A distributed multi-dimensional indexing system for network diagnosis," in *IEEE Infocom, 2006.*

[96] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y., "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 103–116, 2007.

[97] Luo, L., Cao, Q., Huang, C., Abdelzaher, T., Stankovic, J. A., and Ward, M., "Enviromic: Towards cooperative storage and retrieval in audio sensor networks," in *ICDCS*, 2007.

[98] Lyu, M. R., *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1996.

[99] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W., "TAG: a tiny aggregation service for ad-hoc sensor networks," in *OSDI*, 2002.

[100] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.

[101] Moy, J., "OSPF version 2, request for comments 2328," 1998.

[102] Myers, G. J. and Sandler, C., *The Art of Software Testing*. John Wiley & Sons, 2004.

[103] Nagle, D., Serenyi, D., and Matthews, A., "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[104] Oleson, V., Schwan, K., Eisenhauer, G., Plale, B., Pu, C., and D.Amin, "Operational information systems - an example from the airline industry," in *First Workshop on Industrial Experiences with Systems Software (WIESS), 2000.*

[105] Olston, C., Jiang, J., and Widom, J., "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD*, 2003.

[106] Papazoglou, M. P. and Heuvel, W.-J., "Service oriented architectures: approaches, technologies and research issues," 2007.

[107] Patterson, D. A., Gibson, G., and Katz, R. H., "A case for redundant arrays of inexpensive disks (raid)," *SIGMOD Rec.*, vol. 17, no. 3, 1988.

[108] Peterson, Z. and Burns, R., "Ext3cow: a time-shifting file system for regulatory compliance," *Trans. Storage*, vol. 1, no. 2, pp. 190–212, 2005.

[109] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M., "Network-aware operator placement for stream-processing systems," in *ICDE*, 2006.

[110] Plale, B. and Schwan, K., "Dynamic querying of streaming data with the dQUOB system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, 2003.

[111] Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Iron file systems," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 206–220, ACM, 2005.

[112] Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "IRON file systems," *SOSP*, 2005.

[113] Pullum, L. L., *Software fault tolerance techniques and implementation.* Norwood, MA, USA: Artech House, Inc., 2001.

[114] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y., "Rx: Treating bugs as allergies — a safe method to survive software failure," in *SOSP*, Oct 2005.

[115] Ramasamy, H. and Schunter, M., "Architecting dependable systems using virtualization.," in *Workshop on Architecting Dependable Systems in conjunction with DSN*, 2007.

[116] Randell, B., "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*, (New York, NY, USA), pp. 437–449, ACM Press, 1975.

[117] Randell, B., Lee, P., and Treleaven, P. C., "Reliability issues in computing system design," *ACM Comput. Surv.*, vol. 10, no. 2, pp. 123–165, 1978.

[118] Rao, K. K., Hafner, J. L., and Golding, R. A., "Reliability for networked storage nodes," in *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[119] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and William S. Beebee, J., "Enhancing server availability and security through failure-oblivious computing," in *OSDI*, 2004.

[120] Rizzo, L., "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, pp. 24–36, Apr. 1997.

[121] Ronsse, M. and Bosschere, K. D., "Recplay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.

[122] Russinovich, M. and Cogswell, B., "Replay for concurrent non-deterministic shared-memory applications," in *PLDI*, (New York, NY, USA), pp. 258–266, ACM, 1996.

[123] Sacks, D., "Demystifying storage networking, das, san, nas, nas gateways, fibre channel, and iscsi," *IBM Storage Networking*, June 2001.

[124] Schneider, F. B., *Replication management using the state-machine approach.* 1993.

[125] Schroeder, B. and Gibson, G. A., "Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, (Berkeley, CA, USA), p. 1, USENIX Association, 2007.

[126] Schroeder, B. and Gibson, G. A., "Understanding disk failure rates: What does an mttf of 1, 000, 000 hours mean to you?," *TOS*, vol. 3, no. 3, 2007.

[127] Scott, D., "Assessing the costs of application downtime.," *Gartner Group, Stamford, CT*, 1998.

[128] Seshadri, S., Bamba, B., Cooper, B. F., Kumar, V., Liu, L., Schwan, K., and Zhang, G., "Grouping distributed stream query services by operator similarity and network locality," in *SERVICES '08: Proceedings of the 2008 IEEE Congress on Services - Part I*, (Washington, DC, USA), pp. 11–18, IEEE Computer Society, 2008.

[129] Seshadri, S., Chiu, L., Constantinescu, C., Balachandran, S., Dickey, C., Liu, L., and Muench, P., "Enhancing storage system availability on multi-core architectures using recovery conscious scheduling," in *USENIX FAST*, 2008.

[130] Seshadri, S., Chiu, L., and Liu, L., "A systematic approach to system state restoration during storage controller micro-recovery," in *USENIX FAST*, 2009.

[131] Seshadri, S., Kumar, V., and Cooper, B. F., "Optimizing multiple queries in distributed data stream systems," in *NetDB*, 2006.

[132] Seshadri, S., Kumar, V., Cooper, B. F., and Liu, L., "Optimizing multiple distributed stream queries using hierarchical network partitions," in *IPDPS*, 2007.

[133] Seshadri, S., Kumar, V., Cooper, B. F., and Liu, L., "Optimizing multiple distributed stream queries using hierarchical network partitions," in *IPDPS*, 2007.

[134] Seshadri, S., Liu, L., and Chiu, L., "Recovery scopes, recovery groups, and fine-grained recovery in enterprise storage controllers with multi-core processors.," 2009.

[135] Seshadri, S., Liu, L., Cooper, B. F., Chiu, L., Gupta, K., and Muench, P., "A fault-tolerant middleware architecture for high-availability storage services," in *IEEE SCC*, pp. 286–293, 2007.

[136] Shah, M. A., Hellerstein, J. M., Chandrasekaran, S., and Franklin, M. J., "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*, 2003.

[137] Sidiroglou, S., Laadan, O., Keromytis, A. D., and Nieh, J., "Using rescue points to navigate software recovery," in *SP*, 2007.

[138] Sirius, "Sirius enterprise systems group disk drive storage hardware," *Solicitation EPS050059-A4*.

[139] Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Improving Storage System Availability with D-GRAID," in *Proceedings of the Third USENIX Symposium on File and Storage Technologies (FAST '04)*, March 2004.

[140] Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Improving Storage System Availability with D-GRAID," *ACM Transactions on Storage (TOS)*, vol. 1, pp. 133–170, May 2005.

[141] Slember, J. and Narasimhan, P., "Living with nondeterminism in replicated middleware applications," *Middleware*, 2006.

[142] Sommerville, I., *Software engineering (5th ed.)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995.

[143] Srinivasan, S. M., Kandula, S., Andrews, C. R., and Zhou, Y., "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *USENIX ATC*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[144] Srivastava, U., Munagala, K., and Widom, J., "Operator placement for in-network stream query processing," in *PODS*, 2005.

[145] StidhamJr., S., "A last word on $l = \lambda w$," in *Operations Research*, vol. 22, (Montreal, Que., Canada), pp. 417–421, IEEE Computer Society Press, 1974.

[146] Stonebraker, M., "The design and implementation of distributed INGRES," *The INGRES papers: anatomy of a relational database system*, 1986.

[147] Stroustrup, B., *The Design and Evolution of* C++. Addison-Wesley, 1994.

[148] Sullivan, M. and Chillarege, R., "Software defects and their impact on system availability - a study of field failures in operating systems," *FTCS*, 1991.

[149] Trachtenberg, M., "A general theory of software-reliability modeling," *IEEE Trans. on Reliability*, vol. 39, no. 1, pp. 92–96, 1990.

[150] Trivedi, K. S., *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1982.

[151] Vaidyanathan, K., Harper, R. E., Hunter, S. W., and Trivedi, K. S., "Analysis and implementation of software rejuvenation in cluster systems," in *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2001.

[152] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T., "The HP AutoRAID hierarchical storage system," in *SOSP*, 1995.

[153] Williams, R. and others, *R*: An overview of the architecture*.

[154] Wolfram Research, I., *Mathematica*. version 5.0 ed.

[155] Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliççöte, H., and Khosla, P. K., "Survivable information storage systems," *Computer*, vol. 33, no. 8, 2000.

[156] Xiang, S., Lim, H. B., Tan, K.-L., and Zhou, Y., "Two-tier multiple query optimization for sensor networks," in *ICDCS*, 2007.

[157] Yang, J. and Papazoglou, M., "Service components for managing the lifecycle of service compositions," 2003.

[158] Yao, Y. and Gehrke, J., "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, 2002.

[159] Zegura, E. W., Calvert, K. L., and Bhattacharjee, S., "How to model an internetwork," in *Infocom*, 1996.

[160] Zeitler, D., "Realistic assumptions for software reliability models," in *IEEE Int'l Symp. Software Reliability Eng.*, 1991.

[161] Zhang, J., Sivasubramaniam, A., Wang, Q., Riska, A., and Riedel, E., "Storage performance virtualization via throughput and latency control," *Trans. Storage*, vol. 2, no. 3, pp. 283–308, 2006.

VITA

Sangeetha Seshadri was born in Calcutta and raised in Madras, India. She received the BE degree in computer science and the M.Sc degree in Mathematics from the Birla Institute of Technology and Science, Pilani, India, in 2002. She spent the next two years working as a Senior Applications Engineer at Oracle, India. Since Fall 2004, Sangeetha has been working toward the PhD degree at the College of Computing, Georgia Institute of Technology, advised by Prof. Ling Liu. As a member of the Center for Experimental Research in Computer Science (CERCS) and the Distributed Data Intensive Lab (DiSL), Sangeetha's research interests include large scale storage systems and distributed middleware overlay systems particularly techniques to improve the resilience and scalability of such systems. Since May 2005, Sangeetha has actively collaborated with researchers at the Scalable Storage Systems group at IBM Almaden Research Center working towards developing new approaches to develop highly available storage firmware and middleware.