Black-Box Problem Localization in Parallel File Systems

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Michael P. Kasick

B.S., Electrical and Computer Engineering, Carnegie Mellon University

M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, Pennsylvania

December 2015

Acknowledgments

I acknowledge my colleagues for their contributions to our problem diagnosis work: Keith Bare, Eugene Marinelli, and Jiaqi Tan for their contributions to syscall-based problem diagnosis; Rajeev Gandhi for his insights and contributions to data analysis; Soila Kavulya and Raja Sambasivan for their discussions on problem-diagnosis approaches and feedback; Nathan Mickulicz and Utsav Drolia for their feedback on this thesis and related presentations; and Priya Narasimhan for her wordsmithing and incredible tact in conveying the technical details of this work.

I acknowledge Rob Ross, Sam Lang, Phil Carns and Kevin Harms of Argonne National Laboratory for their insightful discussions on PVFS, instrumentation sources, troubleshooting procedures, and anecdotes of performance problems in production PVFS deployments—all of which are instrumental in motivating this work. I especially acknowledge Kevin Harms and William Allcock of Argonne National Laboratory for the opportunity to instrument and collect performance metrics from the Surveyor and Intrepid storage systems, as well as for providing logs and descriptions of problems encountered. I acknowledge the Parallel Data Lab (PDL), specifically Bill Courtright, Greg Ganger, Michael Stroucken, Mitch Franzos, and Zis Economou for providing me with access to the Data Center Observatory (DCO), the computing facility where the laboratory experiments and analysis of case study data for this thesis was performed. I acknowledge John Wilkes of HP Labs (now Google) for his insights on compelling errors for study; John Palmer and Roger Haskin of IBM Research for their insights on GPFS and compelling research problems; and the industry practitioners of the PDL Consortium with whom I've had priceless opportunities to interact and to receive feedback concerning this work.

I acknowledge the members of my Doctoral thesis committee; Greg Ganger, Rajeev Gandhi, and Rob Ross for their readings and evaluations, enlightening discussions, directional motivation, and of course feedback on my thesis and research. I also acknowledge Priya Narasimhan for serving as my graduate advisor and Doctoral committee chair, and for her continued professional and personal support throughout my tenure at Carnegie Mellon.

Research in support of this thesis was sponsored in part by NSF grants #CCF-0621508 and by ARO agreement DAAD19–02–1–0389. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE–AC02–06CH11357. Support for my Ph.D. research was also provided by the NSF Graduate Research Fellowship Program and the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

Special thanks are due to my parents and family, who have always provided unparalleled support and encouragement for my education. Finally, I thank my wife Rena, for her unconditional love and support and for her constant encouragement in the pursuit of my dreams.

Abstract

Parallel file systems target large, high-performance storage systems. Since these storage systems are comprised of a significant number of components (i.e., hundreds of file servers, thousands of disks, etc.), they are expected to (and in practice do) frequently exhibit "problems", from degraded performance to outright failure of one or more components. The sheer number of components, and thus, potential problems, makes manual diagnosis of these problems difficult. Of particular concern are system-wide performance degradations, which may arise from a single misbehaving component, and thus, pose a challenge for problem localization. Even failure of a redundant component with a less-significant performance impact is worrisome as it may, in absence of explicit checks, go unnoticed for some time and increase risk of system unavailability.

As a solution, this thesis defines a novel problem-diagnosis approach, capitalizing upon the parallel-file-system design criterion of balanced performance, that peer-compares the performance of system components to localize problems within storage systems running unmodified, "off-the-shelf" parallel file systems. Performed in support of this thesis is a set of laboratory experiments that demonstrate proof-of-concept of the peer-comparison approach by injecting four realistic problems into 12-server, test-bench PVFS and Lustre clusters. This thesis is further validated by taking the diagnosis approach and adapting it to to work on a very-large, production GPFS storage system consisting of 128 file servers, 32 storage controllers, 1152 disk arrays, and 11,520 total disks. Presented in a 15-month case study is the problems observed through analysis of 624 GB of instrumentation data, in which a variety of performance-related storage-system problems are localized and diagnosed, in a matter of hours, as compared to days or longer with manual approaches.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Identifying and diagnosing problems, especially performance problems, is a difficult task in large-scale storage systems. These systems are comprised of many components: tens of storage controllers, hundreds of file servers, thousands of disk arrays, and tens-of-thousands of disks. Performance problems can arise from different system layers, such as bugs in the application, misconfigurations of file system or protocols, resource exhaustion, network congestion, and redundant component failures. Often, the most interesting and trickiest problems to diagnose are not the outright crash (fail-stop) failures, but rather those that result in a "limping-but-alive" system (i.e., the system continues to operate, but with degraded performance). Even where such problems may not significantly degrade performance (e.g., a redundant disk failure), it is often essential to quickly diagnose and repair these problems before subsequent ones result in a system crash or data loss. Our work targets the diagnosis of such problems in parallel file systems used in high-performance computing (HPC).

Large scientific applications consist of compute-intense behavior intermixed with periods of intense parallel I/O, and therefore depend on storage systems that can support high-bandwidth concurrent writes. The Parallel Virtual File System (PVFS) [11], Lustre [57], and the General Parallel File System (GPFS) [49] are cluster and parallel file systems that are designed to utilize and exploit parallelism across all storage system components to provide HPC applications with

very high-bandwidth concurrent I/O. All three may be deployed in client-server architectures, with many clients communicating with multiple I/O servers and one or more metadata servers. Each also supports the UNIX I/O interface and allow existing UNIX I/O programs to access files without recompilation. To facilitate parallel access to a file, parallel file systems distribute (or "stripe") file data across multiple disks located in multiple storage arrays and accessed by physically distinct file servers, which balances load and performance across all system components.

An interesting class of problems in these systems is hardware component faults. Due to redundancy, generally component faults and failures manifest in degraded performance. Due to careful balancing of the number of components and their connections, the degraded performance of even a single hardware component may be observed throughout an entire storage system, which makes problem localization difficult.

At present, storage system problems are observed and diagnosed through independent monitoring agents that exist within the individual components of a system, e.g., disks (via S.M.A.R.T. [16]), storage controllers, and file servers. However, because these agents act independently, there is a lack of understanding how a specific problem affects overall performance, and thus it is unclear whether a corrective action is immediately necessary. Where the underlying problem is the misconfiguration of a specific component, an independent monitoring agent may not even be aware that a problem exists.

In this thesis we explore the use of peer-comparison techniques to identify and localize performance problems in parallel file systems. As part of a solution to the challenge of problem diagnosis in HPC storage systems, we define a problem-diagnosis approach that capitalizes upon the parallel-file-system design criterion of balanced performance. This approach gathers black-box performance metrics from system components and analyzes them to automatically localize problems within storage systems running unmodified, "off-the-shelf" parallel file systems, enabling diagnosis of the underlying problem by the system's operators. Central to the problem-diagnosis strategy is the hypothesis (borne out by observations of parallel-file-system behavior) that *innocent (i.e., fault-free) storage system components follow similar trends in their (throughput and latency)*

performance metrics, whereas a culprit component appears markedly different in comparison. Based on this hypothesis, a statistical peer-comparison approach is developed that automatically singles out the culprit component through the temporal correlation of histograms and other statistical attributes of gathered performance data across the servers in a storage system.

Even where a performance problem affects *all* components in a storage system (e.g., due to misconfigurations or software/firmware bugs present on all components of the same type) our peer-comparison approach may be used to rule out performance problems affecting specific components. By using our approach in conjunction with other problem diagnosis techniques (§ 7), operators, on complaint of a performance problem, can quickly rule out hardware component failures for most of the system components and focus their search on non-peer hardware components, system software bugs, or poorly-performing workload-I/O patterns. Thus, we envision that our problem-diagnosis approach maybe used in conjunction with component-specific monitoring and other approaches to provide a comprehensive solution to diagnose all classes of performance problems.

We initially demonstrate this problem-diagnosis approach with a proof-of-concept implementation of our peer-comparison algorithm that we use to automatically diagnose performance problems injected during runs of synthetic workloads (dd, IOzone, or PostMark) on a controlled, laboratory test-bench PVFS and Lustre storage clusters of up to 12 file servers. While this prototype demonstrates that peer comparison is a good foundation for diagnosing problems in parallel file systems, it does not, yet, attempt to tackle the practical challenges of diagnosis in large-scale, real-world production systems.

We then adapt the proof-of-concept approach for the primary high-speed storage system of Intrepid, a 40-rack Blue Gene/P supercomputer at Argonne National Laboratory [41], shown in Figure 1.1. In doing so, we tackle the practical issues in making problem diagnosis work in large-scale environment, and we evaluate our approach through a 15-month case study of practical problems that we observe and identify within Intrepid's storage system.

Figure 1.1: Intrepid, consists of 40 Blue Gene/P racks [3].

1.1 Problem Statement

Focusing on problem localization in parallel file systems, the thesis statement is:

> *Through the collection and analysis of black-box, OS-level performance metrics, it is possible to automatically detect and localize (i.e., diagnose) storage system problems to the misbehaving components most responsible for degraded parallel-file-system performance.*

1.2 Hypotheses

We hypothesize that, under a performance-manifesting fault in a parallel-file-system-based storage system, performance metrics should exhibit observable anomalous behavior on the culprit components, e.g., disk arrays, storage controllers, attachments, and file server. Additionally, with knowledge of the parallel file system design criterion of balanced performance, we hypothesize that

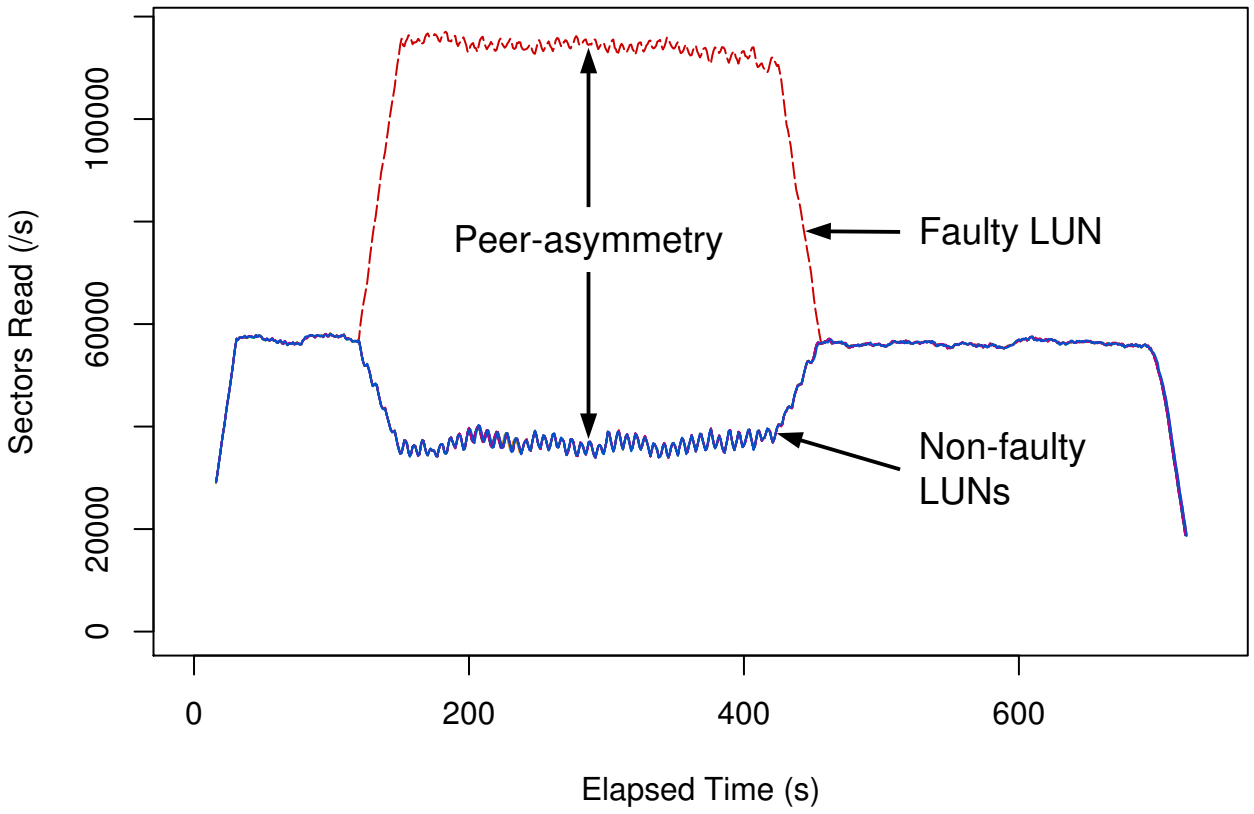

Figure 1.2: Asymmetry in throughput for an injected fault; provides intuition behind the peer-comparison approach that serves as a good foundation for our diagnosis.

the statistical trends of I/O performance data should (i) exhibit symmetry (albeit with inevitable minor differences) across fault-free components, even under workload changes, and (ii) exhibit asymmetries across the faulty (culprit) components, as compared to the fault-free components.

1.2.1 Intuition of Hypotheses

The defining property of parallel file systems is that they parallelize accesses to even a single file, by striping its data across many, if not all, file servers and logical storage units (LUNs) within a storage system. By striping data, parallel file systems maintain similar I/O loads across system components (peers) for all non-pathological client workloads. Figure 1.2 illustrates the intuition behind our hypothesis; the injection of a rogue workload on a spindle shared with a PVFS LUN results in a throughput asymmetry between the faulty and fault-free LUNs, where previously throughput was similar across them.

In the context of our diagnosis approach, *peers* represent components of the same type or functionality that are expected to exhibit similar request patterns. By capturing performance metrics at each peer, and comparing these metrics across peers to locate asymmetries (*peer-comparison*), we expect to be able to identify and localize faults to the culprit peer(s).

1.2.2 Validation of Hypotheses

To validate our hypothesis, we explore a peer-comparison-based approach to automatically diagnose performance problems through a set of fault-injection experiments on controlled PVFS and Lustre test-bench clusters. Through these experiments, we evaluate the accuracy of our peer-comparison algorithm with true- and false-positive rates for diagnosing the correct faulty server and fault type (if an injected fault exists). We then further evaluate our diagnosis approach with the instrumentation, analysis, and case study of the problems observed in a GPFS-based, large-scale, real-world production system.

1.3 Contributions

Through the research that is embodied in this thesis, we make the following contributions:

- A low-overhead method of black-box instrumentation consisting of samples of OS-level storage and network performance metrics.
- A peer-comparison diagnosis algorithm that is able to detect the existence of storage system problems and localize the problems to specific, misbehaving system components.
- Outlining the pragmatic challenges of making problem diagnosis work in large-scale storage systems.
- Adapting our diagnosis approach, from its initial proof-of-concept target of a 12-server experimental cluster, to a 11,000-component, production environment consisting of file servers, storage controllers, disk arrays, attachments, etc.

- Evaluating a case study of problems observed in Intrepid's storage system, including those that were previously unknown to system operators.

1.4 Thesis Roadmap

The remainder of this thesis is organized as follows: We start with a model and description of parallel file systems and the properties that make them amenable to a peer-comparison (§ 2). We then present an overview of our problem-diagnosis approach (§ 3), as it was refined to work in the environment of Intrepid's storage system, to serve as a basis for discussing the evolution of our diagnosis work.

We then describe our approach, as it was originally conceived [36], to work in a small-scale laboratory environment. In doing so, we explore our approach's capability to automatically diagnose performance problems through a set of experiments on controlled PVFS and Lustre test-bench clusters (§ 4).

We then discuss the challenges of taking the initial algorithm from its origin in a limited, controllable test-bench environment, and making it effective in a noisy, 11,000-component production system (§ 5). After meeting these challenges, we present a case study [35], where we evaluate the capability of our approach to localize real-world problems in Intrepid's storage system (§ 5.6).

We then present lessons we've learned throughout our research (§ 6), including a few approaches that, while not part of our primary research results, did contribute to our overall understanding of problem diagnosis in parallel file systems. We also present other problem-diagnosis approaches (§ 7) that my be used in conjunction with our approach for more comprehensive diagnosis of storage-system problems. Finally, we identify areas of future improvement and study, and provide a conclusion of this thesis and our research (§ 8).

Chapter 2

System Model

Although this work specifically focuses on experimentation with parallel file systems, the approach is more broadly applicable to systems with certain characteristics, as described below.

2.1 General Class of Systems

The general class of systems where we may apply a peer-comparison approach is parallel-processing systems, e.g., parallel file systems. These systems are comprised of *nodes* (e.g., disk arrays), nominally-independent entities, each of which processes a portion of a larger data set. To apply peer comparison, we must measure one or more *metrics* (e.g., throughput, latency, etc.) related to the node's processing of data. In order to meaningfully compare these metrics across nodes, each portion of the data set should have the same parameters. That is, while each portion of data need not be identical in value or content, each portion should be the same size, require a similar amount of computational effort to process, and be evenly distributed across nodes.

Beyond the metrics alone, the knowledge of the system topology can be beneficial in enhancing diagnosis.

From the viewpoint of processing, nodes could be considered to be independent. Thus, anomalies in metrics on a particular node would lead us to suspect that node (and any of its components) is at fault. The reality is, that, nodes can often consist of *components* that may be shared with

other nodes; such sharing is often captured in the system topology. The consideration of system topology, along with anomalies in the metrics of nodes, opens up greater possibilities for diagnosis. While the presence of shared components may superficially seem to confound diagnosis, it can also prove to be an advantage when taken in conjunction with the topology of the system's shared components. For example, if anomalies are observed in the metrics of two distinct nodes, this may lead us to suspect that problem originates from a component shared between those nodes.

A node's metrics can often be observed from multiple vantage points in the system. We designate a node's *observer* to be an entity that is capable of seeing and collecting metrics about that particular node. The notion of observers can be further open possibilities for diagnosis. For example, multiple file servers, as observers, may collect metrics about a single disk array (a node). Differences between the metrics observed for that single node, even if not anomalous relative to other nodes, may still indicate a problem between the node and its observer (e.g., a storage attachment).

2.2 Candidate Systems: Parallel File Systems

PVFS, Lustre, and GPFS storage systems consist of multiple file servers (which function as *observers* in our model) that are accessed by one or more clients, as illustrated in Figure 2.1. For large I/O operations, clients issue simultaneous requests across a local area network (e.g., Ethernet, Myrinet, etc.) to each file server. To facilitate storage, file servers may store data on local (e.g., SATA) disks, however, in most systems I/O requests are further forwarded to dedicated storage controllers, either via direct attachments (e.g., Fibre Channel, InfiniBand) or over a storage area network.

Storage controllers expose block-addressable logical storage units (LUNs, which function as *nodes* in our model) to file servers and store file-system content. Each LUN is comprised of a disk array, e.g., in RAID-5 or RAID-6 configurations. Controllers expose different subsets of LUNs to each of its attached the file servers. Usually LUNs are mapped so each LUN primarily serves

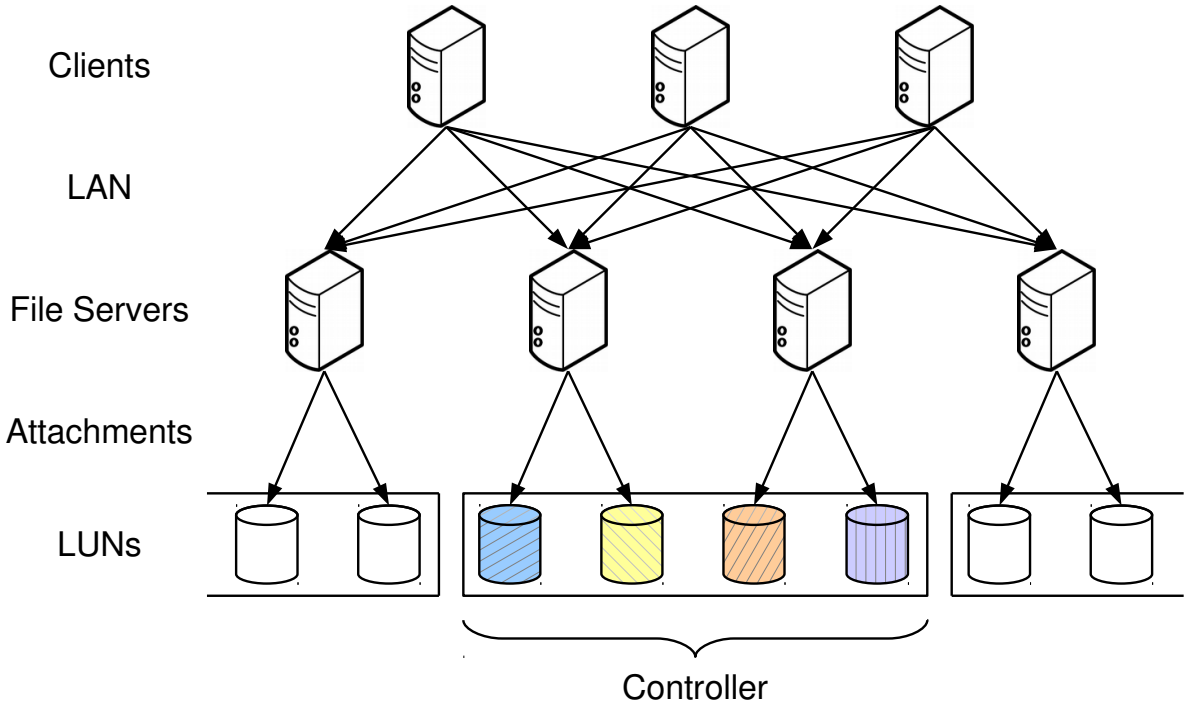

Figure 2.1: Architecture of parallel file systems.

I/O for one (primary) file server, while also allowing redundant access from other (secondary) file servers. This enables LUNs to remain accessible to clients in the event that a small-number of file servers go offline. Controllers themselves may also be redundant so that LUNs remain accessible to secondary file servers in the event of a controller failure.

The defining property of parallel file systems, including PVFS, Lustre, and GPFS, is that they parallelize accesses to even a single file, by striping its data across many (and in a common configuration, across all) file servers and LUNs. For example, when performing large, sequential I/O, clients may issue requests, corresponding to adjacent stripe segments, round-robin to each LUN in the system. LUNs are mapped to file servers so that these requests are striped to each file server, parallelizing access across the LAN, and further striped across the primary LUNs attached to file servers, parallelizing access across storage attachments.

The parallelization introduced by the file system, even for sequential writes to a single file, ensures that non-pessimistic workloads exhibit equal loads across the system, which in turn, should be met with balanced performance. Thus, when "hot spots" and performance imbalances arise in

a storage system, we hypothesize them to be indicative of performance problem. Furthermore, by instrumenting each file server in the system, we can observe the performance of file servers, storage controllers, and LUNs, from multiple perspectives, which enables us to localize problems to the components of the system where performance imbalance is most significant.

2.2.1 PVFS

PVFS storage systems consist of one or more metadata servers and multiple I/O servers that are accessed by one or more PVFS clients. The PVFS server consists of a single monolithic user-space daemon that may act in either or both metadata and I/O server roles. Historically I/O and metadata services were ran on separate servers, but recent versions of PVFS encourage running both services on all servers to enhance performance.

PVFS clients consist of stand-alone applications that use the PVFS library (*libpvfs2*) or MPI applications that use the ROMIO MPI-IO library (that supports PVFS internally) to invoke file operations on one or more servers. PVFS can also plug in to the Linux Kernel's VFS interface via a kernel module that forwards the client's syscalls (requests) to a user-space PVFS client daemon that then invokes operations on the servers. This kernel client allows PVFS file systems to be mounted under Linux similar to other remote file systems like NFS.

With PVFS, file-objects are distributed across all I/O servers in the system. In particular, file data is striped across each I/O server with a default stripe size of 64 kB. For each file-object, the first stripe segment is located on the I/O server to which the object handle is assigned. Subsequent segments are accessed in a round-robin manner on each of the remaining I/O servers. This characteristic has significant implications on PVFS's throughput in the event of a performance problem.

2.2.2 Lustre

Lustre storage systems consist of one active metadata server which serves one metadata target (storage space), one management server which may be colocated with the metadata server, and multiple object storage servers which serve one or more object storage targets each. The metadata

and object storage servers are analogous to PVFS's metadata and I/O servers with the main distinction of only allowing for a single active metadata server per system. Unlike PVFS, the Lustre server is implemented entirely in kernel space as a loadable kernel module. The Lustre client is also implemented as a kernel space file-system module, and like PVFS, provides file system access via the Linux VFS interface. A userspace client library (*liblustre*) is also available.

Lustre allows for the configurable striping of file data across one or more object storage targets. By default, file data is stored on a single target. The `stripe_count` parameter may be set on a per-file, directory, or file-system basis to specify the number of object storage targets that file data is striped over. The `stripe_size` parameter specifies the stripe unit size and may be configured to multiples of 64 kB, with a default of 1 MB (the maximum payload size of a Lustre RPC). The `stripe_offset` parameter determines on which object storage target the first stripe segment resides, with -1 indicating that first segments should be allocated round-robin.

2.2.3 GPFS

GPFS is a cluster and parallel file system used for both high-performance computing and network storage applications. Like PVFS and Lustre, a GPFS storage system consists of instances that run the GPFS software, disk arrays that store file-system content, and storage controllers that export disk arrays as LUNs. For each LUN, the GPFS system configuration defines, in GPFS terminology, a set of Network Shared Disk (NSD) servers to handle network (LAN) I/O requests for those LUNs.

Internally, GPFS instances regard each other as peers, with no software level distinction between "client" instances (those running applications that generate I/O) and "server" instances (those with attached storage). When a GPFS instance issues an I/O request to an attached (direct, storage network, etc.) LUN, the request is made via this attachment even if the instance is not defined as an NSD in the system configuration. Thus, shared storage that is attached to multiple GPFS instances may see I/O requests from all of the attached instances, regardless of their status as NSD servers. When an I/O request is made to a non-attached LUN, the request is forwarded over a LAN (e.g., Ethernet) to the LUN's primary-defined NSD server. If the NSD server is unavailable

or communication is interrupted, the request is made to subsequent NSD servers in their defined order.

This peer-based architecture enables deployment of a variety of cluster layouts that may even change dynamically. In practice, a commonly deployed model is that of separate clients and servers, where the instances (clients) that mount the file system and run applications are distinct from instances (servers) that interact with attached storage. In the case of non-shared (instance-exclusive) storage, all I/O requests from both clients and servers are routed through the singly-defined NSD server for that LUN. In the case of shared storage, I/O requests from all clients and non-attached servers are routed through the highest-priority, presently-available NSD server, and I/O requests from attached servers are made directly via their attachments. Thus, in the client and server model, each LUN's workload is reflected in storage metrics entirely on a single GPFS instance (non-shared storage), or across multiple GPFS instances (shared storage). This is the deployment model that will be discussed in the remainder of this thesis.

2.3 Goals of Problem Diagnosis

To facilitate problem diagnosis in real-world storage systems, we recognize that our diagnosis approach must remain "production friendly", that is, it must minimize the burden of deploying diagnosis tool on a system's operators and users. With this high-level concept in mind, we identify a number of specific goals and non-goals that influence our problem-diagnosis approach.

Goals. Our approach should exhibit:

- *No software modification* of the file system or other system components. Instrumentation should consist of a small software package that can be dropped into an existing system and use already-available instrumentation hooks.
- *Minimal-overhead instrumentation* that does not produce an overwhelming volume of data, does not significantly degrade file system latency or throughput, and does not adversely impact the

system's operation.

- *Application transparency* so that user applications require no modification to enable problem diagnosis, and so that our diagnosis algorithm requires no specific knowledge of workload behavior.

- *Minimal false alarms* of anomalies in the face of legitimate behavioral changes (e.g., workload changes due to increased request rate).

- *Coverage of problems*, particularly those encountered in real-world GPFS storage systems.

Non-Goals. Our approach does not support:

- *Code-level debugging.* Our approach aims for coarse-grained problem diagnosis by identifying the faulty system components (disk arrays, controllers, and servers). We currently do not aim for fine-grained diagnosis that would trace the problem to specific file system modules.

- *Pathological workloads.* Our approach relies on clients exhibiting similar request to logical storage units (LUNs). In parallel file systems, the request pattern for most workloads is similar across all LUNs—requests are either sequential enough to be striped across all LUNs or random enough to result in roughly uniform access. However, some workloads (e.g., overwriting the same portion of a file repeatedly, or only writing stripe-unit-sized records to every stripe-count offset) may make requests distributed to only a subset, possibly one, of the LUNs.

- *Diagnosis of non-peers.* Our approach fundamentally cannot diagnose performance problems on non-peer components (e.g., Lustre's single metadata server).

- *Diagnosis of problems present on all peers.* Our approach also cannot diagnose performance problems that are present on all peer components, including suboptimal workload I/O patterns, misconfigurations, file-system software bugs, or firmware bugs that are simultaneously activated on all components of the same type. However, when a performance problem is otherwise known

to be present (e.g., from user complaints), our approach may be used to *exclude* component-specific problems so operators may focus on diagnosis using other techniques (§ 7).

2.4 Motivation: Real Problem Anecdotes

The faults we study here are motivated by the PVFS developers' anecdotal experience [10] of problems faced/reported in various production PVFS deployments, one of which is Argonne National Laboratory's 557 TFlop Blue Gene/P (BG/P) PVFS cluster. Accounts of experience with BG/P indicate that storage/network problems account for approximately 50%/50% of performance issues [10]. A single poorly performing server has been observed to impact the behavior of the overall system, instead of its behavior being averaged out by that of non-faulty nodes [10]. This makes it difficult to troubleshoot system-wide performance issues, and thus, fault localization (i.e., diagnosing the faulty server) is a critical first step in root-cause analysis.

Anomalous storage behavior can result from a number of causes. Aside from failing disks, RAID controllers may scan disks during idle times to proactively search for media defects [25], inadvertently creating disk contention that degrades the throughput of a disk array [61]. Our *disk-busy* injected problem (§ 4.1) seeks to emulate this manifestation. Another possible cause of a disk-busy problem is disk contention due to the accidental launch of a rogue processes. For example, if two remote file servers (e.g., PVFS and GPFS) are collocated, the startup of a second server (GPFS) might negatively impact the performance of the server already running (PVFS) [10].

Network problems primarily manifest in packet-loss errors, which is reported to be the "most frustrating" [sic] to diagnose [10]. Packet loss is often the result of faulty switch ports that enter a degraded state when packets can still be sent but occasionally fail CRC checks. The resulting poor performance spreads through the rest of the network, making problem diagnosis difficult [10]. Packet loss might also be the result of an overloaded switch that "just can't keep up" [sic]. In this case, network diagnostic tests of individual links might exhibit no errors, and problems manifest only while PVFS is running [10].

Errors do not necessarily manifest identically under all workloads. For example, SANs with large write caches can initially mask performance problems under write-intensive workloads and thus, the problems might take a while to manifest [10]. In contrast, performance problems in read-intensive workloads manifest rather quickly.

A consistent, but unfortunate, aspect of performance faults is that they result in a "limping-but-alive" mode, where system throughput is drastically reduced, but the system continues to run without errors being reported. Under such conditions, it is likely not possible to identify the faulty node by examining PVFS/application logs (neither of which will indicate any errors) [10].

Fail-stop performance problems usually result in an outright server crash, making it relatively easy to identify the faulty server. Our work targets the diagnosis of non-fail-stop performance problems that can degrade server performance without escalating into a server crash. There are basically three resources—CPU, storage, network—being contended for that are likely to cause throughput degradation. CPU is an unlikely bottleneck as parallel file systems are mostly I/O-intensive, and fair CPU scheduling policies should guarantee that enough time-slices are available. Thus, we focus on the remaining two resources, storage and network, that are likely to pose performance bottlenecks.

Chapter 3

Approach

In applying our problem-diagnosis approach to large storage systems like Intrepid's, our primary objective is to localize to the most problematic LUNs (specifically LUN-server attachments that we refer to as "LUNs" henceforth) in the storage system, which in turn, reflect the location of faults with greatest performance impact.

3.1 Synopses of Approach

The flowchart in Figure 3.1 provides a synopses of our problem localization process, which consists of three stages.

Our *instrumentation* consists of software deployed on each file server that collects OS-level performance metrics on the system components, e.g., for every LUN, accessed and utilized by the server. These server-perspective metrics are periodically collected and forwarded, from each file server, to an analysis machine where the remainder of our problem localization takes place.

Our *anomaly detection* takes the component performance metrics and, using a peer-comparison-based algorithm, identifies which of the components exhibit anomalous behavior, relative to its peers, for a window of time.

Our *persistence ordering* takes the list of (server-perspective) anomalous components and localizes to the most problematic components by their persistent impact on overall performance.

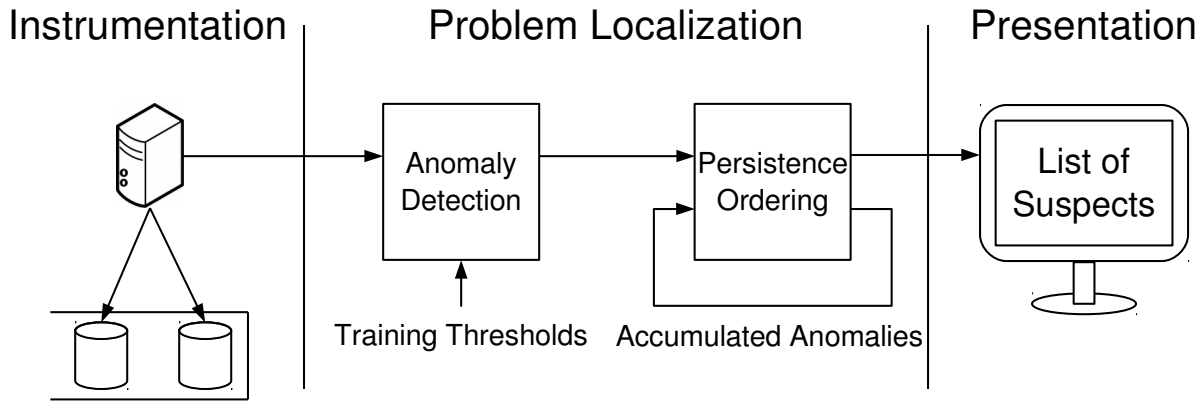

Figure 3.1: Flowchart describing our problem-diagnosis approach.

Finally, we present the problematic components as a list of suspects to an operator. By combining the suspect components with knowledge of the system topology, the operator may then infer that problems originate from a specific set of faulty LUNs, storage controllers, storage attachments, network links, or faulty file servers, and diagnose the problems to a root cause.

3.2 Instrumentation

For our problem diagnosis, we gather and analyze OS-level performance metrics, without requiring any modifications to the file system, the applications, or the OS. For our laboratory experiments (§ 4) we gather and analyze both storage and network metrics. For our case study of Intrepid's storage system (§ 5), as we are principally concerned with problems that manifest at the layer of NSD Servers and below (see § 5.2), we gather and utilize storage-related metrics only.

3.2.1 OS-level Performance Metrics

In Linux, OS-level performance metrics are made available as text files in the `/proc` pseudo file system. Table 3.1 describes the specific metrics that we collect. We use sysstat's `sadc` program [24] to periodically gather storage and network performance metrics at a sampling interval of one second, and record them in activity files. For storage resources, sysstat provides us with

Metric [s/n]*	Significance
tps [s]	Number of I/O (read and write) requests made to (a specific) LUN per second.
rd_sec [s]	Number of sectors read from the LUN per second.
wr_sec [s]	Number of sectors written to the LUN per second.
avgrq-sz [s]	Average size (in sectors) of the LUN's I/O requests.
avgqu-sz [s]	Average number of the LUN's queued I/O requests.
await [s]	Average time (in milliseconds) that a request waits to complete on the LUN; includes queuing delay and service time.
svctm [s]	Average (LUN) service time (in milliseconds) of I/O requests; does not include any queuing delay.
%util [s]	Percentage of CPU time in which I/O requests are made to the LUN.
rxpck [n]	Packets received on (a specific) network interface per second.
txpck [n]	Packets transmitted on the interface per second.
rxbyt [n]	Bytes received on the interface per second.
txbyt [n]	Bytes transmitted on the interface per second.
cwnd [n]	Number of segments (per TCP socket) allowed to be sent outstanding without acknowledgment.

*Denotes storage (s) or network (n) related metric.

Table 3.1: Black-box, OS-level performance metrics collected for analysis.

the throughput (tps, rd_sec, wr_sec) and latency (await, svctm) of the file server's I/O requests to each of the LUNs the file server is attached to. Since our instrumentation is deployed on file servers, we actually observe the compound effect of disk arrays, controllers, attachments, and the file server on the performance of these I/O requests. For network resources, sysstat provides us with throughput (rxpck, txpck, rxbyt, txbyt) metrics for each of the file server's network interfaces. In addition, /proc provides TCP congestion-control data (e.g., cwnd) [55] on a per-socket basis that we can capture with a custom tool.

In § 4.2, we describe properties of the collected storage and network metrics and their application to problem diagnosis. We also find that sadc instrumentation has negligible overhead (< 1% runtime overhead in benchmarks) and an uncompressed data volume of 3.8 kB/s on each file server in our experiment cluster. Intrepid file servers, since they have many more attached LUNs compared to our experiment cluster, yield an uncompressed data volume of 13.6 kB/s on each file server.

In general we find that await is the best single metric for problem diagnosis in parallel file systems as it reflects differences in latency due to both (i) component-level delays (e.g., read errors) and (ii) disparities in request queue length, i.e., differences in workload. Since workload disparities

also manifest in changes in throughput, instances in which `await` is anomalous but not `rd_sec` and `wr_sec` indicate a component-level problem.

3.2.2 Continuous Instrumentation

As our laboratory experiments were of relatively short duration ($\sim$600 s), we were able to spawn instances of `sadc` to record activity files for the duration of our experiments, and perform all analysis once the experiments had finished and the activity files were completely written. For Intrepid, we must continuously instrument and collect data, while also periodically performing offline analysis. To do so, we use a custom daemon, `cycle`, to spawn daily instances of `sadc` shortly after midnight UTC, at which time we are able to collect the previous day's activity files for analysis.

Although the `cycle` daemon performs a conceptually simple task, we have observed a number of practical issues in deployment that motivated the development of robust time-management features. We elaborate on our experiences with these issues in § 5.9. To summarize, the present version of `cycle` implements the following features:

- Records activity files with filenames specified with an ISO 8601-formatted UTC timestamp of `sadc`'s start time.
- Creates new daily activity files at 00:00:05 UTC, which allows up to five seconds of clock backwards-correction without creating a second activity file at 23:59 UTC on the previous day.
- Calls `sadc` to record activity files with a number of records determined by the amount of time remaining before 00:00:05 UTC the next day, as opposed to specifying a fixed number of records. This prevents drifts in file-creation time due to accumulating clock corrections. It also allows for the creation of shorter-duration activity files should a machine be rebooted in the middle of the day.

3.2.3 Other Instrumentation Sources

Although we have settled on OS-level performance metrics as being an ideal for our goals of avoiding software modifications and minimizing instrumentation overheads, in the past we have considered alternate instrumentation sources and briefly compare them here.

Syscall Traces. In [33], we investigate the use of syscalls to diagnose performance problems in addition to propagated errors and crash/hang problems in PVFS. From syscall traces we derived four performance metrics of interest: disk-read and -write service times (`dread` and `dwrite`), and server's and client's network-read times (`nsread` and `ncread`). `dread` and `dwrite` are the values of the wall-clock service times for read and write syscalls, respectively, on file server storage objects. `nsread` and `ncread` represent the amount of time that it takes for the server (client) to read a single PVFS request (response) over the network. Comparing to OS-level metrics, the summed average of `dread` and `dwrite` is analogous to the `await` metric in reflecting storage latency, while `nsread` and `ncread` offer insight into network latency not made available via `/proc`. While we find syscall tracing to have negligible overhead on large-I/O workloads with bulk transfers, we observe on metadata-intensive workloads a runtime overhead of 64%. We also observe that even among the most syscall-conservative workloads a data volume of 1 MB/s on each file server in our experiment cluster [32]. Thus, although syscall tracing is able to diagnosing non-performance problem in PVFS, we find the instrumentation overheads to be cumbersome relative to OS-level metrics.

Sample Profiles and Function-Call Traces. In [34], we investigate the use of CPU instruction-pointer sample profiles and function-call traces to localize performance problems in PVFS. For sample profiling, we run OProfile on each PVFS server to collect samples of the CPU instruction pointer along with execution context, which we then resolve to executing programs and function names (*samples* metric). For function call-tracing, we use a custom C-based module to instrument PVFS function-call sites on both entry and exit, where we count the number of times a function

is called (*count* metric) and record the (wall-clock) time spent executing inside that function (*time* metric). Comparing to syscall tracing and OS-level metrics, *time* for blocked-I/O functions is analogous to `await`, `dread`, and `dwrite` metrics in reflecting storage latency, *samples* for kernel TCP functions reflects network (TCP) throughput, and *count* for PVFS's non-blocking network poll loop reflects network latency similar to the `nsread` metric. We find that sample profiling and function-call tracing both have < 7% runtime overheads for large-I/O workloads, while function-call tracing of metadata-intensive workloads has a runtime overhead of 122%. We also observe that sample profiling and function call-tracing have data volumes of approximately 37 kB/s and 4.5 kB/s–8.0 kB/s (depending on workload) respectively on each file server in our experiment cluster. Furthermore, we note that sample profiling requires (debugging) function symbols to be present in the kernel and file server daemon, while our function-call tracing module requires file server source code to insert instrumentation at compile-time. Thus, while sample profiling and function-call tracing offer code-level insights to diagnosing performance problems, we find these instrumentation approaches difficult to deploy in production environments due to their runtime overheads and required modification of file server software.

3.3 Anomaly Detection

The purpose of anomaly detection is to determine which storage system components (LUNs and servers) are *instantaneously* reflecting anomalous, non-peer behavior. To do so, we developed a statistical peer-comparison algorithm for detecting performance problems in [36], which we refined in our later work [35]. Here we present the final version of the peer-comparison algorithm, as it completes our overall approach. In § 4.6, we describe the first version of the algorithm, and in § 5.3, we detail the differences between the two versions and motivate the specific revisions made between them.

Inevitably, any diagnosis algorithm has configurable parameters that are based on the characteristics of the data set for analysis, the pragmatic resource constraints, the specific analytical

technique being used, and the desired diagnostic accuracy. In the process of explaining our algorithms below, we also explain the intuition behind the settings of some of these parameters.

Overview. To find the faulty component, we peer-compare performance metrics (e.g., storage and network metrics) across components (e.g., LUNs and servers) to determine those with anomalous behavior. We analyze one metric at a time across all components. For each component we first perform a moving average on its metric values. We then generate a distribution function of the smoothed values over a time window of *WinSize* samples. We then compute the distance between the distributions for each pair of components, which represents the degree to which components behave differently. We then flag a component as anomalous over a window if more than half of its pairwise distribution-function distances exceed a predefined threshold. We then shift the window by $WinShift$ samples, leaving an overlap of $WinSize - WinShift$ samples between consecutive windows, and repeat the analysis. We classify a component to be faulty if it exhibits anomalous behavior for at least k of the past $2k-1$ windows.

Downsampling. As Intrepid occasionally exhibits a light workload with requests often separated by periods of inactivity, in our case study of Intrepid we downsample each storage metric to an interval of 15 s while keeping all other diagnosis parameters the same.[1] This ensures that we incorporate a reasonable quantity of non-zero metric samples in each comparison window to detect asymmetries. It also serves as a scalability improvement by decreasing analysis time, and decreasing storage and (especially) memory requirements. This is a pragmatic consideration, given that the amount of memory required would be otherwise prohibitively large.[2]

Since `sadc` records each of our storage metrics as a rate or time average, proper downsampling requires that we compute the metric's cumulative sum, that we then sample (at a different rate), to generate a new average time series. This ensures any work performed between samples is reflected in the downsampled metric just as it is in the original metric. In contrast, sampling the metric

[1] No downsampling is performed in the PVFS and Lustre test-bench experiments.

[2] We frequently ran out of memory when attempting to analyze the data of a single metric, sampling at 1 s, on machines with 4 GB RAM.

directly would lose any such work, which leads to inaccurate peer-comparison. The result of the downsampling operation is equivalent to running `sadc` with a larger sampling interval.

Moving Average Filter. Sampled storage metrics, particularly for heavy workloads, can contain a large amount of high-frequency (relative to sample rate) noise from which it is difficult to observe subtle, but sustained fault manifestations. Thus, we employ a moving average filter with a 15-sample width to remove this noise. As we do not expect faults to manifest in a periodic manner with a periodicity less than 15 samples, this filter should not unintentionally mask fault manifestations.

CDF Distances. For the refined version of our peer-comparison algorithm, we use cumulative histograms to approximate the CDF of a component's smoothed metric values. In generating the histograms we use a modified version of the Freedman-Diaconis rule [19] to select the bin size, $BinSize = 2\,\mathrm{IQR}(x)\,WinSize^{-1/3}$, and number of bins, $Bins = \lceil \mathrm{Range}(x)/BinSize \rceil$ where x contains samples across *all* LUNs in the time window. Even though the generated histograms contain samples from a single component, we compute *BinSize* using samples from all components to ensure that the resulting histograms have compatible bin parameters and, thus, are comparable. Since each histogram contains only *WinSize* samples, we compute *BinSize* using *WinSize* number of observations. Once histograms are generated for each component's values, we compute for each pair of histograms P and Q the (symmetric) distance: $d(P,Q) = \sum_{i=0}^{Bins} |P(i) - Q(i)|$, a scalar value that represents how different two histograms, and thus components, are from each other.

Windowing and Anomaly Filtering. From our laboratory experiments, we found that a *WinSize* of $\sim$60 samples encompassed enough data such that our components were observable as peers, while also maintaining a reasonable diagnosis latency (§ 4.7.2). We use a $WinShift$ of 30 samples between each window to ensure a sufficient window overlap (also 30 samples) so as to provide continuity of behavior from an analysis standpoint. We classify a LUN as faulty if it shows anomalous behavior for 3 out of the past 5 windows ($k = 3$). This filtering process reduces many of the spurious anomalies associated with sporadic asymmetry events where no underlying fault is actu-

ally present, but adds to the diagnosis latency. The $WinSize$, $WinShift$, and k values that we use, along with our moving-average filter width, were derived from our laboratory experiments as having providing the best empirical accuracy rates and are similar to the values we published in [36], while also providing for analysis windows that are round to the half-minute. The combined effects of downsampling, windowing, and anomaly filtering result in a diagnosis latency (the time from initial incident to diagnosis) of 22.5 minutes.

3.3.1 Threshold Selection

In both our laboratory experiments and case study of Intrepid, the pairwise distribution-function (e.g., CDF) distance thresholds used to differentiate faulty from fault-free components are determined through a fault-free training phase that captures the maximum expected deviation in component behavior. For our laboratory experiments, we describe its training phase details in § 4.6.1. For our case study, we use an entire day's worth of data to train thresholds for Intrepid. This is not necessarily the minimum amount of data needed for training, but it is convenient for us to use since our experiment data is grouped by days. We train using the data from the first (manually observed) fault-free day when the system sees reasonable utilization. If possible, we recommend training during stress tests that consist of known workloads, which are typically performed before a new or upgraded storage system goes into production. We can (and do) use the same thresholds in on-going diagnosis, although retraining would be necessary in the event of a system reconfiguration, e.g., if new LUNs are added. Alternatively we could retrain on a periodic (e.g., monthly) basis as a means to tolerate long-term changes to LUN performance. However, in practice, we have not witnessed a significant increase in spurious anomalies during our 15-month study.

To manually verify that a particular day is reasonably fault-free and suitable for training, we generate, for each peer group, plots of superimposed `await`s for all LUNs within that peer group. We then inspect these plots to ensure that there is no concerning asymmetry among peers, a process that is eased by the fact that most problems manifest as observable loads in normally zero-valued non-primary peer groups. Even if training data is not perfectly fault-free (either due to minor

problems that are difficult to observed from `await` plots, or because no such day exists in which faults are completely absent), the influence of faults is only to dampen alarms on the faulty components; non-faulty components remain unaffected. Thus, we recommend that training data should be sufficiently free from observable problems that an operator would feel comfortable operating the cluster indefinitely in its state at the time of training.

3.4 Persistence Ordering

While anomaly detection provides us with a reliable account of instantaneously anomalous components, systems of comparable size to Intrepid with thousands of analyzed components, nearly always exhibit one or more anomalies for any given time window, even in the absence of an observable performance degradation.

Motivation. The fact that anomalies "always exist" is a key fact that requires us to alter our focus as we graduate from test-bench clusters to performing problem diagnosis on real systems. In our laboratory experiments, instantaneous anomalies were rare and either reflected the presence of our injected faults (which we aimed to observe), or the occurrence of false positive (which we aimed to avoid). However, in Intrepid, "spurious" anomalies (even with anomaly filtering) are common enough that we simply cannot raise alarms on each. It is also not possible to completely avoid the alarms through tweaking of analysis parameters (filter width, $WinSize$ and $WinShift$, etc.).

Investigating these spurious anomalies, we find that many are clear instances of transient asymmetries in our raw instrumentation data, due to occasional but regular events where behavior deviates across components. Thus, for Intrepid, we focus our concern on locating system components that demonstrate long-term, or *persistent* anomalies, because they are suggestive of possible impending component failures or problems that might require manual intervention in order to resolve.

Algorithm. To localize persistent anomalies, it is necessary for us to order the list of anomalous components by a measure of their impact on overall performance. To do so, we maintain a

positive-value accumulator for every component in which we add one ($+1$) for each window where the component is anomalous, and subtract one (-1, and only if the accumulator is > 0) for each window where the component is not. We then present to the operator a list of *persistently* anomalous components that are ordered by decreasing accumulator value, i.e., the top-most component in the list is that which has the most number of anomalous windows in its recent history.

Chapter 4

Small Scale Validation: Lab Experiments

To demonstrate proof-of-concept of our peer-comparison algorithm, we first performed a set of laboratory experiments by injecting four realistic problems during execution of synthetic workloads on controlled, test-bench PVFS and Lustre storage clusters of up to 12 file servers.

4.1 Problems Studied for Diagnosis

We separate problems involving storage and network resources into two classes. The first class is *hog* faults, where a rogue process on the monitored file servers induces an unusually high workload for the specific resource. The second class is *busy* or *loss* faults, where an unmonitored (i.e., outside the scope of the server OSes) third party creates a condition that causes a performance degradation for the specific resource. To explore all combinations of problem resource and class, we study the diagnosis of four problems—disk-hog, disk-busy, network-hog, packet-loss (network-busy).

Disk-hogs can result from a runaway, but otherwise benign, process. They may occur due to unexpected `cron` jobs, e.g., an `updatedb` process generating a file/directory index for GNU `locate`, or a monthly software-RAID array verification check. Disk-busy faults can also occur in shared-storage systems due to a third-party/unmonitored node that runs a disk-hog process on the shared-storage device; we view this differently from a regular disk-hog because the increased load on the shared-storage device is not observable as a throughput increase at the monitored servers.

Network-hogs can result from a local traffic-emitter (e.g., a backup process), or the receipt of data during a denial-of-service attack. Network-hogs are observable as increased throughput (but not necessarily "goodput") at the monitored file servers. Packet-loss faults might be the result of network congestion, e.g., due to a network-hog on a nearby unmonitored node or due to packet corruption and losses from a failing NIC.

4.2 Parallel File System Behavior

We highlight our (empirical) observations of PVFS's and Lustre's behavior that we believe is characteristic of stripe-based parallel file systems.

[Observation 1] *In a homogeneous (i.e., identical hardware) cluster, I/O servers (specifically, their LUNs) track each other closely in throughput and latency, under fault-free conditions.*

For N I/O servers, each with one LUN, I/O requests of size greater than $(N-1) \times stripe\_size$ results in I/O on each server (and henceforth, its LUN) for a single request. Multiple I/O requests on the same file, even for smaller request sizes, will quickly generate workloads[1] on all servers. Even I/O requests to files smaller than $stripe\_size$ will generate workloads on all I/O servers, as long as enough small files are read/written. We observed this for all three target benchmarks, dd, IOzone, and PostMark. For metadata-intensive workloads, we expect that metadata servers also track each other in proportional magnitudes of throughput and latency.

[Observation 2] *When a fault occurs on at least one of the I/O servers, the other (fault-free) I/O servers experience an identical drop in throughput.*

When a client syscall involves requests to multiple I/O servers, the client must wait for all of these servers to respond before proceeding to the next syscall.[2] Thus, the client-perceived cluster

[1] Pathological workloads might not result in equitable workload distribution across I/O servers; one server would be disproportionately deluged with requests, while the other servers are idle, e.g., a workload that constantly rewrites the same *stripe_size* chunk of a file.

[2] Since Lustre performs client side caching and readahead, client I/O syscalls may return immediately even if the corresponding file server is faulty. Even so, a maximum of 32 MB may be cached (or 40 MB pre-read) before Lustre

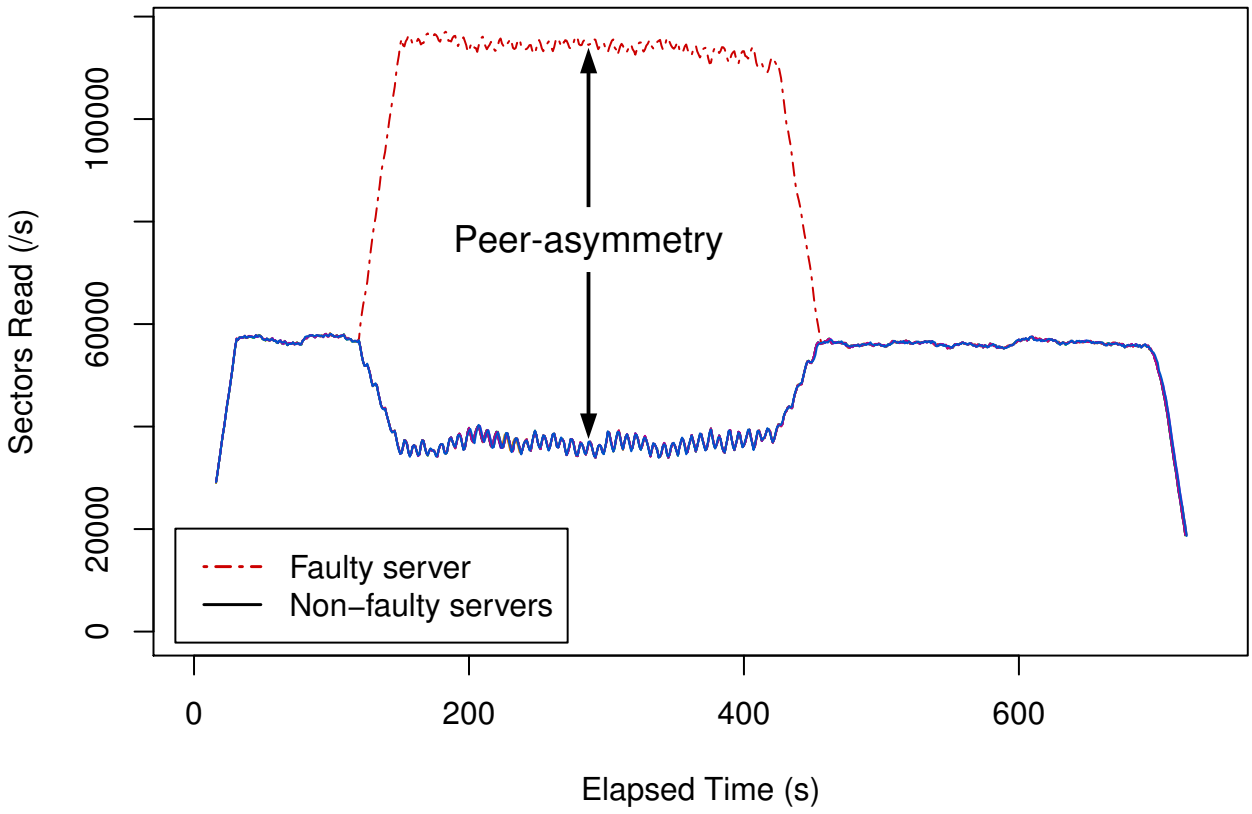

Figure 4.1: Peer-asymmetry of `rd_sec` for `iozoner` workload with *disk-hog* fault.

performance is constrained by the slowest server. We call this the *bottlenecking condition*. When a server experiences a performance fault, that server's per-request service-time increases. Because the client blocks on the syscall until it receives all server responses, the client's syscall-service time also increases. This leads to slower application progress and fewer requests per second from the client, resulting in a proportional decrease in throughput on all I/O servers.

[Observation 3] *When a performance fault occurs on at least one of the I/O servers, the other (fault-free) I/O servers are unaffected in their per-request service times.*

Because there is no server-server communication (i.e., no server inter-dependencies), a performance problem at one server will not adversely impact latency (per-request service-time) at the other servers. If these servers were previously highly loaded, latency might even improve (due to potentially decreased resource contention).

must wait for responses.

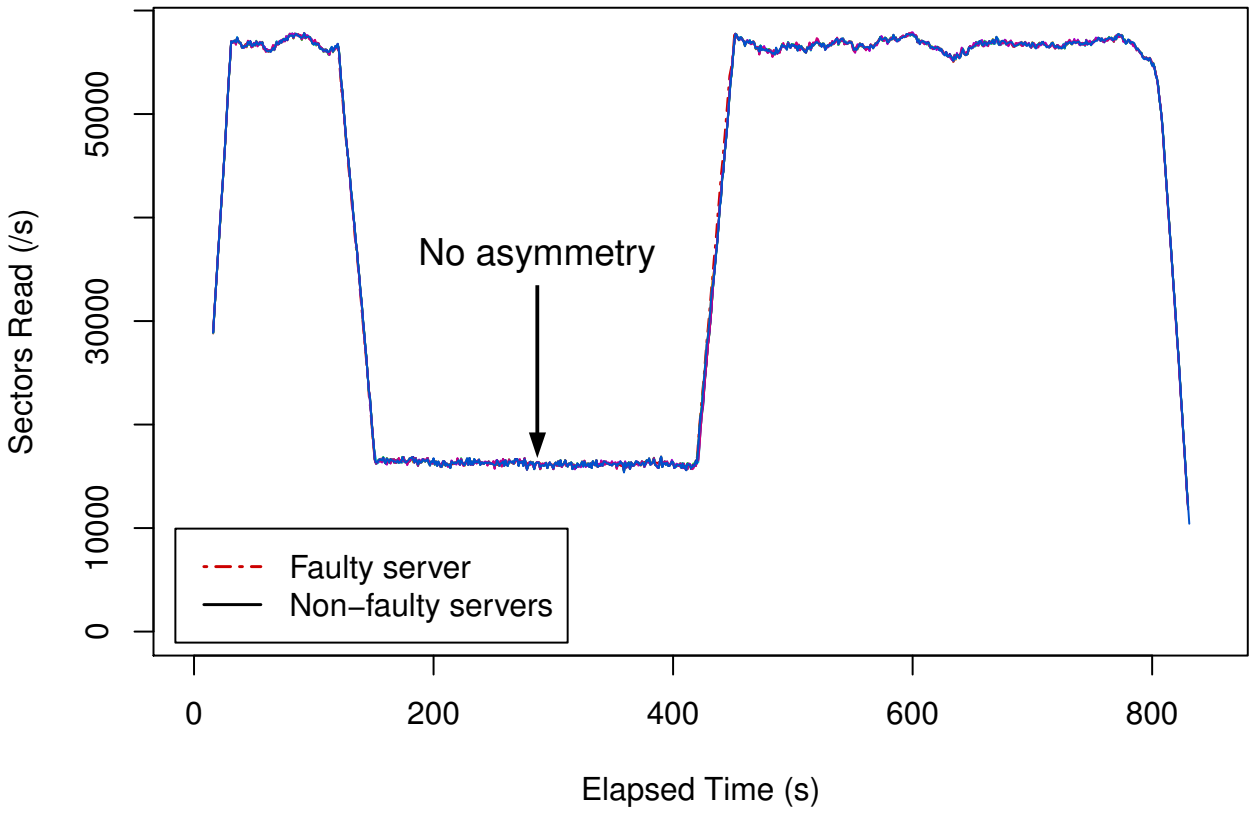

Figure 4.2: No asymmetry of `rd_sec` for `iozoner` workload with *disk-busy* fault.

[Observation 4] *For disk/network-hog faults, storage/network-throughput increases at the faulty server and decreases at the non-faulty servers.*

A disk/network-hog fault at a server is due to a third-party that creates additional I/O traffic that is observed as increased storage/network-throughput. The additional I/O traffic creates resource contention that ultimately manifests as a decrease in file-server throughput on all servers (causing the bottlenecking condition of observation 2). Thus, disk- and network-hog faults can be localized to the faulty server by looking for *peer-divergence* (i.e. *asymmetry* across peers) in the storage- and network-throughput metrics, respectively, as seen in Figure 4.1.

[Observation 5] *For disk-busy (packet-loss) faults, storage- (network-) throughput decreases on all servers.*

For disk-busy (packet-loss) faults, there is no asymmetry in storage (network) throughputs across I/O servers (because there is no other process to create observable throughput, and the server

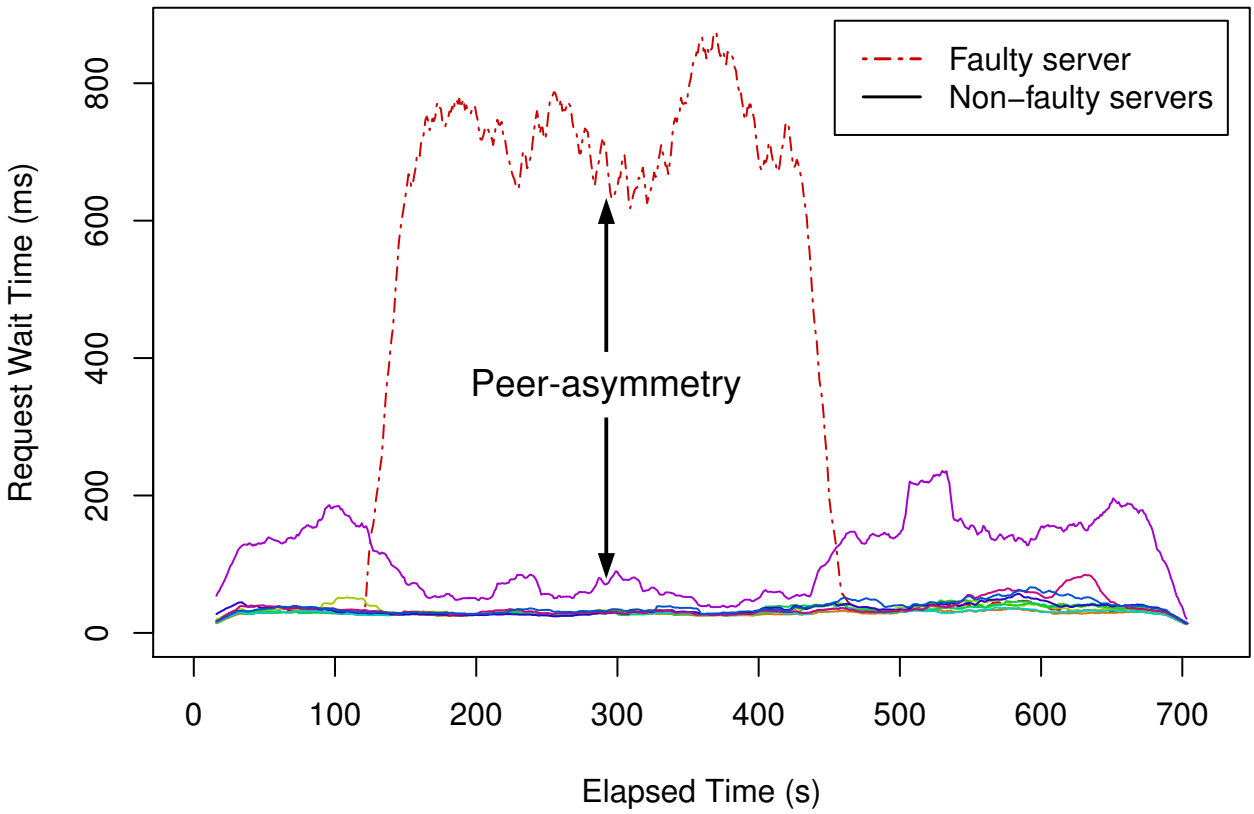

Figure 4.3: Peer-asymmetry of `await` for `ddr` workload with *disk-hog* fault.

daemon has the same throughput at all the nodes). Instead, there is a symmetric decrease in the storage-(network-) throughput metrics across all servers. Because asymmetry does not arise, such faults cannot be diagnosed, as seen in Figure 4.2.

[Observation 6] *For disk-busy and disk-hog faults, storage-latency increases on the faulty server and decreases at the non-faulty servers.*

For disk-busy and disk-hog faults, `await`, `avgqu-sz` and `%util` increase at the faulty server as the disk's responsiveness decreases and requests start to backlog. The increased `await` on the faulty server causes an increased server response-time, making the client wait longer before it can issue its next request. The additional delay that the client experiences reduces its I/O throughput, resulting in the fault-free servers having increased idle time. Thus, the `await` and `%util` metrics decrease asymmetrically on the fault-free I/O servers, enabling a peer-comparison diagnosis of the disk-hog and disk-busy faults, as seen in Figure 4.3.

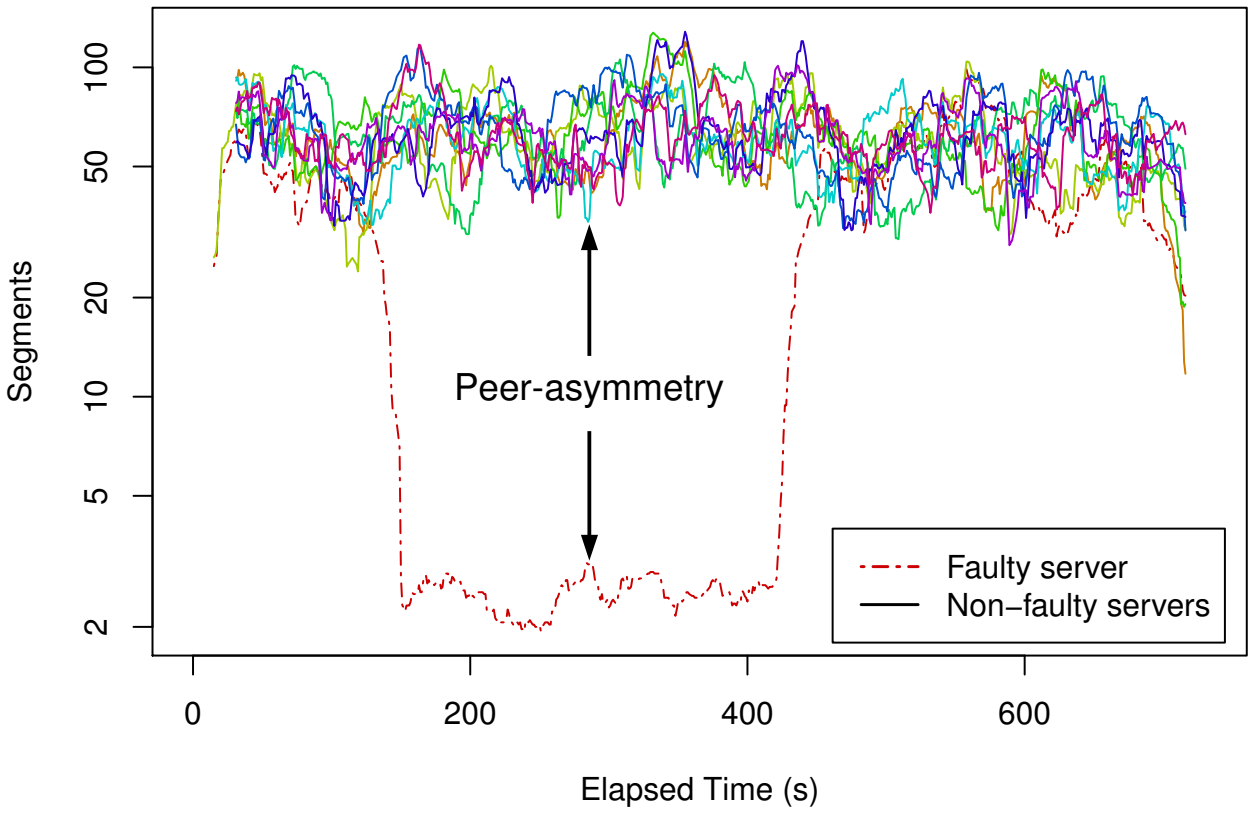

Figure 4.4: Peer-asymmetry of `cwnd` for `ddw` workload with *receive-pktloss* fault.

[Observation 7] *For network-hog and packet-loss faults, the TCP congestion-control window decreases significantly and asymmetrically on the faulty server.*

The goal of TCP congestion control is to allow `cwnd` to be as large as possible, without experiencing packet-loss due to overfilling packet queues. When packet-loss occurs and is recovered within the retransmission timeout interval, the congestion window is halved. If recovery takes longer than retransmission timeout, `cwnd` is reduced to one segment. When nodes are transmitting data, their `cwnd` metrics either stabilize at high (≈100) values or oscillate (between ≈10–100) as congestion is observed on the network. However, during (some) network-hog and (all) packet-loss experiments, `cwnd`s of connections to the faulty server dropped by several orders of magnitude to single-digit values and held steady until the fault was removed, at which time the congestion window was allowed to open again. These asymmetric sustained drops in `cwnd` enable peer-comparison diagnosis for network faults, as seen in Figure 4.4.

4.3 Discussion on Metrics

Although faults present in multiple metrics, we have observed that not all metrics are appropriate for diagnosis as they exhibit inconsistent behaviors. Here we describe problematic metrics.

Storage-Throughput Metrics. There is a notable relationship between the storage-throughput metrics: $\texttt{tps} \times \texttt{avgrq-sz} = \texttt{rd\_sec} + \texttt{wr\_sec}$. While `rd_sec` and `wr_sec` accurately capture real storage activity and strongly correlate across I/O servers, `tps` and `avgrq-sz` do not correlate as strongly because a lower transfer rate may be compensated by issuing larger-sized requests. Thus, `tps` is not a reliable metric for diagnosis.

`svctm`. The impact of disk faults on `svctm` is inconsistent. The influences on storage service times are: time to locate the starting sector (seek time and rotational delay), media-transfer time, reread/rewrite time in the event of a read/write error, and delay time to due servicing of unobservable requests. During a disk fault, servicing of interleaved requests increases seek time. Thus, for an unchanged `avgrq-sz`, `svctm` will increase asymmetrically on the faulty server. Furthermore, during a disk-busy fault, servicing of unobservable requests further increases `svctm` due to request delays. However, during a disk-hog fault, the hog process might be issuing requests of smaller sizes than PVFS/Lustre. If so, then the associated decrease in media-transfer time might offset the increase in seek time resulting in a decreased or unchanged `svctm`. Thus, `svctm` is not guaranteed to exhibit asymmetries for disk-hogs, and therefore is unreliable.

Other Metrics. While problems manifest on other metrics (e.g., CPU usage, context-switch rate), these secondary manifestations are due to the overall reduction in I/O throughput during the faulty period, and reveal nothing new. Thus, we do not analyze these metrics.

4.4 Experimental Set-Up

We perform our experiments on AMD Opteron 1220 machines, each with 4 GB RAM, two Seagate Barracuda 7200.10 320 GB disks (one dedicated for PVFS/Lustre storage), and a Broadcom NetXtreme BCM5721 Gigabit Ethernet controller. Each node runs Debian GNU/Linux 4.0 (etch) with Linux kernel 2.6.18. The machines run in stock configuration with background tasks turned off. We conduct experiments with x/y configurations, i.e., the PVFS x/y cluster comprises y combined I/O and metadata servers and x clients, while the equivalent Lustre x/y cluster comprises y object storage (I/O) servers with a single object storage target each, a single (dedicated) metadata server, and x clients. We conduct our experiments for 10/10 and 6/12 PVFS and Lustre clusters;[3] we explain the 10/10 cluster experiments in detail, but our observations carry to both.

For these experiments PVFS 2.8.0 is used in the default server (`pvfs2-genconfig` generated) configuration with two modifications. First, we use the Direct I/O method (`TroveMethod directio`) to bypass the Linux buffer cache for PVFS I/O server storage. This is required for diagnosis as we otherwise observe disparate I/O server behavior during IOzone's rewrite phase. Although bypassing the buffer cache has no effect on diagnosis for non-rewrite (e.g., `ddw`) workloads, it does improve large write throughput by 10%.

Second, we increase to 4 MB (from 256 kB) the Flow buffer size (`FlowBufferSizeBytes`) to allow larger bulk data transfers and enable more efficient disk usage. This modification is standard practice in PVFS performance tuning, and is required to make our testbed performance representative of real deployments. It does not appear to affect diagnosis capability. In addition, we patch the PVFS kernel client to eliminate the 128 MB total size restriction on the `/dev/pvfs2-req` device request buffers and to `vmalloc` memory (instead of `kmalloc`) for the buffer page map (`bufmap_page_array`) to ensure that larger request buffers are actually allocatable. We then invoke the PVFS kernel client with 64 MB request buffers (`desc-size` parameter) in order to make the 4 MB data transfers to each of the I/O servers.

[3] Due to a limited number of nodes we were unable to experiment with higher active client/server ratios. However, with the workloads and faults tested, an increased number of clients appears to degrade per-client throughput with no significant change in other behavior.

For Lustre experiments we use the etch backport of the Lustre 1.6.6 Debian packages in the default server configuration with a single modification to set the `lov.stripecount` parameter to −1 to stripe files across each object storage target (I/O server).

The nodes are rebooted immediately prior to the start of each experiment. Time synchronization is performed at boot-time using `ntpdate`. Once the servers are initialized and the client is mounted, monitoring agents start capturing metrics to a local (non-storage dedicated) disk. `sync` is then performed, followed by a 15-second sleep, and the experiment benchmark is run. The benchmark runs fault-free for 120 seconds prior to fault injection. The fault is then injected for 300 seconds and then deactivated. The experiment continues to the completion of the benchmark, which ideally runs for a total of 600 seconds in the fault-free case. This run time allows the benchmark to run for at least 180 seconds after a fault's deactivation to determine if there are any delayed effects. We run ten experiments for each workload & fault combination, using a different faulty server for each iteration.

4.4.1 Workloads

We use five experiment workloads derived from three experiment benchmarks: dd, IOzone, and PostMark. The same workload is invoked concurrently on all clients. The first two workloads, `ddw` and `ddr`, either write zeros (from `/dev/zero`[4]) to a client-specific temporary file or read the contents of a previously written client-specific temporary file and write the output to `/dev/null`.

`dd` [58] performs a constant-rate, constant-workload large-file read/write from/to disk. It is the simplest large-file benchmark to run, and helps us to analyze and understand the system's behavior prior to running more complicated workloads. `dd` models the behavior of scientific-computing workloads with constant data-write rates.

Our next two workloads, `iozonew` and `iozoner`, consist of the same file-system benchmark,

[4]We use `/dev/zero` to minimize read impact on system metrics. Reading from other physical block devices creates a confounding I/O workload, `/dev/urandom` is CPU-intensive, and `/dev/mem` does not source enough data. There is no evidence that compression is performed anywhere in the request path or that PVFS/Lustre treat zero data different from arbitrary data.

IOzone v3.283 [7]. We run `iozonew` in write/rewrite mode and `iozoner` in read/reread mode. IOzone's behavior is similar to `dd` in that it has two constant read/write phases. Thus, IOzone is a large-file I/O-heavy benchmark with few metadata operations. However, there is an `fsync` and a workload change half-way through.

Our fifth benchmark is PostMark v1.51 [37]. PostMark was chosen as a metadata-server heavy workload with small file writes (all writes < 64 kB thus, writes occur only on a single I/O server per file).

4.4.2 Configurations of Workloads

For the `ddw` workload, we use a 17 GB file with a record-size of 40 MB for PVFS, and a 30 GB file is used with a record-size 10 MB for Lustre. File sizes are chosen to result in a fault-free experiment runtime of approximately 600 seconds. The PVFS record-size was chosen to result in 4 MB bulk data transfers to each I/O server, which we empirically determined to be the knee of the performance vs. record-size curve. The Lustre record-size was chosen to result in 1 MB bulk data transfers to each I/O server—the maximum payload size of a Lustre RPC. Since Lustre both aggregates client writes and performs readahead, varying the record-size does not significantly alter Lustre read or write performance. For `ddr` we use a 27 GB file with a record-size of 40 MB for PVFS, and a 30 GB file with a record-size of 10 MB for Lustre (same as `ddw`).

For both the `iozonew` and `iozoner` workloads, we use an 8 GB file with a record-size of 16 MB (the largest that IOzone supports) for PVFS. For Lustre we use a 9 GB file with a record-size of 10 MB for `iozonew`, and a 16 GB file with the same record-size for `iozoner`. For `postmark` we use its default configuration with 16,000 transactions for PVFS and 53,000 transactions for Lustre to give a sufficiently long-running benchmark.

4.5 Fault Injection

In our fault-induced experiments, we inject a single fault at a time into one of the I/O servers to induce degraded performance for either network or storage resources. We inject the following faults:

- *disk-hog*: a `dd` process that reads 256 MB blocks (using direct I/O) from an unused storage disk partition.
- *disk-busy*: an `sgm_dd` process [23] that issues low-level SCSI I/O commands via the Linux SCSI Generic (`sg`) driver to read 1 MB blocks from the same unused storage disk partition.
- *network-hog*: a third-party node opens a TCP connection to a listening port on one of the PVFS I/O servers and sends zeros to it (*write-network-hog*), or an I/O server opens a connection and sends zeros to a third party node (*read-network-hog*).
- *pktloss*: a netfilter firewall rule that (probabilistically) drops packets received at one of the I/O servers with probability 5% (*receive-pktloss*), or a firewall rule on all clients that drops packets incoming from a single server with probability 5% (*send-pktloss*).

Real packet-loss due to a failing NIC can show up in black-box metrics as Ethernet frame errors, while our synthetic strategy will not. We believe that our fault-injection strategy is more general because it also emulates silent losses instead of corruptive losses alone. A caveat is that the network-throughput data (packets received) for the faulty server does count the packets as having been lost, making the data slightly inflated compared to reality. From our peer-comparison approach's viewpoint, this is a hindrance as the faulty server's network-throughput data will look closer to those of non-faulty nodes. In reality, the 5% packet-loss is insignificant compared to the order of magnitude decrease in network-throughput due to the behavior of TCP's congestion control.

4.6 Diagnosis Algorithm

For the faults studied in our laboratory experiments our diagnostic algorithm consists of two phases: (i) anomaly detection, where we identify the faulty server, and (ii) root-cause analysis, where we identify the resource at fault.

4.6.1 Anomaly Detection: Finding the Faulty Server

For the first version of our peer-comparison algorithm (§ 3.3 presents an overview of our revised algorithm) we considered several statistical properties (e.g., the mean, the variance, etc. of a metric) as candidates for peer-comparison across servers, but ultimately chose the probability distribution function (PDF) of each metric because it captures many of the metric's statistical properties. Figure 4.5 shows the asymmetry in a metric's histograms/PDFs between the faulty and fault-free servers.

Histogram-Based Approach. We determine the PDFs, using histograms as an approximation, of a specific black-box metric values over a window of time (of size *WinSize* seconds) at each I/O server. To compare the resulting PDFs across the different I/O servers, we use a standard measure, the Kullback-Leibler (KL) divergence [14], as the distance between two distribution functions, P and Q. The KL divergence of a distribution function, Q, from the distribution function, P, is given by $D(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$. We use a symmetric version of the KL divergence, given by $D'(P||Q) = \frac{1}{2}[D(P||Q) + D(Q||P)]$ in our analysis.

We perform the following procedure for each of metric of interest. Using i to represent one of these metrics, we first perform a moving average on i. We then take PDFs of the smoothed i for two distinct I/O servers at a time and compute their pairwise KL divergences. A pairwise KL-divergence value for i is flagged as anomalous if it is greater than a certain predefined threshold. An I/O server is flagged as anomalous if its pairwise KL-divergence for i is anomalous with more than half of the other servers for at least k of the past $2k-1$ windows. We then shift the window by $WinShift$ samples, leaving an overlap of $WinSize - WinShift$ samples between consecutive

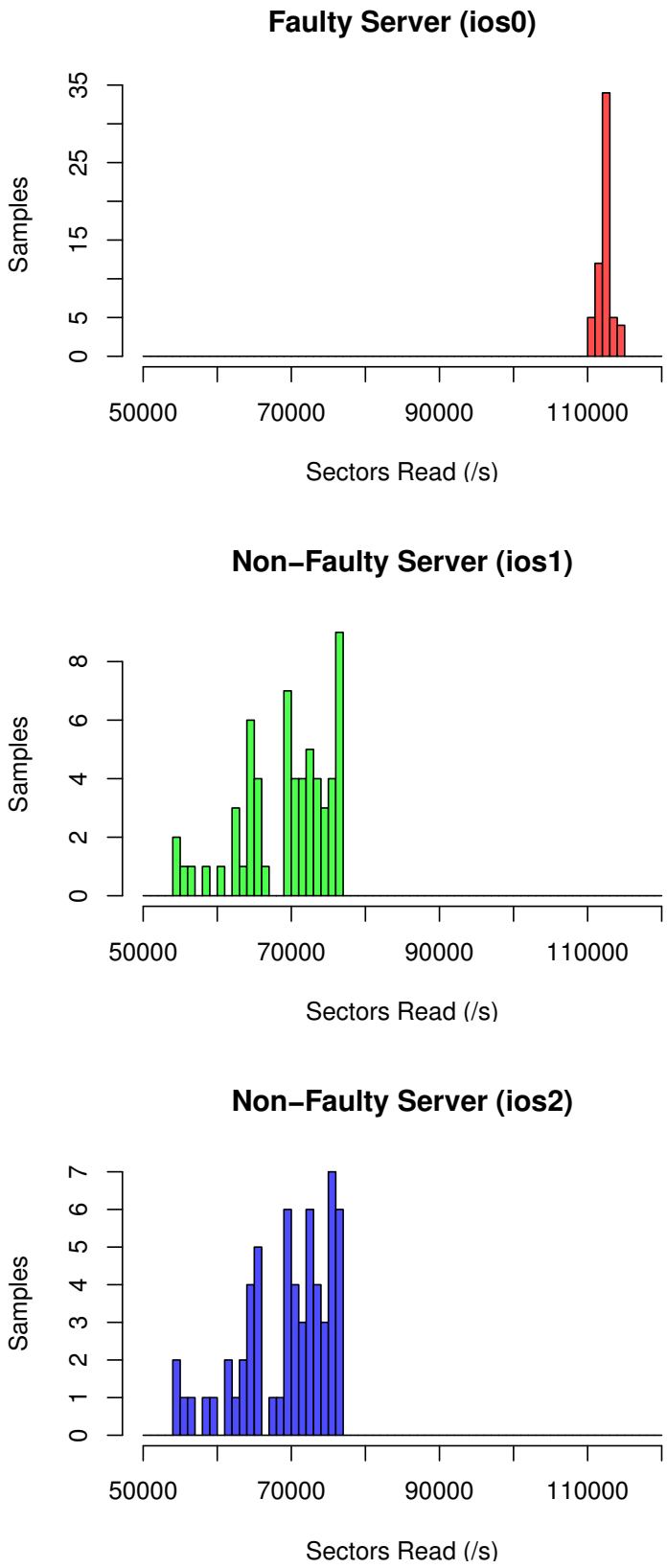

Figure 4.5: Histograms of `rd_sec` (`ddr` with *disk-hog* fault) for one faulty and two non-faulty servers.

windows, and repeat the analysis.

We use a 5-point moving average to ensure that metrics reflect average behavior of request processing. We also use a *WinSize* of 64, a $WinShift$ of 32, and a k of 3 in our analysis to incorporate a reasonable quantity of data samples per comparison while maintaining a reasonable diagnosis latency (approximately 90 seconds). We investigate the useful ranges of these values in § 4.7.2.

Time Series-Based Approach. We use the histogram-based approach for all metrics except `cwnd`. Unlike other metrics, `cwnd` tends to be noisy under normal conditions with no closely-coupled peer behavior. This is expected as TCP congestion control prevents synchronized connections from fully utilizing link capacity. Since we found that the histogram-based approach insufficiently detects network (packet-loss) problems with this metric, we have adopted a time-series based approach that is unique to the analysis of `cwnd` in our laboratory experiments.

Fortunately, there is a simple heuristic that we exploit for detecting packet-loss using `cwnd`. TCP congestion control responds to packet-loss by halving `cwnd`, which results `cwnd` exponential decay after multiple loss events. When viewed on a logarithmic scale, sustained packet-loss results in a linear decrease for each packet lost.

To support analysis of `cwnd`, we first generate a time-series by performing a moving average on `cwnd` with a window size of 31 seconds and computing its base-two logarithm. Based on empirical observation, the moving-average filter attenuates the effect of sporadic transmission timeout events while enabling reasonable diagnosis latencies (i.e., under one minute). Then, every second, a representative value (median) is computed of the log-`cwnd` values. A server is indicted if, at the sample time, its log-`cwnd` is less than a predetermined fraction (threshold) of the median.

Threshold Selection. Both the histogram and time-series analysis algorithms require thresholds to differentiate between faulty and fault-free servers. We determine the thresholds through a fault-free training phase that captures the maximum expected deviation in server performance.

We do not need to train against all potential workloads, instead we train on workloads that are expected to stress the system to its limits of performance. Since server performance devi-

ates the most when resources are saturated (and thus, are unable to "keep up" with other nodes), these thresholds represent the maximum expected performance deviations under normal operation. Less intense workloads, since they do not saturate server resources, are expected to exhibit better coupled peer behavior. In our experiments, we train with 10 iterations of the `ddr`, `ddw`, and `postmark` fault-free workloads. The same metrics are captured during training as when performing diagnosis.

To train the histogram algorithm, for each metric, we start with a minimum threshold value (currently 0.1) and increase in increments (of 0.1) until the minimum threshold is determined that eliminates all anomalies on a particular server. This server-specific threshold is doubled to provide a cushion that masks minor manifestations occurring during the fault period. This is based on the premise that a fault's primary manifestation will cause a metric to be sufficiently asymmetric, roughly an order of magnitude, yielding a "safe window" of thresholds that can be used without altering the diagnosis.

Training the time-series algorithm is similar, except that the final threshold is not doubled as the `cwnd` metric is very sensitive, yielding a much smaller corresponding "safe window". Also, only two thresholds are determined for `cwnd`, one for all servers sending to clients, and one for clients sending to servers. As `cwnd` is generally not influenced by the performance of specific hardware, its behavior is consistent across nodes.

4.6.2 Root-Cause Analysis

In addition to identifying the faulty server, we also infer the resource that is the root cause of the problem through an expert derived checklist. This checklist, based on our observations (§ 4.2) of PVFS's/Lustre's behavior, maps sets of peer-divergent metrics to the root cause. Where multiple metrics may be used, the specific metrics selected are chosen for consistency of behavior (see § 4.3). If we observe peer-divergence at any step of the checklist, we halt at that step and arrive at the root cause and faulty server. If peer-divergence is not observed at that step, we continue to the next step of decision-making.

Do we observe peer-divergence in ...

1.	Storage throughput?	Yes: disk-hog fault
	(`rd_sec` or `wr_sec`)	No: next question
2.	Storage latency?	Yes: disk-busy fault
	(`await`)	No: ...
3.	Network throughput?*	Yes: network-hog fault
	(`rxbyt` or `txbyt`)	No: ...
4.	Network congestion?	Yes: packet-loss fault
	(`cwnd`)	No: no fault discovered

*Must diverge in both `rxbyt` & `txbyt`, or in absence of peer-divergence in `cwnd` (see § 4.8).

4.7 Results

PVFS Results. Tables 4.1 and 4.2 shows the accuracy (true- and false-positive rates) of our diagnosis algorithm in indicting faulty nodes (ITP/IFP) and diagnosing root causes (DTP/DFP)[5] for the PVFS 10/10 & 6/12 clusters.

It is notable that not all faults manifest equally on all workloads. *disk-hog*, *disk-busy*, and *read-network-hog* all exhibit a significant (> 10%) runtime increase for all workloads. In contrast, the *receive-pktloss* and *send-pktloss* only have significant impact on runtime for write-heavy and read-heavy workloads respectively. Correspondingly, faults with greater runtime impact are often the most reliably diagnosed. Since packet-loss faults have negligible impact on `ddr` & `ddw` ACK flows and `postmark` (where lost packets are recovered quickly), it is reasonable to expect to not be able to diagnose them.

When removing the workloads for which packet-loss cannot be observed (and thus, not diagnosed), the aggregate diagnosis rates improve to 96.3% ITP and 94.6% DTP in the 10/10 cluster,

[5]ITP is the percentage of experiments where all faulty servers are correctly indicted as faulty, IFP is the percentage where at least one non-faulty server is misindicted as faulty. DTP is the percentage of experiments where all faults are successfully diagnosed to their root causes, DFP is the percentage where at least one fault is misdiagnosed to a wrong root cause (including misindictments).

and to 67.2% ITP and 58.8% DTP in the 6/12 cluster.

Lustre Results. Tables 4.3 and 4.4 shows the accuracy of our diagnosis algorithm for the Lustre 10/10 & 6/12 clusters. When removing workloads for which packet-loss cannot be observed, the aggregate diagnosis rates improve to 92.5% ITP and 86.3% DTP in the 10/10 cluster, and to 90.0% ITP and 82.1% DTP in the 6/12 case.

Both 10/10 clusters exhibit comparable accuracy rates. In contrast, the PVFS 6/12 cluster exhibits masked network-hogs faults (fewer true-positives) due to low network throughput thresholds from training with unbalanced metadata request workloads (see § 4.8). The Lustre 6/12 cluster exhibits more misdiagnoses (higher false-positives) due to minor, secondary manifestations in storage throughput. This suggests that our analysis algorithm may be refined with a ranking mechanism that allows diagnosis to tolerate secondary manifestations. In § 3.4, we introduce persistence ordering as one such ranking mechanism motivated by our case study of Intrepid.

4.7.1 Diagnosis Overheads & Scalability

Instrumentation Overhead. Table 4.5 reports runtime overheads for instrumentation of both PVFS and Lustre for our five workloads. Overheads are calculated as the increase in mean workload runtime (for 10 iterations) with respect to their uninstrumented counterparts. Negative overheads are result of sampling error, which is high due runtime variance across experiments. The PVFS workload with the least runtime variance (`iozoner`) exhibits, with 99% confidence, a runtime overhead < 1%. As the server load of this workload is comparable to the others, we conclude that OS-level instrumentation has negligible impact on throughput and performance.

Data Volume. The performance metrics collected by `sadc` have an uncompressed data volume of 3.8 kB/s on each server node, independent of workload or number of clients. The congestion-control metrics sampled from `/proc/net/tcp` have a data volume of 150 B/s per socket on each client & server node. While the volume of congestion-control data linearly increases with number

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	0.0%	0.0%	0.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	90.0%	2.0%	90.0%	2.0%
write-network-hog	92.0%	0.0%	84.0%	8.0%
read-network-hog	100.0%	0.0%	100.0%	0.0%
receive-pktloss	42.0%	0.0%	42.0%	0.0%
send-pktloss	40.0%	0.0%	40.0%	0.0%
Aggregate	77.3%	0.3%	76.0%	1.4%

Table 4.1: Results of PVFS diagnosis for the 10/10 cluster.

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	2.0%	0.0%	2.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	100.0%	0.0%	100.0%	0.0%
write-network-hog	42.0%	2.0%	0.0%	44.0%
read-network-hog	0.0%	2.0%	0.0%	2.0%
receive-pktloss	54.0%	6.0%	54.0%	6.0%
send-pktloss	40.0%	2.0%	40.0%	2.0%
Aggregate	56.0%	2.0%	49.0%	8.0%

Table 4.2: Results of PVFS diagnosis for the 6/12 cluster.

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	0.0%	0.0%	0.0%
disk-hog	82.0%	0.0%	82.0%	0.0%
disk-busy	88.0%	2.0%	68.0%	22.0%
write-network-hog	98.0%	2.0%	96.0%	4.0%
read-network-hog	98.0%	2.0%	94.0%	6.0%
receive-pktloss	38.0%	4.0%	36.0%	6.0%
send-pktloss	40.0%	0.0%	38.0%	2.0%
Aggregate	74.0%	1.4%	69.0%	5.7%

Table 4.3: Results of Lustre diagnosis for the 10/10 cluster.

Fault	ITP	IFP	DTP	DFP
None (control)	0.0%	6.0%	0.0%	6.0%
disk-hog	100.0%	0.0%	100.0%	0.0%
disk-busy	76.0%	8.0%	38.0%	46.0%
write-network-hog	86.0%	14.0%	86.0%	14.0%
read-network-hog	92.0%	8.0%	92.0%	8.0%
receive-pktloss	40.0%	2.0%	40.0%	2.0%
send-pktloss	38.0%	8.0%	38.0%	8.0%
Aggregate	72.0%	6.6%	65.7%	12.0%

Table 4.4: Results of Lustre diagnosis for the 6/12 cluster.

Overhead for Workload	File System	
	PVFS	Lustre
`ddr`	0.90% ± 0.62%	1.81% ± 1.71%
`ddw`	0.00% ± 1.03%	–0.22% ± 1.18%
`iozoner`	–0.07% ± 0.37%	0.70% ± 0.98%
`iozonew`	–0.77% ± 1.62%	0.53% ± 2.71%
`postmark`	–0.58% ± 1.49%	0.20% ± 1.28%

Table 4.5: Instrumentation overhead: Increase in runtime w.r.t. non-instrumented workload ± standard error.

of clients, it is not necessary to collect per-socket data for all clients. At minimum, congestion-control data needs to be collected for only a single active client per time window. Collecting congestion-control data from additional clients merely ensures that server packet-loss effects are observed by a representative number of clients.

Algorithm Scalability. Our analysis code requires, every second, 3.44 ms per server and 182 µs per server pair of CPU time on a 2.4 GHz dedicated core to diagnose a fault if any exists. Therefore, realtime diagnosis of up to 88 servers may be supported on a single 2.4 GHz core.

Although the pairwise analysis algorithm is $O(n^2)$, we recognize that it is not necessary to compare a given server against all others in every analysis window. To support very large clusters (thousands of servers), we recommend partitioning n servers into $n-k$ analysis domains of k (e.g., 10) servers each, and only performing pairwise comparisons within these partitions. To avoid undetected anomalies that might develop in static partitions, we recommend rotating partition membership in each analysis window. Although we have not yet tested this technique, it does allow for $O(n)$ scalability.

4.7.2 Sensitivity

Histogram Moving-Average Span. Due to large record sizes, some workload & fault combinations (e.g., `ddr` & disk-busy) yield request processing times up to 4 s. As client requests often synchronize (see § 4.8), metrics may reflect distinct request processing stages instead of aggregate behavior. For example, during a disk fault, the faulty server performs long, low-throughput storage

operations while fault-free servers perform short, high-throughput operations. At 1 s resolution, these behaviors reflect asymmetrically in many metrics. While this feature results in high (79%) ITP rates, its presence in nearly all metrics results in high (10%) DFP rates as well. Furthermore, since the influence of this feature is dependent on workload and number of clients, it is not reliable, and therefore, it is important to perform metric smoothing.

However, "too much" smoothing eliminates medium-term variances, decreasing TP and increasing FP rates. With 9-point smoothing, DFP (11%) exceeds unsmoothed while DTP reduces by 11% to 58.3%. Therefore we chose 5-point smoothing to minimize IFP (2.4%) and DFP (6.7%) with a modest decrease in DTP (64.9%).

Anomalous Window Filtering. In histogram-based analysis, servers are flagged anomalous only if they demonstrate anomalies in k of the past $2k-1$ windows. This filtering reduces false-positives in the event of sporadic anomalous windows when no underlying fault is present. k in the range 3–7 exhibits a consistent 6% increase in ITP/DTP and a 1% decrease in IFP/DFP over the non-filtered case. For $k \geq 8$, the TP/FP rates decrease/increase again. We expect k's useful-range upper-bound to be a function of the time that faults manifest.

`cwnd` Moving-Average Span. For `cwnd` analysis a moving average is performed on the time series to attenuate the effect of sporadic transmission timeouts. This enforces the condition that timeout events sustain for a reasonable time period, similar to anomalous window filtering. Spans in the range 5–31, with 31 the largest tested, exhibit a consistent 8% increase in ITP/DTP and a 1% decrease in IFP/DFP over the non-smoothed case.

WinSize & WinShift. Seven $WinSize$s of 32–128 with 16 sample steps, and seven $WinShift$s of 16–64 with 8 sample steps were tested to determine diagnosis influence. All $WinSize$s ≥ 48 and $WinShifts \geq 32$ were comparable in performance (62–66% DTP, 6–9% DFP). Thus for sufficiently large values, diagnosis is not sensitive.

Histogram Threshold Scale Factor. Histogram thresholds are scaled by a factor (currently 2x) to provide a cushion against secondary, minor fault manifestations (see § 4.6.1). At 1x, FP rates increase to 19%/23% IFP/DFP. 1.5x reduces this to 3%/8% IFP/DFP. On the range 2–4x ITP/DTP decreases from 70%/65% to 54%/48% as various metrics are masked, while IFP/DFP hold at 2%/7% as no additional misdiagnoses occur.

4.8 Experiences & Lessons

We describe some of our experiences, highlighting counterintuitive or unobvious issues that arose.

Heterogeneous Hardware. Clusters with heterogeneous hardware will exhibit performance characteristics that might violate our assumptions. Unfortunately, even supposedly homogeneous hardware (same make, model, etc.) can exhibit slightly different performance behaviors that impede diagnosis. These differences mostly manifest when the devices are stressed to performance limits (e.g., saturated disk or network).

Our approach can compensate for some deviations in hardware performance as long as our algorithm is trained for stressful workloads where these deviations manifest. The tradeoff, however, is that performance problems of lower severity (whose impact is less than normal deviations) may be masked. Additionally, there may be factors that are non-linear in influence. For example, buffer-cache thresholds are often set as a function of the amount of free memory in a system. Nodes with different memory configurations will have different caching semantics, with associated non-linear performance changes that cannot be easily accounted for during training.

Multiple Clients. Single- vs. multi-client workloads exhibit performance differences. In PVFS clusters with caching enabled, the buffer cache aggregates contiguous small writes for single-client workloads, considerably improving throughput. The buffer cache is not as effective with small writes in multi-client workloads, with the penalty due to interfering seeks reducing throughput and pushing disks to saturation.

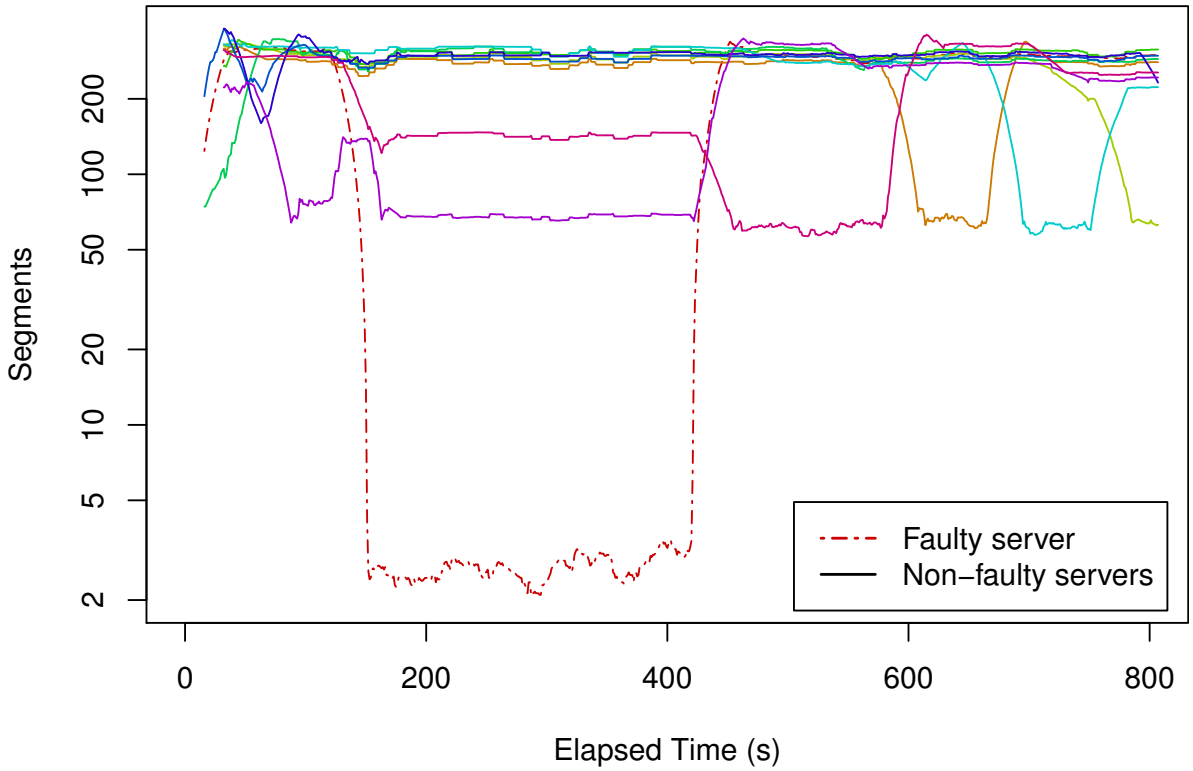

Figure 4.6: Single client `cwnds` for `ddw` workload with *receive-pktloss* fault.

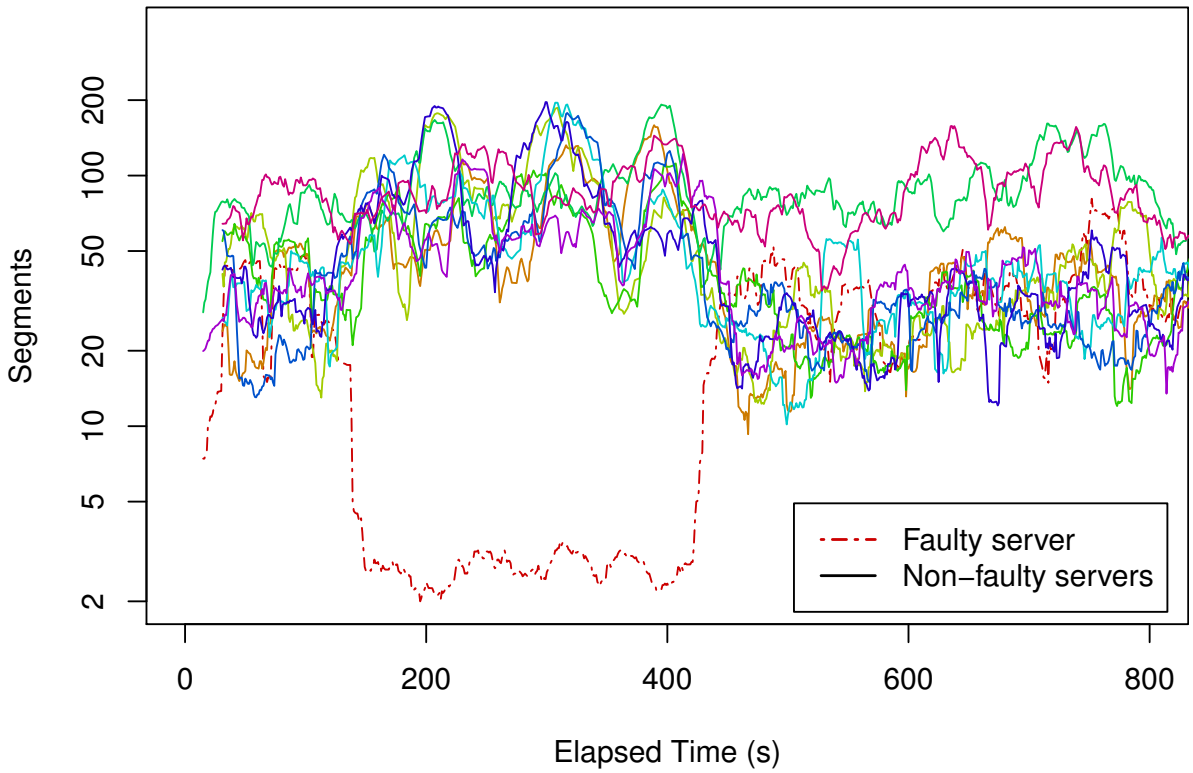

Figure 4.7: Multiple client `cwnds` for `ddw` workload with *receive-pktloss* fault.

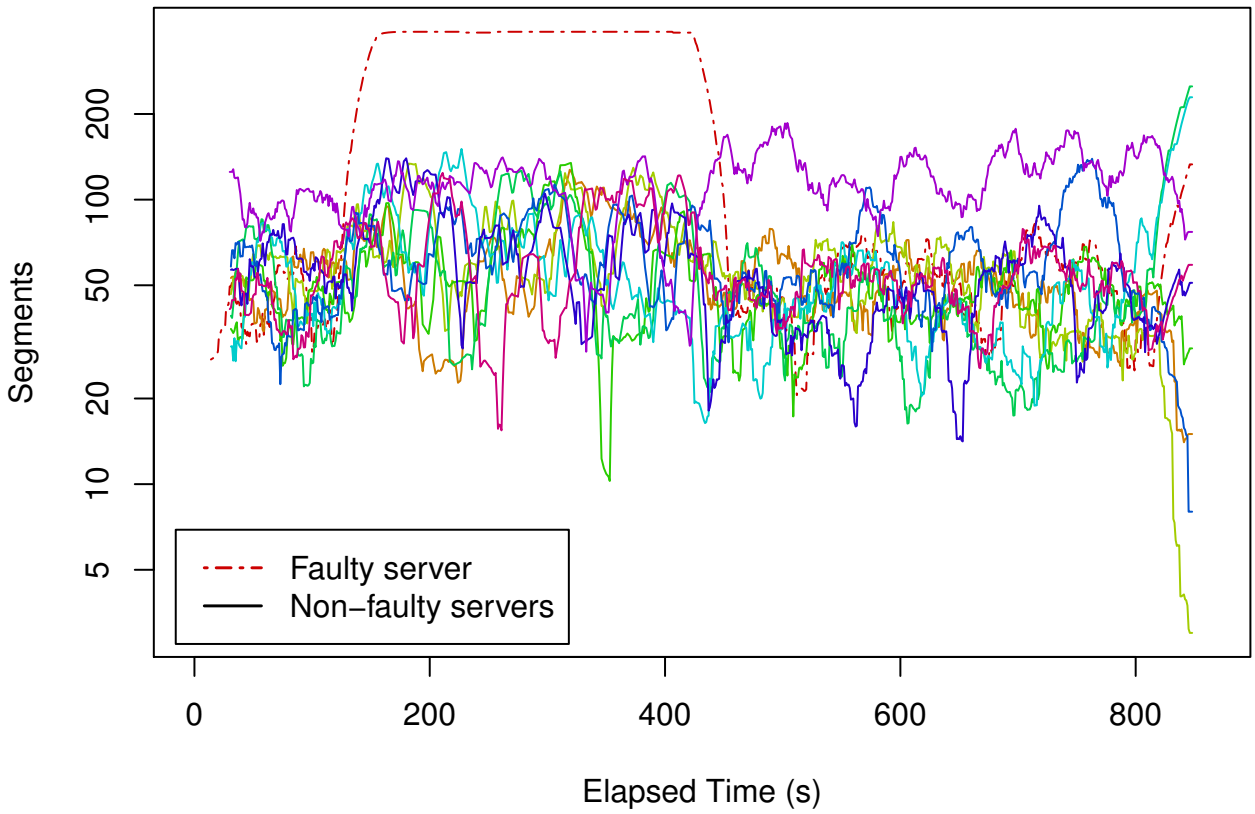

Figure 4.8: Disk-busy fault influence on faulty server's `cwnd` for `ddr` workload.

This also impacts network congestion (see Figures 4.6 & 4.7). Single-client write workloads (Figure 4.6) create single-source bulk data transfers, with relatively little network congestion. This creates steady client `cwnd`s that deviate sharply during a fault. Multi-client write workloads (Figure 4.7) create multi-source bulk data transfers, leading to interference, congestion and chaotic, widely varying `cwnd`s. While a faulty server's `cwnd`s are still distinguishable, this highlights the need to train on stressful workloads.

Cross-Resource Fault Influences. Faults can exhibit cross-metric influence on a single resource, e.g., a disk-hog creates increased throughput on the faulty disk, saturating that disk, increasing request queuing and latency.

Faults affecting one resource can manifest unintuitively in another resource's metrics. Consider a disk-busy fault's influence on the faulty server's `cwnd` for a large read workload (see Figure 4.8). `cwnd` is updated only when a server is both sending and experiencing congestion; thus, `cwnd` does

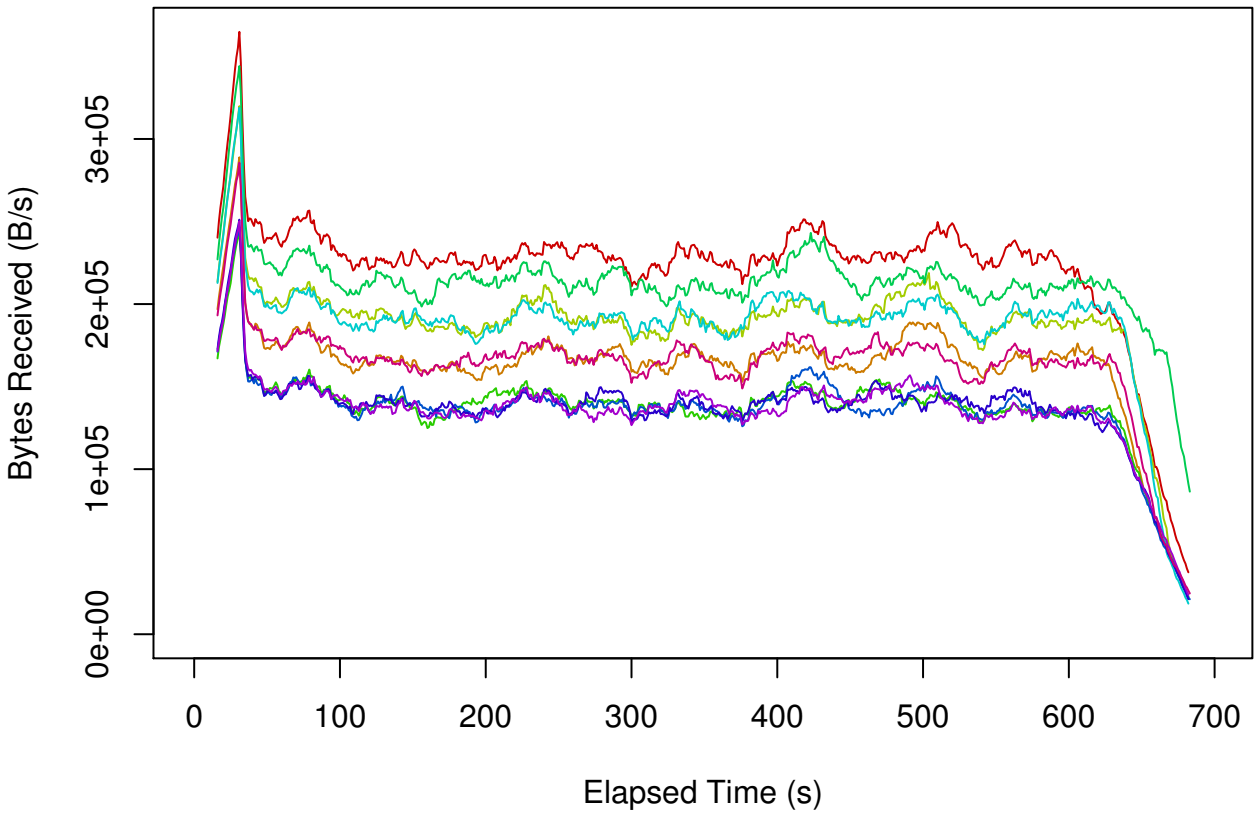

Figure 4.9: Spread in `rxbyt` due to unbalanced metadata requests during `postmark`.

not capture the degree of network congestion when a server is *not* sending data. Under a disk-busy fault, (i) a single client would send requests to each server, (ii) the fault-free servers would respond quickly and then idle, and (iii) the faulty server would respond after a delayed disk-read request.

PVFS's lack of client read-ahead blocks clients on the faulty server's responses, effectively synchronizing clients. Bulk data transfers occur in phases (ii) and (iii). During phase (ii), all fault-free servers transmit, creating network congestion and chaotic `cwnd` values, whereas during phase (iii), only the faulty server transmits, experiencing almost no congestion and maintaining a stable, high `cwnd` value. Thus, the faulty server's `cwnd` is asymmetric w.r.t. the other servers, mistakenly indicating a network-related fault instead of a disk-busy fault.

We can address this by assigning greater weight to storage-metric anomalies over network-metric anomalies in our root-cause analysis (§ 4.6.2). With Lustre's client read-ahead, read calls are not as synchronized across clients, and this influence does not manifest as severely.

Metadata-Request Heterogeneity. Our peer-similarity hypothesis does not apply to PVFS metadata servers. Specifically, since each PVFS directory entry is stored in a single server, server requests are unbalanced during path lookups, e.g., the server containing the directory "`/`" is involved in nearly all lookups, becoming a bottleneck.

We address this heterogeneity by training on the `postmark` metadata-heavy workload. Unbalanced metadata requests create a spread in network-throughput metrics for each server (see Figure 4.9) , contributing to a larger training threshold. If the request imbalance is significant, the resulting large threshold for network-throughput metrics will mask nearly all network-hog faults.

Buried ACKs. Read/write-network-hogs induce deviations in both receive and send network-throughput due to the network-hog's payload and associated acknowledgments. Since network-hog ACK packets are smaller than data packets, they can easily be "buried" in the network-throughput due to large-I/O traffic. Thus, network-hogs can appear to influence only one of `rxbyt` or `txbyt`, for read or write workloads, respectively.

`rxpck` and `txpck` metrics are immune to this effect, and can be used as alternatives for `rxbyt` and `txbyt` for network-hog diagnosis. Unfortunately, the non-homogeneous nature of metadata operations (in particular, `postmark`) result in `rxpck`/`txpck` fault manifestations being masked in most circumstances.

Delayed ACKs. In contradiction to Observation 5, a receive-(send-) packet-loss fault during a large-write (large-read) workload can cause a steady send (receive) network throughput on the faulty node and asymmetric decreases on non-faulty nodes (see Figure 4.10). Since the send (receive) throughput is almost entirely comprised of ACKs, this phenomenon is the result of delayed ACK behavior.

Delayed ACKs reduce ACK traffic by acknowledging every other packet when packets are received in order, effectively halving the amount of ACK traffic that would otherwise be needed to acknowledge packets 1:1. During packet-loss, each out-of-order packet is acknowledged 1:1 resulting in an effective doubling of send (receive) throughput on the faulty server as compared to

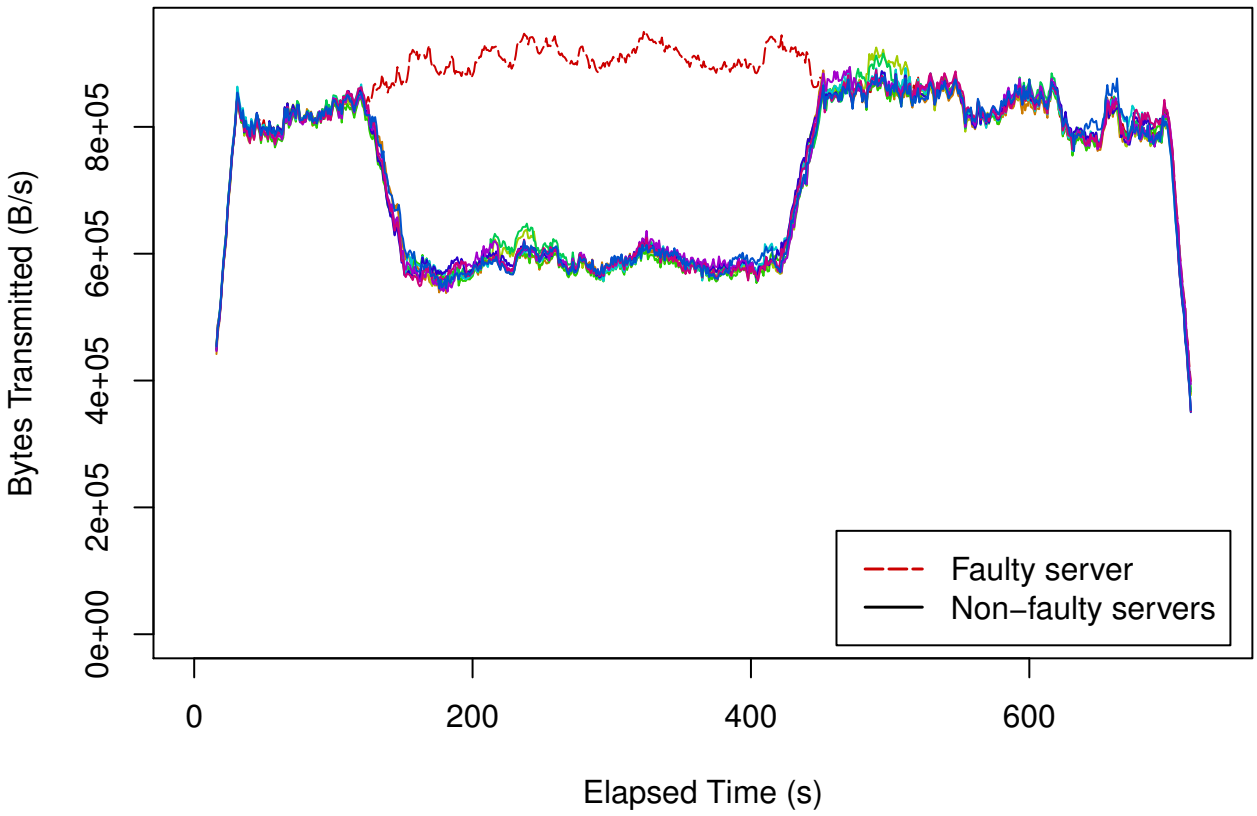

Figure 4.10: Receive packet-loss influence on faulty server's `txbyt` (ACKs) for `ddw` workload.

non-faulty nodes. Since the packet-loss fault itself results in, approximately, a halving of throughput, the overall behavior is a steady or slight increase in send (receive) throughput on the faulty node during the fault period.

Network-Metric Diagnosis Ambiguity. A single network metric is insufficient for diagnosis of network faults because of three properties of network throughput and congestion. First, *write-network-hog*s during write workloads create enough congestion to deviate the client `cwnd`; thus, `cwnd` is not an exclusive indicator of a packet-loss fault. Second, delayed ACKs contribute to packet-loss faults manifesting as network-throughput deviations, on `rxbyt` or `txbyt`; thus, the absence of a throughput deviation in the presence of a `cwnd` does not sufficiently diagnose all packet-loss faults. Third, buried ACKs contribute to network-hog faults manifesting in only one of `rxbyt` and `txbyt`, but not both; thus, the presence of both `rxbyt` and `txbyt` deviations does not sufficiently indicate all network-hog faults.

Thus, we disambiguate network faults in the third root-cause analysis step as follows. If both `rxbyt` and `txbyt` are asymmetric across servers, regardless of `cwnd`, a network-hog fault exists. If either `rxbyt` or `txbyt` is asymmetric, in the absence of `cwnd`, a network-hog fault exists. If `cwnd` is asymmetric regardless of either `rxbyt` or `txbyt` (but not both, due to the first rule above), then a packet-loss fault exists.

Chapter 5

Real-World Validation: Case Study

Following the promising success of our PVFS and Lustre laboratory experiments, we sought to validate our diagnosis approach on Intrepid's primary high-speed GPFS file system.

5.1 New Challenges

At the start of our 15-month case study of Intrepid's storage system, we identified a set of *new challenges* that our diagnosis approach would have to handle:

1. A large-scale, multi-tier storage system where problems can manifest on file servers, storage attachments, storage controllers, and individual LUNs.

2. Heterogeneous workloads of unknown behavior and unplanned hardware-component faults, both of which are outside of our control, that we observe and characterize as they happen.

3. The presence of system upgrades, e.g., addition of storage units that see proportionally higher loads (non-peer behavior) as the system seeks to balance resource utilization.

4. The need for continuous, 24/7 instrumentation and analysis.

5. Redundant links and components, which also exhibit changes in load (as compared to peers) when faults are present, even though the components themselves are operating appropriately.

6. The presence of occasional, transient performance asymmetries that are not conclusively attributable to any underlying problem or misbehavior.

5.1.1 Addressing these New Challenges

While problem diagnosis in Intrepid's storage system is based on the same fundamental peer-comparison process we developed during our laboratory experiments, these new challenges still require us to adapt our approach at every level: by expanding the system model, revisiting our instrumentation, and improving our diagnosis algorithm. Here we map our list of challenges to the sections of this thesis where we address them.

Challenge #2. Tolerating heterogeneous workloads and unplanned faults are inherent features of our peer-comparison approach to problem diagnosis. We assume that client workloads exhibit similar request patterns across all storage components, which is a feature provided by parallel file system data striping for all but pathological cases. We also assume that at least half of the storage components (within a peer group) exhibit fault-free behavior. As long as these assumptions hold, our peer-comparison approach can already distinguish problems from legitimate workloads.

Challenges #1, #3, and #5. Unlike our test-bench clusters, which consisted of a single storage component type (PVFS or Lustre file server with a local storage disk), Intrepid's storage system consists of multiple component types (file servers, storage controllers, disk arrays, attachments, etc.), that may be amended or upgraded over time, and that serve in redundant capacities. Thus, we are required to adapt our system model to tolerate each of these features. Since we collect instrumentation data on file servers (see § 3.2.1), we use LUN-server attachments as our fundamental component for analysis. With knowledge of GPFS's prioritization of attachments for shared storage (see § 5.2.2), we handle redundant components (challenge #5) by separating attachments into different priority groups that are separately analyzed. We handle upgrades (challenge #3) similarly, separating components into different sets based on the time at which they're added to the system,

and perform localization separately within each upgrade set (see § 5.2.1). Furthermore, by knowing which attachments are affected at the same time, along with the storage system topology (see § 5.2), we can infer the most likely tier and component affected by a problem (challenge #1).

Challenge #4. As in our laboratory experiments we use `sadc` to collect performance metrics (see § 3.2.1). To make our use of `sadc` amenable to continuous instrumentation, we also use a custom daemon, `cycle`, to rotate `sadc`'s activity files once a day (see § 3.2.2). This enables us to perform analysis on the previous day's activity files while `sadc` generates new files for the next day.

Challenge #6. Transient performance asymmetries are far more common during the continuous operation of large-scale storage systems, as compared to our short laboratory experiments. Treatment of these transient asymmetries requires altering the focus of our analysis efforts and enhancing our localization algorithm to use persistence ordering (see § 3.4).

5.2 Intrepid's Storage System

The target of our case study is Intrepid's primary storage system, a GPFS file system that, as illustrated in Figure 5.1, consists of 128 Network Shared Disk (NSD) servers (`fs1` through `fs128`) and 16 DataDirect Networks S2A9900 storage arrays, each with two storage controllers [41]. As illustrated in Figure 5.2, each storage array exports 72 LUNs (`ddn6_lun000` through `ddn21_lun071`) for Intrepid's GPFS file system, yielding a 4.5 PB file system comprised from 1152 LUNs and 11,520 total disks. At this size, this storage system demands a diagnosis approach with scalable data volume and an algorithm efficient enough to perform analysis in real-time with modest hardware. In addition, because our focus is on techniques that are amenable to such production environments, we require an approach with a low instrumentation overhead.

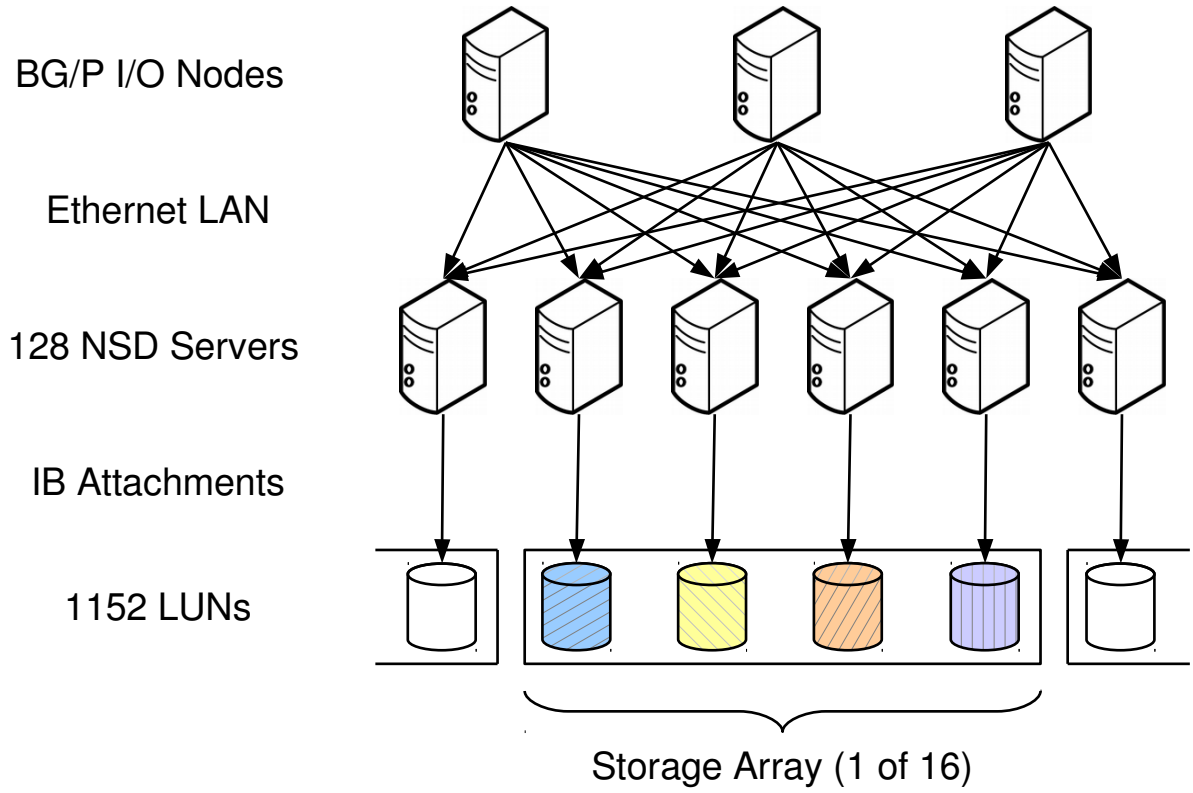

Figure 5.1: Intrepid's storage system architecture.

5.2.1 System Expansion

Of the 72 LUNs exported by each storage array, 48 were part of the original storage system deployment, while the other 24 were added concurrently with the start of our instrumentation to expand the system's capacity. Since the 24 LUNs added in each storage array (384 LUNs total) were initially empty, they observe fewer reads and more writes, and thus, exhibit non-peer behavior compared to the original 48 LUNs in each array (768 LUNs total). As our peer-comparison diagnosis approach performs best on LUNs with similar workloads, we partition Intrepid into "old" and "new" LUN sets, consisting of 768 and 384 LUNs respectively, and perform our localization separately within each set.

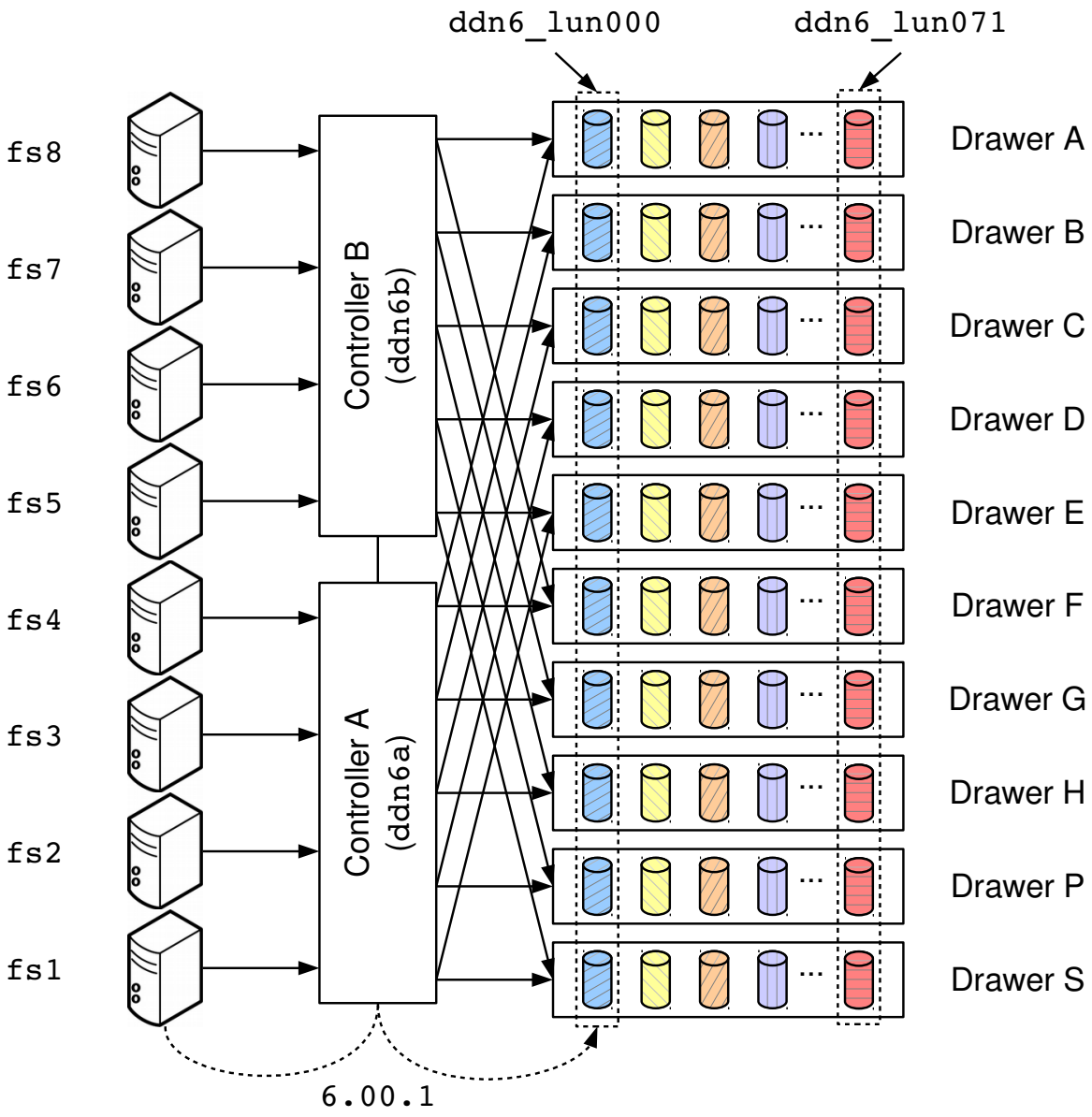

Figure 5.2: Storage array subarchitecture, e.g., `ddn6`.

5.2.2 Shared Storage

Each Intrepid LUN is redundantly attached to eight GPFS file servers with a prioritized server ordering defined in a system-wide configuration. We denote these LUN-server attachments with the convention `controller.lun.server`, e.g., `6.00.1` or `21.71.128`.

GPFS clients, when accessing a LUN, will route all I/O requests through the highest-priority, presently-available server defined for that LUN. Thus, when all servers are online, client I/O requests route through the primary server defined for a given LUN. If the primary server is unavailable, requests route through the LUN's secondary, tertiary, etc., servers based on those servers availability.

Since redundant attachments do not have equal priority for a given LUN, this effectively creates eight system-wide priority groups consisting of equal-priority LUN-server attachments, i.e., the first priority-group consists of all primary LUN-server attachments, the second priority-group consists of all the secondary LUN-server attachments, etc. Combined with the "system expansion" division, the total of 9216 LUN-server attachments (1152 LUNs $\times$ 8 redundantly attached servers) must be analyzed in 16 different peer groups (8 priority groups $\times$ 2 for "old" vs. "new" LUNs) in total.

5.3 Peer-Comparison Algorithm Refinements

In § 3.3, we describe the revised version of our peer-comparison algorithm, which, in conjunction with *persistence ordering* forms the core of our problem-diagnosis approach for Intrepid's storage system. Here, we detail the differences between the two versions of our peer-comparison algorithm and motivate the specific revisions made to it.

A peer-comparison algorithm requires the use of some measure that captures the similarity and the dissimilarity in the respective behaviors of peer components. A good measure, from a diagnosis viewpoint, is one that captures the differences between a faulty component and its non-faulty peer in a statistically significant way. In our explorations with Intrepid, we have sought to

use robust similarity/dissimilarity measures that are improvements over the ones that we used in our laboratory experiments (§ 4.6).

The first of these improvements is the method of histogram-bin selection. In § 3.3, we used Sturges' rule [56] to base the number of histogram bins on *WinSize*. Under both faulty and fault-free scenarios (particularly where a LUN exhibits a small asymmetry), Sturges' rule creates histograms where all data is contained in the first and last bins. Thus, the amount of asymmetry of a specific LUN relative to the variance of all LUNs is lost and not represented in the histogram. In contrast, the Freedman–Diaconis rule selects bin size as a function of the interquartile range (IQR), a robust measure of variance uninfluenced by a small number of outliers. Thus, the number of bins in each histogram adapts to ensure an accurate histogram representation of asymmetries that exceeds natural variance.

One notable concern of the Freedman–Diaconis rule is the lack of a limit on the number of bins. Should a metric include outliers that are orders of magnitude larger than the IQR, then, the Freedman–Diaconis rule will generate infeasibly large histograms, which is problematic as the analysis time and memory requirements both scale linearly with the number of bins. While we found this to not typically be an issue with the `await` metric, `wr_sec` outliers would (attempt) to generate histograms with more than 18 million bins. For diagnosis on Intrepid's storage system, we use a bin limit of 1000, which is the 99$^{\text{th}}$, 91$^{\text{st}}$, and 87$^{\text{th}}$ percentiles for `await`, `rd_sec`, and `wr_sec` respectively, and results in a worst-case (all generated with 1000 bins) histogram-computation time that is only twice the average.

The second improvement of this algorithm is its use of CDF distances as a similarity/dissimilarity measure, instead of the Probability Density Functions (PDFs) distances as we used in our laboratory experiments. Specifically, in § 3.3, we used a symmetric version of Kullback-Leibler (KL) divergence [14] to compute distance using histogram approximations of metric PDFs. This comparison works well when two histograms overlap (i.e., many of their data points lie in overlapping bins). However, where two histograms are entirely non-overlapping (i.e., their data points lie entirely in non-overlapping bins in distinct regions of their PDFs), the KL divergence does not include

Feature	Test Bench	Intrepid	Rationale
Separating upgraded components	✗	✓	Tolerates weighted I/O on recently added storage capacity; addresses challenge #3.
Fundamental component for analysis	LUNs	LUN-server attachments	Provides views of LUN utilization across redundant components; improves problem localization; addresses challenges #1 and #5.
`cycle` daemon	✗	✓	Enables continuous instrumentation with `sadc`; addresses challenge #4.
Downsampling	✗	1 s → 15 s	Tolerates intermittent data, reduces resource requirements; addresses challenge #1.
Histogram bin selection	Sturges' rule	Freedman–Diaconis rule	Provides accurate representation of asymmetries; improves diagnostic accuracy.
Distance metric	KL Divergence (PDF)	Cumulative Distance (CDF)	Accurate distance for non-overlapping histograms; improves diagnostic accuracy.
Persistence Ordering	✗	✓	Highlight components with long-term problems; addresses challenge #6.

Table 5.1: Improvements to diagnosis approach as compared to first version (§ 3.3).

a measure of the distance between non-zero PDF regions. In contrast, the distance between two metric CDFs *does* measure the distance between the non-zero PDF regions, which captures the degree of the LUN's asymmetry.

5.4 Revisiting our Challenges

Table 5.1 provides a summary of the changes to our approach as we moved from our test-bench environment to performing problem diagnosis in a large-scale storage system. This combination of changes both adequately addresses the challenges of targeting Intrepid's storage system, and also improves the underlying algorithm.

Analysis Step	Runtime	Memory
Extract activity file contents (`dump`)	1.8 h	< 10 MB
Downsample and tabulate metrics (`table`)	7.1 h	1.1 GB
Anomaly Detection (`diagprep`)	49 m	6.1 GB
Persistence Ordering (`diagnose`)	1.6 s	36 MB
Total	9.7 h	6.1 GB

Table 5.2: Resources used in analysis of `await`, for the first four peer groups of the 2011-05-09 data set.

5.5 Analysis Resource Requirements

In this section, we discuss the resources (data volume, computation time, and memory) requirements for the analysis of Intrepid's storage system.

Data Volume. The activity files generated by `sadc` at a sampling interval of 1 s, when compressed with `xz` [13] at preset level `-1`, generate a data volume of approximately 10 MB per file server, per day. The median-size data set (for the day 2011-05-09) has a total (includes all file servers) compressed size of 1.3 GB. In total, we have collected 624 GB in data sets for 474 days.

Runtime. We perform our analysis offline, on a separate cluster consisting of 2.4 GHz dual-core AMD Opteron 1220 machines, each with 4 GB RAM. Table 5.2 lists our analysis runtime for the `await` metric, when utilizing a single 2.4 GHz Opteron core, for each step of our analysis for the first four peer groups of the median-size data set. Because each data set (consisting of 24 hours of instrumentation data) takes approximately 9.7 h to analyze, we are able to keep up with data input.

We note that the two steps of our analysis that dominate runtime—extracting activity file contents (which is performed on all metrics at once), and downsampling and tabulation of metrics (includes `await` only)—take long due to our sampling at a 1 s interval. We use the 1 s sample rate for archival purposes, as it is the highest sample rate `sadc` supports. However, we could sample at a 15 s rate directly and forgo the downsampling process, which reduces the extraction time in Table 5.2 by a factor of 15 and tabulation time to 31 m, yielding a total runtime of approximately 1.4h.

Algorithm Scalability. Our CDF distances are generated through the pairwise comparison of histograms is $O(n^2)$ where n is the number of LUNs in each peer group. Because our four peer groups consist of two sets of 768 and 384 LUNs, and our CDF distances are symmetric, we must perform a total of 736,128 histogram comparisons for each analysis window. In practice, we find that our CDF distances are generated quickly, as illustrated by our Anomaly Detection runtime of 49 m for 192 analysis windows (24 hours of data). Thus, we do not see our pairwise algorithm to be an immediate threat to scalability in terms of analysis runtime. In § 4.7.1, we proposed an alternative approach to enable $O(n)$ scalability, but found it unnecessary for use in Intrepid.

Memory Utilization. Table 5.2 also lists the maximum amount of memory used by each step of our analysis. We use the analysis process' Resident Set Size (RSS) plus any additional used swap memory to determine memory utilization. The most memory-intensive step of our analysis is Anomaly Detection. Our static memory costs come from the need to store the tabulated raw metrics, moving-average-filtered metrics, and a mapping of LUNs to CDF distances, each of which uses 101 MB of memory. Within each analysis window, we must generate histograms for each of the 2,304 LUNs in all four of our peer groups. With a maximum of 1000 bins, all of the histograms occupy at most 8.8 MB of memory. We also generate 736,128 CDF distances, which occupy 2.8 MB per window. However, we must maintain the CDF distances across all 192 analysis windows for a given 24-hour data set, comprising a total of 539 MB. Using R's [46] default garbage collection parameters, we find that the steady-state memory use while generating CDF distances to be 1.1 GB. The maximum use of 6.1 GB is transient, happening at the very end when our CDF distances are written out to file. With these memory requirements, we are able to analyze two metrics simultaneously on each of our dual-core machines with 4 GB RAM, using swap memory to back the additional 2–4 GB when writing CDF distances to file.

Diagnosis Latency. Our minimum diagnosis latency, that is, the time from the incident of an event to the time of its earliest report as an anomaly is 22.5 minutes. This figure is derived from our (i) performing analysis at a sampling interval of 15 s, (ii) analyzing in time windows shifted

by 30 samples, and (iii) requiring that 3 out of the past 5 windows exhibits anomalous behavior before reporting the LUN itself as anomalous:

$$15\,\mathrm{s}/\mathrm{samples} \times 30\,\mathrm{samples}/\mathrm{window} \times 3\,\mathrm{windows} = 22.5\,\mathrm{m}$$

This latency is an acceptable figure for a few reasons:

- As a tool to diagnose component-level problems when a system is otherwise performing correctly (although, perhaps at suboptimal performance and reduced availability), the system continues to operate usefully during the diagnosis period. Reductions in performance are generally tolerable until a problem can be resolved.
- This latency improves upon current practice in Intrepid's storage system, e.g., four-hour automated checks of storage controller availability and daily manual checks of controller logs for misbehavior.
- Gabel et al. [20], which targets a similar problem of finding component-level issues before they grow into full-system failures, uses a diagnosis interval of 24 hours, and thus, considers this latency an acceptable figure.

In circumstances where our diagnosis latency would be unacceptably long, lowering the configurable parameters (sample interval, $WinShift$, and Anomaly Filtering's k value) will reduce latency with a potential for increased reports of spurious anomalies, which itself may be an acceptable consequence if there is external indication that a problem exists, for which we may assist in localization. In general, systems that are sensitive to diagnosis latency may benefit from combining our approach with problem-specific ones (e.g., heartbeats, SLAs, threshold limits, and component-specific monitoring) so as to complement each other in problem coverage.

```
Date.Hour:   Value PG:LUN-server
20110417.00:     7 2:18.61.102     6 2:15.65.74      5 2:11.55.48      5 2:12.48.49
20110417.01:    14 2:15.65.74      9 2:16.51.84      8 1:6.39.8        8 1:10.03.36
20110417.02:    19 2:15.65.74     17 2:16.51.84     15 2:10.50.35     15 2:21.56.121
20110417.03:    25 2:16.51.84     15 2:15.65.74     14 2:21.56.121    13 2:10.50.35
20110417.04:    33 2:16.51.84     20 2:15.65.74     15 2:21.56.121    13 2:10.50.35
20110417.05:    41 2:16.51.84     22 2:15.65.74     13 2:19.53.110    10 1:16.30.87
```

Figure 5.3: Example list of persistently anomalous LUNs. Each hour (row) specifies the most persistent anomalies (columns of accumulator value, peer-group, and LUN-server designation), ordered by decreasing accumulator value.

5.6 Evaluation of Case Study

Having migrated our analysis approach to meeting the challenges of problem diagnosis on a large-scale system, we perform a case study of Intrepid over a 474-day period from April 13th, 2011 through July 31st, 2012. We use the second day (April 14th, 2011) as our only "training day" for threshold selection.

In this study we analyze both "old" and "new" LUN sets, for the first two LUN-server attachment priority groups. This enables us to observe "lost attachment" faults both with zero/missing data from the lost attachment with the primary file server (priority group 1), and with the new, non-peer workload on the attachment with the secondary file server (priority group 2). We note that, although we do not explicitly study priority groups 3–8, we have observed sufficient file server faults to require use of tertiary and subsequent file server attachments.

5.6.1 Method of Evaluation

After collecting instrumentation data from Intrepid's file servers, we perform our problem localization (consisting of anomaly detection and persistence ordering as described in § 3.3 and § 3.4 respectively) on the `await` metric, generating a list of the top 100 persistently anomalous LUNs for each hour of the study. See Figure 5.3 for an example list.

In generating this list, we use a feedback mechanism that approximates the behavior of an operator using this system in real-time. For every period, we consider the top-most persistent anomaly, and if it has sufficient persistence (e.g., an accumulator value in excess of 100, which

indicates that the anomaly has been present for at least half a day, but lower values may be accepted given other contextual factors such as multiple LUNs on the same controller exhibiting anomalies simultaneously), then, we investigate that LUN's instrumentation data and storage-controller logs to determine if there is an outstanding performance problem on the LUN, its storage controller, file-server attachments, or attached file servers.

At the time that a problem is remedied (which we determine through instrumentation data and logs, but would be recorded by an operator after performing the restorative operation), we zero the accumulator for the affected LUN to avoid masking subsequent problems during the anomaly's "wind-down" time (the time during which the algorithm would continually subtract one from the former anomaly's accumulated value until zero is reached). For anomalies that persist for more than a few days before being repaired, we regenerate the persistent-anomaly list with the affected LUNs removed from the list, and check for additional anomalies that indicate a second problem exists concurrently. If a second problem does exist, we repeat this process.

5.7 Observed Incidents

Using our diagnosis approach, we have uncovered a variety of issues that manifested on Intrepid's storage system performance metrics (and that, therefore, we suspect to be performance problems). Our uncovering of these issues was done through our independent analysis of the instrumentation data, with subsequent corroboration of the incident with system logs, operators, and manual inspection of raw metrics. We have grouped these incidents into three categories.

5.7.1 Lost Attachments

We use the *lost attachments* category to describe any problem whereby a file server no longer routes I/O for a particular LUN, i.e., the attachment between that LUN-server pair is "lost". Of particular concern are lost primary (or highest priority) attachments as it forces clients to reroute I/O through the secondary file server, which then sees a doubling of its workload. Lost attachments of other

Incident Time	Diagnosis Latency	Recovery Latency	Device	Description
2011-07-14 00:00	1.0 h	25.8 d	ddn19a	Controller failed on reboot; missing data on 19.00.105.
2011-08-01 19:00	17.0 h	8.9 d	fs16	File server down; observed load on secondary (fs9).
2011-08-15 07:31	29 m	16.5 d	fs24	File server down; observed load on secondary (fs17).
2011-09-05 03:23	8.6 h	18.5 d	ddn20b	Controller manually failed; missing data on 20.00.117.
2011-09-11 03:22	11.6 h	35.6 h	fs25	File server down; observed load on secondary (fs26).
2011-10-03 03:09	12.9 h	42.5 d	ddn11a	Controller failed on reboot; observed load on secondary (fs41).
2011-10-17 16:57	22.1 h	28.0 d	ddn12a,20a,21a	Controllers manually failed; observed loads on secondaries (fs53,115,125).
2012-06-14 22:26	7.6 h	3.9 d	ddn8a	Controller manually failed; observed load on secondary (fs20).

Table 5.3: Storage controller failures and down file server lost attachment events.

priorities may still be significant events, but they are not necessarily performance impacting as they are infrequently used for I/O. We observe four general problems that result in lost attachments: (i) failed (or simply unavailable) file servers, (ii) failed storage controllers, (iii) misconfigured components, and (iv) temporary "bad state" problems that usually resolve themselves on reboot.

Failed Events. Table 5.3 lists the observed down file-server and failed storage-controller events. The *incident time* is the time at which a problem is observed in instrumentation data or controller logs. *Diagnosis latency* is the elapsed time between *incident time* and when we identify the problem using our method of evaluation (see § 5.6.1). *Recovery latency* is the elapsed time between *incident time* and when our analysis observes the problems to be recovered by Intrepid's operators. *Device* is the component in the system that is physically closest to the origin of the problem, while the incident's observed manifestation is described in *description*. In particular, "missing data" refers to instrumentation data no longer being available for the specified LUN-server attachment due to the disappearance of the LUN's block device on that file server, while a "0" value means the block device is still present, but not utilized for I/O.

The lengthy *recovery latency* for each of these *failed* events is due to the fact that all (except for fs25) required hardware replacements to be performed, usually during Intrepid's biweekly

Incident Time	Diagnosis Latency	Recovery Latency	Device	Description
2011-05-18 00:21	39 m	49.9 d	fs49,50,53,54	Extremely high await (up to 103 s) due to ddn12 resetting all LUNs, results in GPFS timeouts when accessing some LUNs, which remain unavailable until the affected file servers are rebooted; observed "0" await on 12.48.49.
2011-08-08 19:36	8.4 h	21.9 h	ddn19a	Servers unable to access some or all LUNs due to controller misconfiguration (disabled cache coherency); observed "0" await on 19.50.107.
2011-11-14 19:41	7.6 h	3.9 d	ddn14,18	Cache coherency fails to establish between coupled controllers after reboot, restricting LUN availability to servers; missing data on 14.41.67 and 18.37.103.
2012-03-05 17:50	3.2 h	4.2 d	fs56	GPFS service not available after file server reboot, unknown reason; observed loads on secondary (fs49).
2012-05-10 03:00	9.0 h	4.7 d	ddn16b	LUNs inaccessible from fs84,88, unknown reason; fixed on controller reboot; missing data on 16.59.84.
2012-06-13 03:00	3.0 h	8.7 d	ddn11a	LUNs inaccessible from fs42,45,46, unknown reason; missing data on 11.69.46.

Table 5.4: Misconfigured component and temporary "bad state" lost attachment events.

maintenance window, and perhaps even after consultation and troubleshooting of the component with its vendor. At present, Intrepid's operators discover these problems with syslog monitoring (for file servers) and by polling storage-controller status every four hours. Our *diagnosis latency* is high for file-server issues as we depend on the presence of a workload to localize traffic to the secondary attachment. Normally these issues would be observed sooner through missing values, except the instrumentation data itself comes from the down file server, and so, is missing in its entirety at the time of the problem (although the missing instrumentation data is a trivial sign that the file server is not in operation). In general, *failed* events, although they can be diagnosed independently, are important for analysis because they are among the longest-duration, numerous-LUN-impacting problems observed in the system.

Misconfiguration and Bad-State Events. Table 5.4 lists the observed misconfiguration and temporary "bad state" events that result in lost attachments. We explain the two cache-coherency events as follows: Each storage array consists of two coupled storage-controllers, each attached

to four different file servers, and both of which are able to provide access to attached disk arrays in the event of one controller's failure. However, when both controllers are in healthy operation, they may run in either cache-coherent or non-coherent modes. In cache-coherent mode, all LUNs may be accessed by both controllers (and thus, all eight file servers) simultaneously, as they are expected to by the GPFS-cluster configuration. However, should the controllers enter non-coherent mode (due to misconfiguration or a previous controller problem), then they can only access arrays "owned" by the respective controller, restricting four of the eight file servers from accessing some subset of the controllers' LUNs.

Cascaded Failure. The most interesting example in the "bad state" events is the GPFS timeouts of May 18th, 2011, a cascaded failure that went unnoticed by Intrepid operators for some time. Until the time of the incident, the `ddn12` controllers were suffering from multiple, frequent disk issues (e.g., I/O timeouts) when the controller performed 71 "LUN resets". At this time, the controller delayed responses to incoming I/O requests for up to 103 s, causing three of the file servers to timeout their outstanding I/Os and refuse further access to the affected LUNs. Interestingly, while the controller and LUNs remain in operation, the affected file servers continue to abandon access for the duration of 50 days until they are rebooted, at which point the problem is resolved. This particular issue highlights the main benefit of our holistic peer-comparison approach. By having a complete view of the storage system, our diagnosis algorithm is able to localize problems that otherwise escape manual debugging and purpose-specific automated troubleshooting (i.e., scripts written to detect specific problems).

5.7.2 Drawer Errors

A drawer error is an event where a storage controller finds errors, usually I/O and "stuck link" errors on many disks within a single disk drawer. These errors can become very frequent, occurring every few seconds, adding considerable jitter to I/O operations (see Figure 5.4). Table 5.5 lists four observed instance of drawer errors, which are fairly similar in their diagnosis characteristics.

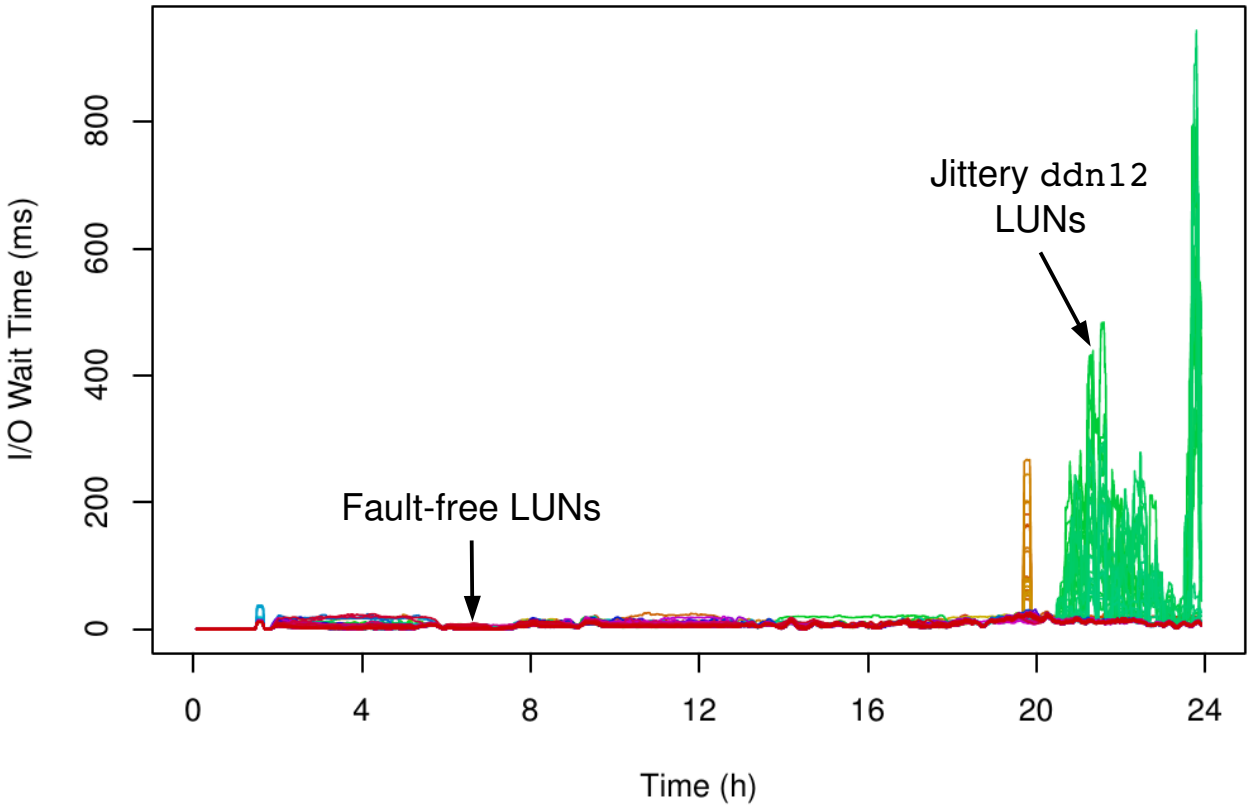

Figure 5.4: I/O wait time jitter experienced by `ddn12` LUNs during the May 17th, 2011 drawer error event.

Incident Time	Diagnosis Latency	Duration	Device	Description
2011-05-17 20:30	3.5 h	3.9 h	`ddn12`	Jittery `await` for 60 LUNs due to frequent "G" drawer errors.
2011-06-20 18:30	1.5 h	48.8 h	`ddn12`	Jittery `await` for 37 LUNs due to frequent "G" drawer errors.
2011-09-08 02:00	12.0 h	27.0 h	`ddn19`	Jittery `await` for 54 LUNs due to frequent "A" drawer errors.
2012-01-16 19:30	3.5 h	24.0 h	`ddn16`	Jittery `await` for 11 LUNs due to frequent "D" drawer errors.

Table 5.5: Drawer error events.

Drawer errors are visible to operators as a series of many verbose log messages. Operators resolve these errors by forcibly failing every disk in the drawer, rebooting the drawer, then reinserting all the disks into their respective arrays, which are recovered quickly via journal recovery.

5.7.3 Single LUN Events

Single LUN events are instances where a single LUN exhibits considerable I/O wait time (`await`) for as little as a few hours, or as long as many days. Table 5.6 lists five such events although as

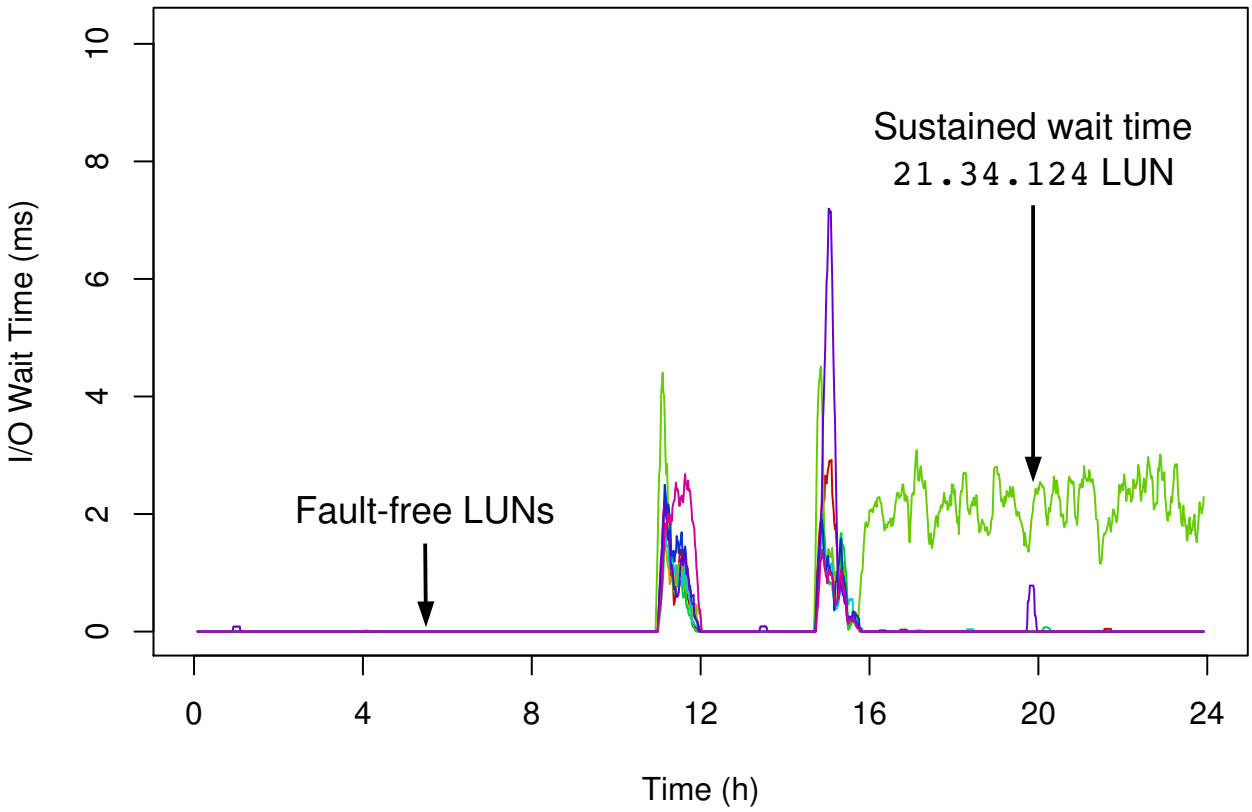

Figure 5.5: Sustained I/O wait time experienced by `21.34.124` during the June 25th single LUN event.

many as 40 have been observed to varying extents during our analysis.

These events can vary considerably in their behavior, and Table 5.6 provides a representative sample. Occasionally, the event will be accompanied by one or more controller-log messages that suggests that one or more spindles in the LUN's disk array is failing, e.g., the June 18th event is accompanied with a message stating that the controller recovered an "$8+2$" parity error. Single LUN events may correspond to single-LUN workloads, and thus, would manifest in one of the throughput metrics (`rd_sec` or `wr_sec`) in addition to `await`. Conversely, the June 25th event in Table 5.6 manifests in `await` in the *absence* of an observable workload (see Figure 5.5), perhaps suggesting that there is a load internal to the storage controller or array that causes externally-visible delay. Unfortunately, since storage-controller logs report little on most of our single LUN events, it is difficult to obtain a better understanding of specific causes of these events.

Incident Time	Diagnosis Latency	Duration	Device	Description
2011-06-18 08:00	18.0 h	10.5 d	`15.37.78`	Sustained above average `await`; recovered parity errors.
2011-08-18 20:00	26.0 h	79.0 h	`19.12.109`	Sustained above average `await`; until workload completes.
2011-09-25 04:00	20.0 h	4.3 d	`11.12.45`	Sustained above average `await`; unknown reason.
2012-04-19 12:00	38.0 h	7.2 d	`9.04.29`	Sustained above average `await`; unknown reason.
2012-06-25 16:00	8.0 h	6.6 d	`21.34.124`	Sustained `await` in absence of workload; unknown reason.

Table 5.6: Single LUN events.

5.8 Alternative Distance Measures

Our use of CDF distances as the distance measure for our peer-comparison algorithm is motivated by its ability to capture the differences in performance metrics between a faulty component and its non-faulty peer. Specifically, CDF distances capture asymmetries in a metric's value (relative to the metric's variance across all LUNs), as well as asymmetries in a metric's shape (i.e., a periodic or increasing/decreasing metric vs. a flat or unchanging metric). The use of CDF distances does require pairwise comparison of histograms, and thus, is $O(n^2)$ where n is the number of LUNs in each peer group. While we have demonstrated that the use of pairwise comparisons is not an immediate threat to scalability (see § 5.5), it is illustrative to compare CDF distances to alternative, computationally-simpler, $O(n)$ distance measures.

The two alternative distance measures we investigate are *median* and *thresh*. For both measures, we use the same Anomaly Detection and Persistence Ordering algorithms as described in § 3.3 and § 3.4, including all windowing, filtering, and their associated parameters. For each time window, instead of generating histograms we use one of our alternative measures to generate, for each LUN, a scalar distance value from the set of *WinSize* samples. For *median*, we generate a median time-series, m by computing for each sample, the median value across all LUNs within a peer group. We then compute each LUN's scalar distance as the sum of the distances between that LUN's metric value x and the median value for each of the *WinSize* samples: $d(x,m) = \sum_{i=0}^{WinSize} |x(i) - m(i)|$. We then flag a LUN as anomalous over a window if its scalar distance exceeds a predefined threshold, which is selected using the approach described in § 3.3.1.

We follow the same procedure for *thresh*, except that each LUN's scalar "distance" value is cal-

Event Type	CDF	Median	Thresh
Controller failure	5	5	5
File server down	3	2	3
Misconfiguration / bad state	6	6	5
Drawer error	4	3	4
Single LUN	5+	2	1

Table 5.7: Number of events observed with each distance measure (CDF distances, *median*, and *thresh*).

culated simply as the maximum metric value x among the $WinSize$ samples: $d(x) = \max x(i) \mid_{i=0}^{WinSize}$. Here, *thresh* is neither truly a measure of *distance*, nor is it being used to perform peer-comparison. Instead we use the *thresh* measure to implement the traditional "metric exceeds alarm-threshold value" within our anomaly detection framework, i.e., an anomalous window using *thresh* indicates that the metric exceeded twice the highest-observed value during the training period for at least one sample.

Performing a meaningful comparison of the *median* and *thresh* measures against CDF distances is challenging with production systems like Intrepid, where our evaluation involves some expert decision making and where we lack ground-truth data. For example, while the events enumerated in Tables 5.3–5.6 represent the most significant issues observed in our case study, we know there exists many issues of lesser significance (especially single LUN events) that we have not enumerated. Thus it is not feasible to provide traditional accuracy (true- and false-positive) rates as we have in our laboratory experiments. Instead, we compare the ability of the *median* and *thresh* measures to observe the set of events discovered using CDF distances (listed in Tables 5.3–5.6), by following the evaluation procedure described in § 5.6.1 for the days during which these events occur.

5.8.1 Comparison of Observations

Table 5.7 lists the number of we events observe with the alternative distance measures, *median* and *thresh*, as compared to the total events observed with CDF distances. Both *median* and *thresh* measures are able to observe all five failed storage-controller events, as well as most down file-server

and misconfiguration/"bad state" events. Each of these events are characterized by missing data on the LUN's primary attachment, and the appearance of a load on the LUN's normally-unused secondary attachment. Unlike CDF distances, neither *median* nor *thresh* measures directly account for missing data, however these events are still observed through the presence of secondary-attachment loads. As the non-primary attachments of LUNs are rarely used, these secondary-attachment loads are significant enough to contribute to considerable distance from the (near zero) median and to exceed any value observed during fault-free training. Both measures are also able to observe most drawer errors as these events exhibit considerable peak `await` that exceed both the median value and the maximum-observed value during training.

Controller Misconfiguration. For the August 8th, 2011 controller misconfiguration event, a zero `await` value is observed on the affected LUNs' primary attachments for the duration of the event, which is observed by the *median* measure. However, this particular event also results in zero `await` on the LUNs' secondary attachments, which are also affected, pushing the load onto the LUNs' tertiary attachments. As we only analyze each LUN's first two priority groups, the load on the tertiary attachment (and the event itself) goes unobserved. Thus, the *thresh* measure requires analysis of all priority groups to locate "missing" loads that are otherwise directly observed with peer-comparison-based measures.

Single LUN Events. Two single LUN events go unobserved by *median* as their manifestations in increased `await` are not sufficiently higher than their medians to result in persistent anomalies. Four of the events go unobserved by *thresh* as `await` never exceeds its maximum value observed during training, except during the June 25th, 2012 incident on a (normally-unused) secondary attachment where sustained `await` is observed in absence of a workload.

5.8.2 Server Workloads

The remaining three events that escape observation by *median* (a down file-server, drawer error, and single LUN events) are each due to the same confounding issue. As described in § 5.2.2, shared storage is normally prioritized such that GPFS clients only use the highest-priority available attachment. However, workloads issued by GPFS file servers themselves preferentially make use of their own LUN attachments, regardless of priority, to avoid creating additional LAN traffic. Thus, for server-issued workloads, we observe loads on each (e.g., 48) of the server's attachments, which span all priority groups, as well as loads on each (e.g., 720) of the primary attachments for LUNs that are not directly attached to those servers. Such workloads, if significant enough, would result in anomalies on each (e.g., 42) of the non-primary attachments.

In practice, Intrepid's storage system does not run significant, long-running workloads on the file servers, so this complication is usually avoided. The exception is that GPFS itself occasionally issues very low-intensity, long-running (multiple day) workloads from an apparently-random file server. These workloads are of such low intensity (throughput < 10 kB/s, `await` < 1.0 ms, both per LUN) that their `await` values rarely exceed our CDF distance algorithm's histogram *BinSize*s, and thus, are regarded as noise. However, server-workload `await` values on non-primary attachments do exceed the (zero) median value, and thus, do contribute to *median* anomalies. The result is that the presence of a server workload during an analysis window often exhibits a greater persistence value than actual problems, which confounds our analysis with the *median* measure. Thus, reliable use of the *median* measure requires an additional analysis step to ignore anomalies that appear across all attachments for a particular file server.

5.8.3 Comparison of Latencies

Table 5.8 lists the differences in diagnosis latencies for events observed with the alternative distance measures, *median* and *thresh*, as compared to the diagnosis latencies observed with CDF distances. Negative values indicate that the alternative measure (*median* or *thresh*) observed the event before CDF distances, while positive values indicate that the alternative measure observed

Event Type	Median (h)	Thresh (h)
Controller failure	−7, 0, 5, 9, 12	−10, 0, 4, 5, 9
File server down	0, 1	−8, 0, 6
Misconfiguration / bad state	−8, −3, 0, 4, 6, 7	−3, −1, 0, 3, 7
Drawer error	−2, −1, 5	−2, −2, −1, 0
Single LUN	−5, 6	−4

Table 5.8: Differences in diagnosis latencies for events observed with alternative measures, as compared to CDF.

the event after. Differences are indicated in integer units as our reporting for the case study is hourly (see Figure 5.3).

With a mean 1.6 h and median 0.5 h increased latency for *median*, and a mean 0.2 h and median 0 h increased latency for *thresh*, diagnosis latency among all three distance measures are comparable. However, for specific events, latencies can vary as much as twelve hours between measures, suggesting that simultaneous use of multiple measures may be helpful to reduce overall diagnosis latency.

5.9 Experiences and Insights

In preparing for our case study of Intrepid's storage system, we made improvements to our diagnosis approach to address the challenges outlined in § 5.1. However, in the course of our instrumentation and case study, we encountered a variety of pragmatic issues, and we share our experiences and insights with them here.

Clock Synchronization. Our diagnosis algorithm requires clocks to be reasonably synchronized across file servers so that we may peer-compare data from the same time intervals. In our laboratory experiments, we used NTP to synchronize clocks at the start of our experiments, but disabled the NTP daemon so as to avoid clock adjustments during the experiments themselves. Intrepid's file servers also run NTP daemons; however, clock adjustments can and do happen during our `sadc` instrumentation. This results in occasional "missing" data samples where the clock adjusts forward, or the occasional "repeat" sample where the clock adjusts backwards. When tabulating

data for analysis, we represent missing samples with R's `NA` (missing) value, and repeated samples are overwritten with the latest recorded in the activity file. In general, our diagnosis is insensitive to minor clock adjustments and other delays that may result in missing samples, but it is a situation we initially encountered in our `table` script.

Discussion on Timestamps. Our activity files are recorded with filenames containing a modified[1] ISO 8601-formatted [39][2] UTC timestamp that corresponds to the time of the first recorded sample in the file. For example, `fs1-20110509T000005Z.sa.xz` is the activity file collected from file server `fs1`, with the first sample recorded at 00:00:05 UTC on 2011-05-09. In general, we recommend the use of ISO 8601-formatted UTC timestamps for filenames and logging where possible, as they provide the following benefits:

- Human readable (as opposed to Unix time).
- Ensures lexicographical sorting (e.g., of activity files) preserves the chronological order (of records).
- Contains no whitespace, so is easily read as a field by awk, R's `read.table`, etc.
- Encodes time zone as a numeric offset; "`Z`" for UTC.

With regard to time zones, ISO 8601's explicit encoding of them is particularly helpful in avoiding surprises when interpreting timestamps. It is an obvious problem if some components of a system report different time zones than others without expressing their respective zones in timestamps. However, even when all components use the same time zone (as Intrepid uses UTC), offline analysis may use timestamp parsing routines that interpret timestamps without an explicit time-zone designation in the local time zone of the analysis machine (which, in our case, is US Eastern).

[1] We remove colons to ensure compatibility with file systems that use colons as path separators.

[2] RFC 3339 is an "Internet profile of the ISO 8601 standard," that we cite due to its free availability and applicability to computer systems.

A more troubling problem with implicit time zones is that any timestamp recorded during the "repeating" hour of transition from daylight savings time to standard time (e.g., 1 am CDT to 1 am CST) are ambiguous. Although this problem happens only once a year, it causes difficulty in correlating anomalies observed during this hour with event logs from system components that lack time zone designations. Alternatively, when when components do encode time zones in timestamps, ISO 8601's use of numeric offsets makes it easy to convert between time zones without needing to consult a time zone database to locate the policies (e.g, daylight savings transition dates) behind time-zone abbreviations.

In summary, ISO 8601 enables easy handling of human readable timestamps without having to work-around edge cases inevitable when performing continuous instrumentation and monitoring of system activity. "Seconds elapsed since epoch" time (e.g., Unix time) works well as a non-human readable alternative as long as the epoch is unambiguous. `sadc` records timestamps in Unix time, and we have had no trouble with them.

Absence of Data. One of the surprising outcomes of our case study is that the *absence of*, or "missing data" where it is otherwise expected among its peers, is the primary indication of problem in five (seven if also including "0" data) of the studied events. This result reflects on the effectiveness of peer-comparison approaches for problem diagnosis as they highlight differences in behavior across components. In contrast, approaches that rely on thresholding of raw metric values may not indicate that problems were present in these scenarios.

Separation of Instrumentation from Analysis. Our diagnosis for Intrepid's storage system consists of simple, online instrumentation, in conjunction with more complex, offline analysis. We have found this separation of online instrumentation and offline analysis to be beneficial in our transition to Intrepid. Our instrumentation, consisting of a well-known daemon (`sadc`), and a small, C-language auditable tool (`cycle`), have few external dependencies and negligible overhead, both of which are important properties to operators considering deployment on a production system. In contrast, our analysis has significant resource requirements and external dependencies

(e.g., the R language runtime and associated libraries), and so is better suited to run on a dedicated machine isolated from the rest of the system. We find that this separation provides an appropriate balance in stability of instrumentation and flexibility in analysis, such that, as we consider "near real-time" diagnosis on Intrepid's storage system, we prefer to maintain the existing design instead of moving to a full-online approach.

Chapter 6

Lessons Learned

We have acquired a number of experiences and "lessons learned" during our exploration of black-box problem diagnosis in parallel file systems. We describe these experiences, as they've pertained to our laboratory experiments and case study of Intrepid's storage system in § 4.8 and § 5.9 respectively. In this chapter, we share additional experiences and lessons learned throughout our research of parallel file systems as a whole.

6.1 Non-Peer Behaviors

The test-bench clusters we used in our laboratory experiments consisted of up to 12 PVFS or Lustre file servers with a single LUN each, and our workloads (dd and IOzone specifically) were I/O bound, pushing our storage hardware throughput to its capacity. In this scenario we witnessed two peculiarities that resulted in observable non-peer behavior, even in the fault-free case, that we had to account for in our experiments.

6.1.1 Disk Saturation

Disk saturation occurs when incoming I/O requests arrive faster than the disk can service them, which results in queuing of backlogged requests and response delays. The most immediate indica-

tion of disk saturation is when `%util` is at or near 100%. Secondary indications are `avgqu-sz` increasing into the 10s or 100s range, and a significant increase in `await` as compared to `svctm` (queuing delay).

For systems or workloads where disks are not (or cannot) be saturated by normal usage, disk saturation is a good indicator of disk-hog/busy conditions. However, many applications and cluster configurations do cause disk saturation for normal workloads. Ideally, disk saturation would occur at the same time across all servers, minimizing peer-asymmetry despite value changes across many metrics. Unfortunately, due to slight differences in device performance characteristics (even among homogeneous hardware), a single disk often reaches saturation before the others; in such cases, that disk tends to remain saturated while the `await` of other disks backoff, creating a *bottlenecking condition* even in the fault-free scenario, effectively mimicking a disk-busy fault on that node despite no increase in workload (see Observation 6 in § 4.2). An example of this behavior is the single elevated `await` (purple line) in Figure 4.3 both before and after the fault period. Thus, we need to distinguish disk-busy faults from normal saturation by the magnitude of the impact (deviation in `await`), and capture that detail as part of our threshold selection (§ 4.6.1).

6.1.2 Buffer Cache

We discovered that the behavior of the Linux buffer cache confounds disk metrics for PVFS write workloads. The modal behavior of the buffer cache greatly influences disk operation despite only minor differences in node state and nearly identical client requests. [54] describes the behavior of the Linux buffer cache which is simplified here to a model with three modes.

In periodic-write mode, write requests are committed to the Linux buffer cache as dirty pages which remain in the cache until they expire, typically after 30 seconds. A `pdflush` thread wake up periodically, typically every 5 seconds, to write pages that have recently expired. Thus, disk throughput consists of a periodic aggregated write profile.

In background-write mode, write requests are received with sufficient rate such that a threshold quantity (`dirty_background_ratio`—typically 10%) of active memory fills with dirty pages

before they expire. Once this threshold is reached, a `pdflush` thread writes a sufficient quantity of unexpired pages to reduce the number of dirty pages below this threshold again. For a steady flow of incoming write requests, this results in a steady state background write behavior where pages are written to disk with the same average throughput as incoming requests.

In foreground-write mode, write requests are received at a rate that exceeds that which `pdflush` can write out in background mode. Once the quantity of surplus pages in the buffer cache exceeds `dirty_ratio` (typically 40%) of active memory, Linux then forces user processes to commit dirty pages directly to disk, blocking any process that attempts to write to the disk until this is done.

Disk performance metrics tends to be similar across nodes as long as all nodes are within periodic or background write modes. However, if nodes are operating in different modes, metrics may deviate if one or more of the nodes is in foreground write mode and the corresponding disks are saturated. Even if all nodes are performing foreground writes, the disk performance metrics may still vary significantly depending on the contents of the cache and the number of queued requests, both of which are dependent on the time at which the foreground write threshold is reached, which itself is a function of node memory and underlying device performance.

The buffer cache negatively influences recovery time due to lingering manifestations even after a fault has been removed. Under a disk-hog or disk-busy fault that results in disk saturation on the faulty node and a buildup of dirty pages, once the fault is removed it may take time to flush the expired dirty pages and this extra recovery workload may carry seek penalties that delays write performance as compared to fault-free nodes.

Thus, we avoid the use of the buffer cache when monitoring performance of PVFS as part of our experimental set-up (§ 4.4). Specifically, introduced in PVFS 2.8.0 is a Direct I/O storage method, which we use to bypass the buffer cache while providing improved performance for high-bandwidth writes. In contrast, Lustre, which uses both client and server side caching, did not exhibit the same behavior.

6.1.3 GPFS Metadata Management

In GPFS we have witnessed non-peer behavior when processes running on the file servers themselves issue I/O requests to the underlying storage LUNs. As we described in § 5.8.2, I/O requests issued by file servers preferentially use their own LUN attachments for any directly-attached LUNs, regardless of the configured NSD server priority. Although we do not know the specific cause of the workloads observed in § 5.8.2, we do know that GPFS centrally manages file system metadata [49] (e.g., inode and indirect block updates through dynamically elected per-file metanodes; and configuration, usage quotas, access control lists, and extended attributes through a central manager). Thus, I/O requests for metadata updates can be made from a different NSD server, and thus, a different storage pathway, than data block requests made by clients.

6.2 Problem Diagnosis at Different Scales

In § 5.1, we discuss and address the challenges of making problem diagnosis work in large-scale environments. While some of the challenges were expected (problems manifesting in different components, heterogeneous workloads, and the need for 24/7 instrumentation) there were both challenges we expected to face, but didn't, as well as challenges that surprised us.

6.2.1 Sustained Non-Peer Behaviors

For our laboratory experiments, the sustained non-peer behavior we witnessed was a result of running I/O bound workloads and saturating our storage throughput capacity, particularly in a small scale cluster. Expecting that this non-peer behavior would be present (if not more prominent) in large scale systems, we spent considerable effort testing these scenarios to ensure our problem diagnosis could accommodate these behaviors and avoid false alarms. However, in practice, we observed that Intrepid's storage system had sufficient (if not much greater) storage throughput capacity for which the day-to-day workloads made use, and sustained (non-faulty) non-peer behavior was uncommon.

At different times this could be a function of the (i) overall system not running at capacity, (ii) locally inefficient I/O operations at compute nodes (thus, bottlenecking I/O before they reach the storage system), or (iii) the provisioning of the storage system as being (slightly more) limited in network throughput versus storage throughput. In general, one of the goals of provisioning large storage systems is to balance the local-area and storage network capacities to avoid the bottlenecking conditions that we observed in our laboratory experiments. Thus, we simply did not have as much trouble with what we expected to be sources of non-peer behavior in our case-study investigation. If did have trouble with sustained non-peer behavior, we expect we would have to retrain regularly to quiesce sustained, but non-faulty anomalies.

6.2.2 Transient Performance Asymmetries

Conversely, one of the challenges that we did not expect was the regular presence of transient performance asymmetries, which motivated the development of our *persistence ordering* algorithm (§ 3.4). Essentially, within a given time window, one or more components (although, generally not more than ten) would indicate anomalous behavior. However, on a per-component basis, these anomalies would not persist across many time windows (again, generally not more than ten windows). When we first observed these anomalies, our thought was to tweak the anomaly detection parameters (e.g., filter width, $WinSize$ and $WinShift$, etc.) in order to suppress them. However, such tuning is not sustainable, and we did observe the underlying asymmetries through manual inspection of plots of the `await` metric. Thus, our anomaly detection was essentially accurate, but not immediately helpful to operators without further guidance of when to investigate alarms. If nothing else, there is little purpose in investigating a component if the asymmetry will go away on its own, and not come back.

Thus, for our case study, persistence ordering serves two purposes: (i) to serve as a severity or ranking metric, by ordering anomalous components by the duration of their asymmetries (i.e., those that are the most persistently anomalous have the highest severity), and (ii) to specifically call attention to the components that are the most persistently anomalous, as, by virtue of experi-

encing asymmetry for the longest duration, they're the least likely components to resolve without intervention.

An aspect of persistence ordering that has not been specifically explored is its function with oscillating inputs. The input to persistence ordering, being the binary-valued output of anomaly detection, can oscillate over time intervals, which corresponds to components that demonstrate intermittent—specifically periodic—asymmetry, and so harbor the qualities of being both transient and sustained. With the current persistence ordering algorithm, components will register as persistently anomalous if they oscillate with a duty cycle greater than 50%, although the persistence value will grow more slowly than components which have sustained asymmetries.

6.2.3 Diagnosis in Tiny Systems

Prior to our instrumentation of Intrepid's storage system, we performed a prototype instrumentation of Surveyor's (a single rack Blue Gene/P system at Argonne National Laboratory) storage system. While the instrumentation and analysis served as a prototype of our later case study with Intrepid, we did encounter a couple of novel experiences while analyzing a system with a very small (nearly minimal) number of peer components.

Surveyor's GPFS-based storage system consists of four NSD servers and a single DataDirect Networks S2A9550 storage array exporting eight LUNs to each of the servers. The LUNs themselves are partitioned into two sets of four, with each set containing a separate file system.

Since each LUN is attached to all NSD servers, this creates a shared storage setup similar to Intrepid's (§ 5.2.2) where each LUN has a prioritized server ordering. GPFS clients, thus, when accessing a LUN, routes all I/O requests through the highest-priority, presently-available server defined for that LUN. Using the terminology we adopted for Intrepid, analysis of Surveyor involves 32 LUN-server attachments separated into eight peer groups with four LUN-server attachments in each group. As part of prototyping our analysis, we experienced two novel challenges.

Histogram Bin Sizes. With four LUNs in each peer group, or anomaly detection algorithm (§ 3.3) should still apply as we meet the minimum requirement of three components to establish a majority symmetry and determine which one asymmetric component is anomalous. In practice though, we found that when generating histograms with the Freedman-Diaconis rule, the Interquartile Range (IQR) of our metric samples was being influenced by a single outlying (anomalous) LUN, resulting in oversized bins and histograms where, like Sturges' rule, all the data was contained in the first and last bins. Thus, for Surveyor, we substituted the Median Absolute Deviation (MAD) of our metric samples in place of IQR when using computing histogram *BinSize*s. With MAD, the computed *BinSize*s were robust to a single anomalous LUN when comparing as few as three LUNs. In contrast, the IQR requires comparing at least five LUNs to be robust to a single outlier. Since our analysis of Intrepid included hundreds of components in each peer group, we did use the IQR as specified by the Freedman-Diaconis rule, but our experience with Surveyor indicates that the MAD can be used when the number of components is near minimal.

Server Workloads. As with Intrepid (§ 5.8.2), we observed server workloads whereby I/O issued by one of the file servers itself would preferentially make use of its own LUN attachments, regardless of its configured priority. Specifically, we witnessed a single, asymmetric workload per-file-system among Surveyor's server nodes, possibly the GPFS quota manager or another metadata management component. We observed that this workload remains associated with a single file server for a long time, for a few weeks or longer in the month-and-a-half period that we studied Surveyor. However, if that file server went offline, the workload did migrate to a different server indefinitely.

Unfortunately the presence of a sustained server workload among client workloads complicated anomaly detection as there existed two potential peer groups for each LUN-server attachment: attachments with the same NSD priority (peers of client workloads) and attachments on a single server (peers of server workloads). The server workload's presence was most troubling when client workloads were light or idle, as it was significant enough to result in sustained asymmetry

that would migrate after a file server restart. Although we could retrain after each migration event, we found that as a viable alternative, we could filter out anomalies due to the server workload by perform anomaly detection separately for attachments in each NSD priority group and for attachments on each server. We then flag a LUN-server attachment if it's anomalous with respect to both its priority-group peers and its server peers. In our study of intrepid we did not generally encounter problems with server workloads except when using the *median* alternative distance measure (§ 5.8). If we were to make greater use of this measure, we could filter out server-workload anomalies using the same strategy.

6.3 Diagnosis of Disk Failures

Following our initial laboratory experiments we sought to observe and characterize realistic storage system problems while still operating in a laboratory setting. Thus, we focused our observations on disk failures in storage arrays by manually failing a single disk during executions of synthetic workloads and observing the behavior of `await` while the storage array operated in degraded (post-failure) and reconstruction (post-reinsertion) modes. Our intention was to use our observations of these failures to diagnose disk failure and reconstruction in the wild, i.e., on Intrepid's storage system. While only one of our observations was applicable to diagnosing failures during read workloads in Intrepid, we did make several other useful observations that both confirm and carry forward the direction of our problem diagnosis work.

We performed our disk-failure experiments on a 16-server GPFS system with eight IBM TotalStorage DS4300 storage arrays exporting four data LUNs. Each data LUN is a five-disk RAID-5 array of size 203 GB, providing for a 6.5 TB file system. Additionally, each storage array maintains three hot-spare disks, each of which can be substituted into one of the four RAID-5 arrays after a disk failure event, to immediately start reconstruction of the failed disk contents onto the spare.

In these experiments we used the same performance metric instrumentation as we used for our earlier experiments, focusing on the `await` metric, and ran the same workloads (e.g., `ddr` and

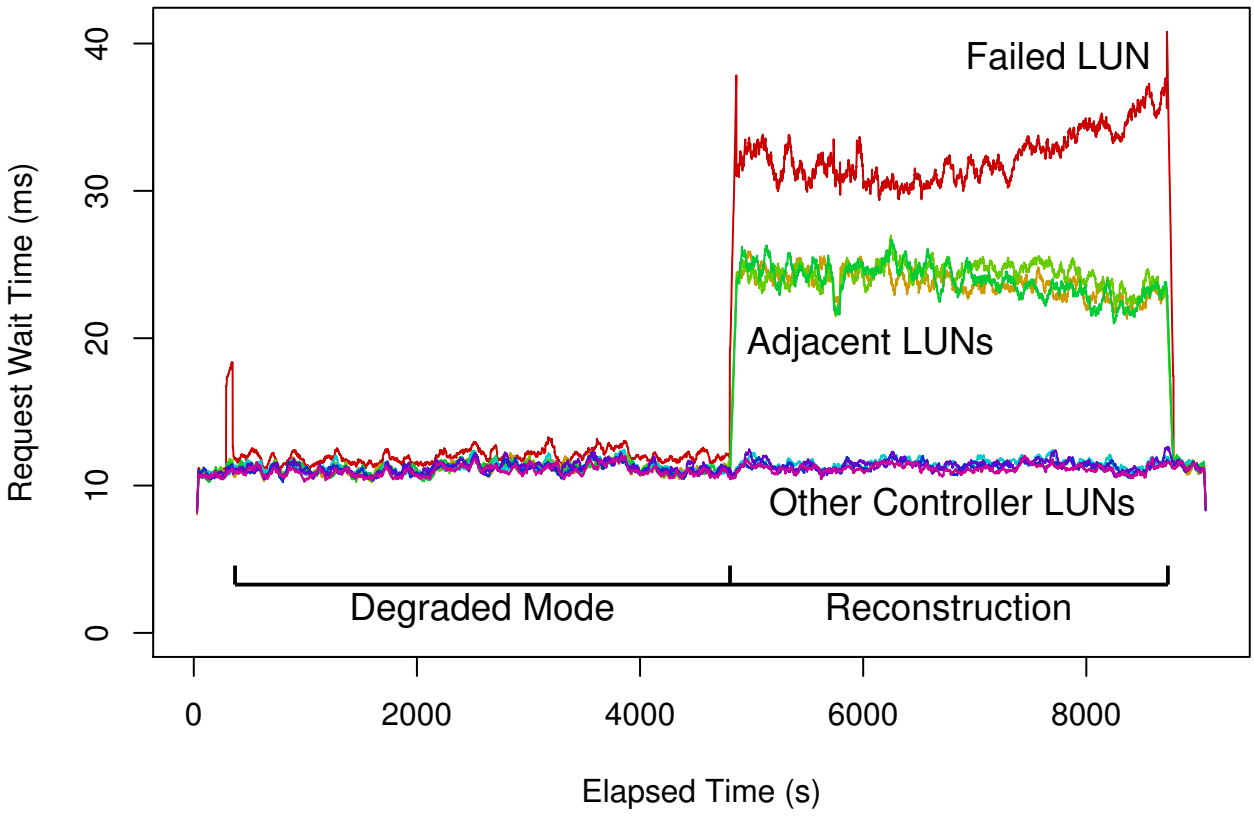

Figure 6.1: Peer-asymmetry of eight LUNs' `await` for `ddr` workload without hot spares.

`ddw`) on four clients, which were executed for the duration of the reconstruction interval and an approximately equivalent duration (≈3600 s) in degraded mode.

Read Workloads without Hot Spares. Figure 6.1 show the asymmetries in `await` when a disk is failed during read (`ddr`) workloads and no hot spare is available. When we fail the disk at 600 s, the array enters degraded mode where we observe slightly asymmetric `await` on the failed LUN alone. At approximately 4800 s we reinsert the failed disk and reconstruction of its contents from the remaining four disks begins. At this point we observe asymmetries between three different sets of LUNs. First, the previously-failed LUN exhibits a 30–40 ms `await`, expectedly, due to read contention between the client workload and the reconstruction process. Unexpectedly, however, we observe that the three *adjacent* LUNs on the same storage controller also exhibit an elevated ≈25 ms `await` despite their respective RAID-5 arrays operating in normal mode, suggesting that the entire storage controller is bandwidth starved by the internal reconstruction I/O. In contrast,

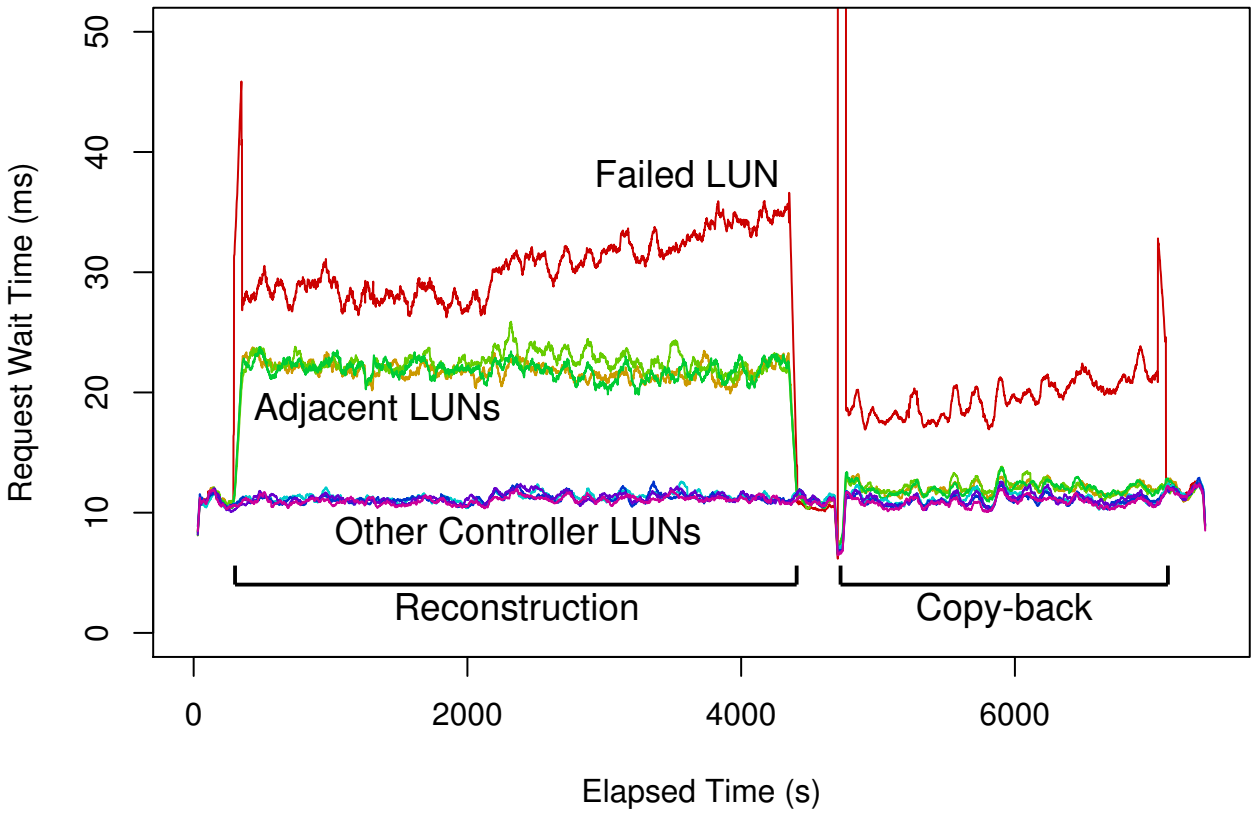

Figure 6.2: Peer-asymmetry of eight LUNs' `await` for `ddr` workload with hot spares.

LUNs on the other seven controllers[1] are unaffected in their `await`s.

Read Workloads with Hot Spares. Figure 6.2 show the asymmetries in `await` when a disk is failed during read (`ddr`) workloads and at least one hot spare is available. When we fail the disk at 600 s, a hot spare is immediately inserted and the controller begins reconstruction of the failed disk's contents onto the spare. Here, the asymmetry is the same as reconstruction in the without-hot-spare case. At approximately 4800 s, after reconstruction is complete, we reinsert the failed disk and the contents of the hot spare are copied back to the reinserted disk so that the spare may be released to the hot spare pool. During this copy-back procedure we again observe asymmetries between the previously-failed LUN with significantly elevated `await`, the adjacent LUNs with slightly elevated `await`, and LUNs on the other controllers. Here we believe the failed LUN experiences significant read contention during copy-back, but since only ¼th the data is being read

[1]Figure 6.1 only shows the four LUNs on a single, representative controller.

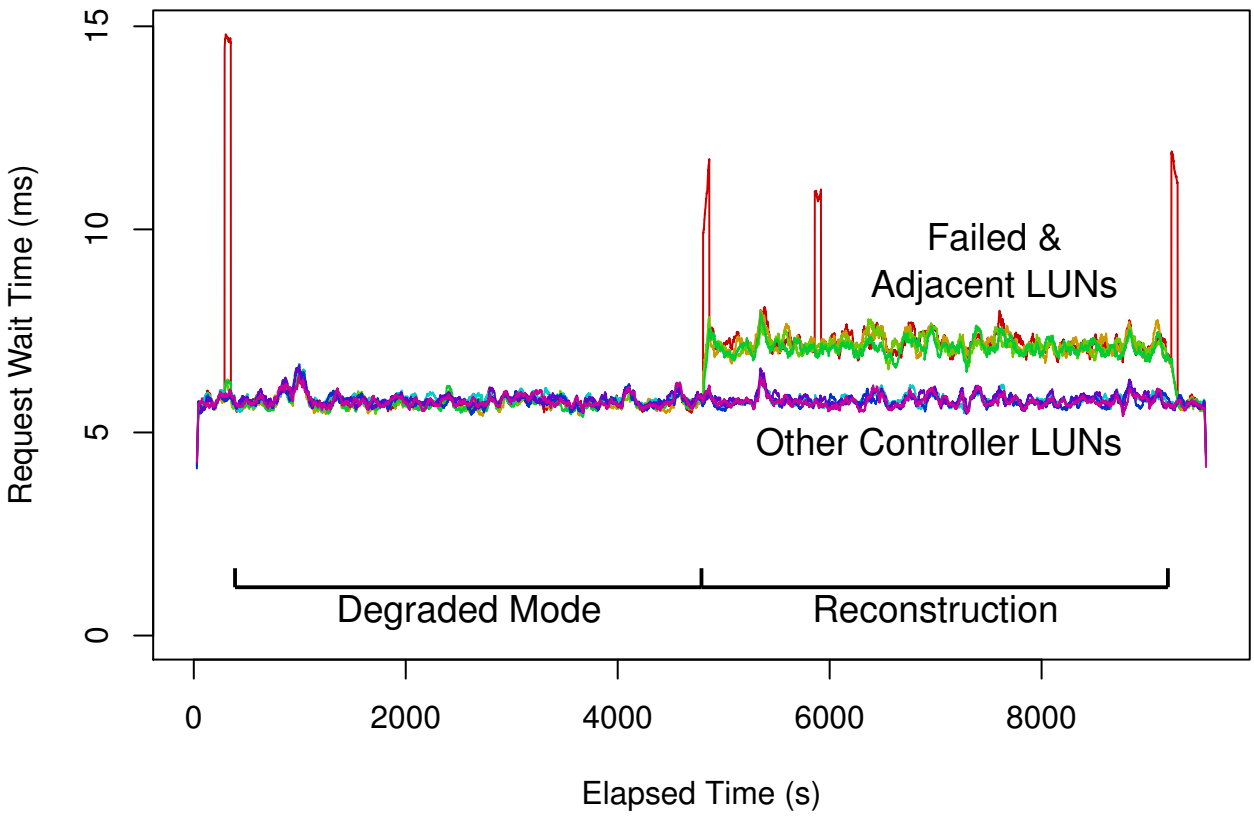

Figure 6.3: Peer-asymmetry of eight LUNs' `await` for `ddw` workload without hot spares.

compared to reconstruction, the controller experiences much less bandwidth starvation.

Write Workloads without Hot Spares. Figure 6.3 show the asymmetries in `await` when a disk is failed during write (`ddw`) workloads and no hot spare is available. When we fail the disk at 600 s, the array enters degraded mode where we observe no asymmetry in `await`, which is expected as the only difference in behavior is that the failed array writes to one less disk. At approximately 4800 s we reinsert the failed disk and reconstruction begins. Here, we only observe asymmetry between two all LUNs on the affected controller and LUNs on the other controllers. All LUNs on the affected controller, including the previously-failed LUN and its three adjacent LUNs, exhibits the same ≈7 ms `await` with controller-wide bandwidth limitations being the most likely culprit.

Write Workloads with Hot Spares. Figure 6.4 show the asymmetries in `await` when a disk is failed during write (`ddw`) workloads and at least one hot spare is available. The behavior when we

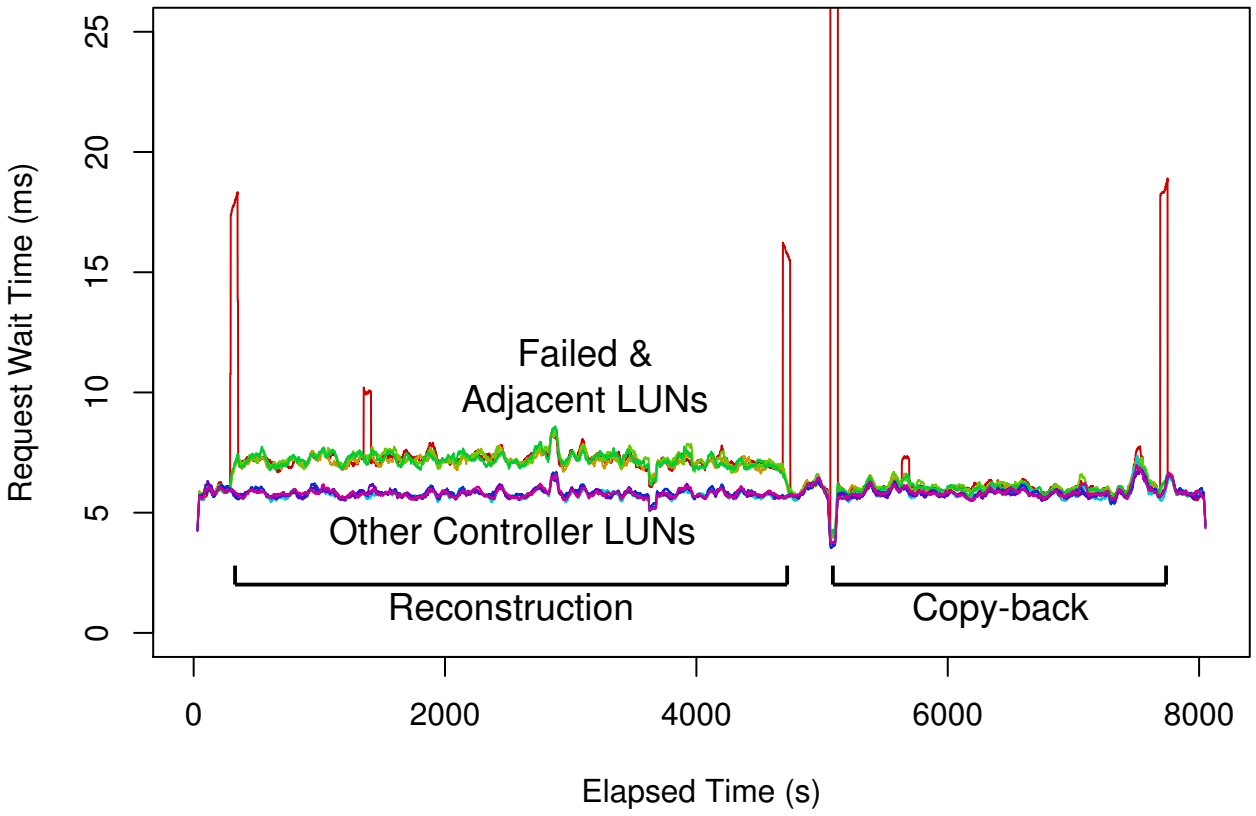

Figure 6.4: Peer-asymmetry of eight LUNs' `await` for `ddw` workload with hot spares.

fail the disk at 600 s and the hot spare is inserted, is identical to reconstruction in the without-hot-spare case. At approximately 5200 s, after reconstruction is complete, we reinsert the failed and copy-back begins. Here we observe a minimal asymmetry with LUNs on the affected controller exhibiting a marginally elevated `await` compared to LUNs on the other controllers.

6.3.1 Applicability of Observations.

From our observations of disk failures, it's not possible to identify storage arrays operating in degraded mode during write-only workloads as large write-back caches serve to avoid read-modify-write operations and perform large writes to all (present) disks in both normal and degraded modes. Indeed, this was an anecdote shared with by the PVFS developers' early in our research (§ 2.4. While it may be possible to identify a storage-controller-related problem during array reconstruction, our observations suggest it's not possible to localize those problems to the specific array af-

fected. Our later observations of Intrepid's storage system suggests that the bandwidth limitations we observed during our experiments may be an artifact of the DS4300 units and not necessarily characteristic of storage arrays in general.

In contrast, it should be possible to identify storage arrays operating in reconstruction and copy-back modes during read-only or read-write workloads due to the presence of significant read contention during the reconstruction process. However, during our study of Intrepid's storage system, we discovered that most instances of "disk failures" were operator initiated in response to logged disk events. In many of these instances, including *drawer errors* (§ 5.7.2), operators would manually fail disks and quickly reinsert them. Since the disks themselves had often not failed, the storage controllers would perform journal replays to quickly recover the disk contents in just a few minutes. Even in instances when disks were outright replaced, the recovery process, taking no more than a couple of hours, would only appear as a low-persistence transience anomaly. Fortunately, the slightly asymmetric `await` that we observed in degraded mode during read workloads did apply in our study of Intrepid where we diagnosed them as *drawer errors* and *single LUN events*.

Finally, our observations of asymmetry between multiple sets of LUNs during reconstruction does illustrate a flaw in our current peer-comparison algorithm. In neither our laboratory experiments or case study of Intrepid's storage system had we encountered one problem known to manifest as observably different asymmetries across both components that are directly involved in the problem (the LUN containing the failed disk) and components that are related to it by a shared resource (the LUNs adjacent to the failed disk sharing the same controller). Applying our current anomaly detection during reconstruction mode with read workloads, we can only indicate that the entire controller is anomalous and are unable to further distinguish between the LUNs despite a significant difference between them in `await` metric magnitude. Although there is no significant consequence of this ambiguity for this particular situation (reconstruction happens after an operator intervenes to replace a failed disk), its existence suggests potential value in further ascribing severity based on degree of peer-asymmetry in the presence of multiple anomalous components.

6.4 Missteps Made During Our Research

As our final lesson learned, we summarize the missteps made during our problem diagnosis research performed in support of this thesis. Having completed our case study of Intrepid's storage system, and with the benefit of hindsight, we highlight two areas of our research that, if we were to start over, we would approach differently.

6.4.1 Alarm-Based Anomaly Detection in Lab Experiments

Our first misstep was our focus on the development of an anomaly detection algorithm providing a binary-valued, alarm-based output (*anomalous* versus *normal*), and evaluating this output with experiment-wide true- and false-positive rates as we did for our laboratory experiments. Although this approach is familiar, being common in literature as a means to evaluate anomaly detection algorithms, we made the incorrect assumption that it is possible to accurately and usefully characterize system components with a single binary state (being either anomalous or normal), without either introducing too many false alarms or masking too many problems. Particularly misleading is that this approach *did* work for our laboratory experiments, disk failure experiments, and our study of Surveyor. It was only when we started analyzing the instrumentation data collected from Intrepid, a system with two orders of magnitude more components (LUNs) than systems we had looked at previously, that we observed the regular occurrence of transient performance asymmetries (§ 6.2.2), requiring us to extend our problem-diagnosis approach with persistence ordering (§ 3.4).

Persistence ordering, with its ranked, integer-valued output provides more information to operators about the relative health of components and the overall system, than independent, binary-valued alarms. For example, when an operator observes that many (10+) LUNs within the same storage array start to experience anomalies, the operator may choose to investigate the problem, likely a drawer error, right away due to the sheer number of LUNs involved. In contrast, when the operator observes that the top-most anomalous LUNs include one or two single LUN events and

there's no external sign of performance concern, the operator may wait for the LUNs' persistence values to increase before investigating knowing the anomalies may resolve without intervention.

In hindsight, due to both the necessity and benefit of the information provided by persistence ordering, we should have made it part of our original problem localization algorithm and applied it during our laboratory experiments. We would also consider approaches to replace the thresholds used in anomaly detection with normalization factors, so that the input to persistence ordering could be continuously-valued, perhaps reflecting the magnitude of asymmetry in the performance metric itself (§ 8.1).

Furthermore, because the information provided by persistence ordering has different utility than binary-valued alarms, especially in the context of 24/7 monitoring, there are alternative—if not better—ways to evaluate diagnosis algorithms than using true- and false-positive rates. For example, in § 5.8.3 we evaluate alternative distance measures by their diagnosis latencies. If we had applied persistence ordering to our laboratory experiments, we would also have considered evaluating other factors including the number of anomalous, but fault-free components, and a comparison of the maximum persistence between faulty and fault-free components.

6.4.2 Disk-Failure Experiments

In § 6.3 we describe a set of disk-failure experiments that we performed in a GPFS cluster following our initial laboratory experiments. Unfortunately our motivations for performing these experiments were misguided and we found that the results, particularly for write-only workloads, were system specific, and thus, not fully applicable to Intrepid's storage system.

Through our laboratory experiments we had validated that our anomaly detection algorithm was capable of identifying component anomalies, at least for injected faults. At that time, our goal was to test our diagnosis approach on problems representative of those in real-world systems. We decided to observe and characterize the behavior of disk failures, particularly in the post-failure modes (degraded mode, reconstruction, and copy-back) as we could reliably initiate them by manually failing disks in a storage array and observe the behavior of the array while operating

in these modes.

We found that we could observe anomalies in all three post-failure modes (degraded mode, reconstruction, and copy-back) for both read and write workload in our experimental GPFS system. However when we applied our anomaly detection to known disk failure events in Intrepid's storage system, we found that we were unable to observe anomalies for write-only workloads. We then determined that the behavior (bandwidth starvation) of our experimental system responsible for the observed component anomalies was simply not present with the storage arrays used in Intrepid's storage system. Thus, we concluded that our characterization of disk failures for write-only workloads was specific to an artifact of our experimental system, and not a behavior common to all storage systems or even GPFS-based systems.

We had better success applying our anomaly detection to Intrepid disk failure events for read-only and read-write workloads. However we had a concern of poor motivation. As Intrepid's storage arrays lack hot spare disks, reconstruction only occurs after an operator manually replaces a disk, and thus, it is redundant to inform the operator of the anomalies occurring due to reconstruction. Furthermore, reconstruction is a relatively short process observed as a transient anomaly.

Observing anomalies during degraded mode is beneficial as operator intervention is required to replace the failed disk. However, from storage-controller logs, we observe that 77% of all disk failure events, and 33% of temporally distinct disk-failure events,[2] are operator initiated. This knowledge changes our motivation for analyzing anomalies related to disk failure events: because operator intervention is often needed to resolve the underlying problems (including drawer errors and single LUN events), it is also important—if not more so—to detect the *pre*-failure anomalies that result in reported disk failure events.

Thus, we approached our disk-failure experiments with a solution, our anomaly detection algorithm, in search of real-world problem on which to validate it. This approach broke down when we applied it to a real-world system, where the details of disk failure events differed significantly from our experiments. In the end, we had to observe Intrepid's storage system as a whole to un-

[2]Since many disk failure events occur within one minute of other disk failure events, we compress all such events that occur within a minute of another as a single, temporally distinct event.

derstand how we could effectively perform problem localization within it. This led us to using our complete problem localization algorithm to observe a set of performance-impacting problems, including drawer errors and single LUN events, with better understanding of the reasons for operator intervention.

Chapter 7

Related Work

While our work is suited to diagnosing problems that affect specific, peer components within a storage system, (i.e., the class of hardware component faults) by using readily available performance metrics, it does not have visibility of other classes of storage system problems including suboptimal workload I/O patterns, misconfigurations and software/firmware bugs activated on all peer components simultaneously, and problems affecting non-peer hardware components. While our approach may be used to exclude hardware component faults, a comprehensive solution to storage-system problem diagnosis would make use of related work to gain visibility of, insights into, and diagnose these other classes of problems.

The related work is categorized in terms of the types of approaches that have been used (trace-based, peer-comparison, etc.). We also describe research efforts on failure diagnosis and failure characterization that are specific to the target systems of interest to this dissertation, namely, file systems and storage systems. We also conclude with a brief overview of other efforts at failure diagnosis in productions systems.

7.1 HPC Storage-System Characterization

Darshan [9] is a tool for low-overhead, scalable parallel I/O characterization of HPC workloads. Darshan shares our goal of minimal-overhead instrumentation by collecting aggregate statistical

and timing information instead of traces in order to minimize runtime delay and data volume, which enables it to scale to leadership-class systems and be used in a "24/7", always-on manner.

Carns et al. [8] combine multiple sources of instrumentation including OS-level storage device metrics, snapshots of file system contents characteristics (file sizes, ages, capacity, etc.), Darshan's application-level I/O behavior, and aggregate (system-wide) I/O bandwidth to characterize HPC storage system use and behavior. These characterization tools enable a better understanding of HPC application I/O and storage-system utilization, so that both may be optimized to maximize I/O efficiency. Our diagnosis approach is complementary to these efforts, in that it locates sources of acute performance imbalances and problems within the storage system, but assumes that applications are well-behaved and that the normal, balanced operation is optimal. Thus, these characterization tools may be used to ensure that our assumptions of optimal workloads is satisfied.

7.2 Failures in HPC and Storage Systems

Studies of HPC and storage-system failures motivate our focus on diagnosing problems in storage-system hardware components. A study of failure data collected over nine-years from 22 HPC systems at Los Alamos National Laboratory (LANL) [50] finds that hardware is the largest root cause of failures at 30–60% across the different systems, with software the second-largest contributor at 5–24%, and 20–30% of failures having unknown cause. The large proportion of hardware failures motivates our concentration on hardware-related failures and performance problems.

A field-based study of disk-replacement data covering 100,000 disks deployed in HPC and commercial storage systems [51] finds an annual disk replacement rate of 1–4% across the HPC storage systems, and also finds that hard disks are the most commonly replaced components (at 18–49% of the ten most frequently replaced components) in two of three studied storage systems. Given that disks dominate the number of distinct components in the Intrepid storage system, we expect that disk failures and (intermittent) disk performance problems comprise a significant proportion of hardware-related performance problems, and thus, are worthy of specific attention.

Jiang et al. [28, 29, 30] studied problem troubleshooting using 636,108 real-world customer cases reported from 100,000 commercially deployed NetApp storage systems over a period of two years. The study focused on correlating the root-cause analysis outcome with the storage-system logs. The study produced a number of interesting insights including the facts that (i) hardware failure and misconfiguration are the two most frequent problem root-cause categories, contributing respectively, 47% and 25% to all customer cases, (ii) software bugs account for a small fraction (3%) of cases, (iii) logs are useful in reducing problem-resolution time. However, as the study observes, while logs are noisy and important log events are hard to locate, it might be possible to extract a signature (a set of relevant log events that uniquely identify a problem root cause) and then use that signature to detect recurring problems and to distinguish problems from each other. Gainaru et al. [21] also analyzes log messages, from large-scale HPC systems, to build models that characterize and predict normal and faulty behaviors (events) of the systems.

DIADS [6] is a tool that aims to provide diagnosis for an enterprise system comprised of databases running on a network-attached storage infrastructure (or SAN). DIADS introduces the concept of Annotated Plan Graphs (APGs) that capture the interaction of the database with the disk, essentially providing the dependency mapping from database operators to the physical disk where the data resides. The elements of the APG are annotated with performance data, allowing for analysis of where the performance issues lie, and also allowing for traceback to higher-level root-cause analysis of where the origin of the problem lies.

Anode [42] is an approach to diagnosing problems in storage systems using a combination of time-series analysis and historical system behavior, in order to pinpoint the affected parts of the system, and to identify the time periods when the impact of the problem is most felt. Anode involves the period collection of metrics (read IOPS, write IOPS, read latency, write latency, etc.) from a storage system, along with a baseline summarization of the expected metric values based on the historically-known behavior of the system. Robustness in anomaly detection is obtained by combining a number of metrics and their respective anomalies in order to produce an overall anomaly score.

These approaches differ from our peer-comparison-based approach in that they use historical data (logs and historical performance metrics) instead of peer behavior, to find events or components of the system most responsible for failures or user complaints of degraded performance. Thus, they are suitable for diagnosing problems (e.g., misconfigurations and software/firmware bugs) where a specific type of component, including non-peer components, exhibits degraded performance compared to past usage.

7.3 Trace-Based Problem Diagnosis

Many previous efforts have focused on path-based [2, 47, 4] and component-based [12, 40] approaches to problem diagnosis in Internet Services. Aguilera et al. [2] treats components in a distributed system as black-boxes, inferring paths by tracing RPC messages and detecting faults by identifying request-flow paths with abnormally long latencies. Pip [47] traces causal request-flows with tagged messages that are checked against programmer-specified expectations. Pip identifies requests and specific lines of code as faulty when they violate these expectations. Magpie [4] uses expert knowledge of event orderings to trace causal request-flows in a distributed system. Magpie then attributes system-resource utilizations (e.g. memory, CPU) to individual requests and clusters them by their resource-usage profiles to detect faulty requests. Pinpoint [12, 40] tags request flows through J2EE web-service systems, and, once a request is known to have failed, identifies the responsible request-processing components.

These trace-based approaches motivated the development of comprehensive end-to-end tracing frameworks [53, 18, 60] that are presently deployed in distributed systems for monitoring, debugging, and diagnostic purposes. Dapper [53] is built into the RPC libraries used in Google services and demonstrates the use of adaptive trace sampling to provide very low instrumentation overheads. X-Trace [18] provides an approach to retrofitting systems with end-to-end tracing and demonstrates it in multiple production services [17]. Stardust [60] builds end-to-end tracing into the Ursa Minor distributed storage system [1] and serves as the instrumentation basis for robust

performance modeling [59] and problem diagnosis [48]. Spectroscope [48] is a diagnostic tool used in conjunction with Stardust that compares distributed system request-flow traces from problem and non-problem execution periods to identify the component-level and software-level sources of performance changes.

In HPC environments, Paradyn [43] and TAU [52] are profiling and tracing frameworks used in debugging parallel applications, and IOVIS [44] and Dinh [15] are retrofitted implementations of request-level tracing in PVFS.

Huang et al. [26, 27] is an interesting approach to failure diagnosis for Web-based applications by transparently profiling all of the file-system-related calls made by the application. While they don't focus on diagnosing problems in file-systems, they use file-related calls as a source of instrumentation to diagnose Web-based applications. System monitoring of the running system is performed through FUSE, without requiring the recompilation of the applications or any modification to the application source-code. The research focuses on studying the effects of normal and (injected) abnormal behavior of the application on the FUSE-instrumented file-related calls; classifiers are built off the labeled failure data, in order to classify future observed data.

In general, trace-based approaches offer significant advantages over performance metrics in terms of request-level localization and detail, making it possible to diagnose problems to specific request types and their impact on performance, which aids in code-level debugging. Thus, where available, trace-based techniques may be used to diagnose problems in both peer and non-peer components. However, at present, there is limited request-level tracing available in production HPC storage deployments, and thus, we concentrate on a diagnosis approach that utilizes aggregate performance metrics as a readily-available, low-overhead instrumentation source.

7.4 Other Peer-Comparison-Based Approaches

Ganesha [45] seeks to diagnose performance-related problems in Hadoop by classifying slave nodes, via clustering of performance metrics, into behavioral profiles which are then peer-compared

to indict nodes behaving anomalously. While the node-indictment methods are similar, our work peer-compares a limited set of performance metrics directly (without clustering). Bodik et at. [5] use fingerprints as a representation of state to generally diagnose previously-seen datacenter performance crises from SLA violations. Our work avoids using previously-observed faults, and instead relies on fault-free training data to capture expected performance deviations and peer-comparison to determine the presence, specifically, of storage performance problems.

Wang et al. [62] analyzes metric distributions to identify RUBiS and Hadoop anomalies in entropy time-series. Our work also avoids the use of raw-metric thresholds by using peer-comparison to determine the degree of asymmetry between storage components, although we do threshold our distance measure to determine the existence of a fault. PeerWatch [31] peer-compares multiple instances of an application running across different virtual machines, and uses canonical correlation analysis to filter out workload changes and uncorrelated variables to find faults. We also use peer-comparison and bypass workload changes by looking for performance asymmetries, as opposed to analyzing raw metrics, across file servers.

7.5 Problem Diagnosis in Other Production Systems

Gabel et al. [20] applies statistical latent fault detection using machine (e.g., performance) counters to a commercial search engine's indexing and service clusters, and finds that 20% of machine failures are preceded by an incubation period during which the machine deviates in behavior (analogous to our component-level problems) prior to system failure. Draco [38] diagnoses chronics in VoIP operations of a major Internet Service Provider by, first, heuristically identifying user interactions likely to have failed, and second, identifying groups of properties that best explain the difference between failed and successful interactions. This approach is conceptually similar to ours in using a two-stage process to identify that (i) problems exists, and (ii) localizing them to the most problematic components.

Theia [22] is a visualization tool that analyzes application-level logs and generates visual sig-

natures of job performance, and is intended for use by users to locate and problems they experience in a production Hadoop cluster. Theia shares our philosophy of providing a tool to enable users (who act in a similar capacity to our operators) to quickly discover and locate component-level problems within these systems.

Chapter 8

Future Work & Conclusion

Through the development of our peer-comparison-based problem-diagnosis approach, our laboratory experiments on test-bench parallel file system clusters, and our case study of large-scale, real-world production storage system we have confirmed our thesis statement: through the collection and analysis of black-box, OS-level performance metrics, we have shown that is possible to localize storage system problems to the misbehaving components most responsible for degraded parallel-file-system performance.

8.1 Future Work

While the work performed in support of this thesis is sufficient to diagnose problems in Intrepid's storage system, during our research we have identified multiple areas for improvement and further investigation.

Severity and Ranking of Anomalies. With our present anomaly detection algorithm, a given component behaves either anomalously (demonstrates peer asymmetry) or normally (demonstrates peer symmetry) for any instantaneous point in time. Persistence ordering provides us with a means to rank components based on the *persistence*, roughly the duration, of anomaly. Thus, components ranked with the highest persistence are generally those that have experience anomalies for the

longest duration.

Missing, however, from this ranking is the magnitude of asymmetry, specifically, the degree to which the problem manifests in the underlying performance metric. As one example, in § 6.3, we observe a single problem (reconstruction of a failed disk) that manifests in the performance metrics of multiple components such that there are two different degrees of peer asymmetry reflecting the immediacy of the component to the underlying problem. Thus, by incorporating the magnitude of asymmetry into a severity metric, we would be able to distinguish between these components, and thus, further localize this problem closer to its source.

As a second example, while our persistence-ordering approach works well to identify longer-term problems in Intrepid, there is a class of problems that escapes our current approach. Occasionally, storage controllers will greatly delay I/O processing in response to an internal problem, such as the "LUN resets" observed on `ddn12` in the May 18th, 2011 cascaded failure event. Although we observed this particular incident, in general, order-of-magnitude increases in I/O response times are not highlighted by a severity metric. Thus, the development of an ranking method that factors in both the magnitude of asymmetry, as well as the anomaly's persistence, would be ideal in highlighting problems that are either severe and short, or not as severe but long in duration.

Coalescing of Anomalous Components. In our study of Intrepid, we observed that problems in storage controllers tend to manifest in a majority of their exported LUNs, and thus, a single problem can be responsible for as many as 50 of the most persistent anomalies. It would be beneficial to extend our approach to recognize that, since these anomalous LUNs are part of the same storage controller and manifest the same problem, that we should coalesce the LUNs' anomalies into a single report covering the entire controller. This in turn, would make it considerably easier to discover multiple problems, affecting different sets of components, that manifest in the same time period.

Discovery of Peer Groups. In our study of Intrepid we defined peer groups based on the combination of NSD server priorities (defined for each LUN as part of GPFS's static, file-system-wide

configuration) and a single system expansion (the addition of 384 empty LUNs). Fortunately NSD server priorities generally never change, and system expansion events are sufficiently rare that manually defining additional peer groups is not onerous, especially relative to the configuration that's necessary to perform on GPFS itself to facilitate a system expansion. However, it would be convenient for our problem diagnosis to automate the process of discovering peer groups, and it may be necessary to do so to support future file systems where peer groups may arise from a dynamic configuration (i.e., to balance free space, or to optimize performance based on long-term component trends, etc.).

We propose two approaches to automate discovery of peer groups. The first is a black-box approach whereby the diagnosis algorithm groups together components with similar throughputs, and then performs peer comparison among latency metrics (e.g., `await`) for each similar-throughput group. This would enable the diagnosis approach to dynamically adapt to different peer groups and, by observation 5 (§ 4.2) still be able to diagnose disk-busy-like problems. Unfortunately, this approach would be unable to diagnose disk-hog-like problems as the asymmetry in throughput would split the anomalous and normal components into two different peer groups. The second is a white-box approach where we modify, or otherwise require that future file systems communicate peer groups and changes to their assignments to the diagnosis agent. Both approaches, however, still assume that peer groups are a function of file-system wide configuration and not determined on a per-request basis.

Automated Localization via System Topology. In our study of Intrepid, problem localization is a function of the set of affected components. By adopting a LUN-server-attachment identification scheme that includes the controller and server, we can easily localize problems to controllers (or servers) when we observe a set of anomalous LUNs with the same controller (or server) identifier. However, for problems where one controller fails in a coupled-controller pair, we can only localize to the specific controller of that pair with knowledge of the storage system topology, specifically by knowing the set of four file servers attached to that controller.

This localization could be automated in our problem diagnosis by analyzing a graph of the system topology. Specifically, when a problem is observed across a set of components, we would determine which one shared component (e.g., a controller within a coupled pair) is most-common to the affected set (the LUNs attached to the four servers via that controller) and is also least-common to the unaffected set (the LUNs inside other storage arrays or attached to other servers). While knowledge of this topology was only necessary for localization of certain controller-related problems in Intrepid, it may be required for sufficient localization of problems in more complex (e.g., greater number of tiers) systems.

8.2 Conclusion

We have shown through the research in support of this thesis that we are able to exploit the peer behavior inherent in parallel file systems to identify and diagnose storage system problems through the collection and analysis of black-box, OS-level performance metrics. We have presented our experiences of taking our problem-diagnosis approach from proof-of-concept on a 12-server test-bench cluster, and making it work on large-scale, real-world production storage system. We have also shared our observations, challenges, solutions, experiences, insights, and lessons learned from performing our problem diagnosis experiments and studies. By diagnosing a variety of performance-related storage-system problems, we have shown the value of our approach for diagnosing problems in large-scale storage systems. Finally, we have provided direction for future research, so that our approach may be applied towards future studies and systems.

Bibliography

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, CA, Dec. 2005.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Bolton Landing, NY, Oct. 2003.

[3] Argonne National Laboratory. Blue Gene / P, Dec. 2007. `https://secure.flickr.com/photos/argonne/3323018571/`.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 259–272, San Francisco, CA, Dec. 2004.

[5] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pages 111–124, Paris, France, Apr. 2010.

[6] N. Borisov, S. Babu, S. Uttamchandani, R. Routray, and A. Singh. DIADS: A problem diagnosis tool for databases and storage area networks. In *Proceedings of the VLDB Endowment*, volume 2, pages 1546–1549, 2009.

[7] D. Capps. IOzone filesystem benchmark, Oct. 2006. `http://www.iozone.org/`.

[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage*, 7(3):8:1–8:26, Oct. 2011.

[9] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *Proceedings of the 1st Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS '09)*, New Orleans, LA, Sept. 2009.

[10] P. H. Carns, S. J. Lang, K. N. Harms, and R. Ross. Private communication, Dec. 2008.

[11] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.

[12] M. Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, pages 595–604, Bethesda, MD, June 2002.

[13] L. Collin. XZ utils, Apr. 2013. `http://tukaani.org/xz/`.

[14] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, NY, Aug. 1991.

[15] T. D. Dinh. Tracing internal behavior in PVFS. Bachelorarbeit, Ruprecht-Karls-Universität Heidelberg, Heidelberg, Germany, Oct. 2009.

[16] M. Evans. Self-monitoring, analysis and reporting technology (S.M.A.R.T.). SFF Committee Specification SFF-8035i, Apr. 1996.

[17] R. Fonseca, M. J. Freedman, and G. Porter. Experiences with tracing causality in networked services. In *Proceedings of the 2010 Internet Network Management Workshop on Research on Enterprise Networking (INM/WREN '10)*, San Jose, CA, Apr. 2010.

[18] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, Apr. 2007.

[19] D. Freedman and P. Diaconis. On the histogram as a density estimator: L2 theory. *Probability Theory and Related Fields*, 57(4):453–476, Dec. 1981.

[20] M. Gabel, A. Schuster, R.-G. Bachrach, and N. Bjørner. Latent fault detection in large scale services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, Boston, MA, June 2012.

[21] A. Gainaru, F. Cappello, J. Fullop, S. Trausan-Matu, and W. Kramer. Adaptive event prediction strategy with dynamic time window for large-scale HPC systems. In *Proceedings of Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML '11)*, Cascais, Portugal, Oct. 2011.

[22] E. Garduno, S. P. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the 26th Large Installation System Administration Conference (LISA '12)*, Washington, DC, Nov. 2012.

[23] D. Gilbert. The Linux sg3_utils package, June 2008. `http://sg.danny.cz/sg/sg3_utils.html`.

[24] S. Godard. SYSSTAT utilities home page, Nov. 2008. `http://pagesperso-orange.fr/sebastien.godard/`.

[25] D. Habas and J. Sieber. Background Patrol Read for Dell PowerEdge RAID Controllers. *Dell Power Solutions*, Feb. 2006.

[26] L. Huang. Assisting failure diagnosis through filesystem instrumentation. Master's thesis, University of Alberta, Edmonton, Alberta, 2011.

[27] L. Huang and K. Wong. Assisting failure diagnosis through filesystem instrumentation. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 160–174, Nov. 2010.

[28] W. Jiang. *Understanding storage system problems and diagnosing them through log analysis*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, 2009.

[29] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Storage system problem troubleshooting and system logs. In *;login*, volume 34, pages 31–40, 2009.

[30] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proceedings of the USENIX File and Storage Technologies*, pages 43–56, 2009.

[31] H. Kang, H. Chen, and G. Jiang. Peerwatch: a fault detection and diagnosis tool for virtualized consolidation systems. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10)*, pages 119–128, Washington, DC, June 2010.

[32] M. P. Kasick. Diagnosing performance problems in parallel file systems. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, May 2009.

[33] M. P. Kasick, K. A. Bare, E. E. Marinelli III, J. Tan, R. Gandhi, and P. Narasimhan. System-call based problem diagnosis for PVFS. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep '09)*, Lisbon, Portugal, June 2009.

[34] M. P. Kasick, R. Gandhi, and P. Narasimhan. Behavior-based problem localization for parallel file systems. In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep '10)*, Vancouver, BC, Canada, Oct. 2010.

[35] M. P. Kasick, P. Narasimhan, and K. Harms. Making problem diagnosis work for large-scale, production storage systems. In *Proceedings of the 27th Large Installation System Administration Conference (LISA '13)*, Washington, DC, Nov. 2013.

[36] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, San Jose, CA, Feb. 2010.

[37] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, Inc., Oct. 1997.

[38] S. P. Kavulya, S. Daniels, K. Joshi, M. Hiltunen, R. Gandhi, and P. Narasimhan. Draco: Statistical diagnosis of chronic problems in large distributed systems. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, Boston, MA, June 2012.

[39] G. Klyne and C. Newman. Date and Time on the Internet: Timestamps. RFC 3339 (Proposed Standard), July 2002.

[40] E. Kıcıman and A. Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, Sept. 2005.

[41] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, Portland, OR, Nov. 2009.

[42] V. Mathur, C. George, and J. Basak. Anode: Empirical detection of performance problems in storage systems using time-series analysis of periodic measurements. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology*, 2014.

[43] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, Nov. 2005.

[44] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross. Visual analysis of I/O system behavior for high–end computing. In *Proceedings of the 3rd Workshop on Large-scale System and Application Performance (LSAP '11)*, San Jose, CA, June 2011.

[45] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Black-box diagnosis of mapreduce systems. In *Proceedings of the 2nd Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics '09)*, Seattle, WA, June 2009.

[46] R Development Core Team. The R project for statistical computing, Apr. 2013. `http://www.r-project.org/`.

[47] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, , and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 115–128, San Jose, CA, May 2006.

[48] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, Mar. 2011.

[49] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Montery, CA, Jan. 2002.

[50] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance-computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*, pages 337–351, Philadelphia, PA, June 2006.

[51] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, Feb. 2007.

[52] S. S. Shende and A. D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, May 2006.

[53] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, Apr. 2010.

[54] G. Smith. The linux page cache and pdflush: Theory of operation and tuning for write-heavy loads, Aug. 2007. `http://www.westnet.com/~gsmith/content/linux-pdflush.htm`.

[55] W. R. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001 (Proposed Standard), Jan. 1997.

[56] H. A. Sturges. The choice of a class interval. *Journal of the American Statistical Association*, 21(153):65–66, Mar. 1926.

[57] Sun Microsystems, Inc. Lustre file system: High-performance storage architecture and scalable cluster file system. White paper, Oct. 2008.

[58] The IEEE and The Open Group. dd, 2004. `http://www.opengroup.org/onlinepubs/009695399/utilities/dd.html`.

[59] E. Thereska and G. R. Ganger. IRONModel: Robust performance models in the wild. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*, pages 253–264, Annapolis, MD, June 2008.

[60] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*, pages 3–14, Saint-Malo, France, June 2006.

[61] J. Vasileff. latest PERC firmware == slow, July 2005. `http://lists.us.dell.com/pipermail/linux-poweredge/2005-July/021908.html`.

[62] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *Proceedings of 12th IEEE/IFIP Network Operations and Management Symposium (NOMS '10)*, Osaka, Japan, Apr. 2010.