ABSTRACT

LIU, YANG. Server-side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. (Under the direction of Xiaosong Ma and Frank Mueller.)

Competing workloads on a shared storage system cause I/O resource contention and application performance vagaries. Such I/O contention and performance interference has been recognized as a severe problem in today's HPC storage systems, and it is likely to become acute at exascale. In this dissertation, we demonstrate (through measurement from Oak Ridge National Laboratory's Titan [4], the world's No. 2 supercomputer [79]) that high I/O variance co-exists with that individual storage units remain under-utilized for the majority of the time. One major reason for this is because of lacking interaction between application I/O requirements and system software tools to help alleviate the I/O bottleneck. More specific, I/O-aware scheduling or inter-job coordination are required on today's supercomputers. For example, knowledge of application-specific I/O behavior potentially allows a scheduler to stagger I/O-intensive jobs, improving both the stability of individual applications' I/O performance and the overall resource utilization. Additionally, for achieving I/O-aware job scheduling, we need detailed information on application I/O characteristics. In this Ph.D. thesis study, we conduct two related studies to achieve such I/O characterization and I/O-aware job scheduling.

The first study is automatic I/O signature identification, which extracts signatures from noisy, zero-overhead server-side I/O throughput logs that are already collected on today's supercomputers, without interfering with the compiling/execution of applications. Traditionally, I/O characteristics have been obtained using client-side tracing tools, with drawbacks such as non-trivial instrumentation/development costs, large trace traffic, and inconsistent adoption. In this work, we designed and implemented IOSI (I/O Signature Identifier), a tool that correlates the aforementioned server-side I/O logs with the batch scheduler job logs, and automatically identify the I/O signature of a target application known to be I/O-intensive. Compared to client-side tracing tools, IOSI is transparent, interface-agnostic, and incurs no overhead. We evaluated IOSI using the Spider storage system at Oak Ridge National Laboratory, the S3D turbulence application (running on 18,000 Titan nodes), and benchmark-based pseudo-applications. Through our experiments we confirmed that IOSI effectively extracts an application's I/O signature despite significant server-side noise. Compared to alternative data alignment techniques (e.g., dynamic time warping), it offers higher signature accuracy and shorter processing time.

In the second study, we significantly extended our work beyond IOSI to remove the requirement of apriori knowledge of an applications I/O intensity. Again through coupling the server-side I/O and job logs, we intend to take a thoroughly data-driven approach to automatically identify applications that appear to cause non-trivial I/O workload. The result is

AID (Automatic I/O Diverter), a system that “mines” a small set of I/O-intensive jobs out of the server-side logs, including per-storage-server traffic history. It then estimates I/O traffic and concurrency levels for applications associated with those jobs. Finally, based on such auto-extracted information, AID provides online I/O-aware scheduling recommendations to steer I/O-intensive applications away from heavy ongoing I/O activities. We evaluated AID on the same supercomputer using both real applications (with extracted I/O patterns validated by contacting users) and our own pseudo-applications. Our results confirm that AID is able to (1) identify I/O-intensive applications, plus their detailed I/O characteristics, and (2) significantly reduce these applications’ I/O performance degradation/variance by jointly evaluating outstanding applications’ I/O pattern and real-time system l/O load.

In summary, this thesis research has investigated automatic application I/O characterization, at two different levels. Part of the proposed techniques has now been incorporated into the official operational workflow of ORNL’s Spider storage system, and we expect the AID/IOSI systems to be useful for next-generation intelligent supercomputer storage systems.

Server-side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems

by
Yang Liu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

BIOGRAPHY

Yang Liu was born in Tangshan, Hebei, China. He received his Bachelor degree in Measurement & Control Technology and Instrument from Hebei University of Technology, China, in 2005. In 2009 summer he entered North Dakota State University to pursue his graduate study majoring in Computer Science, and he got his Master degree in 2011. He then moved to Raleigh, NC, to start his Ph.D. study at North Carolina State University until this thesis was finished in 2016. Since 2011 summer Yang has been conducting research in the PALM group led by Dr. Xiaosong Ma, with a focus on I/O Workload Characterization and Coordination on Large Shared Storage Systems.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Xiaosong Ma, for her support and guidance, and the invaluable knowledge and experience she provided me with, through academics and beyond. I am honored to have had to opportunity to work with her. I would also like to thank my co-advisor, Dr. Frank Mueller, who have dedicated so much of his valuable time, knowledge, and resources to help me finish this project.

I owe my deep gratitude to Dr. Sudharshan Vazhkudai and Raghul Gunasekaran from the Oak Ridge National Lab(ORNL), who provided me with several summer internship opportunities at ORNL, as well as so much constructive advice throughout my Ph.D. research. As our collaborators and co-authors, they first proposed this problem and provided access to the data we use. Especially, they hosted my visits at ORNL for more than a year after Dr. Xiaosong Ma moved to Qatar. This dissertation would not have been possible without the constant guidance and support from them.

I would like to thank Dr. Kemafor Anyanwu Ogan and Dr. Huiyang Zhou for serving on my dissertation committee. Each of them provided insightful feedback on this work that has undoubtedly enhanced the final product. I am also grateful to Dr. Mladen Vouk, Dr. Douglas S. Reeves and Dr. George N. Rouskas from the Computer Science Department at NCSU, who provided so much precious advice throughout my Ph.D. life in the past five years. I acknowledge the National Science Foundation for funding my Ph.D. research, and the Oak Ridge National Laboratory for providing multiple cluster testbeds and valuable job logs.

I would like to thank my former advisor, Dr. Weiyi Zhang, at NDSU. He is the kickoff of my life in research, and he taught me a lot on doing research at the beginning of my graduate study. I would like to also thank Dr. Jun Zhang, who co-directed my Master thesis project and provided me so much advise even after I graduated from NDSU.

I thank all of the past and current members of the PALM group, Ben Clay, Feng Ji, Fei Meng, Heshan Lin, Ye Jin and Zhe Zhang for their friendship and help. I would like to also thank my friends at NCSU and ORNL: Xiaocheng Zou, Jinni Su, Lei Wu, Feiye Wang, Chao Wang, Saurabh Gupta, and Devesh D Tiwari. Their friendship has made my Ph.D. life much more enjoyable.

Finally, I would like to thank my family. Words cannot adequately express my gratitude for the love and support they have given me throughout the years. I thank my parents, Haiyi Liu and Xiaobo Wang, who have always trusted and encouraged me without reservation. I would also like to thank my parents-in-low, Yujun Luo and Shuying Jiang for their unconditional support and help. Finally, I want to express my deepest appreciation to the love and support from my wife, Xiaoqing Luo. She helped me through all the frustrations, shared my successes, calmly alerted me to my mistakes, and created so much happiness in our lives. She is my reader, my crutch, and my best friend during this incredible process. I thank you with all my heart. I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

High-performance computing (HPC) systems cater to a diverse mix of scientific applications that run concurrently. While individual compute nodes are usually dedicated to a single parallel job at a time, the interconnection network and the storage subsystem are often shared among jobs. Network topology-aware job placement attempts to allocate larger groups of contiguous compute nodes to each application, in order to provide more stable message-passing performance for inter-process communication. I/O resource contention, however, continues to cause significant performance vagaries in applications [21, 80]. For example, the indispensable task of checkpointing is becoming increasingly cumbersome. The CHIMERA [18] astrophysics application produces 160TB of data per checkpoint, taking around an hour to write [46] on Titan.

This already bottleneck-prone I/O operation is further stymied by resource contention due to concurrent applications, as there is no I/O-aware scheduling or inter-job coordination on supercomputers [85, 29, 33]. As hard disks remain the dominant parallel file system storage media, I/O contention leads to excessive seeks, significantly degrading the overall I/O throughput.

This problem is expected to exacerbate on future extreme-scale machines (hundreds of petaflops). Future systems demand a sophisticated interplay between application requirements and system software tools that is lacking in today's systems. The aforementioned I/O performance variance problem makes an excellent candidate for such synergistic efforts. For example, knowledge of application-specific I/O behavior potentially allows a scheduler to stagger I/O-intensive jobs, improving *both* the stability of individual applications' I/O performance and the overall resource utilization.

Additionally, in order to achieve I/O-aware job scheduling, we need detailed information on application I/O characteristics. However, the information provided by the traditional client-side tracing tools is not enough for inter-job coordination due to multiple reasons, such as non-trivial instrumentation/development costs, large trace traffic, and inconsistent adoption.

In this Ph.D. thesis study, for achieving such efficient I/O characterization and I/O-aware job scheduling, we propose two novel systems, IOSI (I/O Signature Identifier)[50] and AID (Automatic I/O Diverter)(Submitted for publication). IOSI is designed for automatically identifying the I/O signature of data-intensive parallel applications from existing supercomputer server-side I/O traffic logs and batch job history jobs. AID takes one more step beyond: it mines I/O intensive applications from these two logs fully automatically without requiring any apriori information on the applications or jobs. Based on the automatic I/O characterization results, it further explores the potential of performing I/O-aware job scheduling. Both IOSI and AID achieve the goals without bring any extra overhead to the system, or requiring any effort from developers/users . Below we give high-level introductions on our target problems and solutions.

1.1 I/O Characterization

I/O-aware scheduling requires detailed information on application I/O characteristics, which is very challenging by itself. In this thesis work, we first explore the techniques needed to capture such information in an automatic and non-intrusive way.

Cross-layer communication regarding I/O characteristics, requirements or system status has remained a challenge. Traditionally, these I/O characteristics have been captured using client-side tracing tools [6, 8], running on the compute nodes. Unfortunately, the information provided by client-side tracing is not enough for inter-job coordination due to the following reasons.

- First, client-side tracing requires the use of I/O tracing libraries and/or application code instrumentation, often requiring non-trivial development/porting effort.
- Second, such tracing effort is entirely elective, rendering any job coordination ineffective when only a small portion of jobs perform (and release) I/O characteristics.
- Third, many users who do enable I/O tracing choose to turn it on for shorter debug runs and off for production runs, due to the considerable performance overhead (typically between 2% and 8% [60]).
- Fourth, different jobs may use different tracing tools, generating traces with different formats and content, requiring tremendous knowledge and integration.
- Finally, unique to I/O performance analysis, detailed tracing often generates large trace files themselves, creating additional I/O activities that perturb the file system and distort the original application I/O behavior. Even with reduced compute overhead and minimal information collection, in a system like Titan, collecting traces for individual applications from over 18,000 compute nodes will significantly stress the interconnect and I/O subsystems.

These factors limit the usage of client-side tracing tools for development purposes [34, 47], as opposed to routine adoption in production runs or for daily operations.

Similarly, very limited server-side I/O tracing can be performed on large-scale systems, where the bookkeeping overhead may bring even more visible performance degradations. Centers usually deploy only rudimentary monitoring schemes that collect *aggregate* workload information regarding combined I/O traffic from concurrently running applications.

In Chapter 4, we present IOSI, a novel approach to characterizing per-application I/O behavior from *noisy, zero-overhead server-side I/O throughput logs*, collected without interfering with the target application's execution. IOSI leverages the existing infrastructure in HPC centers for periodically logging high-level, server-side I/O throughput. E.g., the throughput on the I/O controllers of Titan's Spider file system [67] is recorded once every 2 seconds. Collecting this information has no performance impact on the compute nodes, does not require any user effort, and has minimal overhead on the storage servers. Further, the log collection traffic flows through the storage servers' Ethernet management network, without interfering with the application I/O. Hence, we refer to our log collection as zero-overhead.

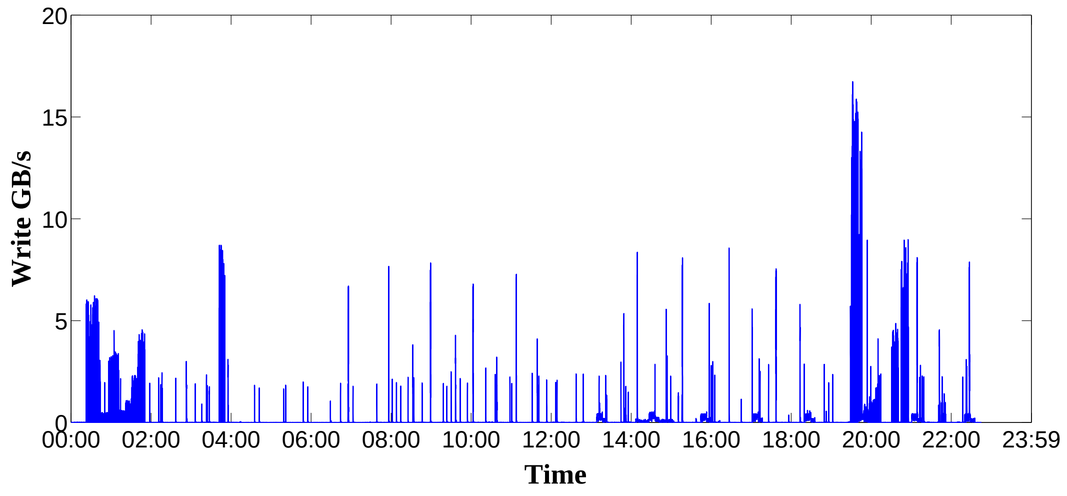


Figure 1.1: Average server-side, write throughput on Titan's Spider storage (a day in November 2011).

Figure 1.1 shows sample server-side log data from a typical day on Spider. The logs are composite data, reflecting multiple applications' I/O workload. Each instance of an application's execution will be recorded in the server-side I/O throughput log (referred to as a *sample* in the rest of this paper). Often, an I/O-intensive application's samples show certain repeated

I/O patterns, as can be seen from Figure 1.1. Therefore, the main idea of IOSI is to *collect and correlate multiple samples, filter out the "background noise", and finally identify the target application's native I/O traffic common across them.* Here, "background noise" refers to the traffic generated by other concurrent applications and system maintenance tasks. Note that IOSI is not intended to record fine-grained, per-application I/O operations. Instead, it derives an estimate of their bandwidth needs along the execution timeline to support future I/O-aware smart decision systems.

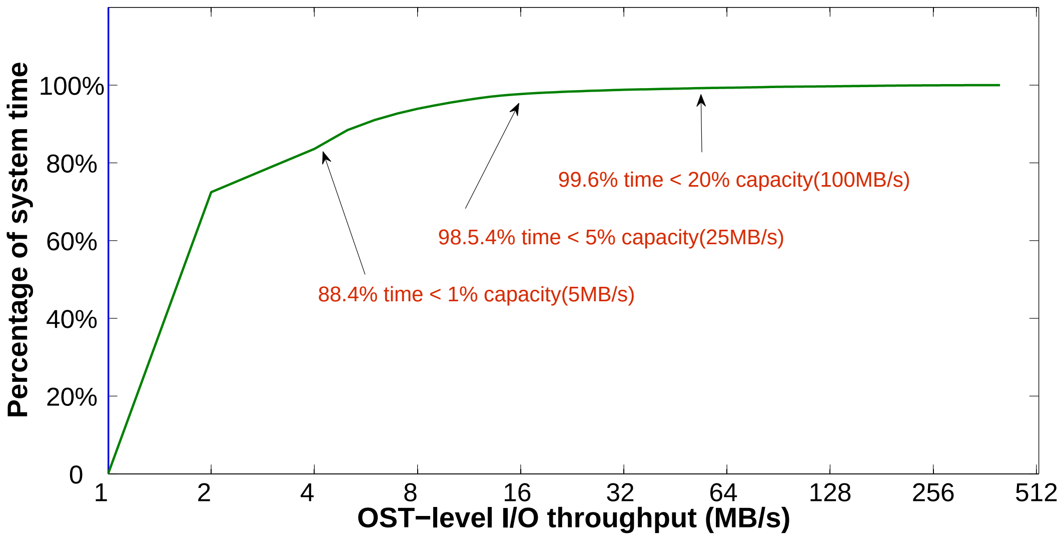
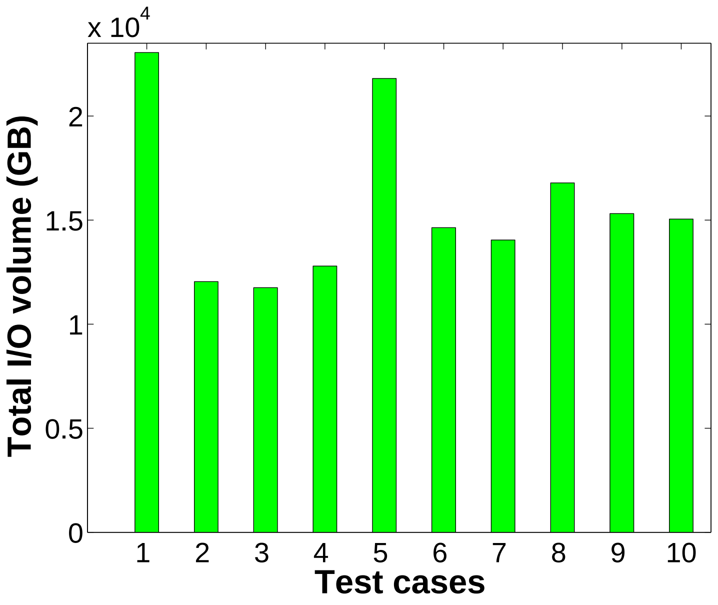


Figure 1.2: CDF of per-OST I/O throughput

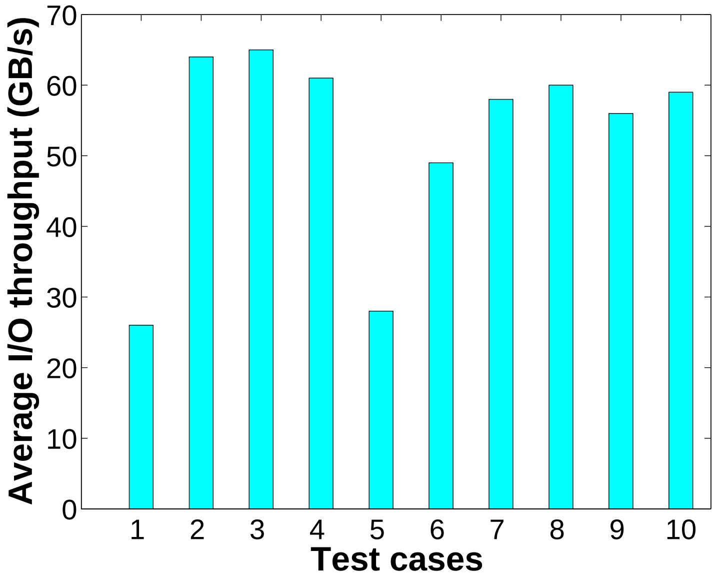
1.2 I/O Aware Job Scheduling

According to our observation of large-scale HPC systems, although significant I/O performance variance is observed frequently, on average, individual pieces of hardware (such as storage server nodes and disks) are often under-utilized.

Figure 1.2 illustrates this with the cumulative distribution of the combined read and write throughput on each of Spider's Lustre OSTs (Object Storage Targets) during a 5 month period (11/2014 to 03/2015), by giving the percentage of time each individual OST is reaching different I/O throughput levels. Overall, most of the OSTs are not busy, experiencing less than 1% (5MB/s) and 20% (100MB/s) of their individual peak throughput during 88.4% and 99.6% of system time, respectively.



(a) RAID controller utilization



(b) Average I/O throughput

Figure 1.3: I/O performance variance test on Titan

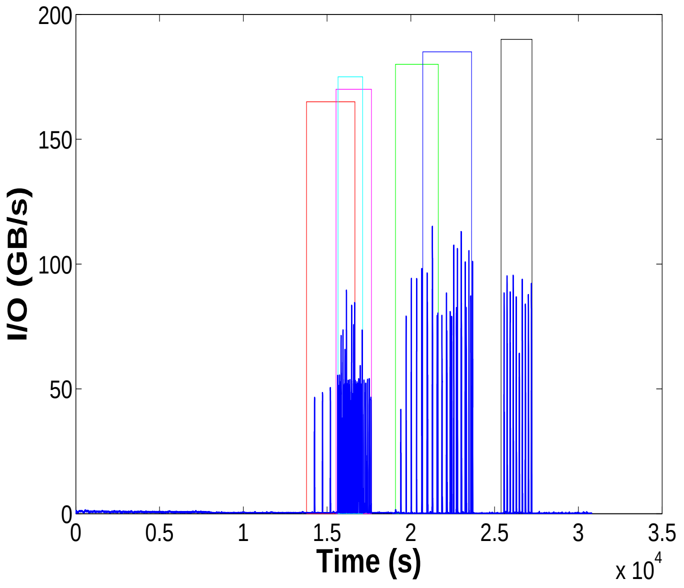
Meanwhile, I/O-heavy jobs may still collide with each other, creating contention-induced performance variance, a challenging obstacle for I/O related execution planning, performance debugging, and optimization [33, 41, 52, 66, 73]. Furthermore, as hard disks remain the dominant storage media, I/O contention leads to excessive seeks, significantly degrading the overall I/O throughput. Figure 1.3 illustrates this situation, using an pseudo-application created with IOR [1], a widely used parallel I/O benchmark. It contains five I/O phases, each writing 2TB data in aggregate from 16348 processes. We repeated the execution 10 times on different days and found that while the majority of runs finish around 3210-3300 seconds, two of them took significantly longer, with the worst one taking 4348 seconds. By checking the server-side aggregate I/O throughput during these execution time windows (Figure 1.3a), we found that the significantly delayed runs correlate with highly elevated overall system bandwidth levels (the 1st and 6th bars), while the other "normal" runs encounter concurrent traffic at a small fraction of the system peak bandwidth. The perceived aggregate I/O throughput for this pseudo-application varies accordingly (Figure 1.3b), from 26.5GB/s to 64.1GB/s. As the 10 runs produce identical I/O traffic, these results clearly demonstrate the impact of "background" I/O activities (from other concurrently running jobs and interactive use) on an individual job.

One major reason for such I/O-induced, large performance variance is the I/O-oblivious job scheduling used in current systems: on HPC systems like Titan (where the tests were conducted) job requests are served in a first-in, first-out (FIFO) order plus backfilling, with further adjustment using priorities, such as job size, queue type, and usage history [3]. *To our best knowledge, I/O behavior/pattern has never been taken into account in production HPC systems' parallel job scheduling.*

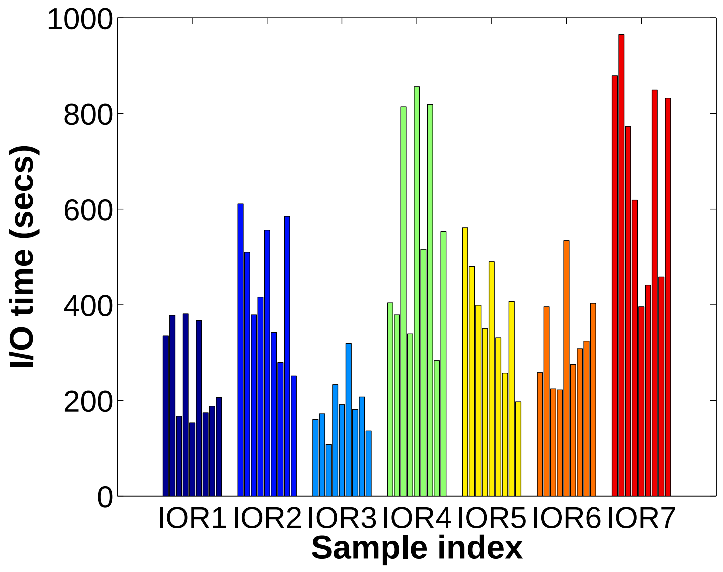
One might wonder that with such an overall under-utilized storage system, I/O-intensive jobs can naturally be spaced apart by the batch queue. To check whether this is the case, we conducted tests using pseudo-applications created with IOR. We defined a set of 7 pseudo-applications, $TEST_1$ - $TEST_7$, each with different configurations, such as node count (64 to 1024), total I/O volume (3 to 100TB), and computation-to-I/O ratio (4% to 20%). We submitted the set of pseudo-applications to the batch scheduler together and repeated such test several times within a week in June 2015. The average queue wait time for each job was over 10 hours.

Figure 1.4a gives a segment of the aggregate server-side I/O traffic data on one Spider partition, captured on 06/13/2015. We highlight the duration of each IOR job with a color rectangle. Though the jobs were queued for much longer than their execution times, their start times were not sufficiently staggered, resulting in remarkable (yet inconsistent) I/O performance variation. Figure 1.4b shows the large distribution of total I/O time across different trials. *E.g.*, the shortest I/O time for $TEST_4$ was 283s, while the longest 856s (around 200% longer).

While there are several studies aimed at reducing inter-job I/O contention [71, 92] (see Chapter 3 for detailed related work discussion), to the best of our knowledge, I/O-aware job scheduling is still not available on production HPC systems. The major obstacle is that HPC applications, while possessing similar periodic I/O behavior, have very different I/O workload patterns. As we mentioned in Chapter 1.1, it is expensive to obtain per-application parallel I/O



(a) I/O traffic with job durations



(b) Job I/O time

Figure 1.4: Sample Inter-job I/O interference

characteristics through tracing/profiling, and is not practical for a supercomputer/cluster to demand such information from application users or developers.

These observations motivate us to propose AID, a system that performs application I/O characterization and I/O-aware job scheduling. AID analyzes existing I/O traffic and batch job history logs, without any application-related prior knowledge or user/developers involvement. As the extension work of IOSI, AID is based on the same intuition: the common behavior observed across multiple executions of the same application is likely attributed to this application. However, IOSI identifies the *I/O signature* of a given I/O-intensive application and assumes its job run instances are identical executions. In contrast, AID takes a fully data-driven approach, sifting out dozens of I/O-heavy applications from job-I/O history containing millions of jobs running thousands of unique applications. These applications can then be given special attention in scheduling to reduce I/O contention. Also, AID utilizes detailed per-OST logs that became available more recently, while the prior work studies aggregate traffic only.

AID correlates the coarse-grain server-side I/O traffic log (both aggregate and OST-level) to (1) identify I/O-intensive applications, (2) "mine" the I/O traffic pattern of applications classified as I/O-intensive, and (3) provide suggestions to the batch job scheduler (regarding whether an I/O-intensive job can be immediately dispatched). Note that AID achieves the above fully automatically, *without requiring any apriori information on the applications or jobs.* It examines the full job log and identifies common I/O patterns across the multiple executions of the same application.

1.3 Contributions

(1) To our best knowledge, this work is the first to explore log-based, automatic HPC application I/O characterization. To this end, we propose and implement IOSI, a tool that identifies given applications I/O signature using existing, zero-overhead, server-side I/O measurements and job scheduling history. Further, we obtain such knowledge of a target application without interfering with its computation and communication, or requiring developers/users' intervention. We evaluated IOSI with real-world server-side I/O throughput logs from the Spider storage system at the Oak Ridge Leadership Computing Facility (OLCF). Our experiments used several pseudo-applications, constructed with the expressive IOR benchmarking tool, and S3D [77], a large-scale turbulent combustion code. Our results show that IOSI effectively extracts an application's I/O signature despite significant server-side noises.

(2) On top of the IOSI techniques, we further explored I/O-aware entirely based on log data analysis, with no apriori information on I/O intensity from users or system admins. To this end, we implemented AID and evaluated it on Titan. We checked its application I/O characterization effectiveness on real applications (partially validated by querying their users) and our own pseudo-applications (where "ground truth" is readily available). For validated I/O-intensive applications, we verified the accuracy of AID's I/O pattern identification. Finally, we assessed the potential gain of I/O-aware job scheduling, by comparing the actual performance

of I/O-intensive jobs dispatched upon different AID suggestions ("run" or "delay"). Our results confirm that AID can successfully identify I/O-intensive applications and their high-level I/O characteristics. In addition, while we currently do not have means to deploy new scheduling policies on Titan, our proof-of-concept evaluation indicates that I/O-aware scheduling might be highly promising for future supercomputing systems.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 introduces related storage infrastructure and existing server-side monitoring infrastructure on supercomputers. Chapter 3 presents the related work about I/O tracing tools and resource-aware job scheduling, etc. In Chapter 4 and Chapter 5, we describe the design and experimental evaluation of IOSI and AID, respectively. Finally, Chapter 6 concludes the dissertation.

Chapter 2

Background

Our work targets petascale or larger platforms, motivated by their extensive storage-side monitoring framework. Below we give a big picture of one such storage system, the ORNL Spider file system [76], supporting Titan and several other clusters.

2.1 Titan's Spider Storage Infrastructure

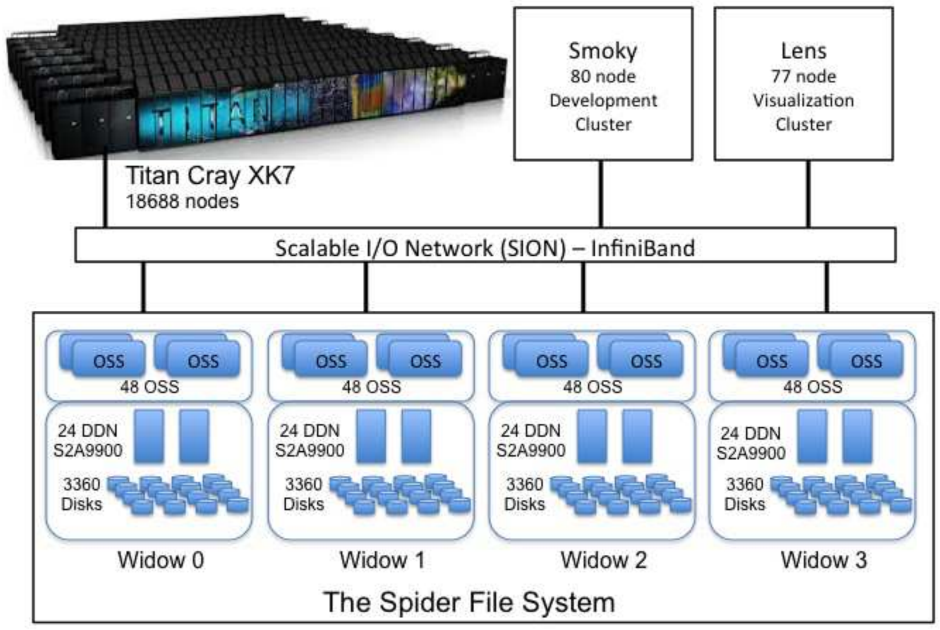


Figure 2.1: Previous Spider storage system architecture

Our prototype development and evaluation use the storage server statistics collected from the Spider center-wide storage system [76] at OLCF, a Lustre-based parallel file system. Spider currently serves the world's No. 2 machine, the 27 petaflop Titan, in addition to other smaller development and visualization clusters. Notice that since Spider has been upgraded to a more powerful system at the beginning of 2014, our IOSI work is based on the previous version of Spider while AID is based on the upgraded system.

Figure 2.1 shows the previous Spider architecture, which comprises of 96 Data Direct Networks (DDN) S2A9900 RAID controllers, with an aggregate bandwidth of 240 GB/s and over 10 PBs of storage from 13,440 1-TB SATA drives. Access is through the object storage servers (OSSs), connected to the RAID controllers in a fail-over configuration. The compute platforms connect to the storage infrastructure over a multistage InfiniBand network, SION (Scalable I/O Network). Spider has four partitions, widow$[0-3]$, with identical setup and capacity.

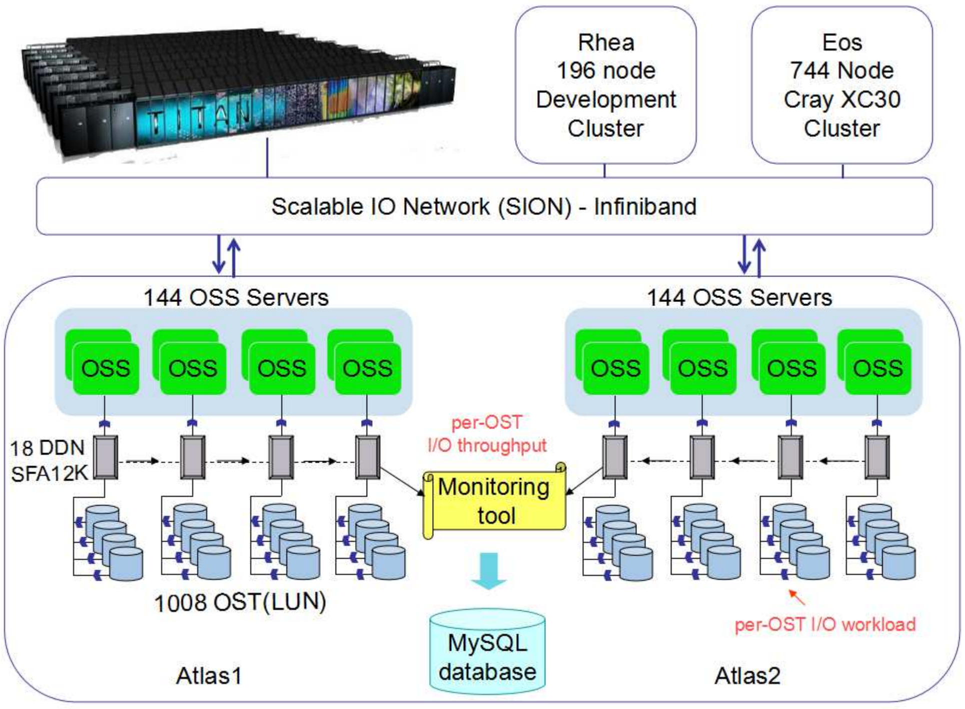


Figure 2.2: Current Spider storage system architecture

Figure 2.2 shows the current Spider architecture. Here 20,160 SATA drives are managed by 36 DDN SFA12K RAID controllers (henceforth referred to as *controllers*). Each 10 disks form a RAID 6 array that makes a Lustre Object Storage Target (OST). Access is via the 288

Lustre Object Storage Servers (OSSs), each with 7 OSTs attached, partitioned into two independent and non-overlapping namespaces, atlas1 and atlas2, for load-balancing and capacity management. Each partition includes half of the system: 144 Lustre OSSs and 1,008 OSTs. The compute nodes (clients) connect to Spider over a multistage InfiniBand network (SION).

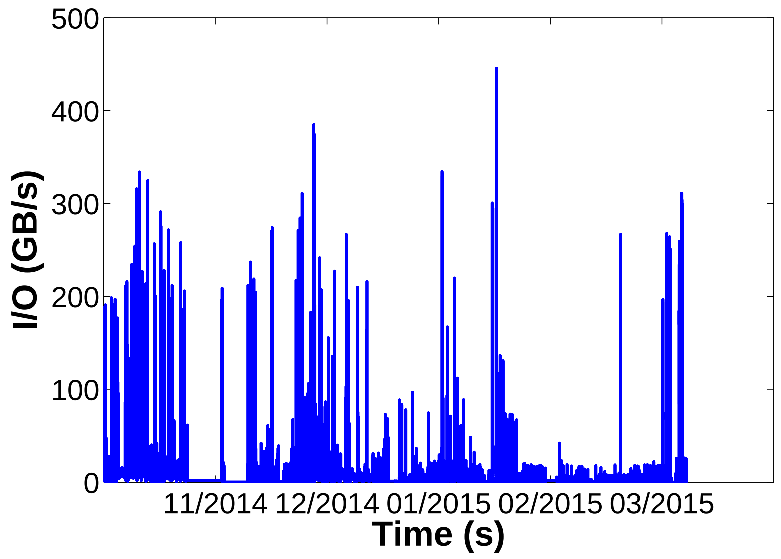
2.2 I/O and Job Data Collection

I/O traffic log Server-side I/O statistics has been collected since 2009 on Spider, via a custom API provided by the DDN controllers. A custom daemon utility [59] polls the controllers periodically and stores the results in a MySQL database. Data collected include read/write I/O throughput and IOPS, I/O request sizes, etc., amounting typically around 4GB log data per day. Unlike client-side or server-side I/O tracing, such coarse-grained monitoring/logging collected over the management Ethernet network (separate from application data traffic) has negligible overhead.

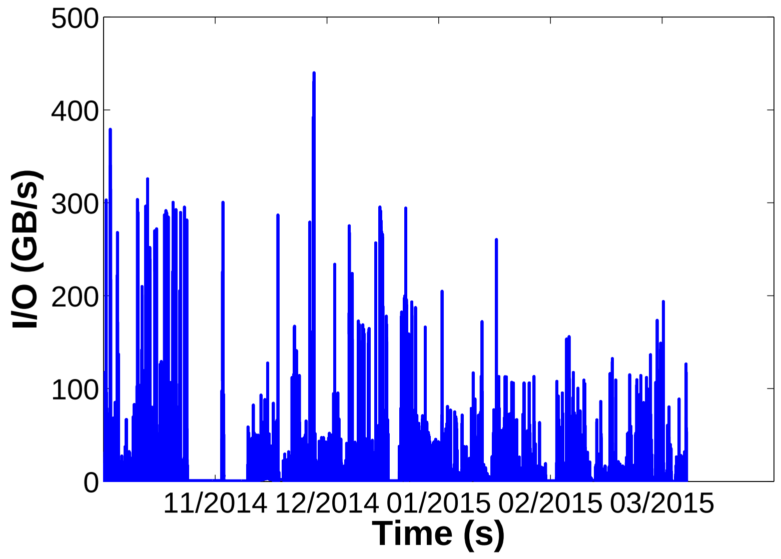
Figure 2.3 shows the aggregated I/O throughput from the two Spider partitions between *2014-11-01* and *2015-03-31*. The observed peak I/O throughput is close to the theoretical peak, around 500 GB/s per partition [65]. However, for most of the time, the I/O throughput is much lower. Also, notable lapses in I/O activity correlated well with recorded system maintenance windows or interruptions to the data collection process. In addition, the two partitions were not utilized equally, with free space on *atlas2* over 40% higher than on *atlas1* (the default partition) in June 2015.

Applications are allocated a set of OSTs. Based on the *stripe width* k, the I/O client round robins across the k *OSTs*. The current Spider has monitoring tools capturing *per-OST* I/O activity. In this work, we leverage this OST-level information for I/O pattern mining and scheduling.

Batch job log Most supercomputers routinely maintain a *batch job log*, recording information on parallel jobs' submission and execution, such as the user/project ID, No. of compute nodes used, job submission/start/end times, job name/ID, etc. By juxtaposing the I/O traffic and job logs, one may "mine" the correlation between I/O traffic and applications' (repeated) executions, to obtain information on application I/O patterns in a lightweight and non-intrusive manner.



(a) Atlas1



(b) Atlas2

Figure 2.3: Spider server-side I/O throughput

Chapter 3

Related Work

3.1 I/O Access Patterns

Miller and Katz observed that scientific I/O has highly sequential and regular accesses, with a period of CPU processing followed by an intense, bursty I/O phase [31]. Carns et al. noted that HPC I/O patterns tend to be repetitive across different runs, suggesting that I/O logs from prior runs can be a useful resource for predicting future I/O behavior [21]. Similar claims have been made by other studies on the I/O access patterns of scientific applications [36, 63, 74]. Such studies strongly motivate IOSI's attempt to identify common and distinct I/O bursts of an application from multiple noisy, server-side logs.

Prior work has also examined the identification and use of I/O signatures. For example, the aforementioned work by Carns et al. proposed a methodology for continuous and scalable characterization of I/O activities [21]. Byna and Chen also proposed an I/O prefetching method with runtime and post-run analysis of applications' I/O signatures [20]. A significant difference is that IOSI is designed to automatically extract I/O signatures from existing coarse-grained server-side logs, while prior approaches for HPC rely on client-side tracing (such as MPI-IO instrumentation). For more generic application workload characterization, a few studies [72, 78, 87] have successfully extracted signatures from various server-side logs.

Very recently, researchers performed comprehensive application I/O characteristics study from several production supercomputer systems [55]. The authors successfully collected profiling data from a large fraction of applications using Darshan and their results provided valuable support on application behavior for AID's design decisions. Meanwhile, AID utilizes existing server-side monitoring and log data (collected with near-zero overhead), and can provide additional application I/O characteristics data to HPC storage/application developers with no user involvement or application modification.

3.2 Client-side I/O Tracing Tools

A number of tools have been developed for general-purpose client-side instrumentation, profiling, and tracing of generic MPI and CPU activity, such as mpiP [81], LANL-Trace [2], HPCT-IO [75], and TRACE [58]. The most closely related to IOSI is probably Darshan [22]. It performs low-overhead, detailed I/O tracing and provides powerful post-processing of log files. It outputs a large collection of aggregate I/O characteristics such as operation counts and request size histograms. However, existing client-side tracing approaches suffer from the limitations mentioned in Chapter 1.1, such as installation/linking requirements, voluntary participation, and producing additional client I/O traffic. Our server-side approach allows us to handle applications using any I/O interface.

3.3 Time-series Data Alignment

There have been many studies in this area [7, 13, 14, 35, 49, 62]. Among them, dynamic time warping (DTW) [14, 62] is a well-known approach for comparing and averaging a set of sequences. Originally, this technique was widely used in the speech recognition community for automatic speech pattern matching [28]. Recently, it has been successfully adopted in other areas, such as data mining and information retrieval, for automatically addressing time deformations and aligning time-series data [23, 38, 42, 90]. Due to its maturity and existing adoption, we choose DTW for comparison against the IOSI algorithms.

3.4 Resource-aware job scheduling

I/O contention has been recognized as an important problem for current and future large-scale HPC platforms [15, 29, 33, 44, 53, 71, 92]. Two studies have proposed platform-level, I/O-aware job scheduling for reducing inter-job interference. Dorier et al. proposed CALCioM [29], which dynamically selects appropriate scheduling policies, coordinating applications' I/O strategy via inter-application communication. Applications on the same compute platform are allowed to communicate and coordinate their I/O strategy with each other to avoid interference. Gainaru et al. proposed a global scheduler [33], which based on applications' past behavior and system characteristics prioritizes I/O operations across applications to reduce I/O congestion. Compared to these systems, AID is much more light-weight and does not require inter-application communication or system-level modification/overhead to schedule I/O operations. In addition, the existing global scheduler [33] only moderates operations from already scheduled applications. Unlike AID, it does not avoid the collision of two I/O-intensive applications with large OST footprints.

A few studies have proposed resource-aware job scheduling to alleviate inter-job resource contention [71, 92], e.g., by considering jobs' communication patterns. Wang et al. [82], developed a library, *libPIO*, that monitors resource usage at the I/O routers, OSSs, OSTs, and the SION InfiniBand Switches; and then based on the load factor allocated OSTs to specific I/O clients. Other systems have explored application-level I/O aware scheduling. Li et al. [48] proposed ASCAR, a storage traffic management framework for improving bandwidth utilization and reducing performance variance. ASCAR also performs I/O pattern classification, but focuses on QoS management between co-running jobs, instead of scheduling high-risk jobs. Novakovic et al. [64] presented DeepDive, a system for transparently identifying and managing interference on cloud services. By monitoring low-level metrics (e.g. hardware performance counters), DeepDive is able to quickly detect that a VM may be suffering from interference and perform follow-up examination plus potentially VM migration. Lofstead et al. [53] proposed an adaptive I/O approach that groups the processes running an application and directs their output to particular storage targets, with inter-group coordination. The same group also presented adaptive I/O methods for the ADIOS I/O middleware to manage IO interference, by dynamically shifting workload from heavily used devices (OSTs) to those more lightly loaded. In Zhang et al. [91] method, when an application performs large I/O, new processes are created to execute the same code, retrieving information on future I/O requests for scheduling. These techniques are complementary to our approach. Meanwhile, AID's global scheduling aims to stagger the relatively small number of high-impact I/O-intensive applications away from each other, reducing the labor and performance cost of application-level scheduling, along with the human effort as well as potential side-effect of uncoordinated optimization by individual applications.

Chapter 4

I/O Signature Identifier

4.1 Background

We first describe the features of typical I/O-intensive parallel applications on supercomputers, which partially enable IOSI.

The majority of applications on today's supercomputers are parallel numerical simulations that perform iterative, timestep-based computations. These applications are write-heavy, periodically writing out intermediate results and checkpoints for analysis and resilience, respectively. For instance, applications compute for a fixed number of timesteps and then perform I/O, repeating this sequence multiple times. This process creates regular, predictable I/O patterns, as noted by many existing studies [31, 68, 83]. More specifically, parallel applications' dominant I/O behavior exhibits several distinct features that enable I/O signature extraction:

Burstiness: Scientific applications have distinct compute and I/O phases. Most applications are designed to perform I/O in short bursts [83].

Periodicity: Most I/O-intensive applications write data periodically, often in a highly regular manner [31, 68] (both in terms of interval between bursts and the output volume per burst). Such regularity and burstiness suggests the existence of steady, wavelike I/O signatures. Note that although a number of studies have been proposed to optimize the checkpoint interval/volume [24, 25, 51], regular, content-oblivious checkpointing is still the standard practice in large-scale applications [70, 89]. IOSI does not depend on such periodic I/O patterns and handles irregular patterns, as long as the application I/O behavior stays consistent across multiple job runs.

Repeatability: Applications on extreme-scale systems typically run many times. Driven by their science needs, users run the same application with different input data sets and model parameters, which results in repetitive compute and I/O behavior. Therefore, applications tend to have a consistent, identifiable workload signature [21]. To substantiate our claim, we have

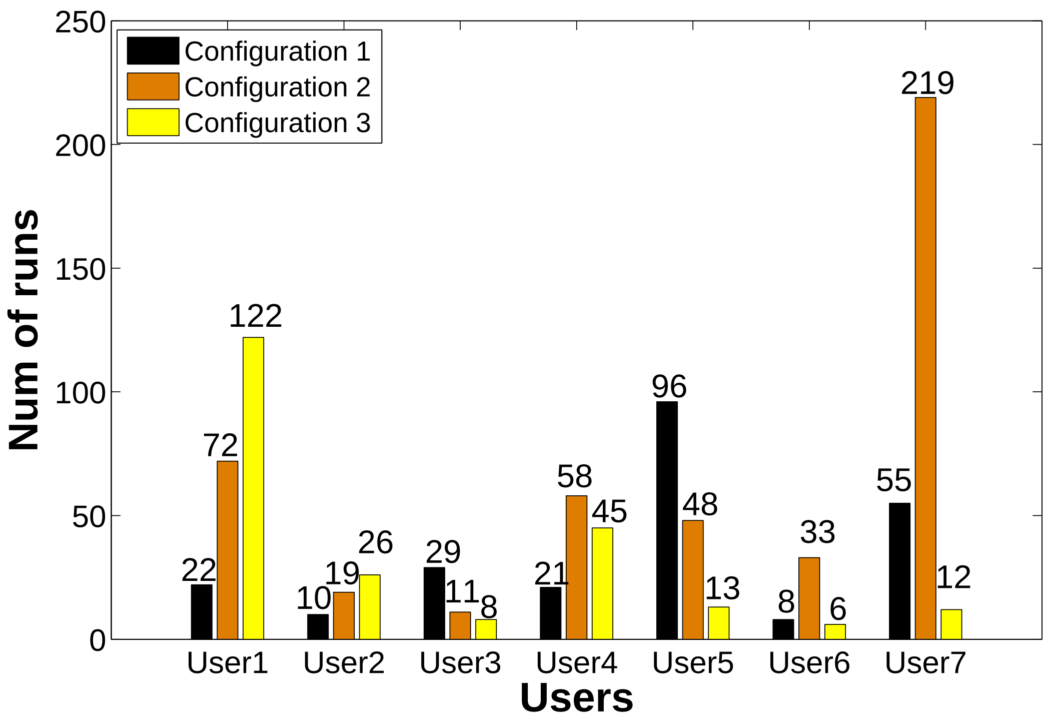


Figure 4.1: Example of the repeatability of runs on Titan, showing the number of runs using identical job configurations for seven users issuing the largest jobs, between July and September 2013.

studied three years worth of Spider server-side I/O throughput logs and Titan job traces for the same time period, and verified that applications have a recurring I/O pattern in terms of frequency and I/O volume. Figure 4.1 plots statistics of per-user jobs using identical job configurations, which is highly indicative of executions of the same application. We see that certain users, especially those issuing large-scale runs, tend to reuse the same job configuration for many executions.

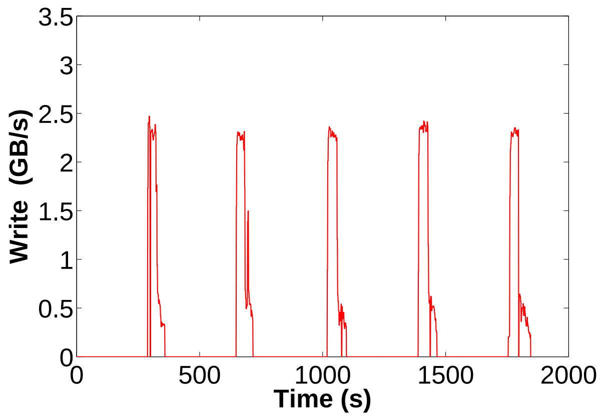
Overall, the above supercomputing I/O features motivate IOSI to find commonality between multiple noisy server-side log samples. Each sample documents the server-side aggregate I/O traffic during an execution of the same target application, containing different and unknown noise signals. The intuition is that *with a reasonable number of samples, the invariant behavior is likely to belong to the target application.*

4.2 Problem Definition: Parallel Application I/O Signature Identification

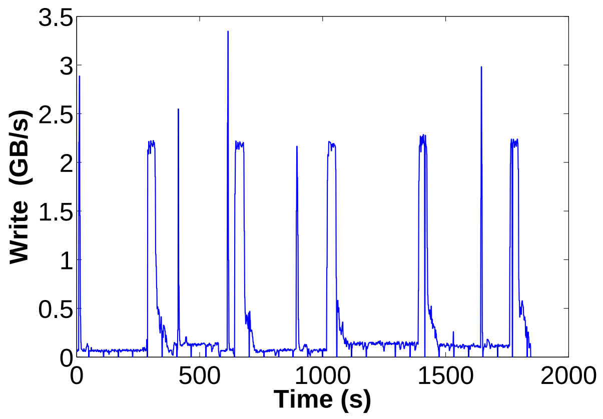
As mentioned earlier, IOSI aims to identify the I/O signature of a parallel application, from zero-overhead, aggregate, server-side I/O throughput logs that are *already* being collected. IOSI's input includes (1) the start and end times of the target application's multiple executions in the past, and (2) server-side logs that contain the I/O throughput generated by those runs (as well as unknown I/O loads from concurrent activities). The output is the extracted I/O signature of the target application.

We define an application's *I/O signature* as the I/O throughput it generates at the server-side storage of a given parallel platform, for the duration of its execution. In other words, if this application runs alone on the target platform without any noise from other concurrent jobs or interactive/maintenance workloads, the server-side throughput log during its execution will be its signature. It is virtually impossible to find such "quiet time" once a supercomputer enters the production phase. Therefore, IOSI needs to "mine" the true signature of the application from server-side throughput logs, collected from its multiple executions. Each execution instance, however, will likely contain different noise signals. We refer to each segment of such a noisy server-side throughput log, punctuated by the start and end times of the execution instance, a *"sample"*. Based on our experience, generally 5 to 10 samples are required for getting the expected results. Note that there are long-running applications (potentially several days for each execution). It is possible for IOSI to extract a signature from even partial samples (e.g., from one tenth of an execution time period), considering the self-repetitive I/O behavior of large-scale simulations.

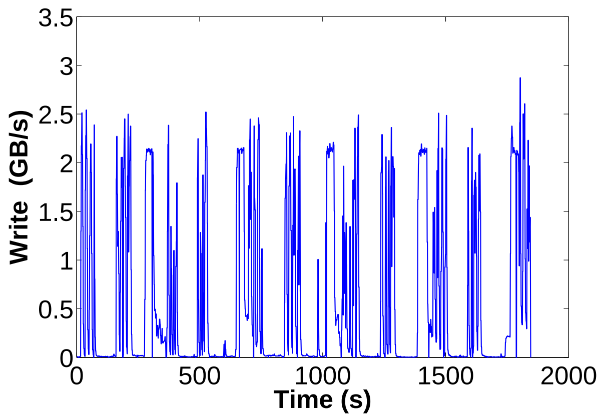
Figure 4.2 illustrates the signature extraction problem using a pseudo-application, IOR_A, generated by IOR. IOR supports most major HPC I/O interfaces (e.g., POSIX, MPIIO, HDF5), provides a rich set of user-specified parameters for I/O operations (e.g., file size, file sharing setting, I/O request size), and allows users to configure iterative I/O cycles. IOR_A exhibits a periodic I/O pattern typical in scientific applications, with 5 distinct *I/O bursts*. Figure 4.2a shows its I/O signature, obtained from a quiet Spider storage system partition during Titan's maintenance window. Figures 4.2b and 4.2c show its two server-side I/O log samples when executed alongside other real applications and interactive I/O activities. These samples clearly demonstrate the existence of varying levels of noise. Thus, IOSI's purpose is to find the common features from multiple samples (e.g., Figures 4.2b and 4.2c), to obtain an I/O signature that approximates the original (Figure 4.2a).



(a) IOR_A target signature



(b) Sample IOR_AS1



(c) Sample IOR_AS6

Figure 4.2: I/O signature of IOR_A and two samples

4.3 Challenges and Approach Overview

Thematic to IOSI is the realization that the noisy, server-side samples contain common, periodic I/O bursts of the target application. It exploits this fact to extract the I/O signature, using a rich set of statistical techniques. Simply correlating the samples is not effective in extracting per-application I/O signatures, due to a set of challenges detailed below.

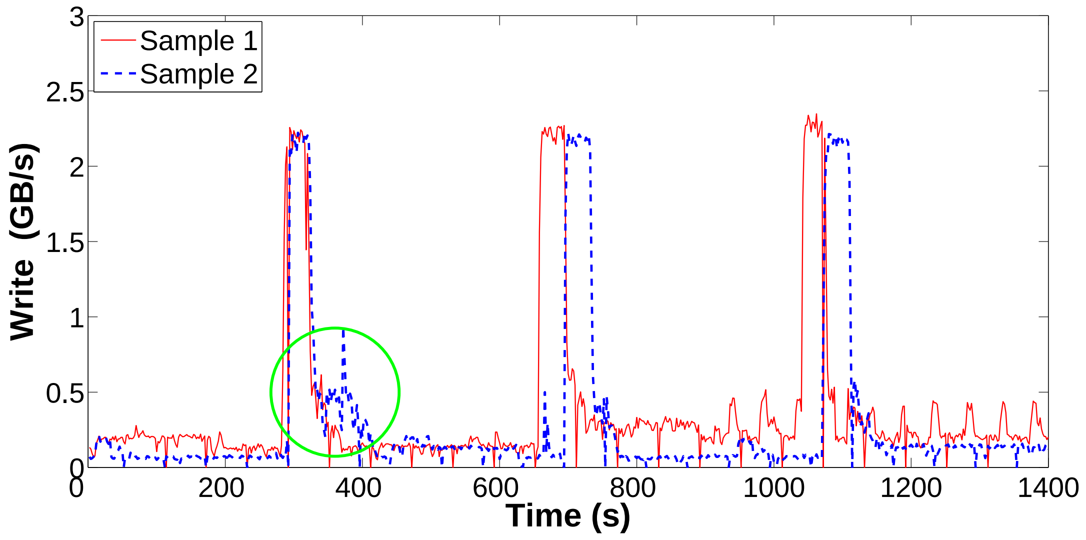


Figure 4.3: Drift and scaling of I/O bursts across samples

First, the server-side logs do not distinguish between different workloads. They contain I/O traffic generated by many parallel jobs that run concurrently, as well as interactive I/O activities (e.g., migrating data to and from remote sites using tools like FTP). Second, I/O contention not only generates "noise" that is superimposed on the true I/O throughput generated by the target application, but also *distorts* it by slowing down its I/O operations. In particular, I/O contention produces *drift* and *scaling* effects on the target application's I/O bursts. The degree of drift and scaling varies from one sample to another. Figure 4.3 illustrates this effect by showing two samples (solid and dashed) of a target application performing periodic writes. It shows that I/O contention can cause shifts in I/O burst timing (particularly with the last two bursts in this case), as well as changes in burst duration (first burst, marked with oval). Finally, the noise level and the runtime variance caused by background I/O further create the following dilemma in processing the I/O signals: IOSI has to rely on the application's I/O bursts to properly *align* the noisy samples as they are the only common features; at the same time, it needs the samples to be reasonably aligned to *identify* the common I/O bursts as belonging to the target application.

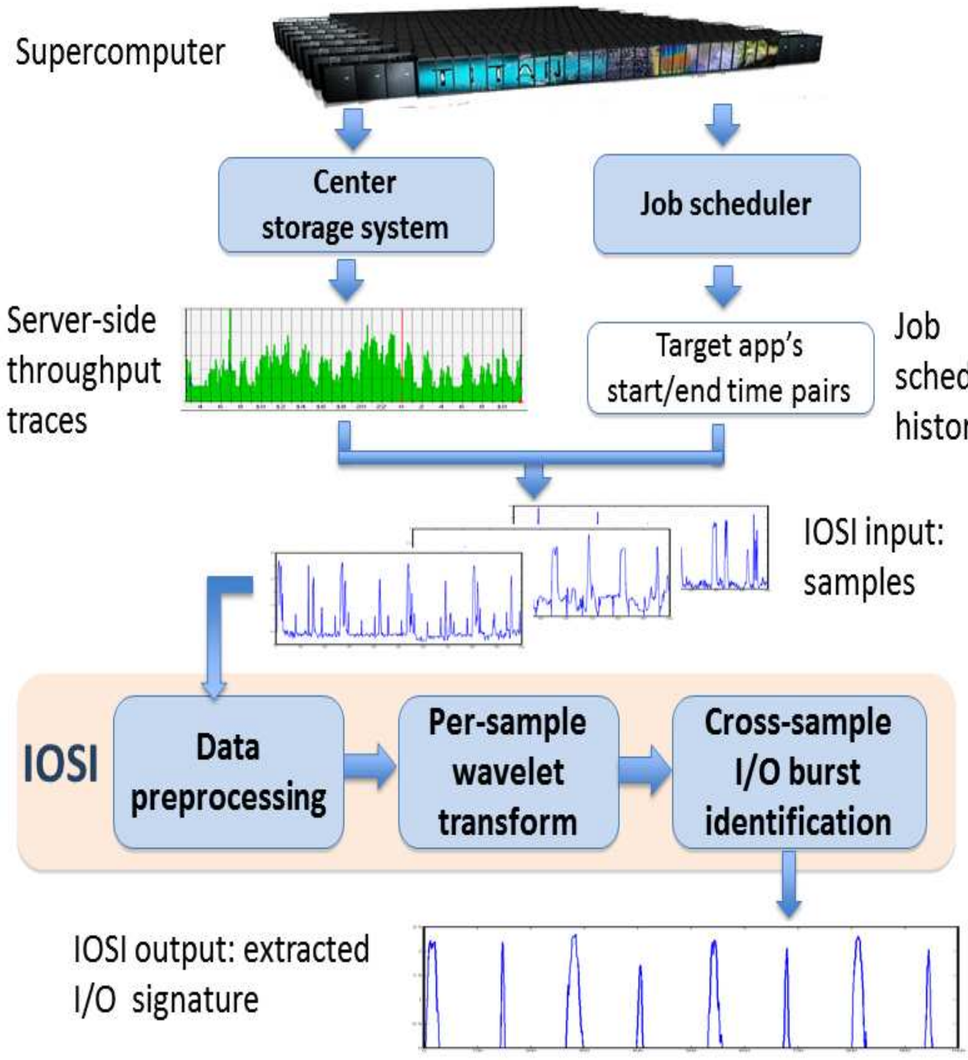

Supercomputer
Center storage system
Job scheduler
Server-side throughput traces
Target app's start/end time pairs
Job scheduling history
IOSI input: samples
IOSI
Data preprocessing
Per-sample wavelet transform
Cross-sample I/O burst identification
IOSI output: extracted I/O signature

Figure 4.4: IOSI overview

Recognizing these challenges, IOSI leverages an array of signal processing and data mining tools to discover the target application's I/O signature using a black-box approach, unlike prior work based on white-box models [22, 80]. Recall that IOSI's purpose is to render a *reliable estimate of user-applications' bandwidth needs*, instead of to optimize individual applications' I/O operations. Black-box analysis is better suited here for generic and non-intrusive pattern collection.

The overall context and architecture of IOSI are illustrated in Figure 4.4. Given a target application, multiple samples from prior runs are collected from the server-side logs. Using such a sample set as input, IOSI outputs the extracted I/O signature by *mining* the common characteristics hidden in the sample set. Our design comprises of three phases:

1. **Data preprocessing**: This phase consists of four key steps: outlier elimination, sample granularity refinement, runtime correction, and noise reduction. The purpose is to prepare the samples for alignment and I/O burst identification.

2. **Per-sample wavelet transform**: To utilize "I/O bursts" as common features, we employ wavelet transform to distinguish and isolate individual bursts from the noisy background.

3. **Cross-sample I/O burst identification**: This phase identifies the common bursts from multiple samples, using a grid-based clustering algorithm.

4.4 IOSI Design and Algorithms

In this section, we describe IOSI's workflow, step by step, using the aforementioned IOR_A pseudo-application (Figure 4.2) as a running example.

4.4.1 Data Preprocessing

Given a target application, we first compare the job log with the I/O throughput log, to obtain I/O samples from the application's multiple executions, particularly *by the same user* and with *the same job size* (in term of node counts). As described in Chapter 4.1, HPC users tend to run their applications repeatedly.

From this set, we then eliminate outliers – samples with significantly heavier noise signals or longer/shorter execution time.[1] Our observation from Spider is that despite unpredictable noise, the majority of the samples (from the same application) bear considerable similarity. Intuitively, including the samples that are apparently significantly skewed by heavy noise is counter-productive. We perform *outlier elimination* by examining (1) the application execution time and (2) the volume of data written within the sample (the "area" under the server-side

[1]Note that shorter execution time can happen with restart runs resuming from a prior checkpoint.

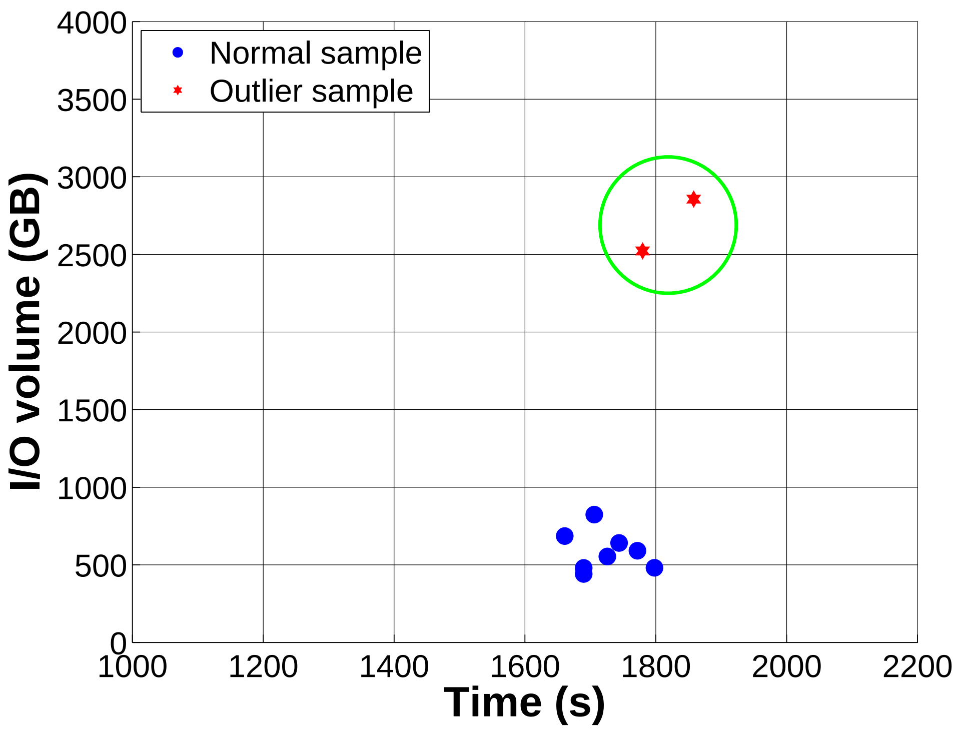


Figure 4.5: Example of outlier elimination

throughput curve). Within this 2-D space, we apply the *Local Outlier Factor (LOF)* algorithm [17], which identifies observations beyond certain threshold as outliers. Here we set the threshold μ as the mean of the sample set. Figure 4.5 illustrates the distribution of execution times and I/O volumes among 10 IOR_A samples collected on Spider, where two of the samples (dots within the circle) are identified by LOF as outliers.

Next, we perform sample granularity refinement, by decreasing the data point interval from 2 seconds to 1 using simple linear interpolation [27]. Thus, we insert an extra data point between two adjacent ones, which turns out to be quite helpful in identifying short bursts that last for only a few seconds. The value of each extra data point is the average value of its adjacent data points. It is particularly effective in retaining the amplitude of narrow bursts during the subsequent WT stage.

In the third step, we perform *duration correction* on the remaining sample data set. This is based on the observation that noise can only prolong application execution, hence the sample with the *shortest* duration received the *least* interference, and is consequently closest in duration to the target signature. We apply a simple trimming process to correct the drift effect mentioned in Chapter 3, preparing the samples for subsequent correlation and alignment. This procedure discards data points at regular intervals to shrink each longer sample to match the shortest one. For example, if a sample is 4% longer than the shortest one, then we remove from it the 1st, 26th, 51st, ..., data points. We found that after outlier elimination, the deviation in sample duration is typically less than 10%. Therefore such trimming is not expected to significantly affect the sample data quality.

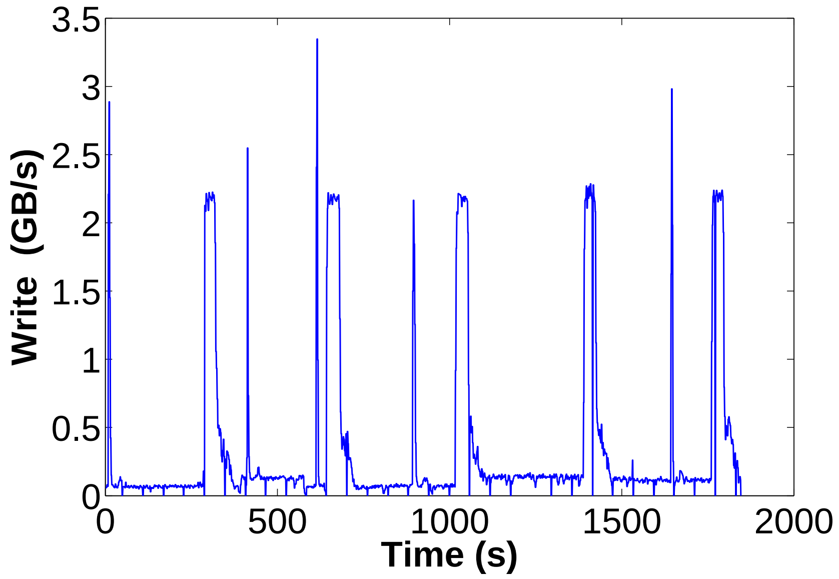
Finally, we perform preliminary *noise reduction* to remove *background noise*. While I/O-intensive applications produce heavy I/O bursts, the server-side log also reports I/O traffic from interactive user activities and maintenance tasks (such as disk rebuilds or data scrubbing by the RAID controllers). Removing this type of persistent background noise significantly helps signature extraction. In addition, although such noise does not significantly distort the shape of application I/O bursts, having it embedded (and duplicated) in multiple application's I/O signatures will cause inaccuracies in I/O-aware job scheduling. To remove background noise, IOSI (1) aggregates data points from all samples, (2) collects those with a value lower than the overall average throughput, (3) calculates the *average background noise level* as the mean throughput from these selected data points, and (4) lowers each sample data point by this average background noise level, producing zero if the result is negative. Figure 4.6b shows the result of such preprocessing, and compared to the original sample in Figure 4.6a, the I/O bursts are more pronounced. The I/O volume of IOR_AS1 was trimmed by 26%, while the background noise level was measured at 0.11 GB/s.

4.4.2 Per-Sample Wavelet Transform

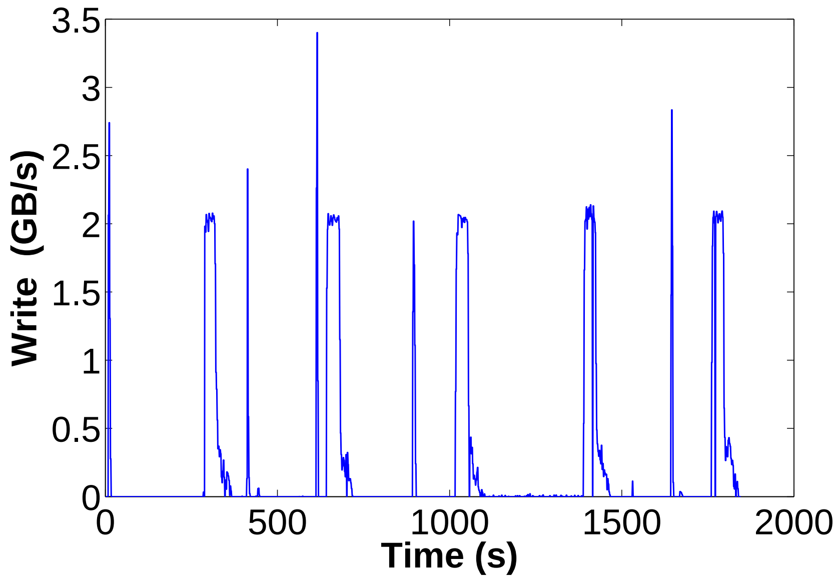
As stated earlier, scientific applications tend to have a bursty I/O behavior, justifying the use of *I/O burst* as the basic unit of signature identification. An I/O burst indicates a phase of high I/O activity, distinguishable from the background noise over a certain duration.

With less noisy samples, the burst boundaries can be easily found using simple methods such as *first difference* [69] or *moving average* [84]. However, with noisy samples identifying such bursts becomes challenging, as there are too many ups and downs close to each other. In particular, it is difficult to do so without knowing the cutoff threshold for a "bump" to be considered a candidate I/O burst. Having too many or too few candidates can severely hurt our sample alignment in the next step.

To this end, we use a WT [26, 57, 86] to smooth samples. WT has been widely applied to problems such as filter design [19], noise reduction [45], and pattern recognition [30]. With WT, a time-domain signal can be decomposed into low-frequency and high-frequency components. The approximation information remains in the low-frequency component, while the detail information remains in the high-frequency one. By carefully selecting the *wavelet function* and *decomposition level* we can observe the major bursts from the low-frequency component. They



(a) Before noise reduction



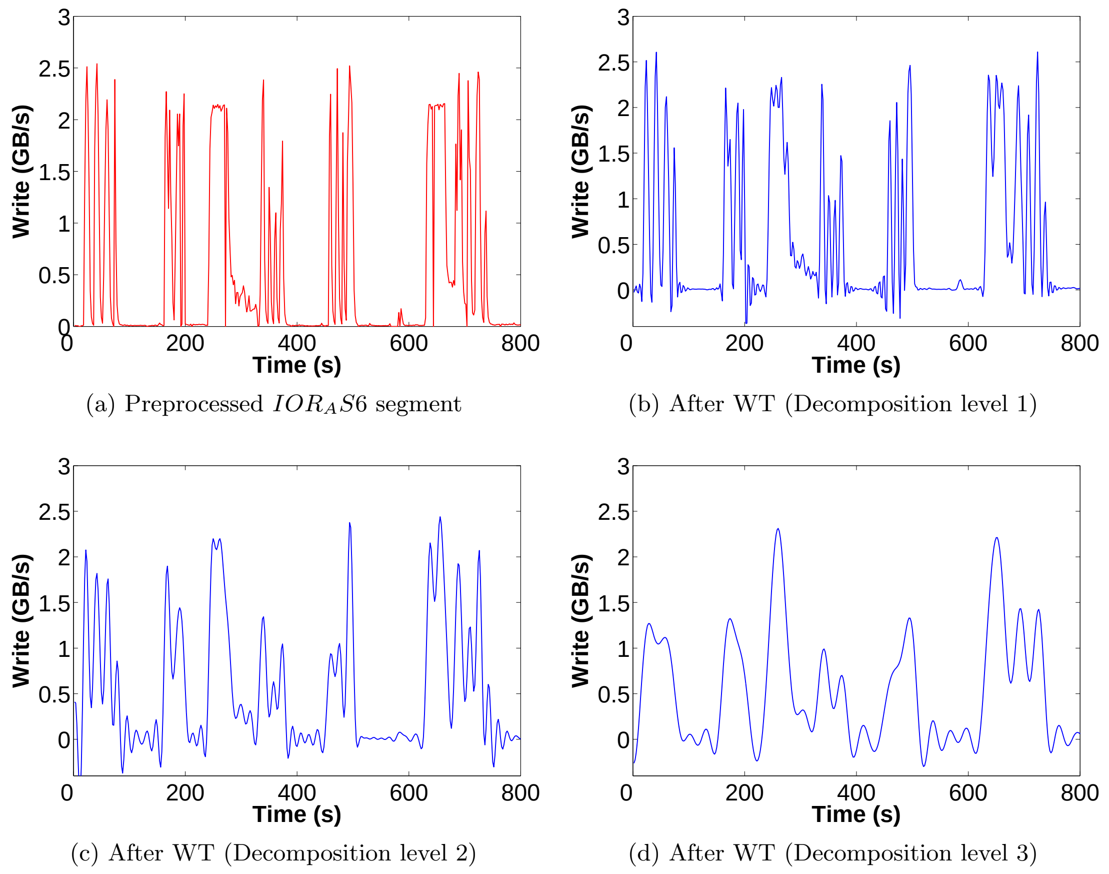
(b) After noise reduction

Figure 4.6: IOR_A samples after noise reduction

contain the most energy of the signal and are isolated from the background noise.

By retaining the temporal characteristics of the time-series data, WT brings an important feature not offered by widely-used alternative techniques such as Fourier transform [16]. We use WT to clarify individual bursts from their surrounding data, without losing the temporal characteristics of the time-series sample.

WT can use quite a few wavelet families [5, 61], such as *Haar*, *Daubechies*, and *Coiflets*. Each provides a transform highlighting different frequency and temporal characteristics. For IOSI, we choose *discrete Meyer (dmey)* [54] as the mother wavelet. Due to its smooth profile, the approximation part of the resulting signal consists of a series of smooth waves. Its output consists of a series of waves where the center of the wave troughs can be easily identified as wave boundaries.



(a) Preprocessed IOR_AS6 segment

(b) After WT (Decomposition level 1)

(c) After WT (Decomposition level 2)

(d) After WT (Decomposition level 3)

Figure 4.7: *Dmey* WT results on a segment of IOR_AS6

Figures 4.7a and Figures 4.7b, 4.7c, 4.7d illustrate a segment of IOR_AS6 and its dmey WT results, respectively. With a WT, the high-frequency signals in the input sample are smoothed, producing low-frequency components that correlate better with the target application's periodic I/O. However, here the waves cannot be directly identified as I/O bursts, as a single I/O burst from the application's point of view may appear to have many "sub-crests", separated by minor troughs. This is due to throughput variance caused by either application behavior (such as adjacent I/O calls separated by short computation/communication) or noise, or both. To prevent creating many such "sub-bursts", we use the mean height of all wave crests for filtering – only the troughs lower than this threshold are used for separating bursts.

Another WT parameter to consider is the *decomposition level*, which determines the level of detailed information in the results. The higher the decomposition level, the fewer details are shown in the low-frequency component, as can be seen from Figures 4.7b, 4.7c and 4.7d. With a decomposition level of 1 (e.g. Figures 4.7b), the wavelet smoothing is not sufficient for isolating burst boundaries. With a higher decomposition level of 3 the narrow bursts fade out rapidly, potentially missing target bursts. IOSI uses a decomposition level of 2 to better retain the bursty nature of the I/O signature.

4.4.3 Cross-Sample I/O Burst Identification

Next, IOSI correlates all the pre-processed, and wavelet transformed samples to identify common I/O bursts. To address the circular dependency challenge mentioned earlier between alignment and common feature identification, we adapt a grid-based clustering approach called CLIQUE [9]. It performs multi-dimensional data clustering by identifying grids (called *units*) where there is higher *density* (number of data points within the unit). CLIQUE treats each such *dense unit* as a "seed cluster" and grows it by including neighboring dense units.

CLIQUE brings several advantages to IOSI. First, its model fits well with our context: an I/O burst from a given sample is mapped to a 2-D data point, based on its *time* and *shape* attributes. Therefore, data points from different samples close to each other in the 2-D space naturally indicate common I/O bursts. Second, with controllable grid width and height, IOSI can better handle burst drifts (more details given below). Third, CLIQUE performs well for scenarios with far-apart clusters, where inter-cluster distances significantly exceed those between points within a cluster. As parallel applications typically limit their "I/O budget" (fraction of runtime allowed for periodic I/O) to 5%-10%, individual I/O bursts normally last seconds to minutes, with dozens of minutes between adjacent bursts. Therefore, CLIQUE is not only effective for IOSI, but also efficient, as we do not need to examine too far around the burst-intensive areas. Our results (Chapter 4.5) show that it outperforms the widely used DTW time-series alignment algorithm [14], while incurring significantly lower overhead.

We make two adjustments to the original CLIQUE algorithm. Considering the I/O bursts are sufficiently spaced from each other in a target application's execution, we limit the growth of the cluster to the immediate *neighborhood* of a dense unit: the units that are adjacent to it. Also, we have modified the density calculation to focus not on the sheer number of data points

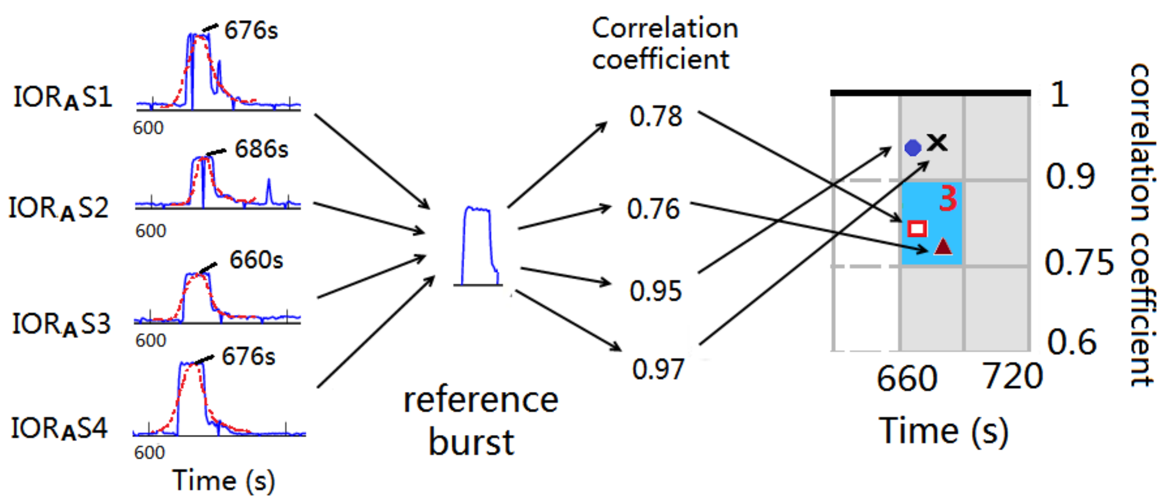


Figure 4.8: Mapping IOR_A I/O bursts to 2-D points

in a unit, but on the number of *different samples* with bursts there. The intuition is that a common burst from the target application should have most (if not all) samples agree on its existence. Below, we illustrate with IOR_A the process of IOSI's common burst identification.

To use our adapted CLIQUE, we need to first discretize every sample s_i into a group of 2-D points, each representing one I/O burst identified after a WT. Given its jth I/O burst $b_{i,j}$, we map it to point $\langle t_{i,j}, c_{i,j} \rangle$. Here $t_{i,j}$ is the time of the wave crest of $b_{i,j}$, obtained after a WT, while $c_{i,j}$ is the correlation coefficient between $b_{i,j}$ and a *reference burst*. To retain the details describing the shape of the I/O burst, we choose to use the pre-WT burst in calculating $c_{i,j}$, though the burst itself was identified using a WT. Note that we rely on the transitive nature of correlation ("bursts with similar correlation coefficient to the common reference burst tend to be similar to each other"), so the reference burst selection does not have a significant impact on common burst identification. In our implementation, we use the "average burst" identified by WT across all samples.

Figure 4.8 shows how we mapped 4 I/O bursts, each from a different IOR_A sample. Recall that WT identifies each burst's start, peak, and end points. The x coordinate for each burst shows "when it peaked," derived using the post-WT wave (dotted line). The y coordinate shows "how similar it is to a reference burst shape," calculated using the pre-WT sample (solid lines).

Therefore, our CLIQUE 2-D data space has an x range of $[0, t]$ (where t is the adjusted sample duration after preprocessing) and a y range of $[0, 1]$. It is partitioned into uniform 2-D grids (units). Defining the unit width and height is critical for CLIQUE, as overly small or large grids will obviously render the density index less useful. Moreover, even with carefully

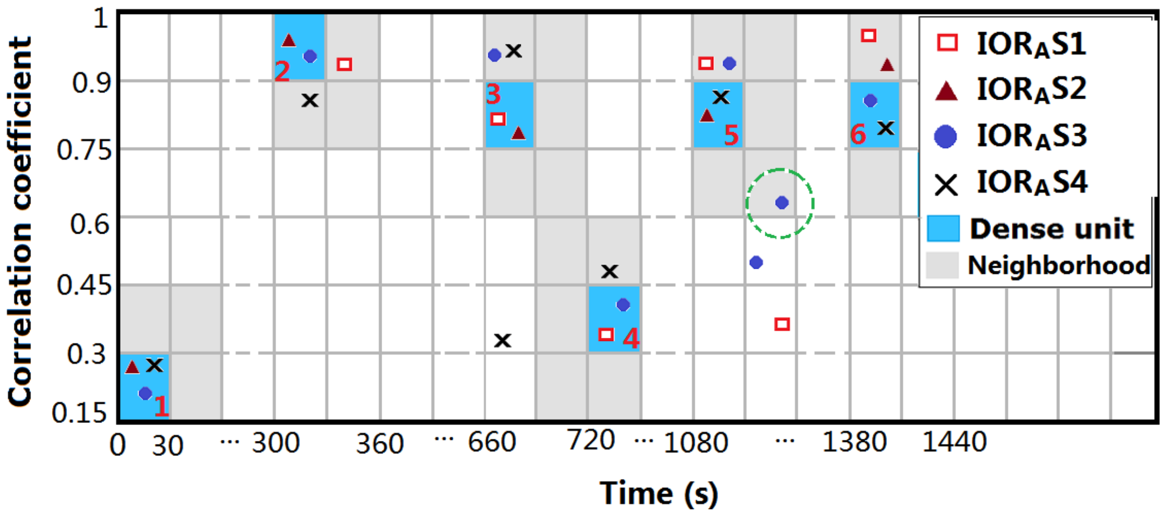


Figure 4.9: CLIQUE 2-D grid containing IOR_A bursts

selected width and height values, there is still a chance that a cluster of nodes are separated into different grids, causing CLIQUE to miss a dense unit.

To this end, instead of using only one pair of width-height values, IOSI tries out multiple grid size configurations, each producing an extracted signature. For width, it sets the lower bound as the average I/O burst duration across all samples and upper bound as the average time distance between bursts. For a unit height, it empirically adopts the range between 0.05 and 0.25. Considering the low cost of CLIQUE processing with our sample sets, IOSI uniformly samples this 2-D parameter space (e.g., with 3-5 settings per dimension), and takes the result that identified the most data points as belonging to common I/O bursts. Due to the strict requirement of identifying common bursts, we have found in our experiments that missing target bursts is much more likely to happen than including fake bursts in the output signature. Figure 4.9 shows the resulting 2-D grid, containing points mapped from bursts in four IOR_A samples.

We have modified the original *dense unit* definition as follows. Given s samples, we calculate the *density* of a unit as "the number of samples that have points within this unit". If a unit meets a certain density threshold $\lceil \gamma * s \rceil$, where γ is a controllable parameter between 0 and 1, the unit is considered dense. Our experiments used a γ value of 0.5, requiring each dense unit to have points from at least 2 out of the 4 samples. All dense units are marked with a dark shade in Figure 4.9.

Due to the time drift and shape distortion caused by noise, nodes from different samples representing the same I/O burst could be partitioned by grid lines. As mentioned earlier, for each dense unit, we only check its immediate neighborhood (shown in a lighter shade in Figure 4.9) for data points potentially from a common burst. We identify *dense neighborhoods* (including the central dense unit) as those meeting a density threshold of $\lceil \kappa * s \rceil$, where κ is another configurable parameter with value larger than γ (e.g., 0.9).

Note that it is possible for the neighborhood (or even a single dense unit) to contain multiple points from the same sample. IOSI further identifies points from the common I/O burst using a *voting* scheme. It allows up to one point to be included from each sample, based on the total normalized Euclidean distance from a candidate point to peers within the neighborhood. From each sample, the data point with the lowest total distance is selected. In Figure 4.9, the neighborhood of dense unit 5 contains two bursts from IOR_AS3 (represented by dots). The burst in the neighborhood unit (identified by the circle) is discarded using our voting algorithm. As the only "redundant point" within the neighborhood, it is highly likely to be a "fake burst" from other concurrently running I/O-intensive applications. This can be confirmed by viewing the original sample IOR_AS3 in Figure4.10b, where a tall spike not from IOR_A shows up around the 1200th second.

4.4.4 I/O Signature Generation

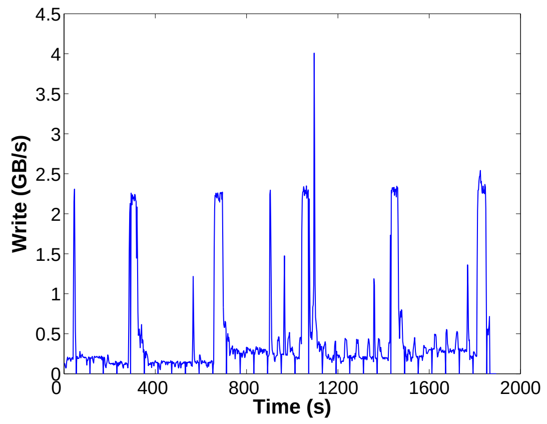
Given the common bursts from dense neighborhoods, we proceed to sample alignment. This is done by aligning all the data points in a common burst to the average of their x coordinate values. Thereafter, we generate the actual I/O signature by sweeping along the x (time) dimension of the CLIQUE 2-D grid. For each dense neighborhood identified, we generate a corresponding I/O burst at the aligned time interval, by averaging the bursts mapped to the selected data points in this neighborhood. Here we used the bursts after preprocessing, but before WT.

4.5 Experimental Evaluation

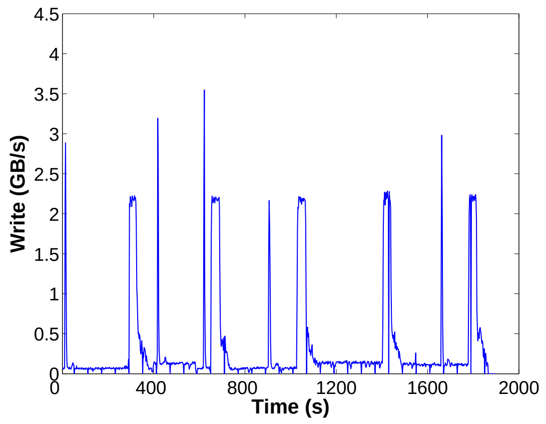
4.5.1 IOSI Experimental Evaluation

We have implemented the proof-of-concept IOSI prototype using Matlab and Python. To validate IOSI, we used IOR to generate multiple pseudo-applications with different I/O write patterns, emulating write-intensive scientific applications. In addition, we used S3D [39, 77], a massively parallel direct numerical simulation solver developed at Sandia National Laboratory for studying turbulent reacting flows.

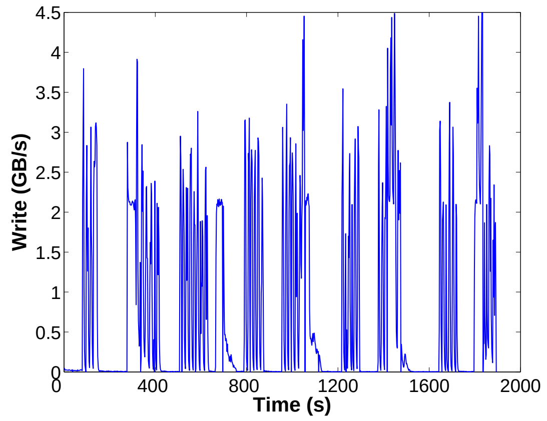
Figure 4.13a, 4.14a and 4.15a are the true I/O signatures of the three IOR pseudo-applications, IOR_A, IOR_B, and IOR_C. These pseudo-applications were run on the Smoky cluster using 256 processes, writing to the Spider center-wide parallel file system. Each process was configured to write sequentially to a separate file (stripe size of 1MB, stripe count of 4) using



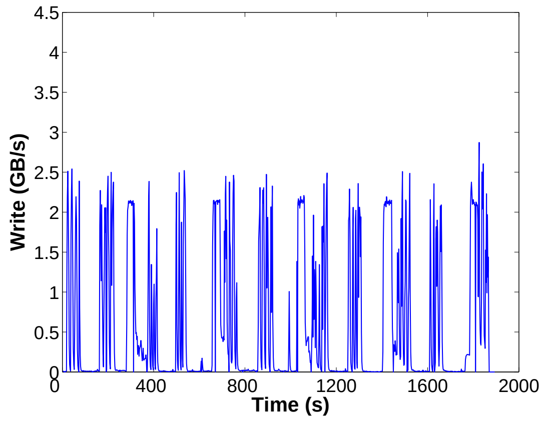
(a) IOR_AS1



(b) IOR_AS2



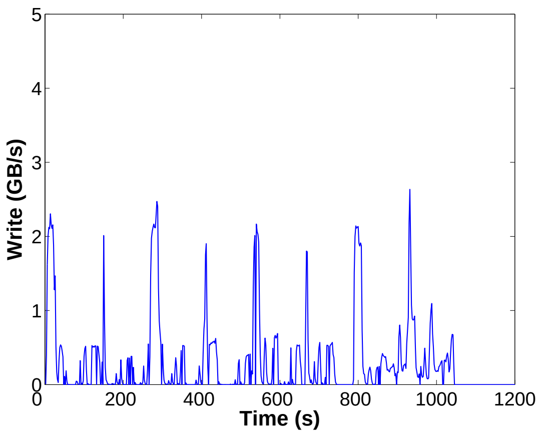
(c) IOR_AS3



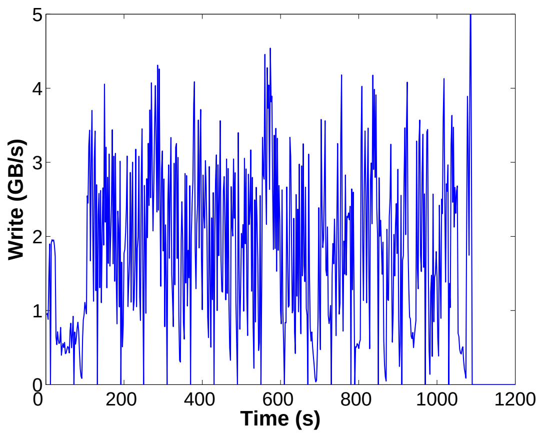
(d) IOR_AS4

Figure 4.10: Samples from IOR_A test cases

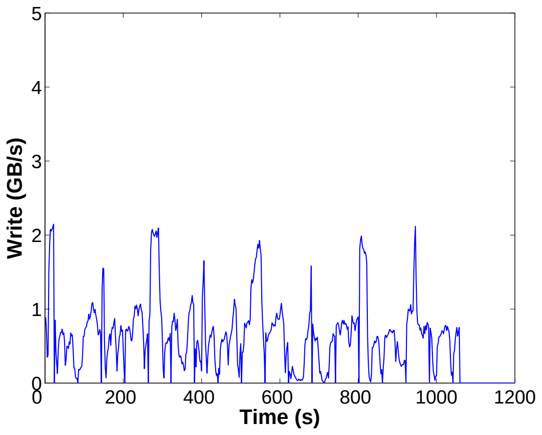
MPI-IO. We were able to obtain "clean" signatures (with little noise) for these applications by running our jobs when Titan was not in production use (under maintenance) and one of the file system partitions was idle. Among them, IOR_A represents simple periodic checkpointing, writing the same volume of data at regular intervals (128GB every 300s). IOR_B also writes periodically, but alternates between two levels of output volume (64GB and 16GB every 120s), which is typical of applications with different frequencies in checkpointing and results writing (e.g., writing intermediate results every 10 minutes but checkpointing every 30 minutes). IOR_C has more complex I/O patterns, with three different write cycles repeated periodically (one output phase every 120s, with output size cycling through 64GB, 32GB, and 16GB).



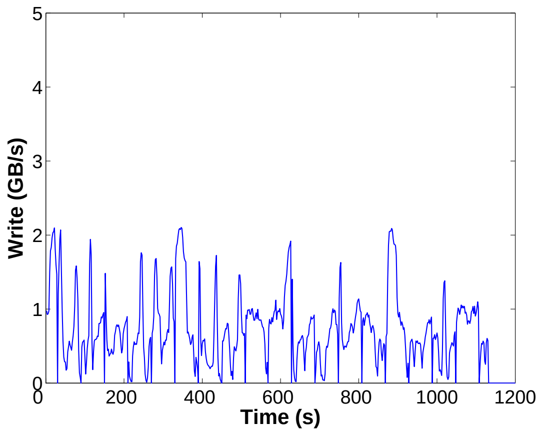
(a) IOR_BS1



(b) IOR_BS2



(c) IOR_BS3



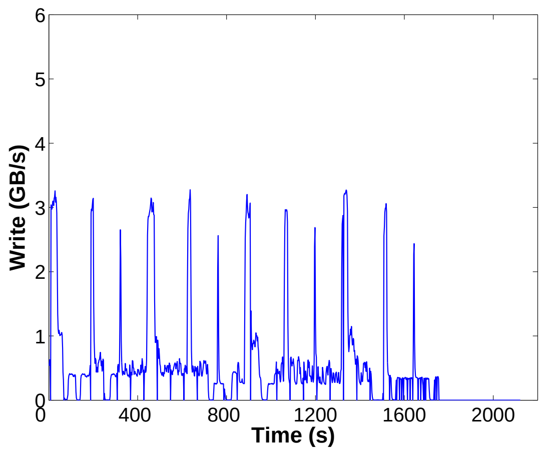
(d) IOR_BS4

Figure 4.11: Samples from IOR_B test cases

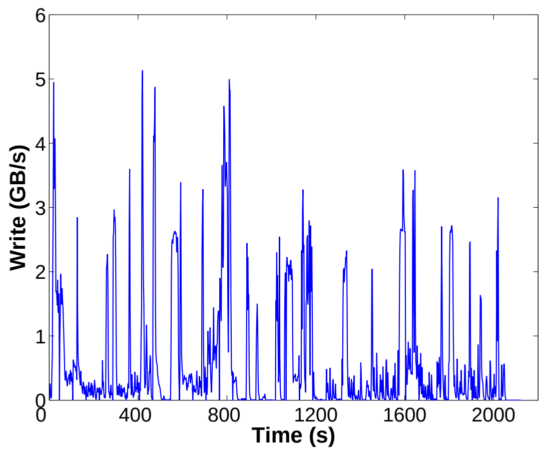
IOR Pseudo-Application Results

To validate IOSI, the IOR pseudo-applications were run at different times of the day, over a two-week period. Each application was run at least 10 times. During this period, the file system was actively used by Titan and other clusters. The I/O activity captured during this time is the noisy server-side throughput logs. From the scheduler's log, we identified the execution time intervals for the IOR runs, which were then intersected with the I/O throughput log to obtain per-application samples.

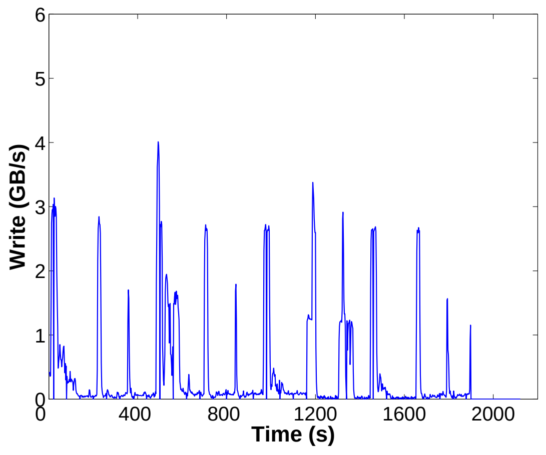
It is worth noting that the I/O throughput range of all of the IOR test cases is designed to be 2-3GB/s. After analyzing several months of Spider log data, we observed that it is this low-bandwidth range that is highly impacted by background noise. If the bandwidth of the



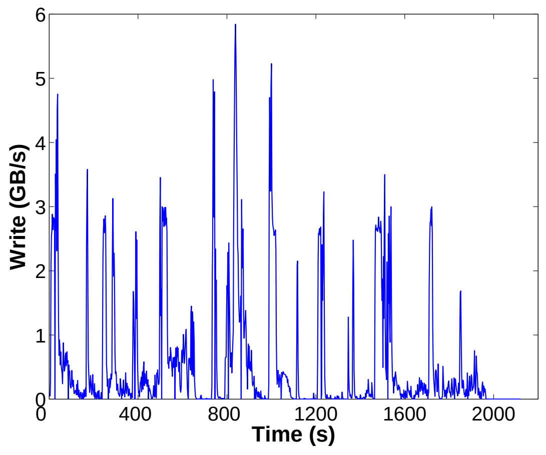
(a) $IOR_C S1$



(b) $IOR_C S2$



(c) $IOR_C S3$

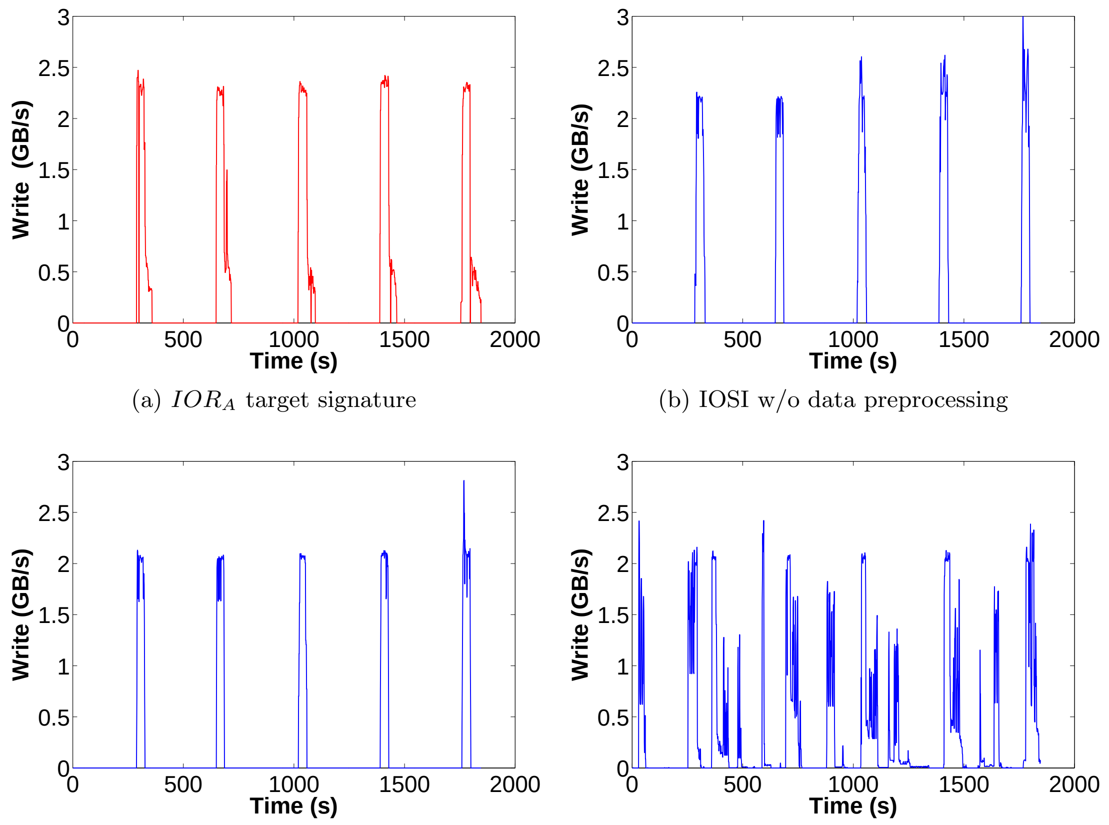


(d) $IOR_C S4$

Figure 4.12: Samples from IOR_C test cases

application is much higher (say 20GB/s), the problem becomes much easier, since there is less background noise that can achieve that bandwidth level to interfere.

Due to the space limit, we only show four samples for each pseudo-app in Figure 4.10, 4.11 and 4.12. We observe that most of them show human-visible repeated patterns that overlap with the target signatures. There is, however, significant difference between the target signature and any individual sample. The samples show a non-trivial amount of "random" noise, sometimes (e.g., $IOR_A S1$) with distinct "foreign" repetitive pattern, most likely from another application's periodic I/O. Finally, a small number of samples are noisy enough to make the target signature appear overwhelmed (which should be identified as outliers and discarded from



(a) IOR_A target signature

(b) IOSI w/o data preprocessing

(c) IOSI with data preprocessing

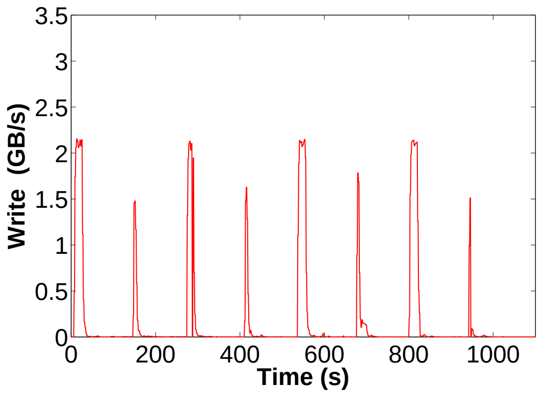
(d) DTW with data preprocessing

Figure 4.13: Target and extracted I/O signatures of IOR_A test cases

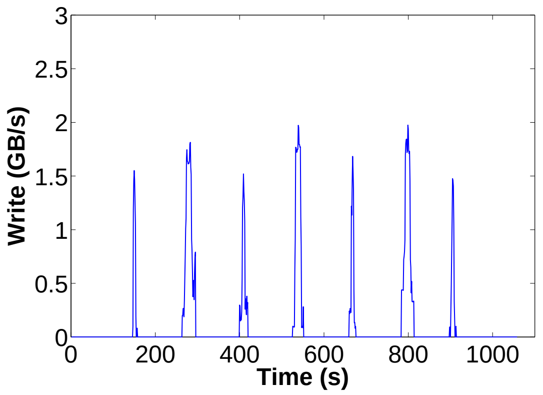
signature extraction).

Figure 4.13, 4.14 and 4.15 presents the original signatures and the extracted signatures using three approaches: IOSI with and w/o data preprocessing, plus DTW with data preprocessing. As introduced in Chapter 3, DTW is a widely used approach for finding the similarity between two data sequences. In our problem setting, similarity means a match in I/O bursts from two samples. We used sample preprocessing to make a fair comparison between DTW and IOSI. Note that IOSI without data preprocessing utilizes samples after granularity refinement, to obtain extracted I/O signatures with similar length across all three methods tested.

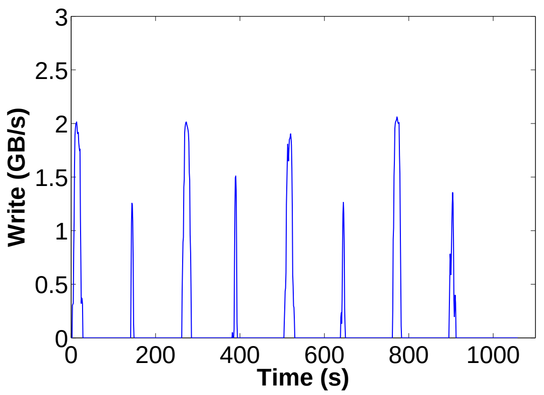
Since DTW performs pair-wise alignment, it is unable to perform effective global data alignment across multiple samples. In our evaluation, we apply DTW as follows. We initially assign a pair of samples as input to DTW, and feed the result along with another sample to DTW again. This process is repeated until all samples are exhausted. We have verified that this approach performs better (in terms of both alignment accuracy and processing overhead) than



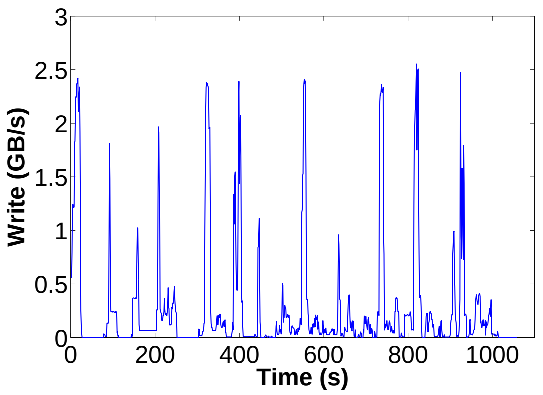
(a) IOR_B target signature



(b) IOSI w/o data preprocessing



(c) IOSI with data preprocessing

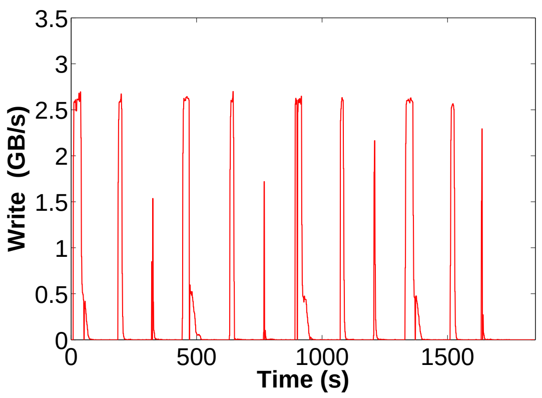


(d) DTW with data preprocessing

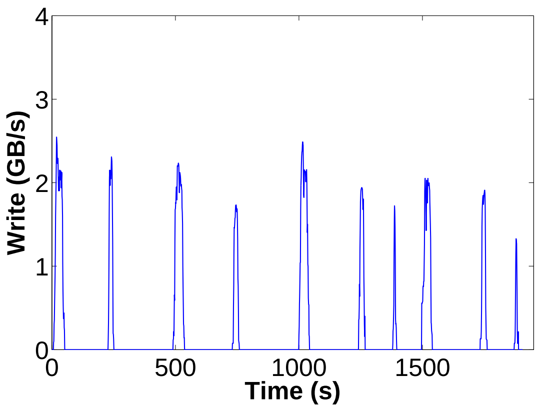
Figure 4.14: Target and extracted I/O signatures of IOR_B test cases

the alternative of averaging all pair-wise DTW results, since it implicitly carries out global data alignment. Still, DTW generated significantly lower-quality signatures, especially with more complex I/O patterns, due to its inability to reduce noise. For example, DTW's IOR_C (Figure 4.15d) signature appears to be dominated by noise.

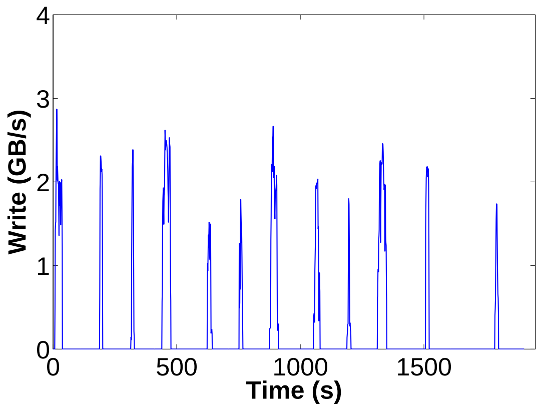
In contrast, IOSI (with or w/o data preprocessing) generated output signatures with much higher fidelity. In both cases, IOSI is highly successful in capturing I/O bursts in the time dimension (with small, yet visible errors in the vertical height of the bursts). Without preprocessing, IOSI missed 3 out of the 25 I/O bursts from all pseudo-applications. With preprocessing, however, the symptom is much improved (no burst missed).



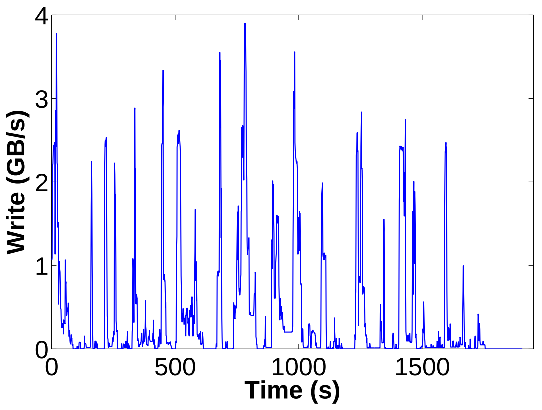
(a) IOR_C target signature



(b) IOSI w/o data preprocessing



(c) IOSI with data preprocessing

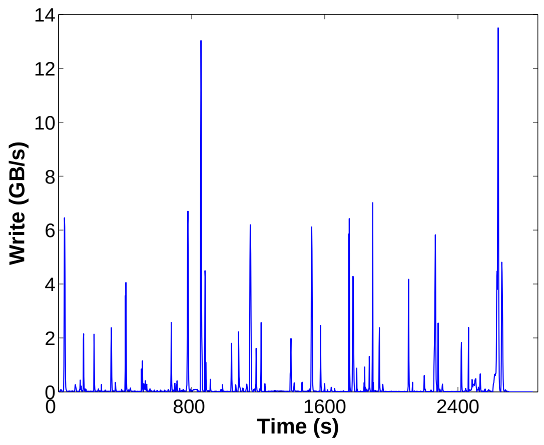


(d) DTW with data preprocessing

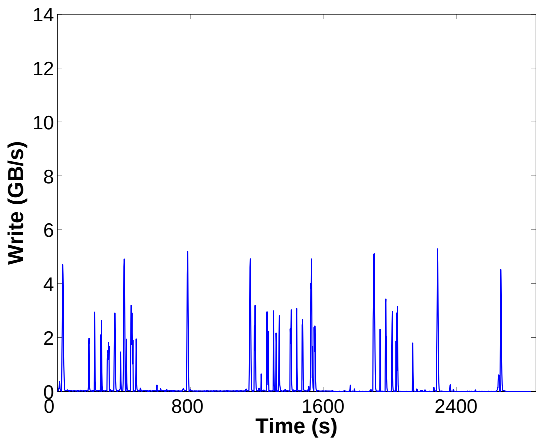
Figure 4.15: Target and extracted I/O signatures of IOR_C test cases

S3D Results

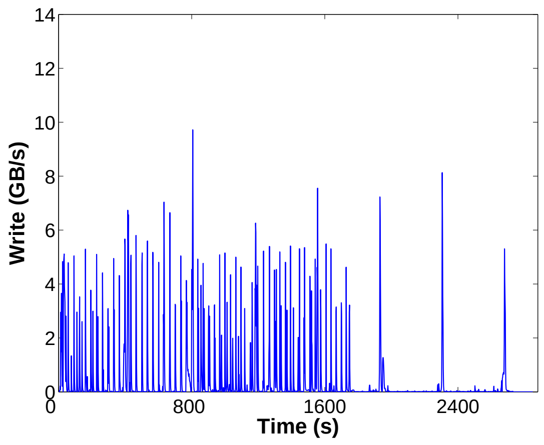
Next, we present results with the aforementioned large-scale scientific application, S3D. S3D was run on Titan and the Spider file system. S3D performs periodic checkpointing I/O, with each process generating 3.4 MB of data. Figure 4.16 shows selected samples from multiple S3D runs, where we see a lot of I/O interference since both Titan and Spider were being used in production mode. Unlike IOR, we were not able to run S3D on a quiescent Spider file system partition to obtain its "clean" signature to validate IOSI. Instead, we had to use client-side I/O tracing, to produce the target I/O signature (Figure 4.17a). The I/O signature also displays variance in peak bandwidth, common in real-world, large job runs. Again, we extracted the I/O signature from the samples using IOSI (with and without data preprocessing), plus DTW with preprocessing (Figure 4.17).



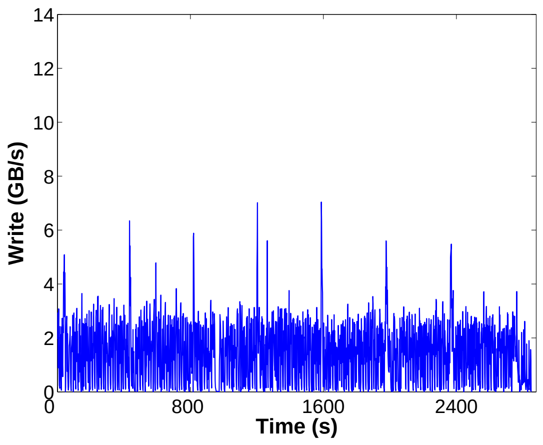
(a) $S3DS1$



(b) $S3DS2$



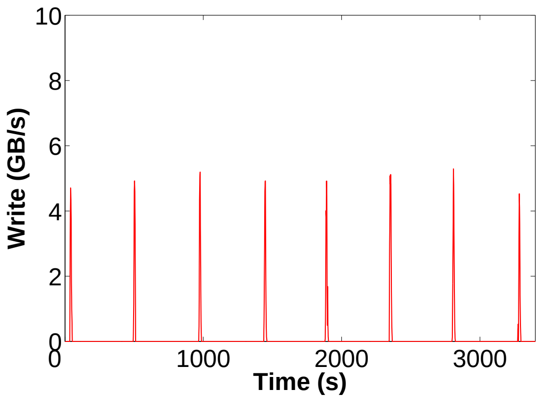
(c) $S3DS3$



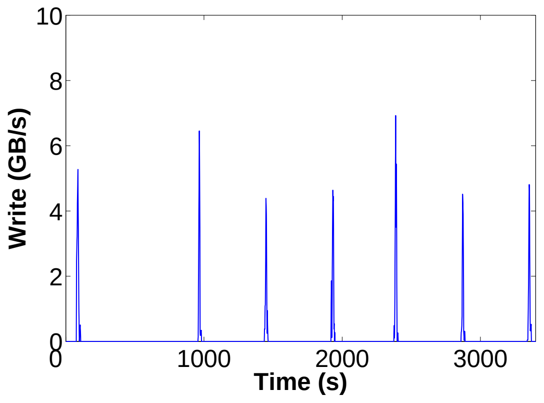
(d) $S3DS4$

Figure 4.16: S3D samples

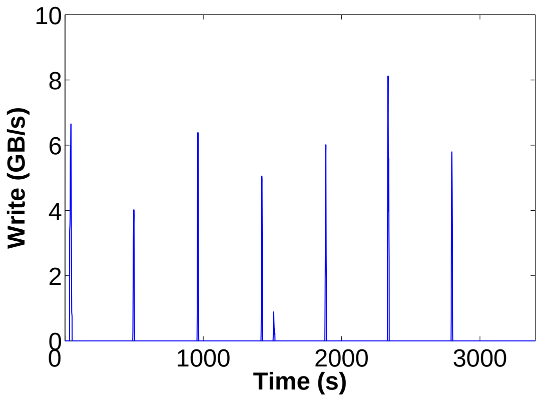
As in the case of IOR, IOSI with data preprocessing performs better than IOSI without data preprocessing and DTW. This result suggests that IOSI is able to the extract I/O signatures of real-world scientific applications from noisy throughput logs, collected from a very busy supercomputing center. While both versions of IOSI missed an I/O burst, the data preprocessing helps deliver better alignment accuracy (discussed in Figures 4.18a and 4.18b). The presence of heavy noise in the samples likely caused DTW's poor performance.



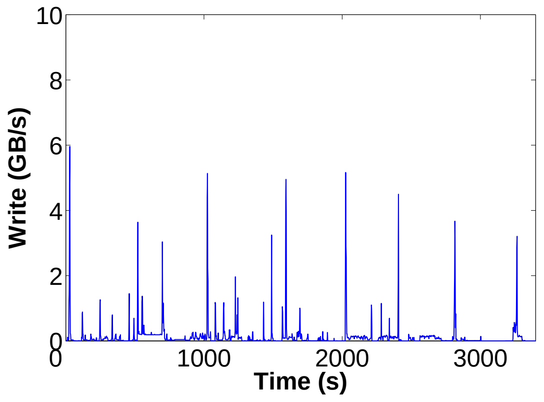
(a) S3D target I/O signature



(b) IOSI w/o data preprocessing



(c) IOSI with data preprocessing



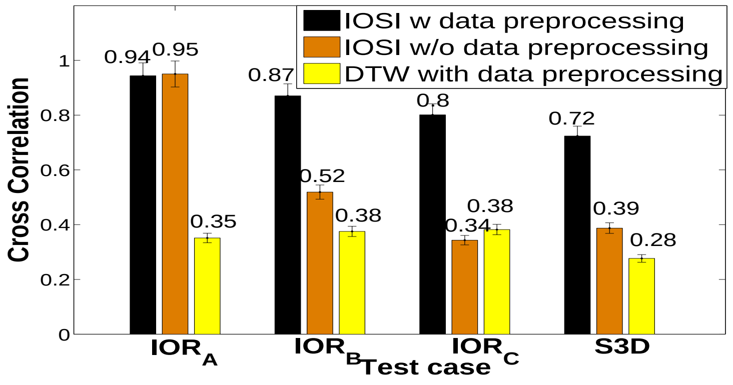
(d) DTW with data preprocessing

Figure 4.17: S3D target I/O signature and extracted I/O signature by IOSI and DTW

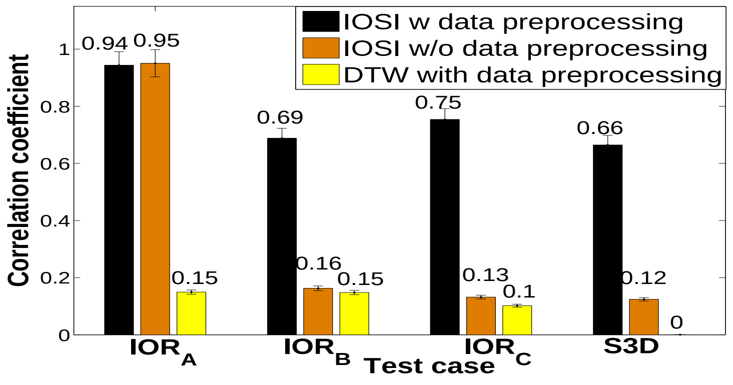
4.5.2 Accuracy and Efficiency Analysis

We quantitatively compare the accuracy of IOSI and DTW using two commonly used similarity metrics, cross correlation (Figure 4.18a) and correlation coefficient (Figure 4.18b). Correlation coefficient measures the strength of the linear relationship between two samples. Cross correlation [88] is a similarity measurement that factors in the drift in a time series data set. Figure 4.18 portraits these two metrics, as well as the total I/O volume comparison, between the extracted and the original application signature.

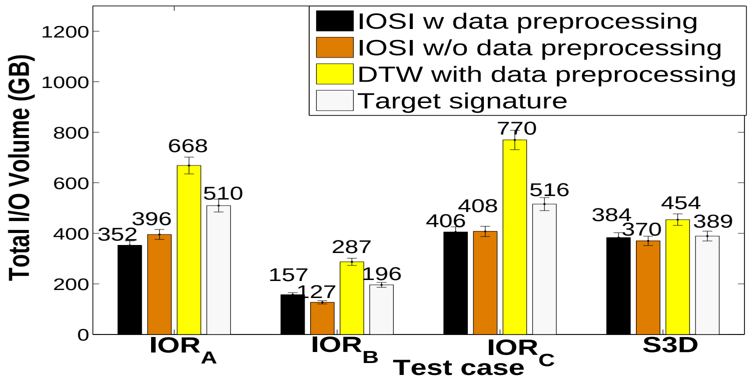
Note that correlation coefficient is inadequate to characterize the relationship between the two time series when they are not properly aligned. For example, with IOR_B, the number of bursts in the extracted signatures by IOSI with and without data preprocessing is very close. However, the one without preprocessing suffers more burst drift compared to the original



(a) Cross correlation



(b) Correlation coefficient



(c) Total I/O volume

Figure 4.18: I/O signature accuracy evaluation

signature. Cross correlation appears more tolerant to IOSI without preprocessing compared to correlation coefficient. Also, IOSI significantly outperforms DTW (both with preprocessing), by a factor of 2.1-2.6 in cross correlation, and 4.8-66.0 in correlation coefficient.

Note that the DTW correlation coefficient for S3D is too small to show. Overall, IOSI with preprocessing achieves a cross correlation between 0.72 and 0.95, and a correlation coefficient between 0.66 and 0.94.

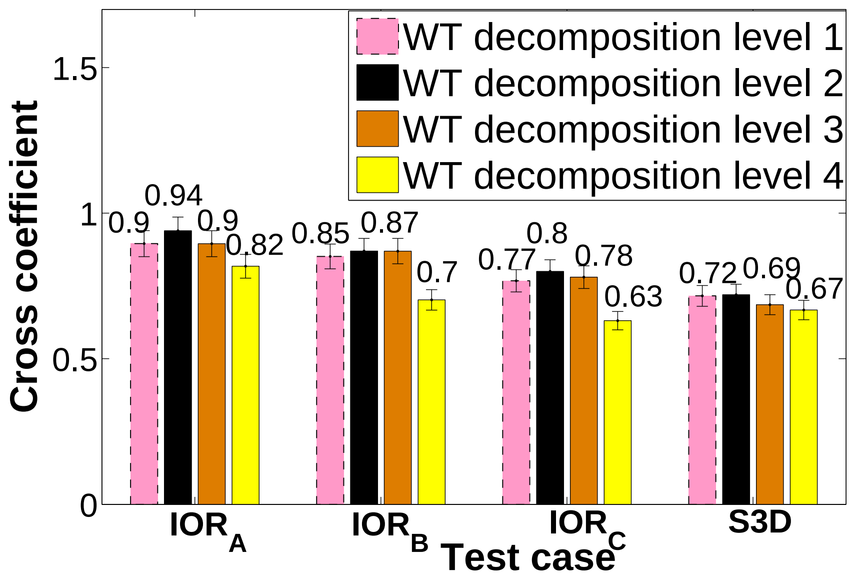
We also compared the total volume of I/O traffic (i.e., the "total area" below the signature curve), shown in Figure 4.18c. IOSI generates I/O signatures with a total I/O volume closer to the original signature than DTW does. It is interesting that without exception, IOSI and DTW err on the lower and higher side, respectively. The reason is that DTW tends to include foreign I/O bursts, while IOSI's WT process may "trim" the I/O bursts in its threshold-based burst boundary identification.

Next, we performed sensitivity analysis on the tunable parameters of IOSI, namely the *WT decomposition level*, and *density threshold/neighborhood density threshold in CLIQUE clustering*. As discussed in Chapter 4.4.2, we used a WT decomposition level of 2 in IOSI. In Figures 4.19a and 4.19b, we compare the impact of WT decomposition levels using both cross correlation and correlation coefficient. Figure 4.19a shows that IOSI does better with a decomposition level of 2, compared to levels 1, 3 and 4. Similarly, Figure 4.19b shows that the correlation coefficient is the best at the WT decomposition level of 2.

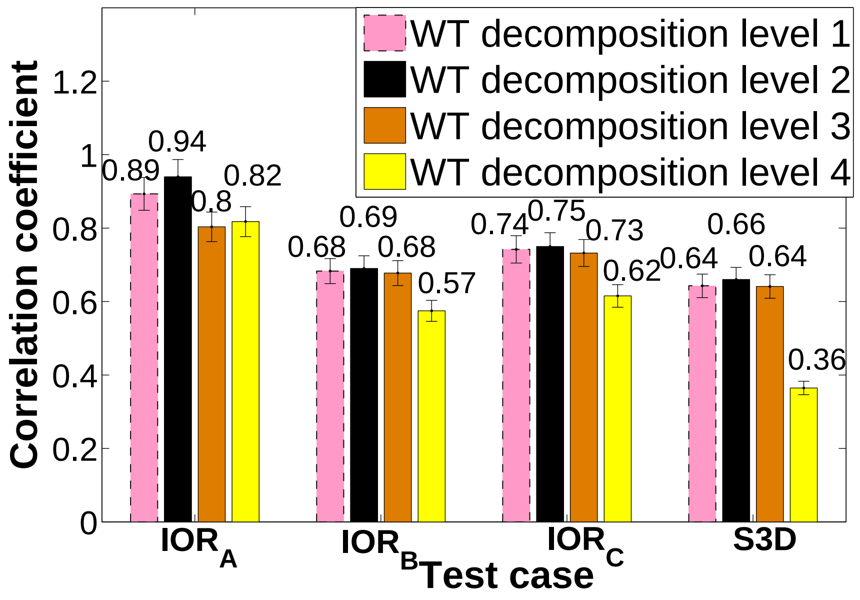
In Figure 4.20a, we tested IOSI with different density thresholds $\lceil \gamma * s \rceil$ in CLIQUE clustering, where γ is the controllable factor and s is the number of samples. In IOSI, the default γ value is 50%. From Figure 4.20a we noticed a peak correlation coefficient at γ value of around 50%. There is significant performance degradation at over 70%, as adjacent bursts may be grouped to form a single burst. In Figure 4.20b, we tested IOSI with different neighborhood density thresholds $\lceil \kappa * s \rceil$, where κ is another configurable factor with value larger than γ. IOSI used 90% as the default value of κ. Figure 4.20b suggests that lower thresholds perform poorly, as more neighboring data points deteriorates the quality of identified I/O bursts. With a threshold of 100%, we expect bursts from all samples to be closely aligned, which is impractical.

Finally, we analyze the processing time overhead of these methods. IOSI involves mainly two computation tasks: wavelet transform and CLIQUE clustering. The complexity of WT (discrete) is *O(n)* [37] and CLIQUE clustering is $O(C^k + nk)$ [40], where k is the highest dimensionality, n the number of input points, and C the number of clusters. In our CLIQUE implementation, k is set to 2 and C is also a small number. Therefore we assume C^k as a constant, resulting in a complexity of *O(n)*, leading to the overall linear complexity of IOSI. DTW, on the other hand, has a complexity of $O(mn)$ [43], where m and n are the lengths of the two input arrays.

Experimental results confirm the above analysis. In Figure 4.21a, we measure the processing time with different sample set sizes (each sample containing around 2000 data points). For IOSI,



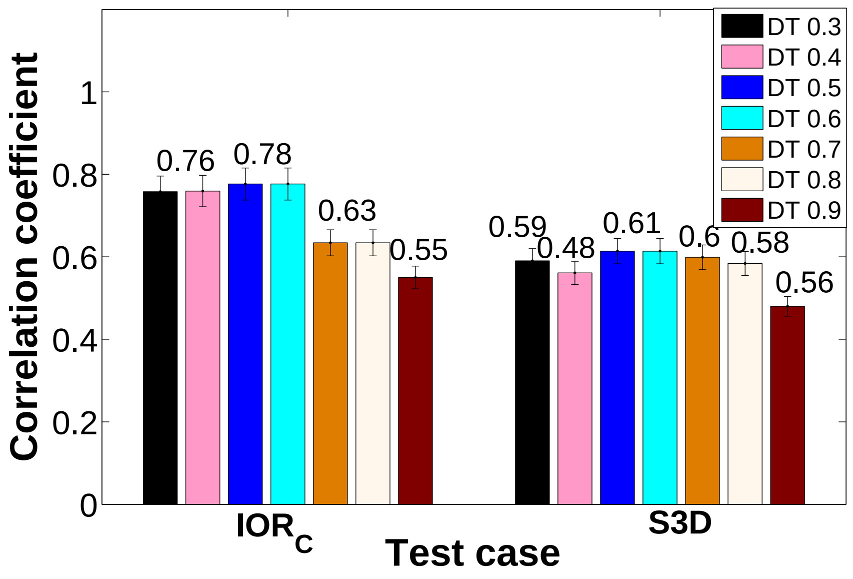
(a) Cross correlation



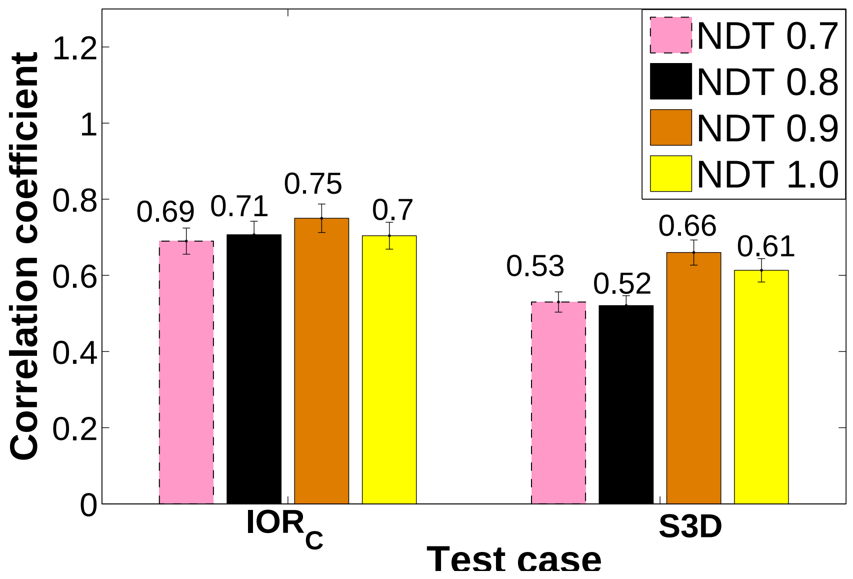
(b) Correlation coefficient

Figure 4.19: IOSI - WT sensitivity analysis

the processing time appears to stay flat as more samples are used. This is because the CLIQUE clustering time, which is rather independent of the number of samples and depends more on the



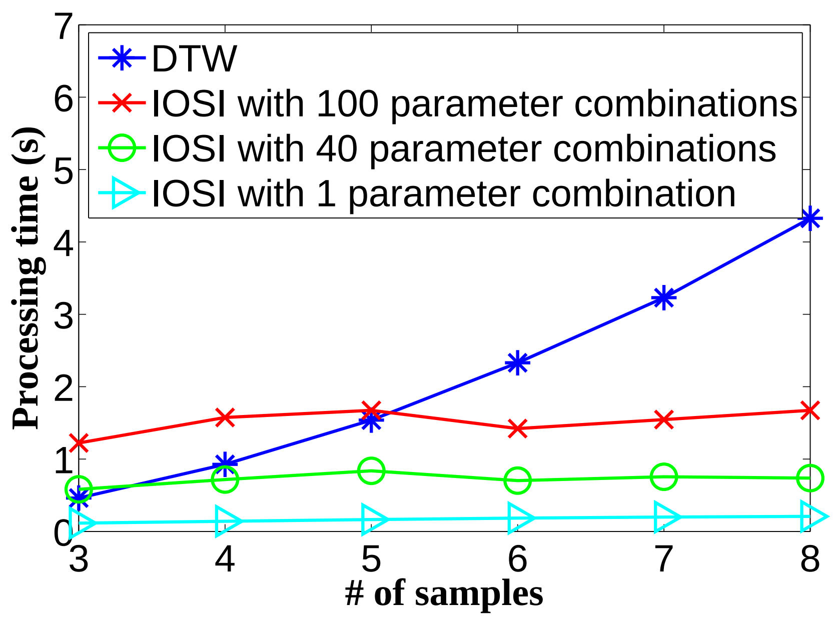
(a) Density threshold (DT)



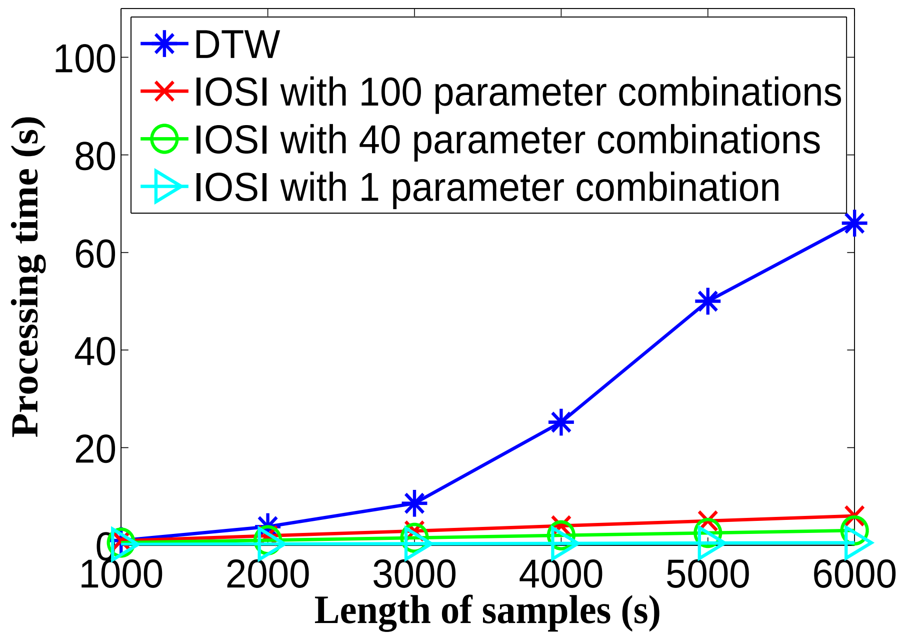
(b) Neighborhood density threshold (NDT)

Figure 4.20: IOSI - Clustering sensitivity analysis

number of grids, dominates IOSI's overhead. Even with 100 2-D IOSI parameter settings (for the CLIQUE grid size), DTW's cost catches up with 5 samples and grows much faster beyond this point. Figure 4.21b shows results with 8 samples, at varied sample lengths. We see that



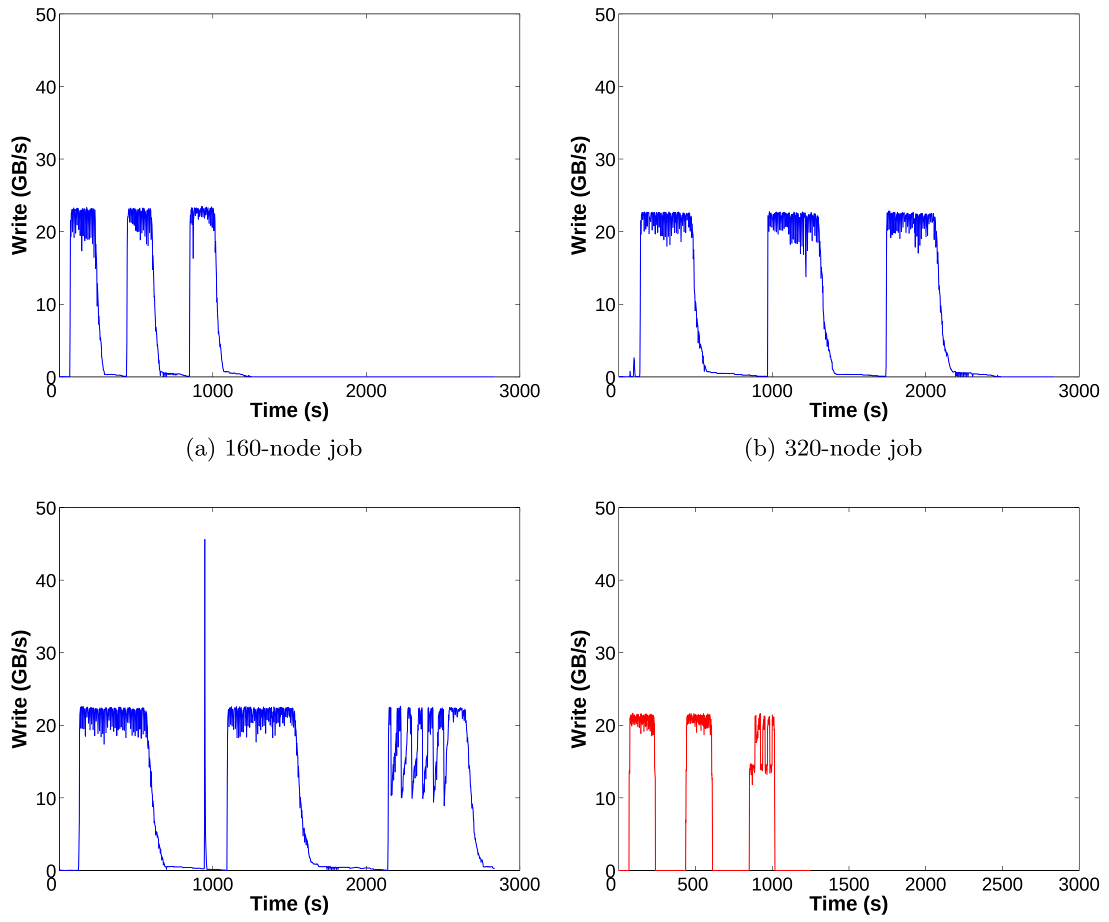
(a) Scalability in # of samples



(b) Scalability in sample duration

Figure 4.21: Processing time analysis

IOSI processing time increases linearly while DTW displays a much faster growth. To give an



(a) 160-node job (b) 320-node job

(c) 640-node job (d) IOSI Result

Figure 4.22: Weak scaling sample and IOSI extracted I/O signature

idea of its feasibility, IOSI finishes processing three months of Spider logs (containing 80,815 job entries) in 72 minutes.

4.6 Discussion

I/O signatures and application configurations Scientific users tend to have a pattern of I/O behavior. However, they do scale applications with respect to the number of nodes, resulting in similar I/O characteristics. In Figure 4.22, we show the I/O pattern of a real Titan user, running jobs with three different node counts (160, 320, and 640). From Figures 4.22a-4.22c, we observe that the total I/O volume increases linearly with the node count (weak scaling), but the peak bandwidth remains almost constant. As a result, the time spent on I/O also increases linearly. IOSI can discern such patterns and extract the I/O signature, as shown in Figure 4.22d. As described earlier, in the data preprocessing step we perform runtime correction and the samples are normalized to the sample with the shortest runtime. In this case, IOSI normalizes the data sets to that of the shortest job (i.e., the job with the smallest node count), and provides the I/O signature of the application for the smallest job size.

- **Identifying different user workloads** Our tests used a predominant scientific I/O pattern, where applications perform periodic I/O. However, as long as an application exhibits similar I/O behavior across multiple runs, the common I/O pattern can be captured by IOSI as its algorithms make no assumption on periodic behavior.

- **False-positives and missing bursts** False-positives are highly unlikely as it is very difficult to have highly correlated noise behavior across multiple samples. IOSI could miscalculate small-scale I/O bursts if they happen to be dominated by noise in most samples. Increasing the number of samples can help here.

4.7 Conclusion

We have presented IOSI, a zero-overhead scheme for automatically identifying the I/O signature of data-intensive parallel applications. IOSI utilizes *existing* throughput logs on the storage servers to identify the signature. It uses a suite of statistical techniques to extract the I/O signature from noisy throughput measurements. Our results show that an entirely *data-driven* approach, exploring existing monitoring and job scheduling history can extract substantial application behavior, potentially useful for resource management optimization. In particular, such information gathering does not require any developer effort or internal application knowledge. Such a black-box method may be even more appealing as systems/applications grow larger and more complex.

Chapter 5

Automatic I/O Diverter

In this Chapter, we present AID, a novel tool that performs automatic I/O-intensive application identification, as well as I/O-ware job scheduling. AID shares several common building blocks as IOSI. More specifically, the feasibility of automatic application I/O characterization is established on the *periodic* and *bursty* I/O pattern of HPC applications [83], with the same application executed *repetitively* through many batch jobs [21]. Meanwhile, AID takes a large step beyond IOSI, by (1) analyzing full job history mining with no prior information or given target applications, and (2) exploring I/O-aware scheduling. More details with be described in the following sections.

5.1 Problem Definition

AID carries two goals:

- To *automatically* identify I/O-intensive applications and their I/O access patterns.
- To explore I/O-aware job scheduling that staggers jobs running such I/O-intensive applications, based on the aforementioned, automatically identified I/O behavior.

While automatic I/O-intensive application identification and characterization can provide useful information and insight for supercomputer design/management, I/O-aware job scheduling is the ultimate goal for approaching the inter-job I/O interference problem that motivated this study. However, to deploy batch scheduler modifications on Titan, a busy production system, is beyond our reach. Meanwhile, there lacks mature parallel file system simulators and it is very hard to generate a realistic background workload mixture. Therefore, rather than developing an enhanced scheduler, we assess the potential benefit of our proposed I/O-aware scheduling by making simple recommendations (“now” or “later”) considering application I/O pattern and current system load. Validation is done by comparing the results of I/O-intensive job execution under different recommendations.

In summary, the input to our proposed system, AID, will be the batch job and server-side I/O traffic logs covering a common time period. AID mines the two logs jointly to identify a set of I/O-intensive applications. Such mining is done continuously and incrementally, with new log data appended daily. For each application labeled as I/O-intensive, AID further identifies its major I/O characteristics (to be detailed below). In job scheduling, the decision of whether to dispatch such applications is augmented by AID's I/O-aware analysis, which considers the automatically identified application I/O characteristics, plus the current system I/O load as additional input. The output of this analysis is a simple recommendation, in the form of "run" (to dispatch the job in question now) or "delay" (to hold the job in queue and re-examine at the next event-driven scheduling point, such as upon the completion of an active job).

5.2 Challenges and Overall Design

As we mentioned above, AID exploits several common features of scientific applications, as we used in IOSI: bursty, periodicity, repeatability.

The bursty behavior results in *I/O bursts*, a segment of time with elevated I/O activities logged at the server side [50], creating observable "I/O waves" that become the basic unit of per-application I/O traffic identification. The periodic behavior establishes a consistent "pattern" which facilitates the attribution of I/O traffic to specific applications. The repetitive behavior allows pattern identification by further correlating multiple *samples* (segments of I/O traffic log intercepted by the job start/end times of the target application), identifying the commonality across samples as application-affiliated "signal" and difference as "noise".

Our main focus here, however, is to find out "suspected I/O-intensive applications", a fraction of the thousands of unique parallel applications executed by millions of batch jobs, without knowing any I/O-related information about them. The IOSI work, in contrast, has I/O-intensive applications *pre-identified* and starts from samples that are from guaranteed identical runs (same executable and input). In examining the 5-month I/O and job logs, we have found such automatic classification quite challenging for several reasons, as explained below.

First, we cannot assume that jobs running the same application are identical. In fact, we have found them to often have *variable duration*, sometimes with significant execution time differences. Jobs are commonly configured with the total number of computation timesteps (specified in batch scripts). Also, large applications in development often have short debug runs. E.g., we observed 20 runs during 11/2014 - 01/2015 of one application (later identified as I/O-intensive), with execution time ranging from 1958 to 86644 seconds. While I/O-intensive applications do tend to possess common periodic I/O patterns, large time variance makes sample alignment and I/O burst identification harder, especially without apriori knowledge on an application's I/O intensity.

Second, in some cases application runs do contain *inconsistent I/O pattern*. As periodic I/O (writing intermediate results and/or checkpoint files) itself is a controllable operation, non-production runs checking algorithmic correctness or tuning performance often turn off such output as a whole or reduce the I/O frequency (typically specified in the form of once every n compute steps).

While there are applications who seldom change such configurations from run to run, the existence of I/O pattern change is more challenging than the execution time variance and further complicates our classification.

Considering these challenges, we decide to perform our proof-of-concept study by focusing on applications with heavy I/O demand, which compose a small fraction of HPC applications. A very recent study carried out at the Argonne Leadership Computing facility (ALCF) using the Darshan parallel I/O profiling tool [55] observed that the aggregate throughput for three-quarters of applications never exceeded 1GB/s, roughly 1% of the peak platform bandwidth. This application-side, white-box investigation is consistent with our server-side, black-box observation.

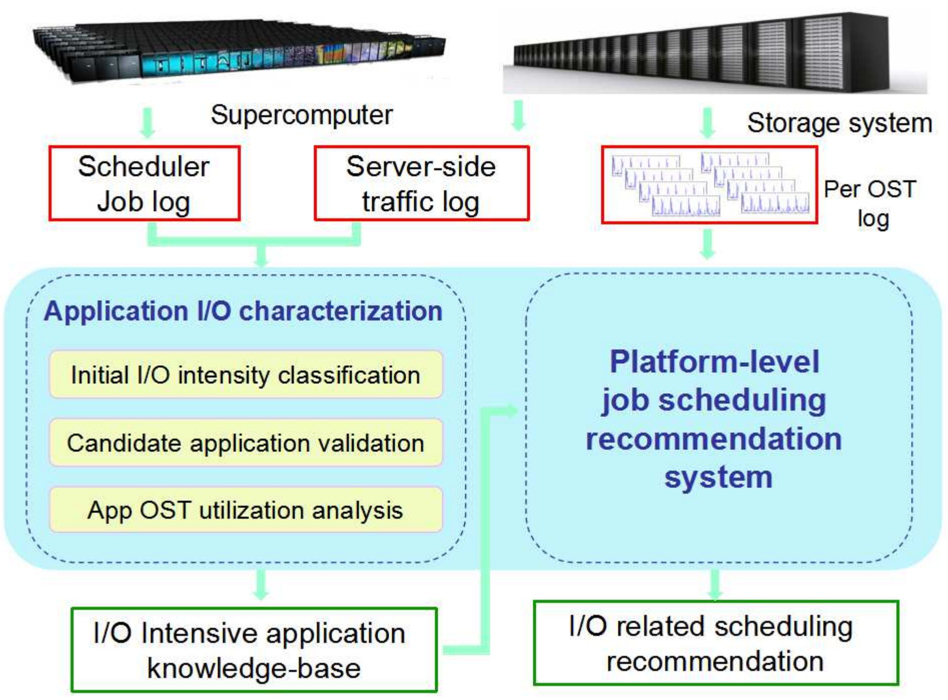

Supercomputer
Storage system
Scheduler Job log
Server-side traffic log
Per OST log
Application I/O characterization
Initial I/O intensity classification
Candidate application validation
App OST utilization analysis
Platform-level job scheduling recommendation system
I/O Intensive application knowledge-base
I/O related scheduling recommendation

Figure 5.1: AID software architecture

Figure 5.1 outlines AID's structure, which comprises two major components: 1) an offline application I/O characterization engine that incrementally processes the I/O traffic logs and batch job logs, saving analysis results in an application knowledge base, and 2) an online I/O-aware scheduler that queries the knowledge base and real-time system load levels to make scheduling recommendations for identified I/O-intensive applications. It utilizes the more recently available per-OST I/O traffic log to analyze the number of OSTs used simultaneously by the application, which turns out to be valuable in both I/O pattern refinement and I/O-aware scheduling. We will discuss the steps involved in these two components in the next two sections, respectively.

5.3 Application I/O Characterization

AID builds its knowledge from scratch and has to rely solely on its two input datasets: the job log and the I/O traffic log (typically over months long to bootstrap the knowledge base). The I/O traffic log comes in at per-OST granularity, which provides valuable additional insight to split I/O activities from concurrent jobs. There is little return, however, for the huge time required to examine the full combination of thousands of OSTs and millions of jobs without knowing *anything* about *any* of the applications.

Besides high cost, there are other complexities involved in analyzing per-OST traffic. Many applications create and write new files during each I/O burst (batch of I/O operations clustered temporally). E.g., an application may aggregate its output from 2048 processes to 64 shared files every 1000 computation timesteps. Though the *OST footprint* (number of OSTs used during each I/O burst) is stable, each wave of files are assigned and striped to their destination OSTs at file creation time. Therefore, the subset of OSTs showing I/O traffic from this application *migrates* between I/O bursts, further perplexing I/O pattern identification.

Considering the above, AID takes a two-phase strategy, to first look at the aggregate traffic log and identify a set of applications suspected to be I/O-intensive, which we call *I/O-intensive application candidates* (*candidates* in short for the rest of the paper). The intuition is that if an application is I/O-intensive enough (having recognizable I/O bursts) and "important" enough (running long or frequently). The combination of these two traits will allow us to have sufficient confidence to mark it as a candidate. Most I/O-light applications (or those I/O-intensive but infrequent or having short execution time), on the other hand, are likely to be ruled out: the chance is quite low for such an application to be lucky enough to have a significant portion of their samples piggy-backed on other I/O-intensive applications (which happen to create consistent I/O patterns). Therefore, instead of setting an arbitrary quantitative standard for an application, which varies with different system configurations and load levels, here "being I/O-intensive" is defined as "having consistent, recognizable, and periodic I/O pattern".

With the short-listed candidates, the second phase will take a much more thorough look by zooming in to the per-OST traffic log, to discover their detailed I/O characteristics. This process serves two-fold purposes to *validate* the I/O intensity of the candidates and for those validated, to collect I/O patterns relevant to subsequent I/O-aware scheduling. More specifically, AID collects the application's aggregate I/O volume per I/O burst and the I/O interval (average time between two adjacent I/O-bursts).

In addition, assisted by the per-OST traffic analysis, AID derives an applications' OST footprint, the number of OSTs it tends to use. Most users leave the file striping width at the Lustre default value (4 at Spider), therefore, the OST footprint depends mostly on how many files are written simultaneously. With n compute nodes, typical I/O-intensive applications may use independent I/O to write n files (n-to-n model), or collective I/O to write one (n-to-1 model) or m (n-to-m) files [12, 21]. Finding the OST footprint also serves two-fold purposes. First, it helps the I/O-aware scheduler to know how many OSTs an I/O-intensive application uses, and assess the chance of two such applications stepping on each other's toes (though there is currently no way for the scheduler to force an application to use a certain group of OSTs). Second, pinpointing the subgroup of OSTs an application used allows our I/O characterization process to refine the I/O patterns collected, as I/O traffic from OSTs considered unused by this application can now be excluded.

5.3.1 Initial I/O Intensity Classification

From jobs to applications: I/O characteristics belong to *applications*, but are observed through *jobs*, each a particular execution of an application. The number of unique applications is typically much smaller than the number of jobs run per year. Meanwhile, the same application run by different users can exhibit different I/O behavior. For example, within one national laboratory group developing a astrophysics code, different members may each run the application many times (issuing many jobs): a domain scientist mostly conducts long runs without I/O to study convergence and stability, plus production runs to collect simulation results, an I/O expert conducts shorter runs with higher-than-normal I/O frequency to study I/O performance and scalability, while a software engineer conducts short debugging runs with normal I/O frequency to test system integration and overall scalability. On the other hand, we observed that it is quite rare for a single user to incur very different I/O patterns running the same application with the same *node count* (the number of compute nodes used by a job).

Therefore, we decide to define a *unique application* in the context of AID with the 3-tuple $\langle$*user name*, *job name*, *node count*$\rangle$. Here "job name" is a user-assigned string identifier included in the job script. This definition allows us to obtain 9998 unique applications from the 5-month log containing 181,969 jobs, resulting in 18.2 job runs per application on average during this period.

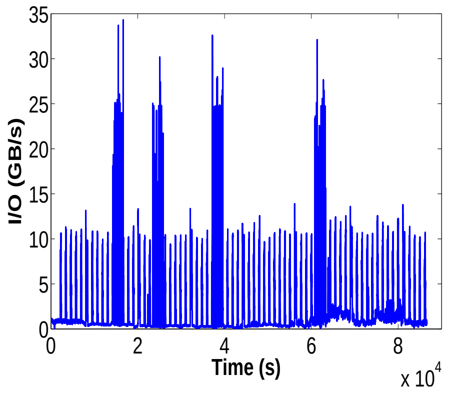
Candidate selection: Now we need to examine the aggregate I/O traffic log to preliminarily nominate I/O-intensive application candidates. This is done by processing the samples of each application (obtained by intercepting the aggregate traffic log using the start/end times of its jobs), in search for consistent and significant I/O activities. Here we exploit several common techniques as used in IOSI, such as Wavelet Transform (WT) to identify I/O bursts in each individual sample, and density-based clustering to identify common I/O bursts across samples. However, for IOSI, the given target application is *known* to be I/O intensive, and samples of this application are *guaranteed* to be from identical executions. This enables IOSI to look for periodic I/O bursts expected to exist in the samples (only blurred by background noises at times).

In contrast, AID has to deal with the irregularity and inconsistency involved in classifying *a large number of unknown real applications* whose job runs may possess variable I/O behavior. The bursts found in each sample could belong to any of the concurrently running applications or even interactive user commands and we have no knowledge to assume any application to possess I/O dominance in these samples. To solve this problem, we adopt a different clustering technique, which transforms each identified I/O burst for an application (all its samples combined) to a point in a 2-D space, using the burst height (peak aggregate I/O throughput) and area (total I/O volume) as x and y coordinate, respectively.

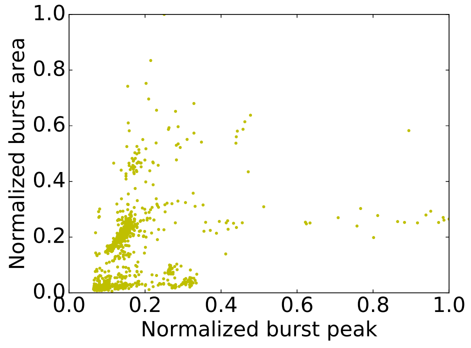
We then perform clustering of these points, aiming to identify groups of highly similar I/O bursts (potentially periodic I/O operations). We experimented with multiple widely-used clustering algorithms, including K-means [56] and DBSCAN [32], and finally selected OPTICS [10], which is very robust with noisy data and does not make any assumption on the number of result clusters or their shapes.

Figure 5.2a plots one original sample of a real application (with job named "scaling" by user). With 5 such samples of "scaling", AID identifies a total of 1070 I/O bursts, as shown in Figure 5.2b. Figure 5.2c further displays the "zoomed-in" area with 4 result clusters identified by OPTICS.

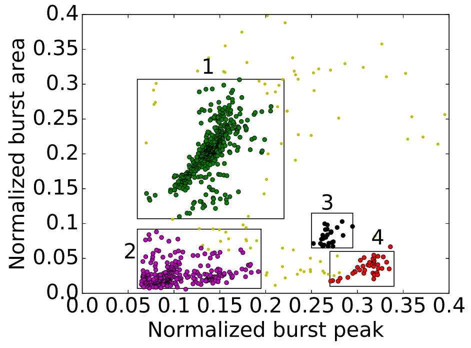
Based on the clustering results, we split the original sample into n *sub-samples*, each containing the I/O bursts from one of n identified clusters. To give an example, Figure 5.3 displays 4 sub-samples from the original sample shown in Figure 5.2a, based on the clustering result in Figure 5.2c and plotted using corresponding colors. Our subsequent processing is based on the intuition that if a group of I/O bursts belongs to the application in question, such I/O bursts need to (1) possess regular periodic pattern, and (2) appear in a significant portion of the original samples. By examining aggregate statistics such as I/O interval (time distance between adjacent I/O bursts), peak I/O throughput, and total I/O volume per-burst, we avoid the "sample alignment" and "full signature mining" tasks based on strong assumptions on the existence and consistency of application I/O pattern. With attention focused on burst "size", "height",



(a) Sample of the application 'scaling'



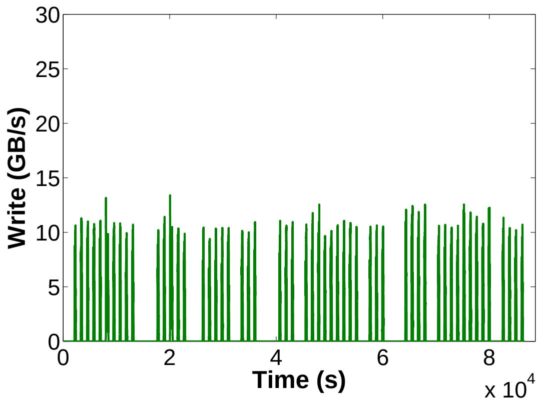
(b) Mapping 'scaling' samples



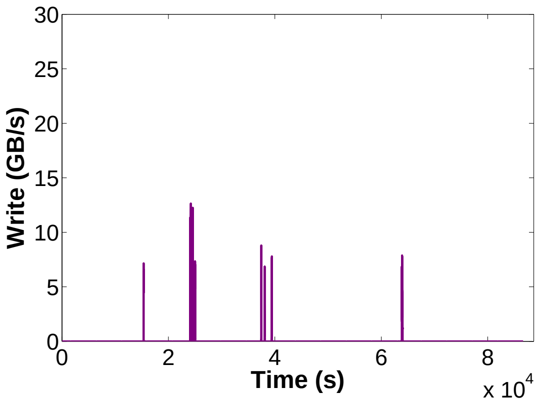
(c) Clustering result of 'scaling' samples

Figure 5.2: Example of OPTICS clustering

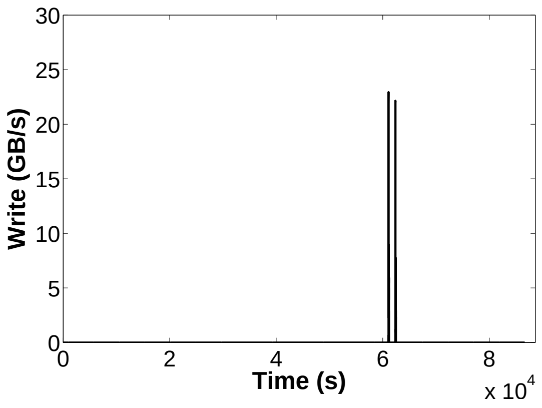
and "frequency", AID can handle variable execution lengths. Similarly, by requiring only a fraction of samples possessing common pattern (using a configurable threshold, which is set to 60% in our evaluation), combined with our strategy to differentiate the same base application's



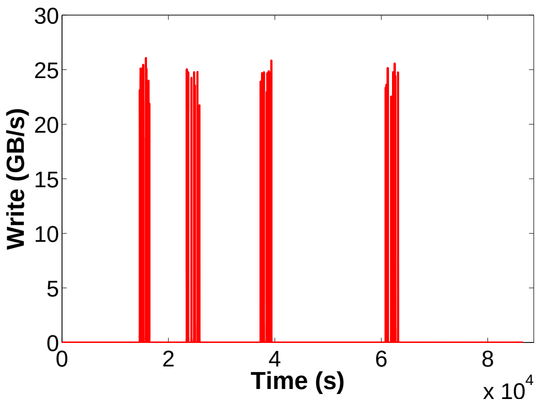
(a) Subsample of cluster 1



(b) Subsample of cluster 2



(c) Subsample of cluster 3



(d) Subsample of cluster 4

Figure 5.3: Restructured sample based on clustering result

executions submitted by different users, AID can handle inconsistent I/O patterns.

Figure 5.3 displays the I/O bursts in the 4 sub-samples for the aforementioned application, from which only one (cluster 1) is identified as a valid I/O pattern, as it is found to be regular, spanning a significant portion of the job execution, and appear across over 60% of sub-samples. The other clusters fail to meet these requirements. As mentioned earlier, applications with such identifiable I/O pattern (at least one verified I/O burst cluster) are preliminarily considered an I/O-intensive application candidate.

5.3.2 Candidate Application Validation

To reduce false positives, AID applies two validation techniques to all candidates, as discussed below.

Scope checking This is to guard against the case where there are long and frequent executions of a true I/O-intensive application, whose samples entirely "cover" certain other applications' shorter samples often enough. In this case, those other applications will have samples bearing exactly the same I/O pattern. The solution is rather intuitive: for each qualifying sample, we look beyond its boundary, left and right, to check the correlation between the I/O pattern's existence and the application's execution. If a detected pattern is indeed incurred by a certain application, it should not be consistently observed before/after its job starts/completes running. Our AID prototype performs this checking by examining 5 times of the detected I/O interval length, each way beyond the sample boundary.

Minimum support requirement Still, there could be relatively rare cases where a candidate happens to piggy-back on true I/O-intensive applications with similar job start and end times. This is more likely to happen when the false positive only has few samples. As mentioned earlier, AID is designed to be a self-learning system, scheduled to run at least daily to incrementally process new samples. Therefore, it maintains a separate watch list for "under-probation" candidates, who need to be validated with more samples. After bootstrapping the knowledge base, all new applications need to go through this probation period. Currently our AID prototype requires 5 minimum samples to have a candidate validated, which, together with the aforementioned scope checking, significantly reduces the chance of a false positive being admitted.

5.3.3 Application OST Utilization Analysis

Now with I/O-intensive applications identified and validated, AID performs another round of more detailed analysis. Note that the "I/O pattern" itself is exactly what we rely on to perform such application classification and at this point, we already have the majority of I/O pattern parameters. Namely, AID knows the total I/O volume per periodic I/O burst, the peak/average I/O throughput during I/O bursts, the I/O burst interval, and the computation-to-I/O time ratio. These parameters describe the *temporal distribution* of an application's I/O traffic and can be obtained by analyzing the aggregate server-side traffic log.

To understand an application's I/O behavior from the *spatial* dimension, AID mines its OST footprint, the number of OSTs it accesses during an I/O burst, as the final parameter of its I/O pattern. Collecting and analyzing OST-level traffic logs is time consuming, as each sample (called *aggregate sample* from now on) becomes 1008 *OST samples*, due to the lack of information on which OSTs were mapped to a job. Fortunately, we have dramatically reduced our scope of examination by identifying I/O-intensive candidates from thousands of applications.

OST footprint identification Our log analysis finds real-world applications consistent in OST footprint across I/O bursts. Note that this observation applies across both read and write operations, as it is unusual for developers to switch between different file access models across non-trivial I/O operations. Therefore, AID assumes a constant OST footprint, κ, for each application. Each individual burst in an aggregate sample then comprises κ OST bursts ($0 < \kappa \leq 1008$). For example, if an application writes collectively a single global shared file (n-to-1 model), with a file stripe width of 16, $\kappa = 16$. If the average OST throughput is 200 MB/s (with no other concurrent activities), the aggregated sample will contain I/O bursts with I/O throughput of around 3.2 GB/s.

After preparing OST samples for each aggregated sample, AID reuses the OPTICS clustering results (described in Chapter 5.3.1). These aggregate bursts have already been certified as "regular and consistent" in the previous steps, hence giving strong hints for the search of similar-shaped bursts on individual OSTs. Suppose we have n aggregate samples from our target application, each bearing m aggregated bursts. For each aggregate burst B_i in the aggregate sample S_j, AID scans all 1008 *OST samples* and counts $o_{i,j}$ corresponding OST bursts. Then the application's OST footprint κ is calculated as

$$\kappa = (\sum_{j=1}^{m} \sum_{i=1}^{n} o_{i,j})/mn \tag{5.1}$$

We have evaluated this simple OST footprint calculation with both real applications and pseudo-applications, and found it to be effective (results in Chapter 5.5.2).

OST-footprint-enabled I/O pattern refinement Finally, we perform another round of fine tuning of the application I/O pattern, by taking into account the AID-identified OST footprint. With this additional piece of information, we can pinpoint the subset of OSTs suspected to have participated in each I/O burst. This allows us to exclude I/O traffic from other OSTs from the aggregate sample, identified as background I/O noise from the target application's point of view. Though not affecting parameters such as I/O interval, such refinement improves the accuracy of per-burst I/O volume and average I/O throughput, as demonstrated by our experimental results in Fig. 5.6.

5.4 I/O-aware Job Scheduling

Based on the application I/O intensity classification and I/O pattern characterization results, AID explores I/O-aware scheduling. As discussed earlier, such proof-of-concept prototype cannot be test-deployed on production supercomputers (especially state-of-the-art machines). Simulation study, on the other hand, cannot reflect the real dynamics of the mixed workload and shared resources of such large-scale systems. AID hence starts with a best-effort *recommendation system*, where it give simple "run" or "delay" recommendations based on its self-learned knowledge *and* real-time system load information. We can then evaluate the validity and potential impact of such recommendations, by comparing what happens with dispatching the application run anyway, according to or despite AID's suggestion.

Another path is *spatial partitioning*, splitting OSTs across applications when possible, given that AID is shown to be able to estimate a candidate's OST footprint reasonably well. This can potentially be applied independently or jointly with temporal staggering of the jobs. However, its deployment requires significant modification of both the scheduler and the parallel file system, making it even harder for us to verify on production platforms. That said, the current AID does take into account the estimated per-application OST footprint in making scheduling recommendations.

5.4.1 Summarizing I/O-Related Information

First, AID needs to establish a global view of I/O requirements of ongoing applications (already running) as well as the real-time storage system status showing how busy individual OSTs are. More specifically, it gathers information from two aspects:

Current application I/O requirement Using its application knowledge base, AID can easily identify I/O-intensive application candidates running or waiting on a supercomputer. It can further query the knowledge base to retrieve each candidate's I/O characteristics, to estimate the resource requirement for upcoming jobs, as well as to estimate the remaining execution time of jobs already running. One thing to keep in mind though is that despite the effectiveness of its I/O workload auto-characterization (results in the next section), AID is a *best-effort* system that does not possess sufficient ground truth on its classification and I/O pattern mining results. Nor can it *guarantee* that the future will repeat the history.

Real-time system I/O load level At the same time, AID continuously monitors the current I/O system load level by maintaining an OST-level *I/O load table (IOLT)*, which is regularly updated by querying the same server-side per-OST traffic log data. In our prototype, the frequency of IOLT update is set at 5 minutes. The IOLT summarizes system I/O load by maintaining a load level histogram for each OST during 4 of the most recent 5-minute windows, in terms of its logged I/O throughput. In other words, it keeps track of a 20-minute sliding window showing detailed recent history of per-OST load level. AID divides the per-OST throughput range $[0, T_p]$ (T_p being the peak throughput) into uniform throughput level bands (with width of $30MB/s$ in our prototype). For each such band (*e.g.*, $[60MB/s, 90MB/s]$) per 5-minute window, AID stores in IOLT the fraction of time that the OST in question is logged to have load level within this interval, say 20%.

The real-time system load information allows AID to supplement its knowledge base with actual run-time system status, making it aware of both the "demand" and "supply" sides of the shared I/O resources. Also, the real-time system load data compensate AID's lack of knowledge on newer applications that do not have (sufficient) samples or ad-hoc user/administrator interactive I/O activities that do not go through batch scheduling.

5.4.2 I/O-aware Scheduling

Finally, AID puts together everything it knows to make I/O-aware scheduling recommendation: whether an upcoming job of a known I/O-intensive application should be dispatched now ("run") or later ("delay"). AID does this by calculating a numeric *OST load score* ("*score*" for short), and making job admission decisions by considering the estimated OST-footprint and I/O traffic of the target application A (whose job is examined for scheduling recommendation). Below we describe the major steps involved in the decision making process.
OST load score calculation This step takes the continuously updated IOLT as input and calculate for each OST the load score s:

$$s = \sum_{i=1}^{n} w_i(f_i + \alpha) \tag{5.2}$$

Here n is the total number of the aforementioned per-OST throughput level bands and f_i is the fraction of time this OST stayed within the ith throughput band, based on the recent history from the IOLT (20 minutes in our implementation). The purpose here is to roughly measure the *chance* and *degree* that A is expected to endure I/O contention with its immediate dispatch. This also explains our choice to consider the "time-at-throughput-level distribution", rather than simply relying on the average/peak/minimum throughput, as this takes into account both the frequency and intensity of existing I/O activities. On top of f_i, we make an additional adjustment, α, leveraging the per-application I/O pattern mined by AID, for ongoing applications that has just started or is about to complete (based on application job history and the maximum execution time submitted to the scheduler and available at real time to AID). Basically, we use their I/O pattern in the knowledge base to add/subtract I/O throughput intervals for newborn/dying jobs. Since we do not have the mapping from application to a particular set of OSTs, we make such adjustment at the top K idle/busy OSTs, where K is the estimated OST footprint of the newborn/dying application.

However, with complex resource sharing behaviors, and without detailed ground truth on the I/O bursts' overlap or the capability to force accurate temporal inter-job placement, we cannot fully understand/predict the impact that the I/O load at each throughput band brings to A's execution on the same OST. To this end, in Equation 5.2 we add a weight, w_i, to the corresponding band. The weight values are in turn to be learned in a black-box style by real-system I/O interference measurement, collected in a 2-month period on Titan, during which we submitted small training jobs with known I/O patterns, to measure their I/O performance behavior under different OST load levels. More specifically, we simply used the I/O time of a training job normalized to the measured shortest time in all trials as s in Equation 5.2, making w the only unknown. With a large number of such training data points, we solve the w values using an over-determined system [11].

Application-specific load threshold calculation Taking a similar approach as in weight calculation above, AID also observes the impact of overall OST load (in terms of average s score over all OSTs used) on a target application by measuring such correlation between system load and training job's I/O performance. Here it maintains a 2-D data structure, partitioning the per-application I/O pattern into coarse intervals using two parameter values: the average I/O throughput per-burst, and the I/O-to-computation time ratio. Each "cell" in this 2-D table saves training data points within the corresponding parameter range, recording the measured average OST load upon dispatch (again mined from history logs) and normalized I/O performance. Therefore, given a known I/O-intensive application A and its I/O pattern retrieved from the AID knowledge base, plus a configurable performance impact factor (*e.g.*, a factor of 1.2 means that 20% longer I/O time can be accepted), we can utilize the pre-computed correlation and derive the threshold average OST load level L.

I/O-pattern-aware OST screening With the per-OST load score s calculated based on recent load history, and the average OST load level L calculated based on A's known I/O pattern, AID checks whether there are enough OSTs with projected I/O load low enough to accommodate A now.

This is done by obtaining A's OST footprint m from the knowledge base, and examine the m OSTs with the lowest load (by s value) in each file system partition. If the average load of such "least busy" OSTs is under A's application-dependent load threshold L, then AID makes the "run" recommendation, encouraging the immediate execution of A. Otherwise, it makes the "delay" recommendation, advising the scheduler to hold A in the batch queue until next (event-prompted) evaluation point.

5.5 Experimental Evaluation

In our experimental evaluation, we focus on verifying several main hypotheses:

1. AID can successfully identify I/O-intensive applications without apriori information.

2. AID can identify I/O-intensive applications' OST footprint with reasonable accuracy.

3. I/O-aware job scheduling based on automatically derived per-job I/O behavior can effectively reduce I/O contention and application I/O performance variance.

Note that AID analyzes real application data, but we have no way to obtain "ground truth" for the majority of user jobs on Titan, which is open to HPC users nationwide. Therefore, we also generated our own pseudo-applications, again with IOR (more details in Chapter 5.5.2). These applications possess typical real I/O patterns observed on Spider, and have been submitted repeatedly during past weeks.

5.5.1 I/O Intensity Classification

Table 5.1: Real application classification results

Name	Value
Total number of logged jobs	181,969
Unique real applications	9,998
Initial I/O-intensive candidates	95
Candidates passing scope checking	67
Candidates passing min support requirement	42
User-verified candidates	8

We implemented a proof-of-concept prototype of AID, in around 3200 lines of Python code. Our evaluation mainly focuses on checking against false positives, as our "I/O-intensive" definition requires "observable" I/O patterns and AID is rather confident when an application cannot even make the under-probation watchlist.

Real applications We fed AID with the aforementioned 5-month Titan I/O traffic and job logs. Table 5.1 summarizes major statistics information regarding logged jobs. Due to space limit, we omit the full CDF plots of all jobs' node count (how many compute nodes are used) and execution time. In summary, a large fraction of jobs have small scales, while a small fraction of jobs are large: the 25th, 75th, 95th, and 99th percentile node counts are 2, 80, 1024 and 8192, respectively. There are similar distributions in execution time, with 25th, 75th, 95th, and 99th values being 1, 70, 180 and 380 minutes.

AID obtained 95 preliminary I/O-intensive candidates, and with its own validation using scope checking and minimum support requirement it cut the shortlist to 42 candidates. We hoped to verify the findings with feedback from the application owners, however contacting Titan users has to comply with ORNL policy and is non-trivial. In the end, we obtained approval to contact 16 candidates, mostly submitted by local users. We contacted these users by email and received response from 8 of them, all positive confirmation of the I/O-intensive classification. Table 5.2 gives basic characteristics of these user-verified candidates.

Pseudo-applications First, Table 5.3 lists the characteristics of our own IOR pseudo-applications, whose trial runs (submitted at diverse times of the day) consumed around 834,682 node-hours on

Table 5.2: User-verified I/O-intensive applications

ID	Node	Time(m)	OST	App. Domain
1	8192	1440	64	Geo-sciences
2	250	6-60	1008	Combustion
3	2048	30-185	1008	Astrophysics
4	1760	720	180	Combustion
5	1024	110-230	1008	Systems research
6	200	30-190	1008	Combustion
7	1008	13-17	1008	Computer Science
8	16388	43-310	800	Environmental

Table 5.3: Pseudo-application configurations

ID	# Nodes	I/O interval	# iter.	Burst vol.
IOR_A	256	300s	6-8	1024GB
IOR_B	512	250s	6-10	2048GB
IOR_C	128	450s	4-6	1024GB
IOR_E	1024	600s	4-5	2048GB
IOR_D	128	100s	10-20	1024GB
IOR_F	128	80s	15-25	1024GB
IOR_G	256	600s	12-20	2048GB
IOR_H	64	500s	4-6	64GB

Titan during the past two months, We specifically varied the number of iterations (computation phase plus I/O phase), to evaluate AID's capability of handling variable-length executions of the same application. Also, we configured IOR_H with smaller output size per compute node and the *n-to-1* model to generate rather low I/O throughput. As expected, AID correctly identified all 7 I/O-intensive pseudo-applications and did not include IOR_H in the candidate list.

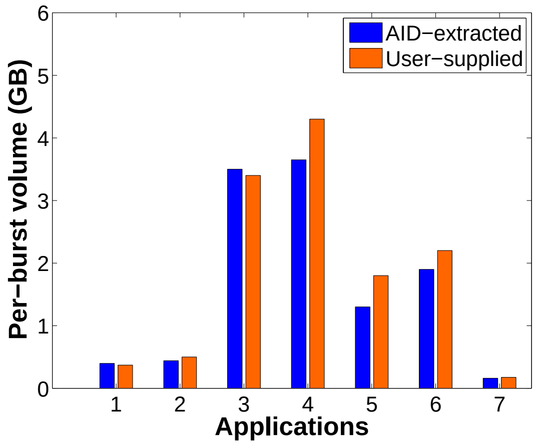
5.5.2 I/O Pattern Identification

Table 5.4: Application 8, AID vs. ground truth

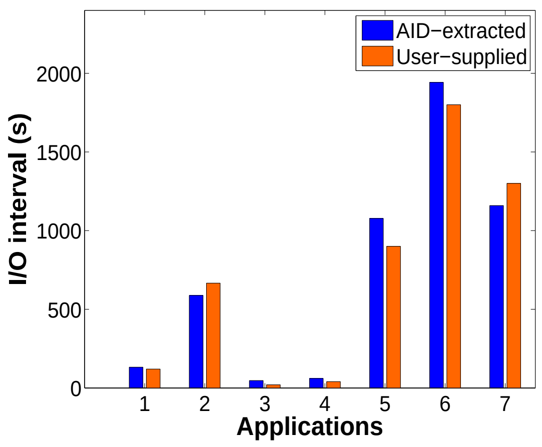
	Burst volume	Interval	Throughput	# OST
AID	34TB	350s	150GB/s	960
Actual	32.4TB	380s	184GB/s	963

Real applications We evaluate AID's capability of mining detailed I/O characteristics using the 8 aforementioned verified real-world applications. Table 5.2 confirms our finding that I/O-intensive applications are indeed run in diverse scales and lengths (producing large ranges of node count and execution time distribution).

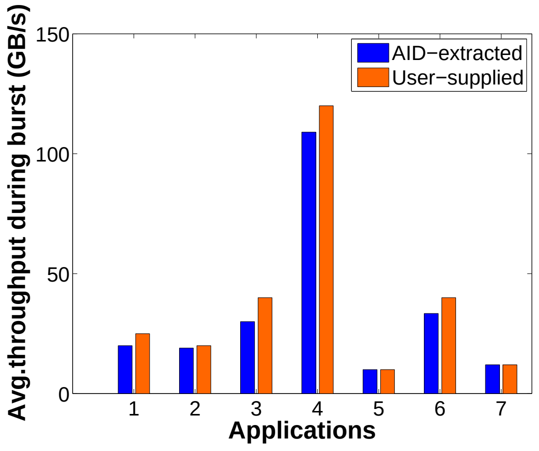
In addition to confirming I/O intensity, their users kindly filled our email questionnaire on



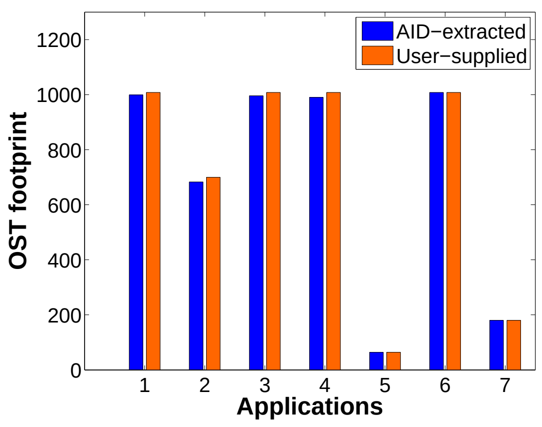
(a) I/O volume per I/O burst



(b) I/O interval



(c) Average I/O throughput



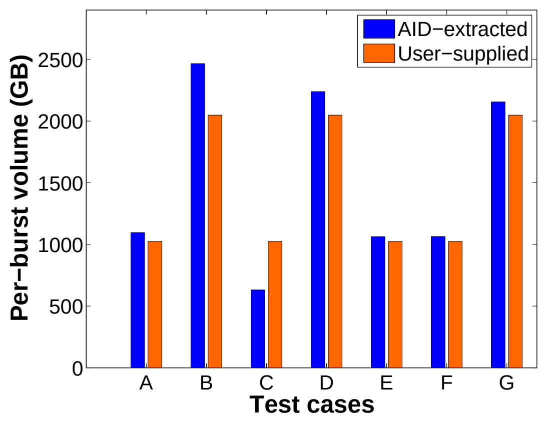
(d) Number of OSTs

Figure 5.4: Real Titan application I/O characterization accuracy

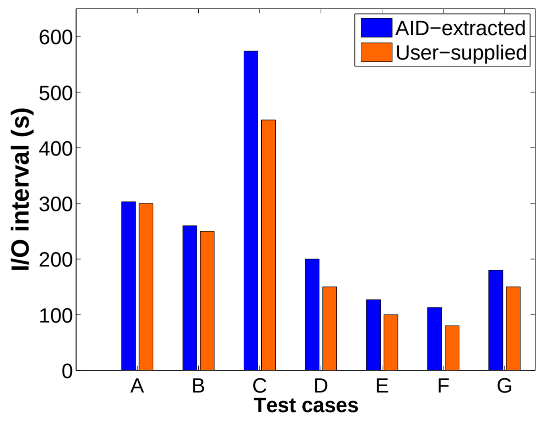
I/O behavior and settings. Figure 5.4 shows the side-by-side comparison between AID-extracted and user-supplied I/O patterns. We examine four key features, namely per-burst I/O volume, I/O interval, average I/O throughput during bursts, and OST footprint. We choose not to normalize the results to show the actual scale of such pattern parameters in real applications. Results for Application 8 are given separately in Table 5.4, due to its exceptionally large I/O volume.

From these results, we can see that AID achieves high accuracy in automatically discovering application-specific I/O characteristics, with errors likely due to noises. Actually, AID estimated I/O volume can sometimes be smaller than the true application volume, indicating that we might excluded I/O traffic from the target application. However, such accuracy suffices for I/O-aware scheduling and workload study purposes.

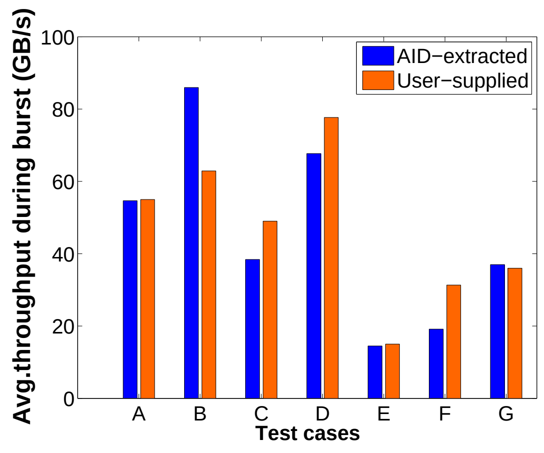
One interesting side discovery here is that the majority of observable I/O-intensive applications have rather large OST footprint, as currently this is still the parallel I/O model that delivers the highest aggregate throughput by avoiding synchronization overhead (even when using the same number of OSTs). More efficient *n-to-m* parallel I/O would allow the applications to obtain high throughput while leaving the system more flexibility in I/O-aware scheduling.



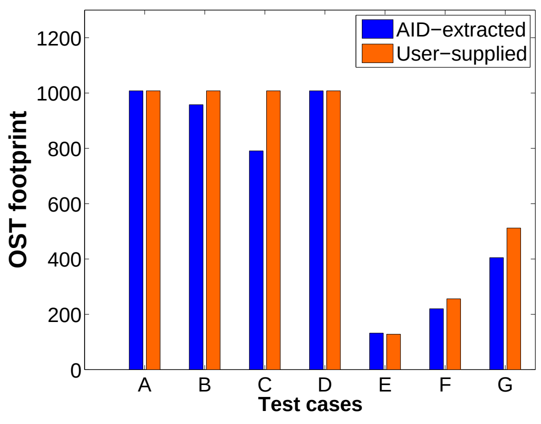
(a) Volume per I/O burst



(b) I/O interval



(c) Average I/O throughput during burst



(d) OST footprint

Figure 5.5: IOR pseudo-application I/O characterization accuracy

Pseudo-applications Unlike with real applications, we possess all ground truth on our IOR pseudo-applications. We designed them to portray the diverse HPC I/O behavior, with contrasting node counts, per-burst I/O volumes, and I/O intervals. Most importantly, they adopt different common HPC file access models (*n-to-1*, *n-to-m*, and *n-to-n*), resulting in different OST footprints. To match the behavior observed in real Titan applications, we intentionally added variability in job behavior (while maintaining the base I/O pattern), by changing the number of computation-I/O iterations, hence producing different sample lengths. In addition, we simulate the "I/O-off" runs by randomly adding 1 - 5 job runs without I/O. The output files use the default Spider setting (stripe count at 4 and stripe size 4MB). As shown in Fig 5.5, AID achieves similarly good accuracy in deriving the application I/O patterns.

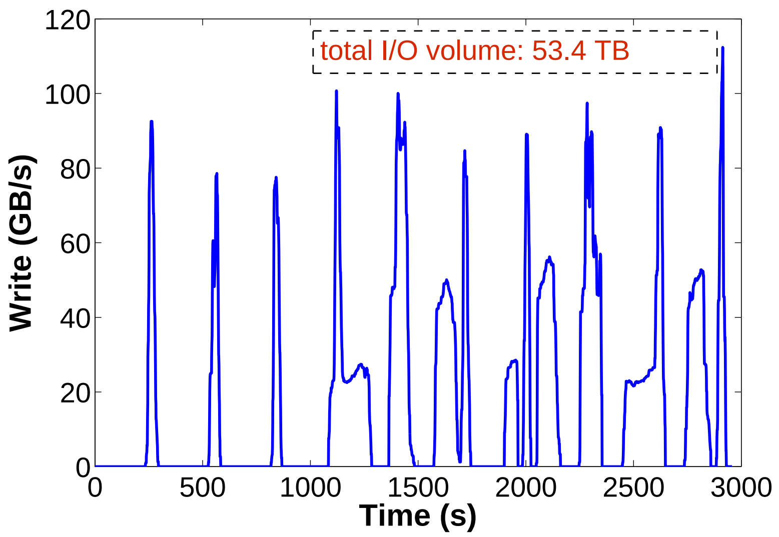
Fig 5.6 demonstrates the effect of I/O pattern refinement on one of the IOR_B samples using the OST footprint results, as described in Chapter 5.3.3. The left figure shows I/O bursts before having OST footprint information and the right one after. The I/O bursts after refinement are visibly more clarified and less noisy. As marked on the top of the figures, the OST-footprint knowledge (verified as quite accurate by previous results) helps AID trim the total I/O volume of this application from 53.4TB to 38.5TB (note that this is single-sample result), by excluding I/O traffic from non-participating OSTs.

5.5.3 I/O-aware Job Scheduling

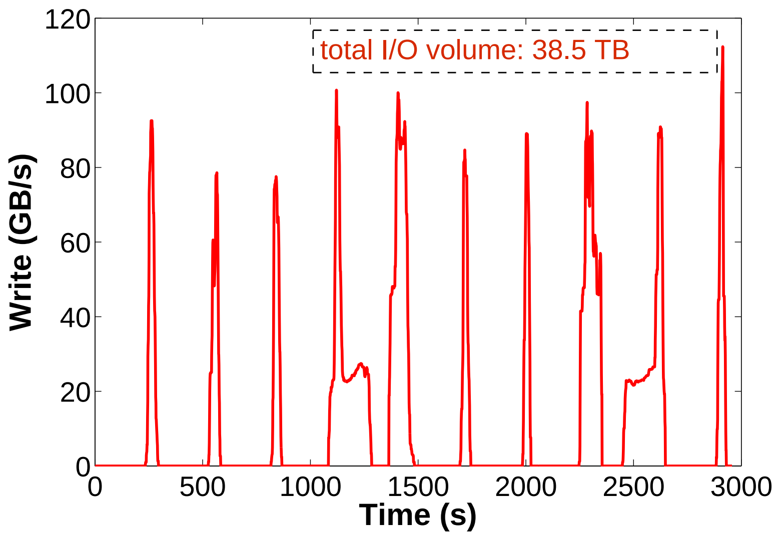
Finally, we evaluate the effectiveness of AID's I/O-aware scheduling recommendation. As we have no way to control the submission of real application jobs, here we have to use pseudo-applications. In this set of experiments, we issued groups of pseudo-applications to create varying levels of inter-job I/O contention as well as system I/O load. We analyze ongoing applications' I/O patterns and calculate the per-OST load scores, as discussed in Chapter 5.4, which leads to either the "run" or "delay" recommendation right before the target pseudo-application's execution. To simulate the scenario with real I/O-aware scheduling enabled, after the target application starts we check and suspend the execution of other queued pseudo-applications, to observe the accuracy of the recommendations.

Here we used another set of IOR pseudo-applications, all using the *n-to-m* model, to control the OST footprint (256, 512, and 1008). The acceptable performance impact factor is set to 1.5 (50% slowdown). Figure 5.7 gives the results, with pairs of bars showing the average per-job I/O time (plus variance in standard deviation). The number above each bar indicates the number of trials. We had to run many jobs to get at least 5 trials receiving each recommendation, as we cannot control whether an individual trial will receive a "run" or a "delay" order. As expected, the runs started despite the "delay" recommendation do spend considerably more time on I/O and often have larger I/O time variances compared to those with the "run" blessing, as seen in Figures 5.7a - 5.7c.

Figure 5.7d plots the 2-D distribution of all three sets of IOR trial data points, in average



(a) Before OST analysis

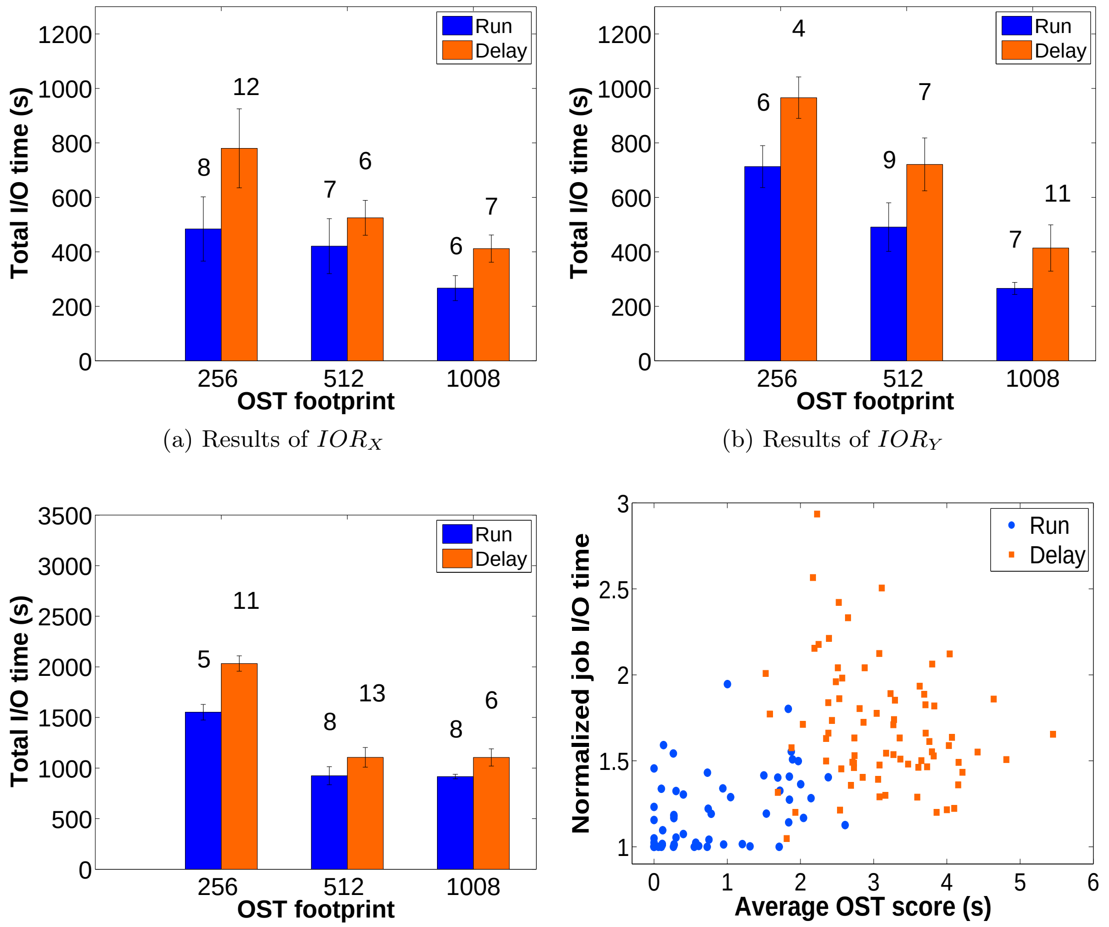


(b) After OST analysis

Figure 5.6: I/O bursts before/after OST footprint analysis

OST score (x) and total I/O time normalized to the shortest measurement (y). It clearly shows that the "run" data points (blue dots) have better and more consistent performance than the "delay" ones (orange squares). More specifically, the "delay" data points have an I/O performance impact factor (slowdown from the shortest I/O time measurement) of 1.69 on average and up to 2.93. The "run" data points, in contrast, have 1.21 on average and up to 1.92.

Several of the "run" data points do get over the 1.5 impact factor threshold and cause larger



(a) Results of IOR_X (b) Results of IOR_Y

(c) Results of IOR_Z (d) Scatter plot

Figure 5.7: Results of scheduling recommendation

variances in the "run" bars in Figures 5.7a - 5.7c. However, the experiments are done on a large production system where we are not really scheduling applications: though we can "hold" other pseudo-applications, real I/O-intensive user jobs do not actually go through AID's approval and may start after its recommendation is made. Therefore we expect AID's advantage to be more significant with fully deployed I/O-aware schedulers.

5.6 Conclusion

We present AID, a mechanism that *mines* application-specific I/O patterns from existing supercomputer server-side I/O traffic logs and batch job history jobs, *without any additional tracing,* *profiling, or user-provided information*. We verified the effectiveness of AID using both user feedback (on real-world HPC applications unknown to us) and our own pseudo-applications on Titan. We further enabled AID to make I/O-aware application scheduling recommendations, and confirmed with experiments on the same production platform that such recommendations can produce significantly lower I/O time and smaller I/O performance variance.

Chapter 6

Conclusion

In this PhD thesis study, we have presented IOSI and AID, two automatic tools for understanding and exploiting supercomputing applications I/O behaviors. To summarize, IOSI is a zero-overhead scheme for automatically identifying the I/O signature of data-intensive parallel applications from existing supercomputer server-side I/O traffic logs and batch job history jobs; while AID is a mechanism that *mines* I/O intensive applications and achieves I/O-aware job scheduling *without requiring any apriori information on the applications or jobs*. Both tools utilize *existing* throughput logs on the storage servers, in addition to routine job logs from the batch scheduler. We verified the effectiveness of IOSI and AID with real scientific applications and pseudo-applications on Titan. Our results demonstrated the accuracy of the tools accuracy in identifying I/O-intensive applications, the I/O signature of such applications, as well as the effectiveness of potential I/O-aware job scheduling leveraging such automatically I/O characterization results.

Our work demonstrates that in large, complex, and highly dynamic shared environments, where detailed tracing/profiling is often intrusive and costly (both performance and human effort wise), we can still learn a lot about unknown applications just by examining low-overhead, coarse-granule system logs that have been routinely collected. The key observation here is that resource-heavy applications tend to have consistent behavior to be noticed, and the future, to a large extent, does repeat history.

Finally, though designed and evaluated entirely in the supercomputing environment, we believe that nothing regarding the approaches, techniques, and underlying tools is limited to supercomputers. Such data-driven automatic workload characterization and resource scheduling can potentially benefit multi-tenant executions on commercial virtual/cloud platforms as well.

REFERENCES

[1] IOR HPC Benchmark, https://asc.llnl.gov/sequoia/benchmarks/#ior.

[2] Los Alamos National Laboratory open-source LANL-Trace, http://institute.lanl.gov/data/tdata.

[3] OLCF Policy Guide, https://www.olcf.ornl.gov/support/system-user-guides/olcf-policy-guide/.

[4] Titan, http://www.olcf.ornl.gov/titan/.

[5] Wavelet, http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html.

[6] Using Cray Performance Analysis Tools. *Document S-2474-51, Cray User Documents, http://docs.cray.com*, 2009.

[7] John Aach and George M Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, 17:495–508, 2001.

[8] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCtoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[9] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[10] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod Record*. ACM, 1999.

[11] I Barroda and FDK Roberts. Solution of an overdetermined system of equations in the l1 norm [f4]. *Communications of the ACM*, 1974.

[12] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

[13] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, 2000.

[14] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, 1994.

[15] Rupak Biswas, MJ Aftosmis, Cetin Kiris, and Bo-Wen Shen. Petascale computing: Impact on future nasa missions. *Petascale computing: architectures and algorithms*, 2007.

[16] Ronald N. Bracewell. *The Fourier transform and its applications*. McGraw-Hill New York, 1986.

[17] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[18] SW Bruenn, A Mezzacappa, WR Hix, JM Blondin, P Marronetti, OEB Messer, CJ Dirk, and S Yoshida. 2d and 3d core-collapse supernovae simulation results obtained with the chimera code. *Journal of Physics: Conference Series*, 2009.

[19] C Sidney Burrus, Ramesh A Gopinath, Haitao Guo, Jan E Odegard, and Ivan W Selesnick. *Introduction to wavelets and wavelet transforms: a primer*, volume 23. Prentice Hall Upper Saddle River, 1998.

[20] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '08)*, 2008.

[21] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 2011.

[22] Philip H. Carns, Robert Latham, Robert B. Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 Characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS'09)*, 2009.

[23] Selina Chu, Eamonn J Keogh, David Hart, and Michael J Pazzani. Iterative Deepening Dynamic Time Warping for Time Series. In *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM'02)*, 2002.

[24] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Proceedings of the 1st International Conference on Computational Science (ICCS'03)*, 2003.

[25] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[26] Ingrid Daubechies. Orthonormal bases of compactly supported wavelets II: Variations on a theme. *SIAM Journal on Mathematical Analysis*, 24:499–519, 1993.

[27] Carl de Boor. *A practical guide to splines.* Springer-Verlag New York, 1978.

[28] John R Deller, John G Proakis, and John HL Hansen. *Discrete-time processing of speech signals.* IEEE, 2000.

[29] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *Parallel and Distributed Processing Symposium*, 2014.

[30] Pan Du, Warren A Kibbe, and Simon M Lin. Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics*, 22(17):2059–2065, 2006.

[31] E.L.Miller and R.H.Katz. Input/output behavior of supercomputing applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'91)*, 1991.

[32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[33] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Yves Robert, and M. Snir. Scheduling the of HPC applications under congestion. In *IPDPS*, 2015.

[34] Gregory R. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. In *Proceedings of the Computer Measurement Group (CMG'95)*, 1995.

[35] Zhenhuan Gong and Xiaohui Gu. PAC: Pattern-driven Application Consolidation for Efficient Cloud Computing. In *Proceedings of the 18th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'10)*, 2010.

[36] Raghul Gunasekaran, David Dillow, Galen Shipman, Richard Vuduc, and Edmond Chow. Characterizing application runtime behavior from system logs and metrics. In *Proceedings of the Characterizing Applications for Heterogeneous Exascale Systems (CACHES'11)*, 2011.

[37] Haitao Guo and C Sidney Burrus. Fast approximate Fourier transform via wavelets transform. In *Proceedings of the International Symposium on Optical Science, Engineering, and Instrumentation*, 1996.

[38] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, 2005.

[39] Evatt R Hawkes, Ramanan Sankaran, James C Sutherland, and Jacqueline H Chen. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65, 2005.

[40] MR Ilango and V Mohan. A Survey of Grid Based Clustering Algorithms. *International Journal of Engineering Science and Technology*, 2(8):3441–3446, 2010.

[41] Yasuhiko Kanemasa, Qingyang Wang, Jack Li, Masazumi Matsubara, and Calton Pu. Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation. In *Services Computing (SCC), IEEE International Conference*, 2013.

[42] Eamonn Keogh and Chotirat Ann Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[43] Eamonn J Keogh and Michael J Pazzani. Scaling up dynamic time warping for datamining applications. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Dining*, 2000.

[44] Youngjae Kim, Scott Atchley, Geoffroy R Vallée, and Galen M Shipman. LADS: Optimizing data transfers using layout-aware data scheduling. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[45] Nick G Kingsbury. The dual-tree complex wavelet transform: a new technique for shift invariance and directional filters. In *Proceedings of the 8th IEEE Digital Signal Processing (DSP) Workshop*, 1998.

[46] Douglas Kothe and Ricky Kendall. Computational science requirements for leadership computing. *Oak Ridge National Laboratory, Technical Report*, 2007.

[47] Zachary Kurmas and Kimberly Keeton. Synthesizing Representative I/O Workloads Using Iterative Distillation. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*, 2003.

[48] Yan Li, Xiaoyuan Lu, Ethan L Miller, and Darrell DE Long. Ascar: Automating contention management for high-performance storage systems. In *Mass Storage Systems and Technologies (MSST'15)*, 2015.

[49] Constanze Lipowsky, Egor Dranischnikow, Herbert Göttler, Thomas Gottron, Mathias Kemeter, and Elmar Schömer. Alignment of noisy and uniformly scaled time series. In *Proceedings of the Database and Expert Systems Applications (DEXA'09)*, 2009.

[50] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014.

[51] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'08)*, 2008.

[52] Jay Lofstead and Robert Ross. Insights for exascale IO APIs from building a petascale IO API. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[53] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010.

[54] Yan Long, Liu Gang, and Guo Jun. Selection of the best wavelet base for speech signal. In *Proceedings of the International Symposium on Intelligent Multimedia, Video and Speech Processing*, 2004.

[55] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[56] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, 1967.

[57] S.G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[58] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O'Hallaron. //trace: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[59] Ross Miller, Jason Hill, David A. Dillow, Raghul Gunasekaran, Shipman Galen, and Don Maxwell. Monitoring tools for large scale systems. In *Proceedings of the Cray User Group (CUG'10)*, 2010.

[60] Kathryn Mohror and Karen L Karavanic. An Investigation of Tracing Overheads on High End Systems. Technical report, PSU, 2006.

[61] Walid G Morsi and ME El-Hawary. The most suitable mother wavelet for steady-state power system distorted waveforms. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, 2008.

[62] Meinard Müller. Dynamic time warping. *Information Retrieval for Music and Motion*, pages 69–84, 2007.

[63] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M.L. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1996.

[64] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[65] Sarp Oral, David A Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, et al. OLCF's 1 TB/s, next-generation lustre file system. In *Proceedings of Cray User Group Conference (CUG 2013)*, 2013.

[66] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, et al. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[67] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[68] B.K. Pasquale and G.C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'93)*, 1993.

[69] Thomas C Peterson, Thomas R Karl, Paul F Jamason, Richard Knight, and David R Easterling. First difference method: Maximizing station density for the calculation of long-term global temperature change. *Journal of Geophysical Research: Atmospheres (1984–2012)*, 1998.

[70] James S Plank and Michael G Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *Journal of Parallel and distributed Computing*, 61(11):1570–1590, 2001.

[71] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-aware adaptive scheduling for mapreduce clusters. In *Middleware 2011*, pages 187–207. Springer, 2011.

[72] Anna Povzner, Kimberly Keeton, Arif Merchant, Charles B Morrey III, Mustafa Uysal, and Marcos K Aguilera. Autograph: automatically extracting workflow file signatures. *ACM SIGOPS Operating Systems Review*, 43(1):76–83, 2009.

[73] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who is your neighbor: Net I/O performance interference in virtualized clouds. *Services Computing*, 2013.

[74] Philip C Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: held in conjunction with SC'07*, 2007.

[75] Seetharami Seelam, I-Hsin Chung, Ding-Yong Hong, Hui-Fang Wen, and Hao Yu. Early experiences in application level I/O tracing on Blue Gene systems. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'08)*, 2008.

[76] Galen Shipman, Dave Dillow, Sarp Oral, and Feiyi Wang. The spider center wide file system: From concept to reality. In *Proceedings of the Cray User Group (CUG'09)*, 2009.

[77] Kyle Spafford, Jeremy Meredith, Jeffrey Vetter, Jacqueline Chen, Ray Grout, and Ramanan Sankaran. Accelerating S3D: a GPGPU case study. In *Euro-Par 2009 Parallel Processing Workshops*, 2010.

[78] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. Extracting flexible, replayable models from large block traces. In

Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12), 2012.

[79] TOP500 Supercomputer Sites, `http://www.top500.org/`.

[80] A. Uselton, M. Howison, N.J. Wright, D. Skinner, N. Keen, J. Shalf, K.L. Karavanic, and L. Oliker. Parallel I/O performance: From events to ensembles. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'10)*, 2010.

[81] Jeffrey S Vetter and Michael O McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming (PPoPP'01)*, 2001.

[82] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan Vazhkudai. Improving Large-Scale Storage System Performance via Topology-aware and Balanced Data Placement. In *The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.

[83] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. Mclarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the IEEE 21th Symposium on Mass Storage Systems and Technologies (MSST'04)*, 2004.

[84] William Wu-Shyong Wei. *Time series analysis.* Addison-Wesley Redwood City, California, 1994.

[85] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[86] Y. Xu, J. B. Weaver, D. M. Healy, and J. Lu. Wavelet transform domain filters: a spatially selective noise filtration technique. *IEEE Transactions on Image Processing*, 3(6):747–758, 1994.

[87] Neeraja J Yadwadkar, Chiranjib Bhattacharyya, K Gopinath, Thirumale Niranjan, and Sai Susarla. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[88] Jae-Chern Yoo and Tae Hee Han. Fast normalized cross-correlation. *Circuits, Systems and Signal Processing*, 28(6):819–843, 2009.

[89] John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[90] Daren Yu, Xiao Yu, Qinghua Hu, Jinfu Liu, and Anqi Wu. Dynamic time warping constraint learning for large margin nearest neighbor classification. *Information Sciences*, 181(13):2787–2796, 2011.

[91] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[92] Hongbo Zou, Xian-He Sun, Siyuan Ma, and Xi Duan. A source-aware interrupt scheduling for modern parallel I/O systems. In *Parallel & Distributed Processing Symposium (IPDPS)*, 2012.