Fine-grained Policy-driven I/O Sharing for Burst Buffers

Ed Karrels*
University of Illinois Urbana Champaign

Lei Huang*
Texas Advanced Computing Center

Yuhong Kan
The University of Texas at Austin

Ishank Arora
The University of Texas at Austin

Yinzhi Wang
Texas Advanced Computing Center

Daniel S. Katz
University of Illinois Urbana Champaign

William D. Gropp
University of Illinois Urbana Champaign

Zhao Zhang
Rutgers University

ABSTRACT

A burst buffer is a common method to bridge the performance gap between the I/O needs of modern supercomputing applications and the performance of the shared file system on large-scale supercomputers. However, existing I/O sharing methods require resource isolation, offline profiling, or repeated execution that significantly limit the utilization and applicability of these systems. Here we present ThemisIO, a policy-driven I/O sharing framework for a remote-shared burst buffer: a dedicated group of I/O nodes, each with a local storage device. ThemisIO preserves high utilization by implementing opportunity fairness so that it can reallocate unused I/O resources to other applications. ThemisIO accurately and efficiently allocates I/O cycles among applications, purely based on real-time I/O behavior without requiring user-supplied information or offline-profiled application characteristics. ThemisIO supports a variety of fair sharing policies, such as user-fair, size-fair, as well as composite policies, e.g., group-then-user-fair. All these features are enabled by its statistical token design. ThemisIO can alter the execution order of incoming I/O requests based on assigned tokens to precisely balance I/O cycles between applications via time slicing, thereby enforcing processing isolation. Experiments using I/O benchmarks show that ThemisIO sustains 13.5–13.7% higher I/O throughput and 19.5–40.4% lower performance variation than existing algorithms. For real applications, ThemisIO significantly reduces the slowdown by 59.1–99.8% caused by I/O interference.

ACM Reference Format:
Ed Karrels*, Lei Huang*, Yuhong Kan, Ishank Arora, Yinzhi Wang, Daniel S. Katz, William D. Gropp, and Zhao Zhang. 2023. Fine-grained Policy-driven I/O Sharing for Burst Buffers. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3581784.3607041

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607041

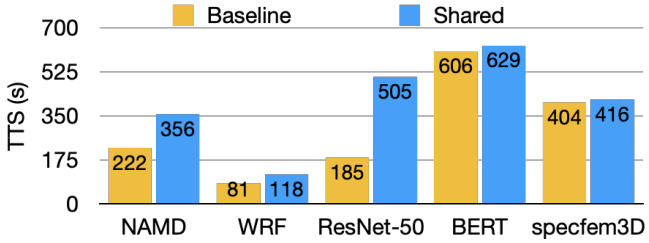


Figure 1: Time-to-solution comparison of five applications using a remote burst buffer. Baseline = measurements with exclusive access to the burst buffer. Shared = measured with a background I/O benchmark job running.

1 INTRODUCTION

High-performance computing (HPC) architects have deployed burst buffers on supercomputers to bridge the I/O gap between compute and storage by absorbing bursty I/O. However, when burst buffers are shared among applications, researchers have observed I/O interference [9, 15, 18, 28], where one application is slowed down by the I/O of other applications.

To illustrate this, we run a set of five applications in a controlled environment with an in-house remote-shared burst buffer using two nvdimm nodes on the Frontera supercomputer. Each node is equipped with a 6.2 TB Intel Optane persistent memory. We first measure the baseline performance of each application making exclusive use of the burst buffer, then we measure each application's runtime with a background I/O benchmark job, which is 3–173% longer than the baseline, as shown in Figure 1.

The root cause of I/O interference is that today's production systems generally process I/O requests in a first-in-first-out (FIFO) manner, which means that highly concurrent and bursty I/O traffic from one application can saturate the I/O system's queue, then block the I/O of another application for a period of time. The amount of the slowdown depends on the state of the queue when I/O requests enter; specifically, how many I/O requests from one application are ahead of the new requests from the other application.

Computer scientists have proposed various ideas to enable efficient I/O sharing [9, 12, 15, 16, 19, 23] . However, these solutions require prior knowledge of the application's I/O behavior to enable

*Equal contributions to this work

fair-sharing of I/O resource. E.g., Gift [19] assumes applications are repeatedly run, it uses a throttle-and-reward system to maximize I/O bandwidth by relaxing the fairness window. This throttled execution in Gift may introduce a unbounded delay. TBF [23] and CARS [16] require users to supply the I/O throughput of applications, then they statically allocate sufficient I/O resource to meet the requirement. Because I/O workload can vary during an application's execution, a static allocation can lead to wasted I/O resources. These systems cannot dynamically adjust I/O resource allocations based on the real-time I/O requirement for new applications, or even existing applications with new configurations, e.g., running at a larger scale.

To address the I/O interference issue in burst buffer systems and to enable **efficient** and **dynamic** I/O resource sharing with **flexible policies**, we have designed, built, and tested ThemisIO. To ensure ThemisIO is always operating with **maximal I/O throughput**, it implements opportunity fairness, meaning that fairness is only enforced when the overall I/O requests received by the I/O system exceeds its capacity. While using ThemisIO when the I/O system is partially loaded, applications will get the same amount of I/O resources as they would when running without ThemisIO (see §5.3.1). Thus, the delay of an applications due to enforced fairness is bounded, as ThemisIO guarantees the assigned I/O resources for each application are no less than its fair share as specified in a policy (see §5.5). We have designed a statistical, token-based strategy for time-sliced sharing, which offers every I/O-active application an opportunity for fair-sharing, unlike previous work that uses a mandatory I/O bandwidth assignment. To enable ThemisIO to **dynamically** adjust I/O resources to applications based on their real-time workloads, we embed job-related information, such as job id, user id, and job size, in the I/O request. ThemisIO can allocate or reclaim tokens based on the number of I/O requests from each user, each job, or each group (see §5.4). This token-based design also makes ThemisIO **flexible in sharing polices**. For example, ThemisIO can assign the same number of tokens to each user to achieve primitive sharing policies, such as user-fair (i.e., I/O cycles are evenly split across users.) It can also assign an appropriate number of tokens to enable composite policies such as user-then-job-fair, i.e., the I/O cycles are evenly split across users, then for each user, the allocated cycles are further evenly split among their jobs (see §5.3.2).

To make ThemisIO compatible with existing supercomputing applications, we implement a POSIX-compliant interface, so applications can take advantage of ThemisIO as a tradition file system; users do not have to make changes to their code to use ThemisIO.

We have integrated ThemisIO with an in-house burst buffer system and used I/O benchmarks and five real-world applications to examine the efficacy of the ThemisIO design. Our results show that ThemisIO is efficient, as it achieves the hardware I/O throughput limit, which is ~22 GB/sec per I/O server combining read and write. Global fairness can be preserved with a controlled delay, which is negligible compared to the time-to-solution of jobs (see §5.6). The sharing enabled by ThemisIO is more efficient and stable than that of existing frameworks GIFT and TBF. With ThemisIO, the sustained I/O throughput of benchmarks is 13.5–13.7% higher than when using the GIFT and TBF algorithms, and the I/O throughput variation in I/O sharing is 19.5–40.4% lower with ThemisIO.

For real applications, running a 64-node NAMD job with a background I/O benchmark job is 37.7% faster with ThemisIO size-fair policy than that of FIFO. Compared to the baseline with exclusive I/O resource access, the size-fair policy is 0.1% slower while the FIFO policy is 60.3% slower, which is a ~600x reduction in the performance slowdown due to I/O interference. Across the applications in §5.5, ThemisIO size-fair reduces the slowdown by 59.1–99.8% compared to FIFO.

The contributions of this paper are:

- the statistical token-based time sharing design of ThemisIO, which can flexibly support various upstream sharing policies,
- the global fairness enforcement algorithm on multiple I/O servers
- the design of primitive and composite I/O sharing policies, and
- the open source implementation of ThemisIO (https://github.com/bbThemis/ThemisIO).

The rest of the paper is structured as follows. Section 2 provides background on modern supercomputing applications and states the need and requirements of a policy-driven I/O sharing system. The foundational statistical token is presented in §3. We then discuss the system design and implementation in §4 and present experiments and their results in §5. Existing I/O sharing work is summarize and compared to ThemisIO in §6. Finally, we conclude and envision future work in §7.

2 BACKGROUND AND MOTIVATION

In this section, we first review some modern supercomputing applications and present a suite that we have selected to drive the design of ThemisIO, then discuss the motivation for I/O sharing.

2.1 Background

In addition to traditional numerical simulation applications, such as WRF (Weather Research and Forecasting) [25] and NAMD (scalable molecular dynamics) [22], researchers now also use a big data approach (e.g., parallel scripting and MapReduce) and deep learning tools to conduct disciplinary research. Researchers also train neural networks for image classification, object detection, and scientific literature analysis. Some notable models ResNet [7], Mask R-CNN [6], and BERT [4]. Those models are being actively applied to research fields from astronomy to zoology [1, 2, 13].

These diverse applications, run daily on supercomputers, use I/O libraries such as MPI-IO [26, 27], HDF5 [5], and POSIX IO. When running at large scale, some individual executions can cause file system saturation and unresponsiveness. To absorb the bursty I/O workload, supercomputer architects have deployed burst buffer systems on supercomputers, such as the Cray DataWarp on NERSC's Cori and DDN Infinite Memory Engine (IME) on the OSC Pitzer and Owens clusters. The system we use here, Frontera, features 16 compute nodes that each have ~5.4 TB of Intel Optane memory, configured as 2.1 TB memory extension and 3.3 TB local storage.

There are generally two types of burst buffers. In a node-local burst buffer, a storage device such as an SSD is attached to each compute node, such as the NVDIMM nodes on Frontera. ThemisIO is designed for the second type, remote-shared burst buffer, where storage devices are attached to a number of dedicated I/O nodes,

such as DataWarp on Cori. In both cases, compute nodes and I/O nodes are connected via a high-speed interconnect, e.g., InfiniBand.

2.2 Motivation

Most of today's supercomputers provide processing isolation for computing resources by granting exclusive access to compute nodes. However, such isolation does not exist in I/O resources, which causes problems. In this section, we discuss these problems in detail and provision a multi-tenant I/O framework to address I/O sharing challenges.

2.2.1 State of the Practice and Problems. From our daily experience of operating HPC systems at multiple institutions, we have noticed cases where the I/O of a small job can dominate the systems I/O resources due to its high frequency and high volume. The root causes of this phenomena are that 1) the I/O queue is packed with requests from the small job and the I/O system processes them in a FIFO way, and 2) the file system's throughput is insufficient. Depending on when other I/O requests arrive in the I/O queue, the small job can indefinitely block the I/O requests of all other jobs. If we define fairness of I/O resources to be proportional to a job's size, i.e., the number of compute nodes, **this unfair sharing of the I/O resource certainly does not reflect the priority of the jobs.**

In some other cases, the I/O workload of a job can be heavy in metadata access, which eventually saturate the metadata server. While this blocks other jobs from accessing metadata, the data servers, such as OSTs in Lustre, may be idle during this process. Again, **it is the FIFO processing of I/O requests that causes this huge resource waste**. Although using burst buffers can absorb the bursty workload to some extent, the problem still exists as long as the I/O requests are processed in a FIFO manner.

It is challenging, if not impossible, for today's I/O systems to provide a quality of service (QoS) guarantee. For example, an I/O and shared file system that can provide 10M IOPS typically cannot guarantee 1M IOPS to each of ten jobs on the machine, regardless of the job's size. One solution to this is to disassociate I/O control from actual processing. With rich job metadata, control logic can alter the processing order of the I/O requests to enforce defined sharing policies and achieve fairness.

Those practices and problems motivate ThemisIO. Clearly, there is a critical need for an I/O sharing framework that can 1) isolate I/O request processing so one job does not block others, 2) assign idle I/O cycles to jobs with high I/O demand when possible to achieve high utilization, and 3) provide flexible sharing policies to enforce certain fairness, as discussed in detail in the next subsection.

2.2.2 Definition of Fairness. There are many ways to define fairness. Allocating I/O resources in proportion to job size is one type of fairness, which we refer to as **size-fair**. Evenly splitting I/O resources among all active jobs is another type of fairness, and we refer to this as **job-fair**. Splitting I/O resources among all users regardless of the number of jobs each user is running is a third type of fairness, which we call **user-fair**.

These are just three examples of primitive fairness definitions. One can argue that assigning more I/O resources to prioritized jobs is fair, for example, during the hurricane season where there is an urgent need for computing resources to forecast potential disasters; we refer to this as **priority-fair**.

Some cases may need composite sharing policies, for example, to first split I/O resources evenly among users, then to split a user's resources in proportion to their job sizes. We call this **user-then-size-fair**. There could also be **group-then-size-fair** and **group-then-user-fair** policies.

The ThemisIO design can support all of these sharing policies with a single parameter, so that system administrators can specify the sharing policy when starting ThemisIO. Internally, ThemisIO uses a token-based design to enable time-sliced sharing of I/O resources. It maps the above sharing polices and fairness definitions to token management, where I/O requests are processed with corresponding job tokens, and tokens are recycled after requests are processed. This means that sharing policies can be implemented by assigning a number of tokens to each job. The details of the token-based design is discussed in §3.

3 SHARING VIA STATISTICAL TOKENS

We designed ThemisIO to be a generic I/O sharing system, in the sense that it can flexibly support **primitive sharing policies** as well as **composite sharing policies**. For primitive policies, e.g., **job-fair**, **user-fair**, **size-fair**, and **priority-fair**, we assign an identical number of I/O cycles among jobs, users, or in proportion to the node count or the job priority within a time unit. We refer the notion of job, user, and size as sharing entities.

Leveraging the classic token mechanism, allocating an appropriate number of tokens to each sharing entity is a straight-forward and effective approach to enable primitive polices, as illustrated in Figure 2(a). In Figure 2(a), we assign two tokens to each job to enable job-fair sharing. However, this token-based approach is limited in enabling composite policies, e.g., user-then-job-fair, where we want to evenly assign I/O resource among users, then enforce job-fair sharing within the scope of a user. A naive approach to enable this composite policy would to have two tiers of token queues, as shown in 2(b), each representing the sharing entities of users and jobs that belong to a user. With a composite sharing policy that involves N layers of sharing entity, we have to maintain N tiers of token queues and actual number of token queues is exponential to N, which significantly limits the scalability of the token mechanism. In addition, managing the token queues requires frequent locking and unlocking to ensure consistency, which introduces extra system overhead.

We notice that managing the tokens for each sharing entity to enforce a policy is equivalent in a statistical approach: we divide the range of $[0, 1]$ into several segments, with the segment length proportional to the token counts. Then an I/O worker draws a random number within $[0, 1]$. The I/O request of a job is processed if the random number falls in its corresponding segment. For primitive policies, the range is split according to the number of jobs, users, or the node counts. Figure 3(a) and (b) show the statistical token assignments of the example in Figure 2(a) and (b), respectively.

The statistical token assignment can be calculated as a chain of transition matrix multiplication. Figure 4 shows the transition matrices of the user-then-job-fair example. At each sharing entity level, each row represents a token queue and each column represent

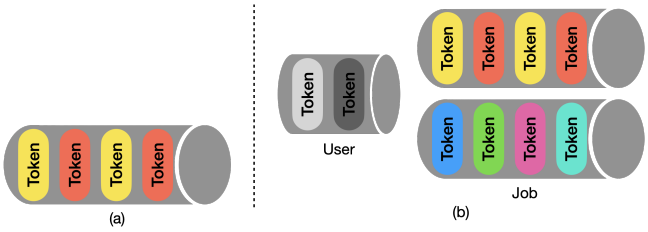


Figure 2: (a) An job-fair token assignment example with two jobs. (b) A user-then-job-fair token assignment of a two-tier token design with two users, one running two jobs and the other running four jobs.

Figure 3: (a) An job-fair statistical token assignment example with two jobs. (b) A user-then-job-fair statistical token assignment of a two-tier token design with two users, one running two jobs and the other running four jobs.

the entities in this level. For example, in the job matrix of Figure 4, the first row represents the top job queue, where there are two jobs. The second row represents the lower job queue with four jobs. To derive the statistical token assignment, we multiply the two matrices and obtain the results as in Figure 3(b).

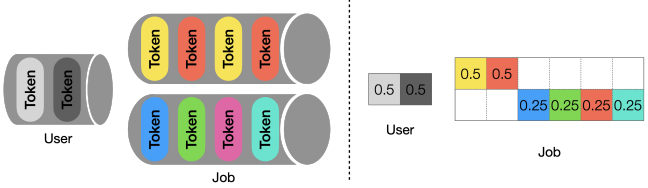


Figure 4: A transition matrix example of the user-then-job-fair policy.

More formally, we refer to the transition matrix as T^i. The value of $T^i_{j,k}$ is the fair share of the sharing entity as a percentage of its local scope. The sum of each row is one, and only one entry in each column can have a non-zero value, as the sharing percentage is applied within the local sharing entity scope. The statistical token assignment is evaluated as:

$$\prod_{i=0}^{N-1} T^i, \quad \text{N: the depth of the sharing policy} \tag{1}$$

Conceptually, this statistical token design is capable of supporting any composite sharing policy with an arbitrary depth. It reduces the the complex data structure management to a chain production of transition matrices. It removes the frequent use of a locking mechanism in synchronized queues at runtime. The statistical assignment can be easily adjusted by recalculating the matrix multiplication. One limitation introduced by this approach is that an application has to have a sufficiently large I/O workload to make the statistical token design effective, but this is commonly the case for modern supercomputing applications.

3.1 Local vs. Global Fairness

With multiple burst buffer servers, the job information on each server may not always be globally consistent. If the stripes of every file are spread across all burst buffer servers, e.g., with a sufficiently large stripe number, then every server has the global job status without communication. Otherwise, if files land on a disjoint set of burst buffer servers, every server initially has only local job information, which may not globally consistent. Synchronization is required to determine global job state and thus, there is a delay before global fairness is reached.

Figure 5 shows an example of delayed fairness with **size-fair**. Here, Server 1 sees two jobs: Job 1 and Job 2. After checking the size of the jobs (16 and 8 compute nodes, respectively), Server 1 assigns $[0, 0.66]$ and $[0.66, 1]$ for the two jobs, respectively. Similarly, Server 2 assigns $[0, 0.66]$ and $[0.66, 1]$ for Job 1 and Job 3. Now Job 1 gets 67% of the available I/O resources. However, by looking at the job size globally, we see the correct sharing ratio should be 16:8:8, which means Job 1 should only have 50% of the I/O resources.

To address this, ThemisIO introduces λ-**delayed fairness**, where controllers perform an all-gather on the job status table every λ time interval. This design guarantees that a globally unfair state will not last longer than λ. After this communication, both Server 1 and 2 see that Job 1 has a larger range than it should. So every server adjusts the statistical token of Job 1, and global fairness is reached. An effectiveness study of the length of λ is in §5.6.

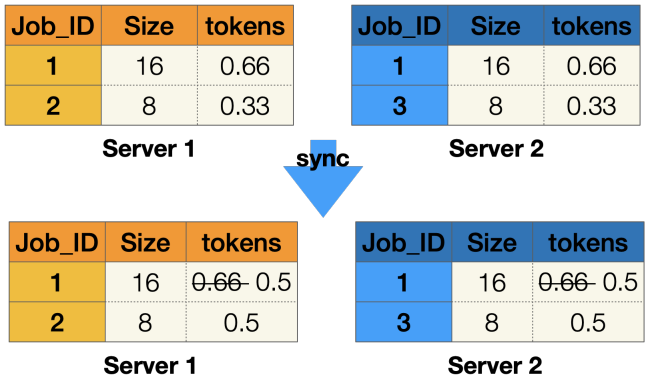


Figure 5: A Simple Example of Job Status Table Synchronization. Server 1 and 2 start with local job status and allocated tokens, then synchronize the tables by exchanging the entries and adding token counts.

4 DESIGN AND IMPLEMENTATION

In this section, we present the ThemisIO system's overall design and discuss key components and algorithms that enable generic and global fairness guarantees.

4.1 Architecture

ThemisIO exploits a server-client design, as shown in Figure 6. **Clients** reside with application processes on compute nodes and **servers** run on the burst buffer nodes.

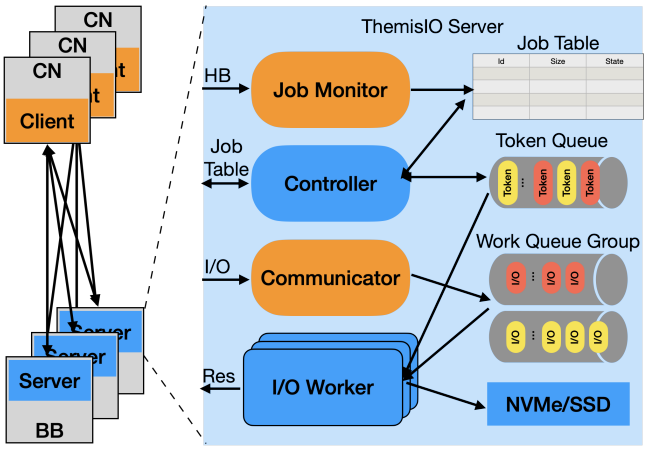


Figure 6: Overview of ThemisIO Architecture. Clients run on compute nodes with the applications. Servers run on dedicated burst buffer nodes. CN, Compute node; BB, Burst Buffer node; HB, Heartbeat; Res, Results.

The client intercepts I/O functions, gathers job metadata including user id, job id, and job size, then forwards I/O requests to servers. The client also sends heartbeats to servers, so that the servers can track job status in real time.

The server has four components: a **job monitor**, an **I/O request communicator**, a **controller**, and a group of **workers**. The **job monitor** may receive heartbeats from multiple clients of multiple applications. It maintains a job status table that summarizes job id, size, user, user group, and status. Job status is set to active when the corresponding job is new to the server. It is changed to inactive if a job heartbeat is not received for a predefined period of time. The **communicator** receives I/O requests from applications. Those I/O requests are grouped into queues based on the fair sharing policy. For example, with size-fair, where I/O resources are proportionally shared with respect to node count, requests are pushed into queues that are identified by job ids. The **controller** synchronizes with other servers to get the global status of active jobs, and allocates a number of tokens according to the fair sharing policy. Each **worker** pops one token at a time and an I/O request identified by the token, then processes the I/O request. There can be multiple workers for higher I/O throughput.

4.2 Communication

ThemisIO uses Unified Communication X (UCX [24]) for low latency and high bandwidth communication. Each ThemisIO server maintains two types of UCP workers: one for client-server communication and the other for server-server communication. A UCP worker is an opaque object in UCX that represents an instance of a local communication resource and the progress engine associated with it. When a ThemisIO server accepts a connection request from a ThemisIO client, it assigns a UCP worker to the client and keeps a mapping from the client to the UCP worker. One UCP worker can be shared between multiple ThemisIO clients.

Each application process/thread launches a ThemisIO client, which initializes a local UCP worker with a ThemisIO server. When the endpoint connection is established on the local UCP worker for the destination ThemisIO server address, job metadata is transferred to the servers. I/O requests are forwarded to the ThemisIO servers via the UCP worker to be processed, and the results are returned to the ThemisIO client. The heartbeat monitor in ThemisIO servers monitors the job status, and if a job is inactive for a predefined period of time, the server marks the job's status as inactive and destroying all the UCP worker resources mapping entries associated with that job. When a client exits, it notifies the ThemisIO servers to destroy the corresponding mapping entry.

The UCP workers are persistent during the lifetime of ThemisIO. ThemisIO servers synchronize the job status table to have a global view of the jobs and the assigned tokens. This synchronization enables the controller to adjust the token count to achieve λ-fairness, as discussed in §3.1.

4.3 File System

We integrate ThemisIO with a byte-addressable file system to support NVMe or SSD. To gain complete control and native speed, we implement a user space file system ourselves, however, ThemisIO can work with any shared file system in either kernel or user space. In this file system, both directories and files are stored as files, and files and metadata are spread across ThemisIO servers using a consistent hash function. Striping is supported with corresponding records in file metadata.

The location of a file is on one or more servers, determined by a hash function, and on those servers, an index specifies the NVMe region of the file's contents. Directory and file creation updates the content of the parent directory. Queries over a directory return the content in that directory. Reading a file returns the contents specified by the path and offset range. Writing a file writes/overwrites a range of allocated byte-addressable space in NVMe, and the metadata update is done on the same ThemisIO server. Concurrent read operations on the same file are executed without locking. Concurrent write operations to the same file proceed without any limitation if the byte ranges do not conflict. However, a locking mechanism is used when multiple threads are updating the file metadata.

4.4 I/O Function Interception

One of the design principles of ThemisIO is to be compatible with existing applications, i.e., applications do not have to make code change to leverage ThemisIO. However, most production supercomputers do not grant root privilege, which makes a kernel module implementation infeasible. So ThemisIO uses a I/O function interception technique. In this way, ThemisIO provides a POSIX-compliant interface, with which users can simply point I/O to a path that is prefixed by ThemisIO namespace, e.g., `/fs/input/path`. All I/O to/from this path will be intercepted by ThemisIO then processed in burst buffers.

To implement this, we either intercept the 64-bit version of I/O functions in the GNU C Library by simply exposing functions with the same name, or rewrite the first several instructions of a function with a jump instruction to the function implemented in ThemisIO library, then jump back to the original function if necessary. The first method is referred to as **override** [14] and the second method as **trampoline** [10].

Listing 1 summarizes the intercepted functions.

```
int open(const char *filename, int flags[, mode_t mode])
int close(int fd)
ssize_t read(int fd, void *buffer, size_t size)
ssize_t write(int fd, const void *buffer, size_t size)
off_t lseek(int fd, off_t offset, int whence)
DIR * opendir (const char *dirname)
struct dirent * readdir(DIR *dirstream)
int closedir (DIR *dirstream)
int stat(const char *filename, struct stat *buf)
```

Listing 1: Example Intercepted Functions

5 EXPERIMENTS AND RESULTS

To validate the effectiveness of the ThemisIO design and the correctness of our implementation, we run both benchmark and real applications with various sharing policies. We also implement the sharing algorithms in GIFT and TBF in ThemisIO and perform a comparative study. In addition, we investigate the impact of system performance with various communication intervals in λ-delayed fairness. In summary:

- Benchmarks show that ThemisIO can efficiently share I/O resources between jobs with both primitive and composite policies (details in §5.3).
- The comparison study with FIFO, GIFT, and TBF shows that ThemisIO shares I/O resources more efficiently and stably (details in §5.4).
- Real applications show that ThemisIO reduces the I/O intervention slowdown drastically or completely (details in §5.5).
- The communication interval in λ-delayed fairness can be as large as 500 ms without significant impact on global fairness. (details in §5.6)

All the experiments are run on the TACC Frontera supercomputer, which consists of 8,008 CPU nodes (CLX), 16 large-memory nodes (NVDIMM), and 90 GPU nodes (RTX). The CPU nodes have two Intel Xeon Platinum 8280 processors and 192 GB RAM. Each large-memory node has four Intel Xeon Platinum 8280 processors and 2.1 TB RAM, supported by Intel Optane memory. Each GPU node has four Nvidia Quadro RTX 5000 cards and 128 GB RAM. In all experiments, ThemisIO runs on the CLX nodes with RAM as storage devices.

5.1 Benchmark and Application Configuration

Throughout the experiments, we used IOR and mdtest for simple tests and a customized benchmark to measure I/O sharing performance. The customized benchmark simulates two workloads: 1) *iops_stat* repeatedly calls stat() to query file metadata with randomly generated file names; 2) *iops_write_read* writes a small (1 MB) file then reads the same file repeatedly. We disable client caching in all tests as ThemisIO is designed for remote-shared burst buffer, and we are investigating the I/O sharing capability in particular.

To simplify validation of the sharing capability, we run each application at a fixed size. Some of the applications take too long to finish, so we only run cases with a reasonable time length, which are still representative of their I/O workloads. The **WRF** benchmark uses the 12 KM CONUS Benchmark dataset from https://www2.mmm.ucar.edu/wrf/WG2/benchv3/#_Toc212961288. It is a 48-hour, 12-km resolution case over the Continental U.S. (CONUS) domain October 24, 2001 with a time step of 72 seconds. We run the WRF benchmark on 4 nodes each with 56 MPI process per node. The **NAMD** benchmark uses the 1M atom Satellite Tobacco Mosaic Virus system from https://www.ks.uiuc.edu/Research/namd/benchmarks/. It runs on 64 nodes with 8 MPI processes per node and 7 threads per process. The input was modified to save trajectory every 48 steps. The **SPECFEM3D** benchmark runs a small-scale regional seismic wave propagation simulation tweaked from the benchmark data set published by NVIDIA (https://www.nvidia.com/en-sg/data-center/gpu-accelerated-applications/specfem3d-globe/). The grid is defined in one cubed-sphere chunk of the globe and sliced into 224x256 elements. The simulated record length is 100 minutes. We run the benchmark on 16 nodes with 56 MPI processes per node. The **ResNet-50** case uses an open source PyTorch implementation [21] with the ImageNet [3] dataset that contains 1,331,167 images. The total size is ~156 GB and the average size of the image is about 116 KB. We run ResNet-50 on 16 RTX nodes with a 128 batch size per GPU. The complete training time is ~20 hours, so we only use the first three epochs (3×157 steps) in this case. The **BERT** case is a PyTorch implementation [20] with the English wikitext and Toronto Book Corpus datasets. The text is reorganized as 512 HDF5 files, with a total size of 71 GB and an average size of ~48 MB. We run BERT on four RTX nodes with a 16 batch size per GPU. BERT training has two phases, where phase 1 takes 393 hours to finish on four RTX nodes, so we only use the first three steps in this case.

5.2 Scaling Performance

Figure 7 represents the unidirectional aggregate throughput achieved running the ThemisIO server on 1 to 128 nodes. For each set of server nodes, an equal number of nodes were each running eight IOR processes, writing and reading 1 GB files in 1 MB blocks. We include a comparison of the performance of simple FIFO queuing versus job-fair queuing. With one server node, this achieved a maximum throughput of 11.7 GB/s. With eight server nodes, the slowest result was for FIFO reads at 77.1 GB/s, a scaling efficiency of 82%. With 128 server nodes, the throughput reached 1017 GB/s, a scaling efficiency of 68%. For comparison, the largest Lustre file system on Frontera has 32 OST nodes with an aggregate throughput of 120 GB/s. The DataWarp burst buffer on NERSC Cori has a peak throughput of 1.7 TB/s with each burst buffer node contributing 6.5 GB/s. The sustained I/O throughput of ThemisIO is comparable to the state of the art production system. It is worth noting that these experiments were unidirectional, just writing or just reading, unlike the read/write tests in Figure 8, where only half the interconnect throughput is available.

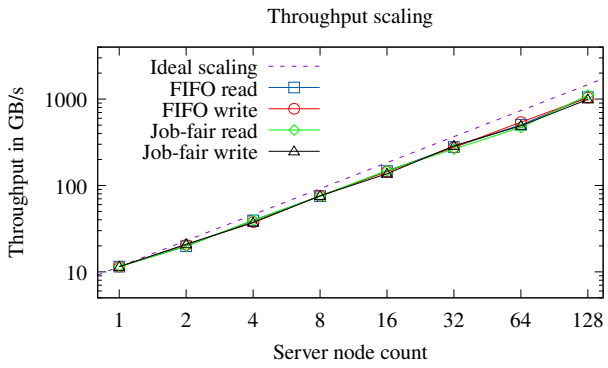


Figure 7: Aggregate throughput when using multiple ThemisIO server nodes

5.3 Sharing with Various Policies

In this section, we present ThemisIO's sharing capability with various primitive and composite policies.

5.3.1 Primitive Policy. Figure 8 shows the results of three primitive sharing policies, namely, **size-fair**, **job-fair**, and **user-fair**. We concurrently run two benchmark programs on different numbers of nodes and report the throughput (MB/s) to examine the sharing efficacy. The benchmark program in these experiments opens one file per process. Each process writes 10 MB of data to its file, then reads it back, and continues to repeat this write/read cycle for a set length of time. Figure 8 demonstrates the measured I/O throughput with samples taken at 1-second intervals. The actual response time of each I/O operation is on the order of 1 microsecond. The two-second delay is a measurement artifact.

Figure 8(a) demonstrates a server running in **size-fair** mode with a 224-process benchmark job running on four nodes competing for server throughput with a benchmark job consisting of 56 processes running on one node. The first job runs for 60 seconds, while the second job run for 30 seconds, starting 15 seconds after the first job starts. The median throughput of the 4-node job when running unopposed is 21.8 GB/sec. The median throughput of the 4-node job and the 1-node job when both are running is 17.4 GB/sec and 4.4 GB/sec, respectively. This represents a throughput ratio of 3.96x, which closely approximates the 4x ratio of job sizes.

Figure 8(b) demonstrates the same pair of benchmark jobs, but with a server running in **job-fair** mode. The first job still consists for four times as many client processes, but the overall throughput for the jobs when both are running is nearly equal, with a median throughput of 10.6 GB/sec.

In Figure 8(c), the server is configured in **user-fair** mode, and three jobs from two users compete for throughput. User A is running two jobs, each of which uses two nodes, while User B is running one job on one node. When all three jobs are running, User A's jobs have a median total throughput of 10.85 GB/sec, which is roughly the same as the throughput of User B's job: 10.80 GB/sec.

5.3.2 Composite Policy. Next, we examine the I/O sharing efficiency with composite policies. In particular, we choose the **user-then-size-fair** and **group-user-then-size-fair** policies.

In the first experiment, we run four jobs owned by two users with different node count. As shown in Figure 9, user-fair sharing is achieved at the first level, as the two jobs of user 1 get a throughput of 10.1 GB/s, while user 2 gets 9.9 GB/s. Looking deeper into the two jobs of user 1, Job 1 gets 3.4 GB/s and Job 2 gets 6.7 GB/s, which matches the ratio between node count of 1:2. Similarly, Job 3 and 4 get 3.9 GB/s and 6.0 GB/s, respectively. It is close to the node count ratio of 4:6. The aggregated throughput is 20 GB/s, which is ~1.7 GB/s lower than the primitive policy. This throughput degradation is caused by the slow startup of ThemisIO when setting up the connections between clients and servers, which is negligible compared to the long runtime of modern supercomputing applications.

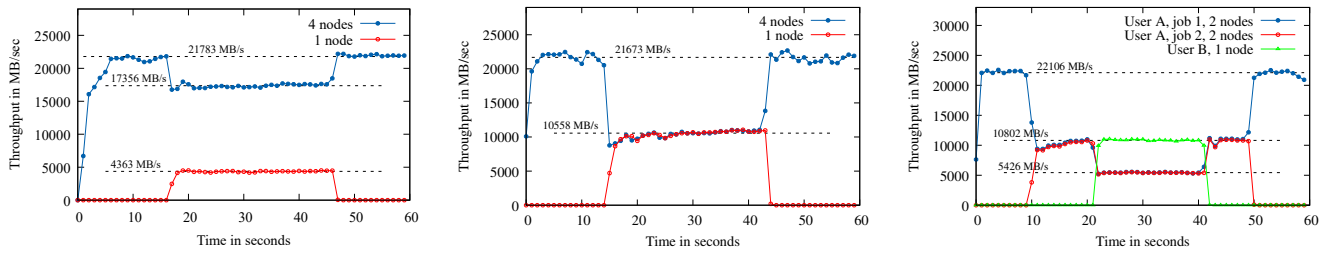
To demonstrate the flexibility of ThemisIO in supporting composite policies, we also implement a three-tier **group-user-size-fair** policy. This policy should enforce an even I/O throughput partition across groups, then across users in each group, allocating the I/O resource among jobs of each user in proportion to the job size. Figure 10 shows the results of the experiment with two groups, four users, and eight jobs, and Figure 11 shows the results as a hierarchy tree. Group 1 gets 9.5 GB/s and Group 2 gets a total of 11.2 GB/s. This is not an exactly even split due to the slow startup in first 10 seconds, as in the previous experiment. However, I/O resource are almost fair-shared after the startup period. Inside Group2, User 2, 3, and 4 get total throughput of 3.8 GB/s, 3.7 GB/s, and 3.7 GB/s, respectively. I/O through is evenly split between the three users. For each user, all jobs get throughput proportional to the job size. That is to say, the three jobs of User 2 get 1.1 GB/s, 1.6 GB/s, and 1.1 GB/s, which is almost the ratio of 2:3:2. The overall throughput is 20.7 GB/s, which is only 1 GB/s lower than the maximum throughput.

In summary, ThemisIO can effectively and efficiently support primitive policies that are as easy as size-fair, job-fair, and user-fair. It is also capable of supporting composite polices such as user-size-fair and group-user-size-fair.

5.4 Comparison with Existing Solutions

In this experiment, we compare the I/O resource sharing performance of ThemisIO with existing solutions of GIFT and TBF using the metrics of overall I/O throughput, latency to fair-sharing, and the standard deviation of I/O throughput.

To understand the performance of ThemisIO vs. that of GIFT, we copy the GIFT core algorithms, BSIP (Basic Synchronous I/O Progress) and the linear programming algorithm, from the GIFT codebase into ThemisIO and replace the I/O resource allocation and throttling mechanisms of LINUX cgroups with the ThemisIO probabilistic token design. GIFT uses pending I/O requests every μ time interval to determine bandwidth allocation. The default setting of μ is ten seconds, which leads to a long delay in I/O resource adjustment. We experiment with a series of μ values and conclude that 0.5 sec is an appropriate interval for our reference implementation. The original implementation of TBF on Lustre directly manages tokens, which are assigned based on I/O request type. Similarly to GIFT, we



(a) Size-fair, 4-node job competing with 1-node job

(b) Job-fair, 4-node job competing with 1-node job

(c) User-fair, Two 2-node jobs competing with a 1-node job

Figure 8: Effectiveness of I/O Resource Sharing with Size-, Job-, and User-fair Policies with Single ThemisIO Server.

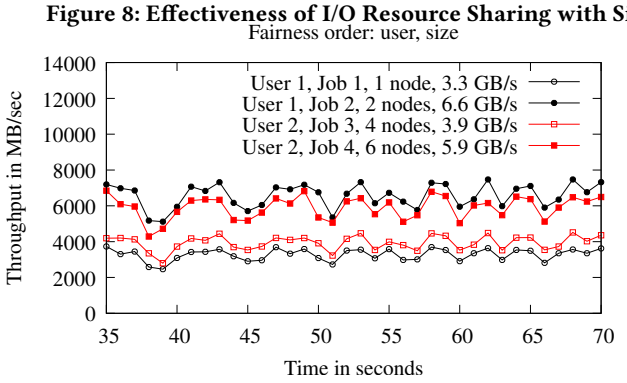


Figure 9: Four competing jobs, two from each user. Total throughput is balanced between users and between jobs belonging to each user.

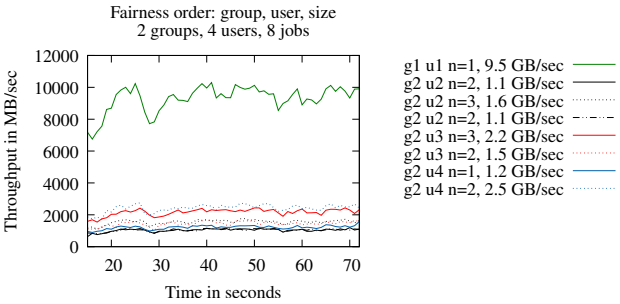


Figure 10: Eight competing jobs from two groups containing four users. Throughput is balanced by group, then by user, then by size.

implement the core HTC (Hard Token Compensation) and PSSB (Proportional Sharing Spare Bandwidth) strategies and integrate them with ThemisIO's I/O resource allocation mechanism.

Figure 12 presents the comparison of ThemisIO with GIFT and TBF using a pair of single node benchmark jobs. In each experiment, Job 1 runs for 60 seconds and Job 2 is started 15 seconds after Job 1 and runs for 30 seconds. ThemisIO runs in job-fair mode. The sustained peak throughput of ThemisIO is 19.8 GB/s, which is 13.5% and 13.7% higher than that of GIFT and TBF, respectively. During the sharing phase, the throughput of Job 2 is 10.2 GB/s, which is 7.9% and 14.7% higher than GIFT and TBF. ThemisIO also shows a

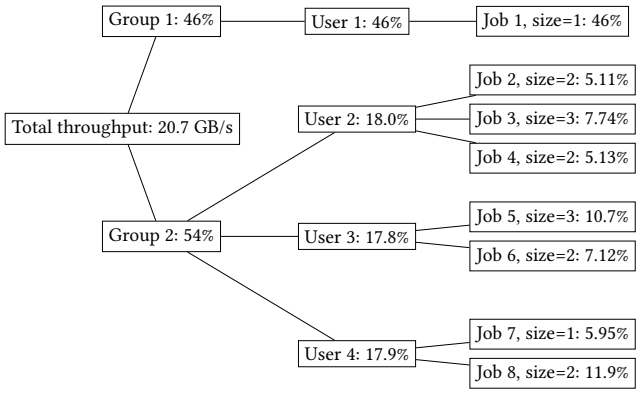
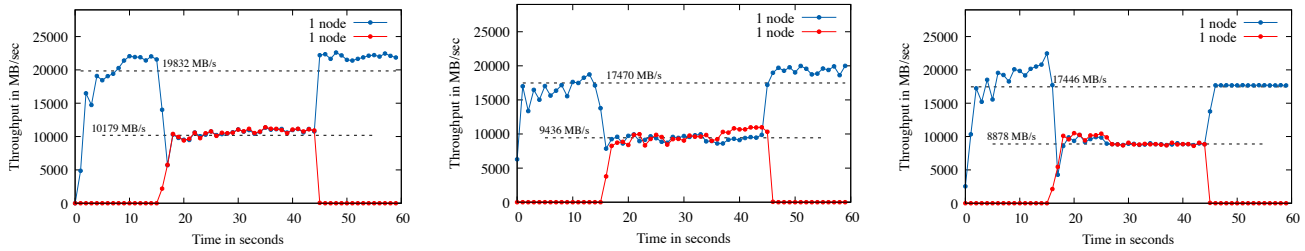


Figure 11: Tree depiction of group-user-size-fair policy experiment. Each node lists the percentage of the overall throughput allocated to that job, user, or group of users. Note that the throughput is approximately balanced across groups and users within a group, and proportional to job size across jobs for each user.

lower standard deviation of the throughput of Job 2 with a value of 504 MB/s, compared with 626 MB/s for GIFT and 845 MB/s for TBF. Compared with existing solutions, ThemisIO shares I/O resources more efficiently and more stably. In addition, GIFT and TBF only support job-fair sharing and require prior knowledge, e.g., the repeated pattern and the I/O rate. In contrast, ThemisIO is more versatile in sharing policies and it gathers all necessary information at runtime from the I/O traffic, which makes I/O resource sharing adaptive to the real workload without requiring user-supplied information.

5.5 Sharing with Applications

In previous experiments, we showed that ThemisIO can correctly and efficiently share I/O resources among jobs with benchmarks. In this experiment, we study the overall impact of policy-driven I/O resource sharing on applications. We run the five applications 1) with exclusive access to a ThemisIO deployment of one server except ResNet-50 which runs with two servers due to space limit, 2) with FIFO policy and a background benchmark job (one compute node), and 3) with size-fair policy and the background benchmark job. We expect that with **size-fair**, ThemisIO can significantly reduce the impact of I/O interference. Figure 13 demonstrates the



(a) Job-fair Sharing with ThemisIO. (b) Job-fair Sharing with GIFT. (c) Job-fair Sharing with TBF.

Figure 12: I/O Resource Sharing Comparison with Existing Solutions with Single ThemisIO Server.

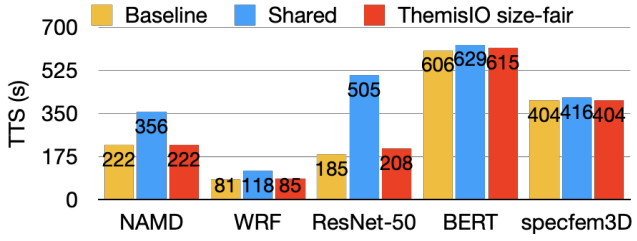


Figure 13: Relative FIFO and size-fair time-to-solution to the baseline with exclusive ThemisIO access. This experiment uses two ThemisIO servers.

performance measurements of the five applications with the three settings. With FIFO and the background job, NAMD, WRF, BERT, and SPECFEM3D are slowed down by 60.6%, 45.3%, 3.8%, and 3.0%, respectively. With size-fair and the background job, the slowdown of each application is 0.1%, 4.6%, 1.6% , and 0.0%. All the slowdowns of size-fair are bounded by its fair share of the I/O resource in proportion to node count. For example, NAMD is run on 64 nodes, so the maximum possible slowdown with a background benchmark job from one node should be $1/65 = 1.5\%$, assuming NAMD is entirely I/O. However, due to the non-trivial computation in NAMD, the measured slowdown is only 0.1%.

The only observed exception is ResNet-50, which uses asynchronous I/O. The 2.7x slowdown with FIFO is reduced to 12.9% with size-fair. The 12.9% slowdown exceeds the maximum possible value of 5.9%. which is calculated as the ratio of the background job size to the total node count of the two jobs (1/17). We believe this is due to the bounding factor change: with asynchronous I/O, ResNet-50 is bounded by the computation and communication. As the I/O latency increases, I/O becomes the dominating factor, which introduces a non-linear increase in time-to-solution. To further validate the effectiveness of size-fair, we change the I/O of ResNet-50 to be synchronous. Although the synchronous I/O is slower than asynchronous I/O (with a 62.1% overhead), the size-fair policy with ThemisIO only introduces a 1.1% slowdown compared to the exclusive case. In contrast, the FIFO policy slows ResNet-50 down by 2.0x.

The results clearly show that ThemisIO can dramatically reduce the impact of I/O interference by proportionally sharing I/O capability with the size-fair policy. With ThemisIO fair-share, these slowdown caused by I/O interference is reduced by 59.1–99.8% across applications. The resulting time-to-solutions are all below the maximum possible slowdown with the appropriate amount of I/O capability, which shows that the delay introduced by ThemisIO sharing policies is bounded.

5.6 λ-delayed Fairness

In §3.1, we introduced λ-delayed fairness to mitigate an imbalanced sharing of I/O resources due to using a local job view. In this experiment, we vary the communication interval (λ) and study the impact of this parameter on the overall system. This experiment has three jobs with associated files spread across two ThemisIO servers exclusively, so ThemisIO starts in a unfair sharing state. Our tests set the communication interval to {10, 50, 200, 500} ms. Figure 14 shows the sharing percentage of each job's I/O usage in the four test cases. Using a communication interval of 50, 200, and 500 ms, ThemisIO reaches global fairness by the second interval. With a 10 ms interval, it takes five intervals for ThemisIO to reach global fairness. One other thing to note is that using a shorter communication interval produces a higher variance in the I/O resources allocated among the jobs, as discussed in §3. We observe that ~50 ms is the effectiveness boundary of ThemisIO on Frontera. Although this boundary depends on the processing speed of the server, the interconnect, and the underlying shared file system, we find the 500 ms communication interval is a reasonable value for real applications and benchmarks.

6 RELATED WORK

Traditional I/O research, such as MPI-IO [26, 27] and ADIOS [17], optimizes individual job performance by intelligently mapping memory region to file system data structures. ZOID [11] is an operating system level I/O component that decouples file and socket I/O and enables customized application I/O interface at scale.

Researchers have noticed interference when sharing I/O resources among applications [18]. In practice, production burst buffer systems, such as DataWarp [8] on the Cori supercomputer, integrate with SLURM, so that users can provision burst buffers with two simple policies: *bandwidth* and *interference*. The *bandwidth policy* allocates burst buffer servers to a job to maximize its I/O throughput. The *inference policy* assigns a minimal number of burst buffer servers to a job, but with exclusive access. Both policies are

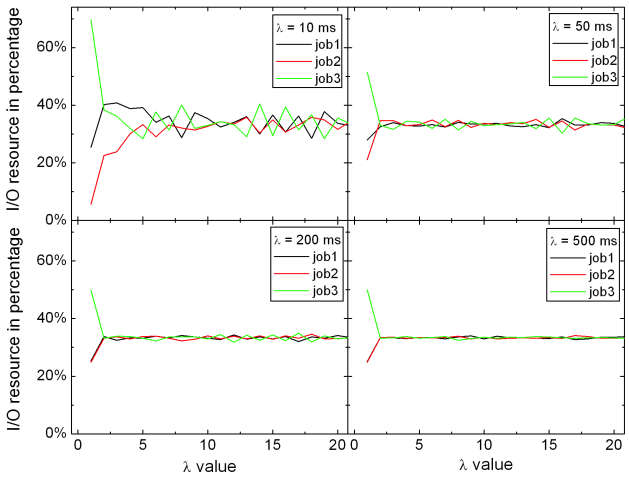


Figure 14: λ-delayed global fairness with various interval lengths (λ).

resource underutilization prone: The *bandwidth policy* can underutilize the allocated I/O resource while the *interference policy* can lead to resource starvation.

Other research has investigated the origin of this interference and proposed different solutions. For example, I/O-aware job scheduling [9] attributes the interference to the aggregated I/O throughput of interconnect switches, then proposes an I/O provisioning algorithm with users' input of their required throughput. This method guarantees that the aggregated bandwidth of jobs does not exceed that of switches; however, multiple jobs can still be placed under the same switch, which processes I/O requests in a FIFO manner and results in possible indefinite blocking of an application due to the lack of isolation. Also, a user can easily specify a higher I/O throughput than the application needs to over-provision resources. Physical isolation [15] maps I/O workloads from different jobs to a disjoint group of file system servers, which provides isolation but can result in low overall efficiency as jobs may not be able to fully utilize the resources. In particular, this approach allocates file system servers based on the output file size specified by users, which can be gamed with boosted values. Liang et al. analyze the impact of I/O process count on the contention problem, and propose the CARS system to map jobs to burst buffer servers to avoid such I/O contention [16]. Similar to I/O-aware scheduling, this approach does not support isolation, and the policy can be tampered with through user input of the I/O process count.

Recent research on I/O forwarding resource sharing such as GIFT [19], TBF [23], and DFRA [12] investigates system design and algorithms to enable efficient and fair sharing of I/O resources. GIFT and DFRA use the fact of 80% of HPC applications are run more than five times. They design a specific throttle-and-reward mechanism and profile-based job placement, respectively. In contrast, ThemisIO assigns I/O resources based on real-time I/O dynamics and is effective for both known and new applications. Similar to ThemisIO, TBF enables I/O resource sharing among compute nodes, jobs, or I/O operations. It requires user-supplied upper and lower I/O request rate and provides QoS accordingly. However, it is difficult to know the exact I/O request rate of an application, even for an experienced user. In addition, the user-supplied request rate may not be accurate.

7 CONCLUSION AND FUTURE WORK

This paper has presented ThemisIO, an automatic, policy-driven, and efficient I/O sharing framework for burst buffer. It enables policy-driven I/O resource sharing and minimizes the impact of I/O interference with a statistical token design to time-slice I/O request processing cycles and assigning cycles based on runtime information of jobs. We introduced λ-delayed fairness to mitigate the sub-optimal sharing problem due to the job information discrepancy. We demonstrate the sharing policy flexibility of ThemisIO with three primitive sharing policies and two composite policies. Our benchmark results show that ThemisIO can correctly and efficiently enforce specified sharing policies to assign I/O resources using various policies. The I/O sharing enabled by ThemisIO sustains a 13.5–13.7% higher I/O throughput and a 19.5–40.4% lower performance variation than existing algorithms. In a controlled environment, ThemisIO significantly reduces or eliminates the application slowdown caused by I/O interference compared to the FIFO baseline. As future work, we are investigating various log-structure byte-addressable file system designs and persistent data structure strategy to enable fault tolerance in ThemisIO.

8 ACKNOWLEDGEMENTS

This work was supported by NSF OAC-2008388 and OAC-2008286.

REFERENCES

[1] Lorenzo Casalino, Abigail C Dommer, Zied Gaieb, Emilia P Barros, Terra Sztain, Surl-Hee Ahn, Anda Trifan, Alexander Brace, Heng Ma, Hyungro Lee, et al. 2020. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. *BioRxiv* (2020).

[2] Tom Charnock and Adam Moss. 2016. Deep Recurrent Neural Networks for Supernovae Classification. *arXiv preprint arXiv:1606.07442* (2016).

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)*. 248–255.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[5] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *SC'99: International Conference for High Performance Computing, Networking, Storage and Analysis*, Vol. 99. 5–33.

[6] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *IEEE International Conference on Computer Vision*. 2961–2969.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[8] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. 2016. Architecture and design of Cray DataWarp. In *Cray User Group meeting*.

[9] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. 2016. Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 69–80.

[10] Galen Hunt and Doug Brubacher. 1999. Detours: Binary interception of Win32 functions. In *3rd USENIX Windows NT Symposium*.

[11] Kamil Iskra, John W Romein, Kazutomo Yoshii, and Pete Beckman. 2008. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 153–162.

[12] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, Xiyang Wang, Nosayba El-Sayed, Jidong Zhai, Weiguo Liu, and Wei Xue. 2019. Automatic, application-aware I/O forwarding resource allocation. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 265–279.

[13] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. 2019. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature* 568, 7753 (2019), 526–531.

[14] Michael Kerrisk and P Zijlstra. 2014. Linux Programmer's Manual. *The Linux man-pages project* 3 (2014).

[15] Anthony Kougkas, Matthieu Dorier, Rob Latham, Rob Ross, and Xian-He Sun. 2016. Leveraging burst buffer coordination to prevent I/O interference. In *2016 IEEE 12th International Conference on e-Science (e-Science)*. IEEE, 371–380.

[16] Weihao Liang, Yong Chen, Jialin Liu, and Hong An. 2019. CARS: A contention-aware scheduler for efficient resource management of HPC storage systems. *Parallel Comput.* 87 (2019), 25–34.

[17] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *6th International workshop on Challenges of Large Applications in Distributed Environments (CLADE)*. ACM, 15–24.

[18] Misbah Mubarak, Philip Carns, Jonathan Jenkins, Jianping Kelvin Li, Nikhil Jain, Shane Snyder, Robert Ross, Christopher D Carothers, Abhinav Bhatele, and Kwan-Liu Ma. 2017. Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers. In *IEEE International Conference on Cluster Computing*. IEEE, 204–215.

[19] Tirthak Patel, Rohan Garg, and Devesh Tiwari. 2020. GIFT: A coupon based throttle-and-reward mechanism for fair and efficient i/o bandwidth management on parallel storage systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 103–119.

[20] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. 2021. KAISA: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[21] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. 2020. Convolutional Neural Network Training with Distributed K-FAC. *International Conference for High Performance Computing, Networking, Storage and Analysis* (2020).

[22] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. 2005. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 26, 16 (2005), 1781–1802.

[23] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 2017. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[24] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[25] William C Skamarock, Joseph B Klemp, and Jimy Dudhia. 2001. Prototypes for the WRF (Weather Research and Forecasting) model. In *Preprints, Ninth Conf. Mesoscale Processes*. Amer. Meteorol. Soc., J11–J15.

[26] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On implementing MPI-IO portably and with high performance. In *Proceedings of 6th Workshop on I/O in Parallel and Distributed Systems*. 23–32.

[27] Rajeev Thakur, Ewing Lusk, and William Gropp. 1997. *Users guide for ROMIO: A high-performance, portable MPI-IO implementation.* Technical Report. Argonne National Laboratory.

[28] Sagar Thapaliya, Purushotham Bangalore, Jat Lofstead, Kathryn Mohror, and Adam Moody. 2016. Managing I/O interference in a shared burst buffer system. In *45th International Conference on Parallel Processing (ICPP)*. IEEE, 416–425.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://zenodo.org/badge/latestdoi/333189409

ARTIFACT IDENTIFICATION

ThemisIO is a first of its kind software-defined I/O system for supercomputers. It enables policy-driven I/O capacity sharing on supercomputers. At its core, ThemisIO disassociates I/O control (i.e., I/O request processing order) from processing by incorporating job metadata such as user, job id, and job size (i.e., node count) ThemisIO can precisely balance the I/O cycles between applications via time slicing to enforce processing isolation, enabling a variety of fair sharing policies. ThemisIO can precisely allocate I/O resources to jobs so that every job gets at least its fair share of the I/O capacity as defined by the sharing policy. ThemisIO can decrease the slowdown of real applications due to I/O interference by two to three orders of magnitude when using fair sharing polices compared to the first-in-first-out (FIFO) baseline.

The open source ThemisIO software can run on supercomputer with x86 architecture, and researchers can use this software to support HPC applications and do further research.

The software was only tested on TACC Frontera supercomputer with both nvdimm and CLX nodes. All experiments in the paper are reproducible.

REPRODUCIBILITY OF EXPERIMENTS

1. This instruction records how to reproduce results in paper. We assume the AD/AE team has access to TACC Frontera. If not, please reach out to zzhang@tacc.utexas.edu and iwang@tacc.utexas.edu. We can facilitate test accounts for AD/AE evaluation.

2. The total time lasts about 30 minutes.

3. The output of the experiments are in the text files. E.g., https://github.com/bbThemis/ThemisIO/blob/main/testfair/size-fair.1vs1.txt. We visualize these numerical results in the figures of the paper.

The complete reproduction instruction is in https://github.com/bbThemis/ThemisIO/blob/main/instructions.md.

4. Step 6 corresponds to Figure 6(a); Step 7 corresponds to Figure 6(b); Step 10 corresponds to Figure 6(C)

To reproduce Figure 7, please follow https://github.com/bbThemis/ThemisIO/blob/main/instructions.md#instructions-for-recreating-throughput-scaling-results

For Figure 19, checkout the "composite-sharing" branch by running: git checkout composite-sharing Then run "test-fair/group_radical.sh" to produce "group_8_jobs.txt" output file.

To reproduce GIFT results (Figure 12(b)), please follow https://github.com/bbThemis/ThemisIO/blob/main/instructions.md#instructions-for-recreating-gift-results-on-themisio.

To reproduce TBF results (Figure 12(c)), please follow ttps://github.com/bbThemis/ThemisIO/blob/main/instructions.md#instructions-for-recreating-tbf-results-on-themisio.

ARTIFACT DEPENDENCIES REQUIREMENTS

1. We use the TACC Frontera supercomputer. 2. CentOS Linux 7 3. Intel MPI 19.0.5, ibverbs, pthreads

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

Compile server and wrapper.so, git clone https://github.com/bbThemis/ThemisIO cd ThemisIO mkdir obj make cd src/client ./compile.sh

You need to revise the impi path in Makefile.

Run a server, cd ThemisIO ./server

Run on client side export MYFS_CONF="/full_path/ThemisIO/myfs.param" export LD_PRELOAD="/full_path/ThemisIO/wrapper.so"

ls -l /myfs touch /myfs/a ls -l /myfs

Run Client within a container Assume you are on a system with the server running already (see above for instructions of launching a server), you may run the client within our pre-built container. To pull the container with Docker, run:

1. docker pull ghcr.io/bbthemis/themisio:client This can also be done with other container runtimes, for example the Apptainer:

1. apptainer pull themisio.sif docker://ghcr.io/bbthemis/themisio:client Then, you may run a shell within the container (using apptainer as example here):

1. apptainer run themisio.sif bash You can then access the filesystem with:

1. export MYFS_CONF="/full_path/ThemisIO/myfs.param" 1. export LD_PRELOAD="/ThemisIO-client/wrapper.so" 1. ls -l /myfs 1. touch /myfs/a 1. ls -l /myfs You may also build a new container using our client container as the base to run any applications with ThemisIO support.