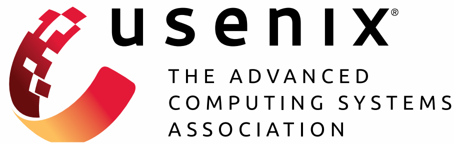


GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems

Tirthak Patel, *Northeastern University;* Rohan Garg, *Nutanix;*
Devesh Tiwari, *Northeastern University*

https://www.usenix.org/conference/fast20/presentation/patel-gift

This paper is included in the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
February 25–27, 2020 • Santa Clara, CA, USA
978-1-939133-12-0

Open access to the Proceedings of the
18th USENIX Conference on File and
Storage Technologies (FAST '20)
is sponsored by

GIFT: A Coupon Based Throttle-and-Reward Mechanism for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems

Tirthak Patel
Northeastern University

Rohan Garg
Nutanix

Devesh Tiwari
Northeastern University

Abstract

Large-scale parallel applications are highly data-intensive and perform terabytes of I/O routinely. Unfortunately, on a large-scale system where multiple applications run concurrently, I/O contention negatively affects system efficiency and causes unfair bandwidth allocation among applications. To address these challenges, this paper introduces GIFT, a principled dynamic approach to achieve fairness among competing applications and improve system efficiency.

1 Introduction

Problem Space and Gaps in Existing Approaches. Increase in computing power has enabled scientists to expedite the scientific discovery process, but scientific applications produce more and more analysis and checkpoint data, worsening their I/O bottleneck [7,45]. Many applications spend 15-40% of their execution time performing I/O, which is expected to increase for exascale systems [12,15,22,31,53,55]. Unfortunately, multiple concurrent applications on a large-scale system lead to severe I/O contention, limiting the usability of future HPC systems [11,45].

Recognizing the importance of the problem, there have been numerous efforts to mitigate I/O contention from both I/O throughput and fairness perspectives [13, 14, 17, 25, 37, 42, 75, 76, 78, 88, 89]. Unfortunately, ensuring fairness and maximizing throughput are conflicting objectives, and it is challenging to strike a balance between them under I/O contention. For parallel HPC applications, the side-effect of I/O contention is further amplified because of the need for *synchronous I/O progress*. HPC applications are inherently tightly synchronized; during an I/O phase, MPI processes of an HPC application must wait for all processes to finish their I/O before resuming computation (i.e., synchronous I/O progress among MPI processes is required) [28,31,39,57,90].

MPI processes of an HPC application perform parallel I/O access to multiple back-end storage targets (e.g., an array of disks) concurrently. These back-end storage targets are shared among concurrently running applications and have different degree of sharing over time and hence, a varying level of contention. A varying level of I/O contention at the shared back-end parallel storage system makes different MPI processes progress at different rates and hence, leads to non-synchronous I/O progress. In Sec. 2, we quantify non-synchronous I/O progress as a key source of inefficiency in shared parallel storage systems. It results in (1) wastage of compute cycles on compute nodes, and (2) reduction in effective system I/O bandwidth (i.e., the bandwidth that contributes toward synchronous I/O progress), since full bandwidth is not utilized toward synchronous I/O progress.

Recent works have noted that non-synchronous I/O progress degrades application and system performances on modern supercomputers like Mira, Edison, Cori, and Titan [9,31,32,39,69,83]. Thus, there is an emerging interest in improving the quality-of-service (QoS) of parallel storage systems [24,80,86]. Previous works have proposed rule-based or ad-hoc bandwidth allocation strategies for HPC storage [14,17,23,36,42,88,89]. However, existing approaches do not systematically implement synchronous I/O progress to balance the competing objectives: improving effective system I/O bandwidth and improving fairness.

To bridge this solution gap, *this paper describes GIFT, a coupon-based bandwidth allocation approach to ensure synchronous I/O progress of HPC applications while maximizing I/O bandwidth utilization and ensuring fairness among concurrent applications on parallel storage systems.*

Summary of the GIFT Approach. GIFT introduces two key ideas: (1) *Relaxing the fairness window:* GIFT breaks away from the traditional concept of instantaneous fairness at each I/O request, and instead, ensures fairness over multiple I/O phases and runs of an application. This opportunity is enabled by exploiting the observation that HPC applications have multiple I/O phases during a run and are highly repetitive, often exhibiting similar behavior across runs; and (2) *Throttle-and-reward approach for I/O bandwidth allocation:* GIFT opportunistically throttles the I/O bandwidth of certain applications at times in an attempt to improve the overall effective system I/O bandwidth (i.e., it minimizes the wasted I/O bandwidth that does not contribute toward synchronous I/O progress). GIFT's throttle-and-reward approach intelligently exploits *instantaneous* opportunities to improve effective system I/O bandwidth. Further, relaxing the fairness window enables GIFT to reward the "throttled" application at a *later point* to ensure fairness.

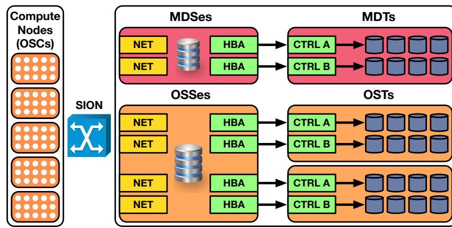


Figure 1: Overview of HPC storage system architecture.

First, GIFT allocates I/O bandwidth to all competing applications in a fair manner and ensures synchronous I/O progress among all processes of the same application at *all times* - this fundamental design principle eliminates the key source of parallel storage system inefficiencies (Sec. 3.1). This allows GIFT to estimate the amount of wasted I/O bandwidth (i.e., bandwidth which does not contribute toward the synchronous I/O progress). Then, GIFT exploits the "opportunity" to reduce the bandwidth waste by identifying and throttling the I/O bandwidth share of some applications and expanding the I/O bandwidth share of other applications (Sec. 3.2). To minimize the I/O bandwidth waste, GIFT uses constraint-based, linear programming to optimally allocate bandwidths to applications (Sec. 3.4). GIFT issues "coupons" to the throttled applications – the worth of these coupons is proportional to the degree of throttling. At a later point, GIFT "redeems" the previously issued coupons to throttled applications to ensure fairness (Sec. 3.3). In cases where GIFT cannot redeem issued coupons for an application, it rewards the application with proportional compute node-hours (credited from a bounded "system regret budget"). This system regret budget acts as a credit bank of compute node-hours, which GIFT uses to achieve fairness when coupons cannot be redeemed.

The contributions of GIFT include:

Design and Implementation. GIFT designs and develops an efficient and practical coupon-based management system for I/O bandwidth allocation among competing applications on shared parallel storage systems. GIFT develops new lightweight and effective techniques to identify throttle-friendly applications, determine the degree of throttling and expansion of I/O bandwidth share of competing applications, and redeem coupons to ensure fairness. GIFT shows that the usage of the "system regret budget" upon failure to redeem coupons is minimal, and that the compute node-hours required for the system regret budget are much less than compared to the increase in system throughput due to faster I/O. GIFT implements all the core ideas in a real-system prototype based on the FUSE file system, demonstrating that GIFT's' ideas can be realized in practice, open to the community for reproducibility, and do not require heroic optimization efforts or system-specific parameter tunings to realize the performance gains. GIFT is available at `https://github.com/GoodwillComputingLab/GIFT`.

Evaluation of GIFT. Our evaluation confirms that GIFT reduces the "bandwidth waste" caused by I/O contention on a HPC storage system, and thereby, improves the I/O bandwidth utilization toward synchronous I/O progress, application performance and fairness, and system job throughput. Our evaluation is based on extensive real system experimental results, guided by real-world, large-scale HPC system and application parameters, and supported by simulation results. GIFT is shown to improve the mean effective system I/O bandwidth by 17% and the mean application I/O time by 10%, compared to multiple competing schemes. GIFT is also shown to be effective under various scenarios including high contention levels and different application characteristics.

2 Background and Motivation

HPC Storage Systems. This section describes the key components of storage systems attached to large-scale HPC systems, such as Mira, Edison, Titan, Cori, and Stampede2 [1,22, 54,73]. HPC systems use parallel file systems, such as Lustre, Ceph, GPFS, and PVFS, to perform parallel I/O [58–60,79]. For simplicity, this works targets widely-used Lustre-like HPC storage system. A Lustre-like architecture consists of multiple building blocks (Fig. 1). The most basic of these is an Object Storage Target (OST), a RAID array of disks. A file is typically distributed across multiple OSTs for parallelism and can be accessed in parallel from multiple MPI processes. The OSTs serve the Object Storage Servers (OSS), which are connected to the front-end compute nodes via an I/O network. Applications running on compute nodes communicate with the OSSes via file system clients. The Meta Data Server (MDS) is the starting point for all file metadata operations. MDS consults with the Meta Data Targets (MDT), which maintain the metadata of all I/O requests.

Day in the Life of an I/O Request in a HPC System. Large-scale applications run on multiple nodes and spawn multiple (MPI) processes. These processes periodically write (or read) analysis output and checkpoint data to (or from) the storage system – referred to as an I/O phase. Processes from the same application may perform I/O on separate files or stripe a single file across multiple OSTs for concurrent access [8].

We refer to an I/O operation (read/write) accessing one OST from an MPI process of an application as an I/O request. First, the file system client on the compute node issues a remote procedure call (RPC) to the MDS, which returns information about the file stripe and OST mappings. For a new file creation request, the MDS first assigns OSTs in a capacity-balanced manner. For existing files, the MDS returns previously assigned OST information to the file system client. Then, the file system client issues an I/O request over the network to the OSS corresponding to the target OST [81]. In practice, during the I/O phase, an HPC application issues multiple I/O requests from different MPI processes.

Table 1: I/O characteristics of large-scale HPC applications.

	< 1 min	1-15 mins	> 15 mins
I/O Phase Length	HACC [63], Chombo-Crunch [52]	HIMMER [63], Chombo-Crunch [52] WRF [48], S3D [30, 33]	PTF [32], VPIC [9], Plasma Based Accelerators [19]
	< 5 min	5-30 mins	30 mins - 3hr
I/O Interval	GTC [33], Titan Apps [39], GYRO [33]	WRF [48], S3D, Chombo-Crunch [52], Titan Apps [39]	VPIC [9], CHIMERA [33], Chombo-Crunch [52], VULCAN [33]
	< 100 GB	100 GB - 1 TB	> 1 TB
I/O Output Size	GTC [33], POP [33], GYRO [33]	WRF [48], VULCAN [33], Titan Apps [39], HACC [63]	VPIC [9], XGC1 [57], HIMMER [63], S3D [30, 33]

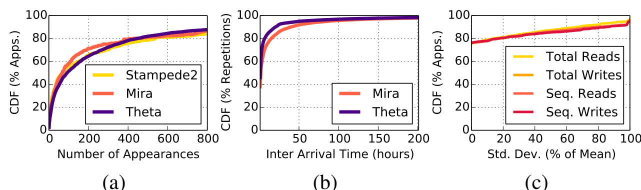


Figure 2: CDF of the (a) number of times that applications make appearances, (b) inter arrival times between each appearance, and (c) variation of I/O characteristics between two appearances.

I/O Phases of HPC Applications. HPC applications are typically long-running and perform I/O at regular intervals [28, 31, 39, 57, 90]. Their execution time ranges from a few hours to a few weeks [4, 5, 33, 57, 62, 83], and the compute period between two I/O phases can be from minutes to hours [9, 33, 39, 48, 52]. The I/O phases typically produce large amounts of data (up to hundreds of GBs) in the form of checkpoints and post-simulation results [8, 9, 33, 39, 48, 57, 62, 63]. Table 1 highlights the I/O characteristics of some popular HPC applications collected from multiple supercomputers. It shows that I/O phases can be as long as 30 min and the I/O interval (compute period) can be between 5 min and 3 h. Also, large amounts of data (100 GB - 5 TB) are transferred during each I/O phase. Next, we discuss some HPC I/O observations.

Observation 1. *HPC applications are highly repetitive in nature – that is, HPC applications typically run repeatedly and exhibit similar I/O behavior across their execution instances, though different applications have different I/O behavior.* Previous studies have shown that many HPC applications execute multiple times with similar execution characteristics [4, 5, 12, 22, 62, 63]. This is because scientific applications often model and simulate physical phenomena. This is an iterative process and requires repeated simulations for model refinement. Analysis of job scheduler logs for the last five years, two years, and one year from the leading supercomputers (Mira, Theta, and Stampede2) shows strong repetition (Fig. 2). More than 40% of the applications appear more than 200 times and about 15% of the applications appear more than 1000 times. Only less than 20% of the applications are run less than 5 times. Interestingly, we also found that the inter-arrival times between re-occurrences of HPC applications is relatively short on Mira and Theta (inter-arrival times

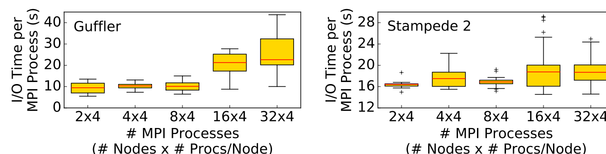


Figure 3: I/O variability among I/O performing processes of an HPC application on two HPC systems.

for Stampede2 were unavailable) (Fig. 2(b)). In fact, 80% of repetitions occur within 24 hours of each other.

Furthermore, Fig. 2(c) shows that applications exhibit only a small variation in their I/O characteristics across repetitions. This data was obtained by instrumenting HPC applications with Darshan on the Mira supercomputer [63, 83]. More than 80% of the applications that repeat more than five times show less than 5% standard deviation (as % of mean) in total amount of data read and written. We observe similar trends for different types of I/O requests (sequential and random).

Unfortunately, a shared storage back-end with no contention mitigation strategies results in severe contention among competing HPC applications [10, 28, 34, 47, 81, 85]. The I/O contention issue is further exacerbated by the need for *synchronous I/O progress* in HPC applications – an MPI process of an HPC application, exiting from an I/O phase, must wait for the slower processes to also finish their I/O [28, 31, 39, 57, 90]. Previous studies have noted that OSTs are the most contended resource on the I/O storage path (i.e., compute node, I/O routers, and OSSes) [10, 34, 47, 81, 85], since they have the lowest bandwidth among the different resources. *We note that the Meta Data Server (MDS) attempts to capacity-balance the OSTs by mapping files uniformly across OSTs, but since the MDS has no knowledge of future access patterns, its decisions cannot avoid runtime I/O contention on OSTs caused due to access patterns.* Next, we provide experimental evidence to demonstrate the impact of I/O contention and how it affects synchronous I/O progress of HPC applications.

Observation 2. *MPI processes from the same application experience significantly different I/O progress during an I/O phase – resulting in non-synchronous I/O progress across processes. This problem cannot be solved by simply identifying and speeding up a straggler process.* To demonstrate the effects of non-synchronous I/O progress, we performed a set of IOR benchmark [41] experiments on a local, production HPC system, Engaging. Engaging consists of over 100 compute nodes, and runs a production Lustre parallel file system with 44 OSTs, 44 OSSes, and 1 MDS. We ran IOR with different number of MPI processes, with each MPI process writing to a different OST. Other concurrently running applications were not controlled. We performed these experiments multiple times and from different compute nodes to eliminate transient and spatial biases. From Fig. 3, we observe that the

I/O time of different MPI processes can vary significantly (up to 4x) across runs and the number of nodes (2-32 nodes, with 4 MPI processes per node). This non-synchronous I/O progress is attributed to the difference in degrees of contention encountered by different MPI processes on their respective OSTs. Similar experiments on Stampede2 showed up to 83% variation in I/O time. Previous studies have reported similar results on non-synchronous I/O progress of MPI processes on other large-scale supercomputers including Cori, Mira, Edison, and Hopper [9, 40, 63, 83]. On further analysis, we discovered that often different processes finish at very different speeds (covering a large spectrum), and the ordering of processes in terms of their completion time changes significantly across different runs, because the I/O contention at different OSTs changes over time. *This shows that the non-synchronous I/O progress problem is not the same as the traditional straggler problem – and hence, cannot be solved by simply identifying and speeding up a straggler MPI process or OST.*

Observation 3. *Non-synchronous progress among MPI processes is caused due to unmanaged, varying I/O contention at the OSTs in the HPC storage back-end. Naïve strategies to ensure synchronous I/O progress cannot find the right balance between competing objectives: maximizing effective I/O bandwidth and fairness among applications.* To further analyze the I/O contention behavior, we ran another set of IOR experiments on Engaging, measuring the observed I/O bandwidth at each OST. Each experiment consists of writing to a particular OST from one process. Fig. 4 shows the contention (defined as the inverse of bandwidth) faced on a few OSTs (other OSTs show similar trends). Results of this simple experiment show that the degree of contention is different on each OST and varies over time. Unfortunately, allocating I/O bandwidth among competing applications to achieve conflicting objectives (fairness, effective system I/O bandwidth, synchronous I/O progress) is non-trivial. To achieve fairness, POFS (Per-OST Fair Share) scheme allocates I/O bandwidth to all competing applications equally on each individual OST (as shown in Fig. 5). But, this fair scheme may generate non-synchronous I/O progress and lead to lower effective system I/O bandwidth (i.e., sum of all bandwidths that contribute toward synchronous I/O progress). For example, under POFS, a part of the bandwidth assigned to all applications on OST3 and a part of the bandwidth assigned to A on OST1 are wasted. This is because additionally allocated bandwidths do not contribute toward synchronous I/O progress.

To ensure synchronous I/O progress, one can allocate bandwidth on each OST determined by the fair allocation on the bottlenecked OST. In Fig. 5, BSIP (Basic Synchronous I/O Progress) scheme performs such an allocation. Essentially, BSIP scheme allocates the I/O bandwidth to an application as determined by its most contended OST (e.g., A's allocations on other OSTs is determined by its bottlenecked or the most-contended OST (i.e.,OST2)). Unfortunately, this scheme also creates bandwidth gaps on less contended OSTs and lowers effective system I/O bandwidth because the bandwidth share is limited by the most-contended OST. On the other hand, a greedy approach to minimize bandwidth gaps by preferentially allocating bandwidth to applications that maximize effective system I/O bandwidth, while still ensuring synchronous I/O progress results in unfair allocations. Fig. 5 illustrates such a scheme, referred as MBW (Minimum Bandwidth Wastage), which minimize bandwidth gaps by allocating more bandwidth to certain applications and unfairly hurting other applications (e.g., it reduces the bandwidth share of application E to zero in Fig. 5). In summary, allocating I/O bandwidth among competing applications presents challenging trade-offs and GIFT strikes a balance between them as described in the next section.

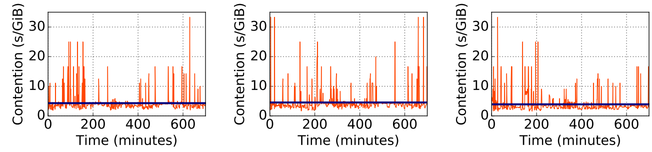


Figure 4: I/O contention on 3 of the 44 OSTs on Engaging (blue line indicates the mean contention level).

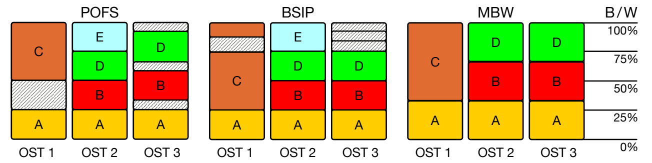


Figure 5: Bandwidth allocation among five applications spanning on three OSTs with (1) Per-OST Fair Share (POFS), (2) Basic Synchronized I/O Progress (BSIP), and (3) Minimum Bandwidth Wastage (MBW) schemes. Checkered boxes indicate bandwidth waste (not contributing toward synchronous I/O progress).

3 GIFT: Design and Implementation

3.1 Overview of GIFT

First, GIFT enforces synchronous I/O progress among processes of an application by allocating bandwidth using the BSIP scheme (Fig. 5). BSIP determines the bandwidth allocation to an application according to its most contended OST. As shown in Fig. 5, BSIP scheme can create bandwidth gaps on OSTs, GIFT attempts to "fill" these bandwidth gaps by carefully throttling the bandwidth share of some applications and expanding the bandwidth share of some other applications, such that a net gain in the overall effective system I/O bandwidth is achieved. This requires identifying which applications to throttle, when to throttle, whom to expand, and how to compensate throttled applications for fairness. GIFT uses a simple and low-overhead approach to dynamically identify "throttle-friendly applications": applications which GIFT can throttle with high confidence of rewarding the stolen band-

width at a later point. The later point could be during the same I/O phase, a later I/O phase during the same run, or a future run of the same application (Sec. 3.2). GIFT issues "coupons" to throttled application which can be redeemed at later points. At regular intervals (also referred as "decision instance"), GIFT considers all throttle-friendly applications (i.e., applications which can redeem a high fraction of issued coupons - "high redemption rates") and solves a linear programming (LP) based optimization problem to maximize the effective I/O bandwidth (Sec. 3.4). This step determines which applications are throttled, which ones are expanded, and by how much. Expanded applications (which can also include throttle-friendly applications) get more than their fair share of the bandwidth, which reduces the bandwidth wastage.

Finally, GIFT bounds the unfairness toward throttle-friendly applications by using a dynamic limiting strategy (Sec. 3.2). GIFT periodically assess its fairness and compensates for the unfair treatment in the form of compute time (i.e., node-hours on the HPC system). GIFT also bounds the node-hours given out to a maximum specified "system regret budget" of compute node-hours. Algorithm 1 outlines the steps that GIFT takes at the start of every decision instance.

Algorithm 1 GIFT Decision Algorithm.

1: $X \leftarrow$ All apps performing I/O
2: $\forall i \in X$, Determine fair share of bandwidth as per $b_{i,bsip}$
3: Redeem previously issued coupons if possible (Sec. 3.3)
4: $\uparrow$ Redemption rate of apps with redeemed coupons
5: Determine the set of throttle-friendly apps Y (Sec. 3.2)
6: Allocate bandwidth using LP optimization (Sec. 3.4)
7: Issue coupons to throttled apps $\subseteq Y$
8: $\downarrow$ Redemption rate of apps with issued coupons

3.2 Identifying Throttle-friendly Applications

To identify throttle-friendly applications, GIFT throttles, issues coupons, and observes the coupon redemption rate of throttled applications. Redemption rate can be estimated with high accuracy if the whole system state (e.g., information about all concurrently running applications, their OST mapping, I/O phase length, etc.) is stored with every coupon issuance and redemption event. However, this can impose a high storage and access overhead. Also, note that, some application's OST-level I/O behavior might change over a long period (e.g., the number of OSTs, and OST mappings), causing the application's throttle-friendly status to change.

Therefore, GIFT uses the concept of receding window at the application-level that captures the recent history of an application's coupon redemption behavior (Sec. 3.5 and 4 show it is both lightweight and effective). The recent coupon redemption behavior of an application is estimated at the start of every decision instance by taking the ratio of the coupons redeemed to the last N coupons issued, where N denotes the length of the receding window (Table 2). For fairness and simplicity, length of the receding window (N) is kept the same for all applications, although each application may take a different amount of time to accumulate N coupons depending upon its OST mappings, I/O phase length, and system I/O contention level, etc. At the start of decision instance, k, the coupon redemption rate of an application i is expressed as $c_i(k) = n_i(k)/N$, where $n_i(k)$ is the number of coupons redeemed (out of N) by application i. GIFT considers an application throttle-friendly, if its redemption rate is greater than a set threshold τ: $Y(k) = \{i \in X(k),\ \text{if}\ c_i(k) \geq \tau\}$, where $X(k)$ is the set of all applications performing I/O and $Y(k)$ is the set of throttle-friendly applications. As the receding window moves forward, more coupons are issued only until $c_i(k) \geq \tau$. Once the redemption rate breaches the τ limit, GIFT avoids issuing more coupons to the application until it redeems its existing coupons and its redemption rate goes above τ. Using this method, GIFT ensures that unfairness is bounded for each application in case the application's redemption rate cannot go over the threshold. GIFT gives out compute node-hours as regret for unfairly treated applications periodically - this period is referred as "regret assessment period" and, as Sec. 4 shows, it can be much larger to allow applications sufficient time for redeeming the coupons.

Table 2: GIFT model parameters.

N	Length of the receding window of applications (unit: number of coupons issued)
τ	Minimum redemption rate required for an application to be eligible for throttling and for the system to throttle applications (unit: ratio)
B_{thres}	Upper threshold of the factor by which each application's I/O request can be throttled

Throttling applications based on threshold-based redemption rate at the application-level helps constrain the "regret" the system experiences from giving out node-hours (out of the system's regret budget) for unfair treatment toward one single application. But, in a system with multiple applications, the system's "cumulative" regret in terms of compute node-hours given to *all* applications can still grow sufficiently large. To address this challenge, GIFT employs a receding window at the system-level too, where it tracks the aggregate redemption rate of coupons issued by the system to *all* the applications, in order to minimize the "system regret budget" level. GIFT makes sure that the system only hands out coupons until its redemption rate is above τ (same threshold as the one used for the applications). However, unlike applications' redemption rates, GIFT resets the system's redemption rate at the end of each regret assessment period. This prevents the system's redemption rate from being saturated at τ because of non-throttle-friendly applications which never get redeemed, which can cause GIFT to miss the opportunity of throttling even throttle-friendly applications. Our evaluation (Sec. 4) shows that GIFT's approach of using τ at the system- and application- level helps keep the outstanding node-hours

("system regret budget") to a reasonably low level (e.g., less than 7% of the total gain in compute node-hours obtained via system throughput improvement due to GIFT). We also observed that keeping the same τ for applications and system is simple and effective; a higher τ at the system-level does not yield additional improvements.

Finally, we note that GIFT carefully chooses the length of receding window (N) to balance competing trade-offs: bound on unfairness toward applications vs. stability of application's status (throttle-friendly or non-throttle-friendly). If N is too large, it increases the upper bound on unfairness toward individual applications (i.e., possibility of higher number of coupons that cannot be redeemed). If N is small, an application's redemption rate $c_i(k)$ can vary erratically as the window glides, and the application's status can toggle frequently between throttle-friendly and non-throttle-friendly. GIFT achieves stable behavior by maintaining the variance of the mean redemption rate of the receding window to be small. For samples within a given receding window, the maximum variance occurs when half of the coupons can be successfully redeemed, and the other half cannot be redeemed. Hence, the maximum possible variance is $v^2 = 0.25$ (independent of N). The variance of the mean redemption rate is defined as $\sigma^2 = \frac{v^2}{N}$, which is bounded by $\sigma^2 \leq \frac{0.25}{N}$. Statistically, σ less than 0.001 can achieve reasonable stability [49]. GIFT's choice of receding window length is guided by this principle. In fact, GIFT's evaluation demonstrates that its improvements are not sensitive to the choice of parameters N (receding window size) and τ (redemption rate threshold), and that GIFT performs effectively well without the need to fine-tune.

3.3 Coupon Redemption Policy

Recall that redeeming previously issued coupons is critical to ensuring fairness. GIFT does not simply attempt to redeem an application's coupons the very next I/O phase after they were issued. This is because if redeeming a coupon requires throttling another application, then it would lead to a zero-sum result in terms of improvements in efficiency (e.g., effective system bandwidth). Thus, GIFT redeems coupons only when it does not require throttling applications. Before performing optimal bandwidth allocation and picking applications to throttle, GIFT first attempts to redeem coupons of previously throttled applications (Algorithm 1 line 3).

Coupons are redeemed when GIFT finds gaps on the OSTs on which a coupon-bearing application is running. After making the basic fair synchronous-I/O progress (BSIP) bandwidth allocation, GIFT searches through the coupon database of active applications. If all of the OSTs on which the coupon-bearing application is performing I/O have a bandwidth gap, then the coupon is redeemed either partially (if the gap is less than the coupon value) or fully or multiple coupons can also be redeemed (if the gap is large enough). By redeeming coupons in this manner, GIFT avoids throttling other applications. Also, GIFT, by design, allows coupons to be issued and redeemed on different OSTs for any given application.

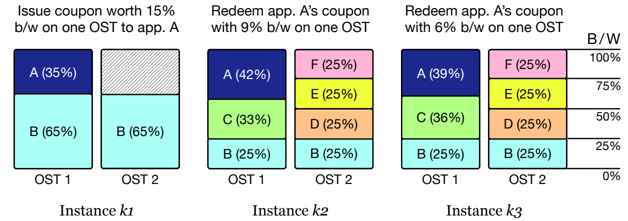


Figure 6: GIFT redeems coupons in a manner which is fair and efficient, without throttling other applications.

GIFT intelligently allocates spare bandwidth toward redeeming coupons to maintain fairness and efficiency. We note that redeeming coupons without throttling other applications requires availability of "spare I/O bandwidth". One may reason that since spare I/O bandwidth is available, applications would have naturally been allocated higher I/O bandwidth allocation, irrespective of GIFT's I/O bandwidth allocation policies. Consequently, why should GIFT refer to this additional allocated I/O bandwidth as "coupon redemption" and claim this as a mechanism to achieve fairness? Below, we discuss a simple example to illustrate the wide range of choices to allocate spare I/O bandwidth. But, GIFT carefully allocates this spare bandwidth such that (1) it redeems previously issued coupons (i.e., maintains fairness over longer term), but (2) without throttling any application at the current decision instance, otherwise it would cause more unfairness and lead to a zero-sum result in terms of efficiency.

As shown in Fig. 6, let us consider a simple example: two OSTs and bandwidth allocation decisions at three decision instances ($k1$, $k2$ and $k3$). At instance $k1$, OST1 is shared by two applications (A and B), but OST2 is only serving application B. The fair share of application A is 50% on OST1. But, if A was given its fair share on OST1, then half of the bandwidth on OST2 would be wasted since it would have not contributed toward synchronous I/O progress even if it was allocated to application B. Therefore, GIFT decides to throttle application A to reduce the overall I/O bandwidth waste. Application A's share on OST1 is reduced to 35% and a corresponding coupon is issued, and application B's share on both OSTs is increased to 65% which results in 15% reduction in I/O bandwidth waste on OST2.

At instance $k2$, OST1 is shared by three applications (A, B, and C), and OST2 is now shared by four applications (B, D, E and F). Note that application B's bandwidth share is decided by its bottlenecked OST (OST2). Application B's share on OST1 and OST2 is 25% – this ensures synchronous I/O progress and is not unfair to application B and other applications on OST1 or OST2. Due to application B's bottleneck on OST2, 9% of spare bandwidth is available on OST1. The fair share for application A and C on OST1 is 33% each. A GIFT-less approach that does not issue coupons to maintain fairness over longer time windows, would equally divide this spare

bandwidth on OST1 (9%) to both application A and C. However, GIFT decides to allocate this spare bandwidth fully to application A (increases its share to 42%, *partially* redeeming a coupon issued to application A at instance $k1$). Application C is still treated fairly even though it is not allocated any part of the spare bandwidth. Application C's fair share was 33% and it still receives it. At instance $k3$ (same OST sharing scenario as instance $k2$), application A receives 6% of the spare bandwidth (completely redeeming the coupon issued at $k1$) and the remaining 3% bandwidth can be allocated in any way (it is allocated to application C in this case).

In summary, application A was throttled in the past to increase the effective system I/O bandwidth utilization. Application A was kind then, and is later picked to receive the reward (larger share in the available spare bandwidth), without being unfair to C or throttling any other application below its fair share. This way, GIFT's decision to throttle A in the past proves to be useful. Using a throttle-and-reward approach, GIFT reduces the overall bandwidth utilization over these three time steps, while ensuring fairness to other applications and maintaining synchronous I/O progress. A GIFT-less BSIP approach (instantaneous fairness and synchronous I/O ensuring allocation at each decision instance but without throttle-and-reward approach) would have been fair but incurred 50% bandwidth waste on OST2 at instance $k1$; in comparison GIFT incurs only 35% bandwidth waste, while remaining fair over multiple decision instances. These are the kind of opportunities that GIFT detects and exploits. Such situations are not deterministic or predictable, which is why GIFT learns using the concepts of redemption rate and system regret budget.

Lastly, we note that GIFT can track coupon issuance and redemption at the user-level if the same application is being shared across multiple users and maintaining fairness at the user-level is deemed more appropriate. This will simply require including and tracking different types of identifiers per I/O request. GIFT can be extended to support different variations of "fair share" instead of being limited to treating all applications equally important. This can be achieved via encoding and tracking relative priority levels, or weights.

3.4 Optimal Bandwidth Allocation

Once a set of throttle-friendly applications is determined and coupons are redeemed, GIFT proceeds to make the bandwidth allocations to maximize the effective bandwidth. Inputs to this step include the set of throttle-friendly applications, the set of all applications concurrently performing I/O, and the set of OSTs being used by each application.

First, GIFT calculates the fair share of each application on the OSTs it is performing I/O on, to ensure synchronous I/O progress. These allocations are the same as in the BSIP scheme (Fig. 5). Next, GIFT maximizes the effective I/O bandwidth by adjusting the bandwidth of all applications subject to multiple constraints: (1) only throttle-friendly applications are allowed a lower bandwidth assignment than their fair share, (2) the total effective bandwidth is always equal to or greater than what is achieved by the BSIP scheme, and (3) *the gains from reducing the bandwidth wastage should be more than the worth of issued coupons (i.e., bandwidth waste with BSIP - bandwidth waste with GIFT > aggregate worth of coupons)*. GIFT formulates and solves this problem as a constraint-based, linear programming (LP) bandwidth allocation optimization problem, as discussed below.

Bandwidth allocation LP optimization: GIFT accounts for constraints from both, the applications' and system's perspectives. For the applications, at each decision instance k:

- All I/O requests (r_i) of application i issued across all assigned OSTs (S_i) should get the same bandwidth in order to facilitate synchronized I/O progress, i.e., for application i, $b_{ij} = b_i \forall j \in S_i$, where b_{ij} is bandwidth allocated to application i's I/O request on OST j and b_i is the bandwidth allocated to application i's I/O request running on the most contented OST.
- The final bandwidth allocation b_i should be s.t.

  (a) $b_{i,bsip}(1 - B_{thres}) \leq b_i \leq 1 \quad \text{if } i \in Y$
  (b) $b_{i,bsip} \leq b_i \leq 1 \quad \text{otherwise}$

The second constraint essentially allows GIFT to reduce the bandwidth share of a throttle-friendly application (belonging to set Y) by a configurable parameter (B_{thres}) (Table 2). Higher values of B_{thres} create more opportunity of reducing bandwidth wastage, but also result in higher coupon values. Our evaluation shows that GIFT delivers performance for a wide range of B_{thres} values and does not require tuning.

From the system's perspective, the bandwidth allocation at each OST is constrained by its full capacity. That is, $\forall j \in Z$, where Z is the set of all OSTs, if L_j is the set of applications served by j, then $\sum_{i \in L_j} b_i \leq 1$. With these constraints in mind, at every instance, k, we have the following polynomial-time optimization problem: maximize the effective system I/O bandwidth by making allocations b_i for each application i:

$$\text{maximize} \sum_{j \in Z} \sum_{i \in L_j} b_i \tag{1}$$

We make two important remarks: (1) throttle-friendly applications are not always necessarily throttled. In fact, if it is optimal to give more bandwidth to a throttle-friendly application (i.e., expand a throttle-friendly application), given a set of contending applications, then the GIFT's LP-based optimization solution does so. (2) At any time instance, the throttling decision is not limited to picking only one candidate. In fact, the GIFT's LP-based optimization solution might select to throttle multiple throttle-friendly applications simultaneously and expand multiple applications (including throttle-friendly applications) if it leads to highest effective system I/O bandwidth while honoring the constraints.

3.5 GIFT Implementation

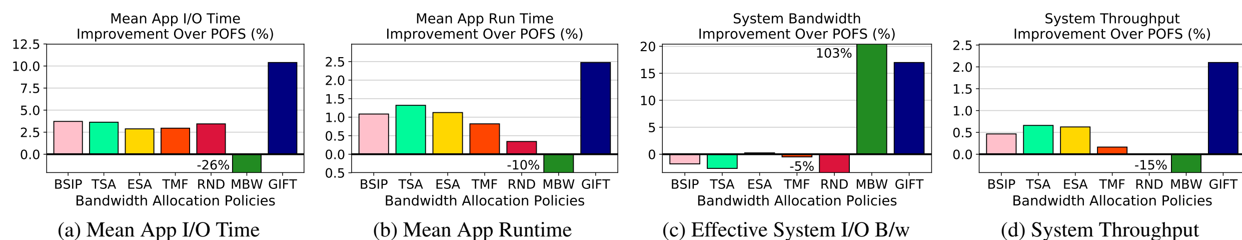
To evaluate GIFT, we implemented it using FUSE [67] as the base file system. Our prototype extends FUSE to capture the functionality of a parallel file system. The architecture of the GIFT implementation is similar to that of a Lustre-based HPC storage system (Sec. 2). Compute nodes mount the remote partition through FUSE. A local service daemon acts as a file system client on each compute node and monitors the mounted partition. An application's requests for file system operations are intercepted by the service daemon and executed on remote storage targets through RPC calls over the network. The file system client forwards file metadata requests to a remote metadata service (MDS), which decides the remote storage target (OST) mappings of a file. Once a file is open, data requests are directly sent over the network to the appropriate OST without involving the MDS. Each I/O request is augmented with metadata about application identity. A local service daemon (OSS) running on the storage node persists the application's data to the OST. Similar to Lustre, our implementation uses two separate network channels: "Lnet" for internal messages (for example, heartbeat, control messages, etc.) and "Dnet" for application data.

The MDS daemon broadcasts a heartbeat message to all the OSSes at a user-configurable time interval. Each OSS responds to the heartbeat message with a list of currently active data requests. The OSSes send a set of <application, I/O requests> tuples for each application they are serving. The MDS uses this data to look up its "coupon" table, make redemption decisions, and determine a set of throttle-friendly applications. Then, it makes a LP optimal bandwidth allocation decision and sends a set of <application, bandwidth allocation> tuples to each OSS. The optimal bandwidth allocation algorithm is implemented using the COIN-OR CLP [43] library. The `blkio` control group (cgroup) is used to enforce bandwidth limits. GIFT uses the MDS as a centralized coordination and decision-making service for all OSSes. OSSes incurring transient failures can be synchronized at the next decision instance. GIFT uses a 1 second timeout and makes a new decision if more than 80% of the OSSes respond. GIFT's decision instance interval is configurable and set to 10 seconds by default, that is decisions are made every 10 sec (Sec. 4). Note that GIFT operates and makes decisions at the system-level without requiring any input from the user applications or changing user applications.

We chose FUSE instead of a production parallel file system such as Lustre or GPFS to implement GIFT's core ideas because the current underlying implementation of bandwidth control support provided in Lustre and GPFS cannot be used for GIFT purposes. This is because the current bandwidth control support does not guarantee synchronous I/O progress and may create imbalance across contended OSTs – a key source of inefficiency that GIFT attempts to solve. GPFS provides bandwidth control only for maintenance tasks [24]. We experimented with recent QoS control features of Lustre as provided by LIME and other frameworks (TBF-NRS algorithm) [56, 80, 86], but found that fairness mechanism does not work as expected because the QoS support does not account for the OST mappings. Even simple experiments such as running a few applications with equal QoS support results in significant performance differences (up to 25%) because of varying level of contention at OST level which leads to non-synchronous progress – these issues and OST mapping information is not accounted by existing early QoS support features. GIFT solves these issues.

4 Evaluation

Methodology. GIFT is evaluated on a real system using system and application characteristics of supercomputers Mira, Theta, and Stampede2. GIFT's experimental setup includes 64 OSSes (and corresponding 64 OSTs) and one MDS running on a cluster with Intel Xeon E5-2686 v4 servers – similar to the Stampede2 OSS and MDS configuration. A total of 192 file system clients are connected to OSSes. The servers and clients are connected to each other via Ethernet with a measured peak bandwidth of 4.5 GB/s. Each OST is connected to a single HDD with a peak bandwidth of 102 MB/s. Experiments are driven by an application set of 250 applications, where applications are executed with repetitions as per the typical number of distinct applications submitted on Stampede2 during a week [1, 16]. The characteristics of applications, such as number of nodes, total compute time and amount of I/O data, are taken from applications running on Stampede2 [16, 62, 66]. Number of MPI processes, and length of compute interval and I/O intervals is based on Darshan logs from Mira and Theta [83]. We use a transparent checkpointing library (DMTCP [3]) to produce periodic I/O from HPC applications such as CoMD [51], SNAP [87], and miniFE [26]. The application arrival times follow a Gamma distribution [1, 44] and are scheduled on the system using an FCFS strategy with easy-backfilling, as used by contemporary HPC schedulers [65]. For practical repeatability, the real-system evaluation scales down the compute and I/O phases to get one week's system wall clock time to finish within a few days. We also evaluate GIFT using simulations to gain deeper insights into GIFT's performance on large-scale systems. The simulations allow us to study aspects of GIFT which are too time consuming to be feasible for a representative real-system evaluation. Specifically, we use simulations to explore the effect of GIFT model parameters and high contention on GIFT performance – these explorations require hundreds of runs to cover the full parameter space. The simulations use the same parameters as the real-system evaluation, but the default application set size is increased to 500 and the simulated time period is 25 days of system wall clock time. As discussed later, the simulation results support the real-system evaluation results and demonstrate the robustness of GIFT.



(a) Mean App I/O Time (b) Mean App Runtime (c) Effective System I/O B/w (d) System Throughput

Figure 7: GIFT's implementation provides improvement for both application- and system- level objectives (higher is better).

Scheduling Policies. We evaluate GIFT against seven competing I/O scheduling policies: Per-OST Fair Share (POFS), Basic Synchronous I/O Progress (BSIP), Minimum Bandwidth Wastage (MBW), Throttle Small Applications (TSA), Expand Small Applications (ESA), Throttle Most Frequent Applications (TMF), and Throttle Randomly (RND). POFS, BSIP, and MBW are implemented as discussed in Sec. 2. TSA attempts to increase the effective system bandwidth by throttling small applications, while ESA attempts to improve the system throughput by increasing the bandwidth allocation for longer-running, smaller applications that generally do small I/O [2, 4, 5]. We also compare against other simple, intuitive strategies such as TMF and RND, which pick the "most frequently appearing" and "random" applications for bandwidth throttling, respectively. POFS is used as the baseline policy.

Objective Metrics. *Application I/O Time* is the amount of time spent in I/O by an application during its run. *Application Run Time* is the run time of the application. *Effective System Bandwidth* is the average effective I/O bandwidth during the run of an application set, defined as overall system bandwidth minus the wasted bandwidth (Sec. 2). *System Throughput* is the number of jobs completed per unit time.

GIFT's real-system implementation provides better application- and system- level performances. First, our results show that GIFT outperforms all competing techniques significantly. Fig. 7 (a)-(d) show that GIFT performs better for mean application I/O time, mean application runtime, effective system bandwidth, and system throughput, respectively. The mean application I/O time with GIFT is 10% better than with POFS, and 3.5% better than the next best technique, BSIP. Interestingly, when applications are throttled based on their characteristics (TSA, ESA, and TMF), or are arbitrarily throttled (RND), the performance remains similar to that of BSIP. This shows that naïve, rule-based techniques cannot match the performance delivered by the GIFT approach.

GIFT also improves the effective system bandwidth by more than 17% compared to POFS and other techniques, except MBW. Expectedly, MBW improves the effective system bandwidth the highest because it solely focuses on this metric. Next, we note that by compromising fairness one could design techniques that solely focus on improving system throughput (e.g., favor small jobs). GIFT does not compromise fairness,

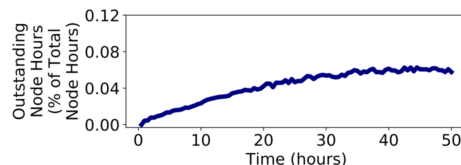


Figure 8: GIFT implementation bounds outstanding node-hours using application- and system-level redemption rate thresholds.

and it neither directly manipulates nor aims to improve the system job throughput, but by virtue of reducing I/O bandwidth waste and mean application I/O time, GIFT yields 2% improvement in system throughput. We note that even a small improvement in system throughput leads to large monetary savings in operational cost of HPC systems [18, 71, 84].

Next, we recall that GIFT gives out compute node-hours as regret, but it is minimal compared to the system throughput improvement it enables (2% savings in total compute node-hours). Fig. 8 shows that GIFT gave out less than 0.06% hours of total compute node-hours from the system regret budget in a more than two-day long experimental run – this result shows that application- and system-level redemption rate thresholds keep the system regret budget under control. Even if one were to award outstanding node-hours every day, GIFT would give out only 0.12% of node-hours, which is much smaller than the gains in system throughput (2%); this trend is also later supported by simulation results.

Next, we discuss the effectiveness of GIFT in terms of fairness. First, recall that the design of GIFT introduces two ideas: (1) opportunistically rewarding applications, and (2) compensating unfairness in I/O performance via additional compute hours. These ideas do not naturally align with the traditional notion of fairness - where a scheme tends to distribute the "benefits" equally among all applications and the "currency" of fairness measurement remains the same. In contrast, GIFT is designed to distribute the benefit opportunistically among applications because, as discussed earlier, distributing the benefits equally among all applications leads to benefit (system bandwidth) wastage due to non-synchronous I/O progress. GIFT achieves fairness by compensating I/O unfairness with compute resources. Therefore, GIFT's performance cannot be directly compared with POFS to establish its fairness effectiveness. Nevertheless, we provide this comparison for completeness and to demonstrate that GIFT is not unfair.

Fig. 9(a) and (b) show that GIFT implementation provides similar fairness in terms of both the I/O and runtime performance as the baseline fairness strategy (POFS). First, as expected, GIFT indeed provides better performance than POFS for many applications. In fact, GIFT is able to improve the I/O performance of one-third of the applications by more than 20%, while competing techniques cannot. But, this improvement is not evenly distributed among all beneficiary applications. This is because, as noted earlier, GIFT rewards certain applications opportunistically by increase their I/O bandwidth if it helps reduce the overall bandwidth waste. We note that these decisions are not systematically biased toward preferring certain applications over others.

Therefore, next, we focus on applications that receive worse performance than under the POFS scheme. This set of application provides us a better quantification of "unfairness" of GIFT and other competing schemes. First we note that other competing schemes, besides GIFT, tend to provide worse performance than POFS for a large fraction of applications compared to POFS - indicating that they are not consciously fairness aware. To further quantify this better, we use a more intuitive and traditional way to measure unfairness - the fraction of applications that achieve worse performance than POFS. As Fig. 9(c) shows GIFT outperforms other schemes in this metric as well (32% for GIFT vs. more than 45% for all other schemes, and 76% for MBW which aggressively focuses only on performance and not fairness). More importantly, even though 32% of the all the applications under GIFT achieve worse performance POFS, we calculated that the average magnitude of I/O time degradation for applications performing worse than POFS is approx. 1.2%. This shows that GIFT is able to provide a similar fair performance compared to our baseline fairness scheme (POFS). These are applications which get throttled initially but are unable to redeem coupons, for which they get compensated in node-hours. Finally, we note, unlike other competing schemes, GIFT indeeds compensates these applications via compute resources and hence, achieves fairness over the long term.

GIFT improves performance across different parameters and the required system regret budget level needed to award outstanding hours is fairly low even under pessimistic scenarios. To study the impact of model parameters on GIFT performance accurately, we perform a simulation-based exploration. First, we briefly present the simulation results for the same objective metrics as the real system evaluation. We find that GIFT's simulation results support and closely match the trends observed with the real system evaluation (Fig. 10 vs. Fig. 7). Fig. 10 shows that compared to POFS, GIFT improves the mean application I/O time by 15% and effective system bandwidth by 25%. Similarly, GIFT improves the mean application run time by more than 4% and system throughput by approx. 2%. We note thatthe absolute improvement values are higher than real system evaluation because

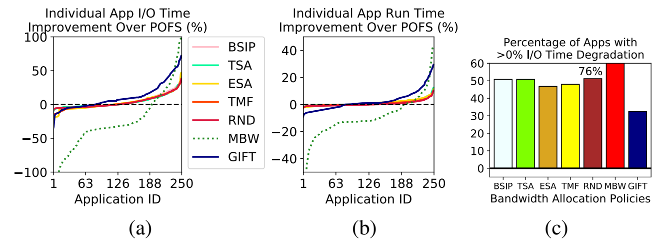


Figure 9: GIFT implementation provides I/O and runtime performance fairness to individual applications.

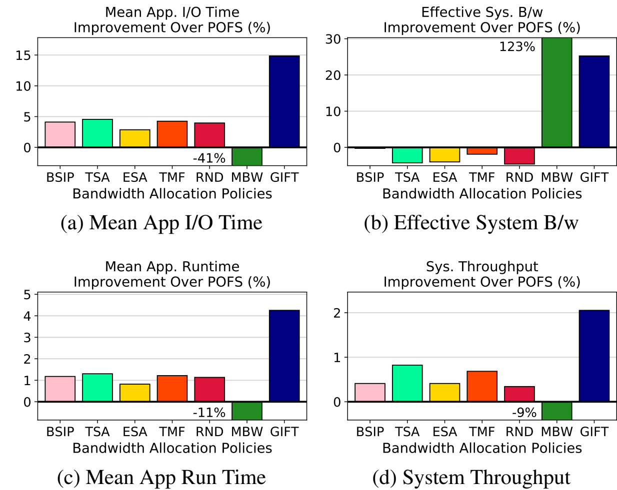


Figure 10: GIFT simulation results support GIFT real-system-based implementation results and show significant improvements.

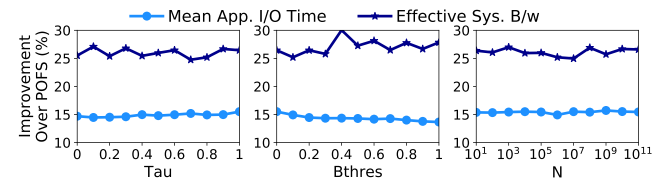
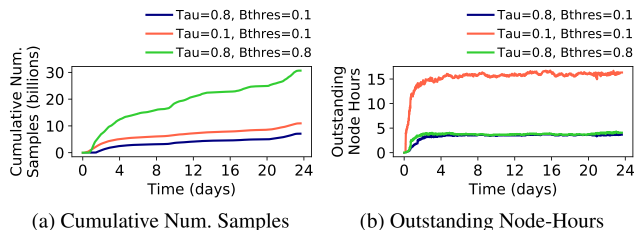


Figure 11: GIFT improves performance across different values of throttle-and-reward parameters.

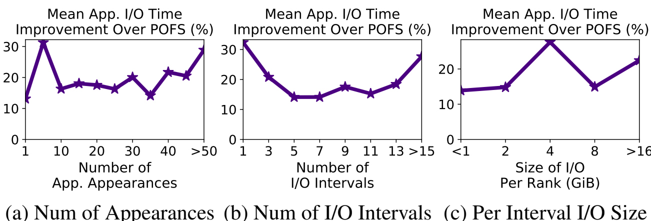
simulation study covers a longer time frame (25 days) and a larger application set (500); this provides more opportunities for GIFT to make better throttle-and-reward decisions.

Next, our results (Fig. 11) illustrate that GIFT performs effectively across the parameter space and does not require tuning. Recall that τ is the minimum redemption rate for the system to throttle and for an application to be considered throttle-friendly. Therefore, it is expected that at higher values of τ, the I/O time would improve slightly. GIFT also continues to provide significant improvement in effective system bandwidth, even with high τ values. Recall that B_{thres} is the maximum factor by which an application's bandwidth can be throttled. Fig. 11(b) shows that GIFT is effective at different B_{thres} values. Note that GIFT increases the effective system bandwidth by as much as 5% points for higher B_{thres} values. This trend is expected: a higher B_{thres} value implies higher



(a) Cumulative Num. Samples (b) Outstanding Node-Hours

Figure 12: GIFT is able to collect high cumulative number of samples and bound the node-hours awarded.



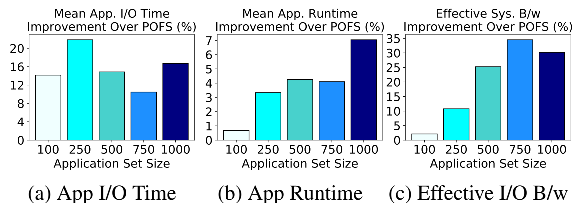
(a) Num of Appearances (b) Num of I/O Intervals (c) Per Interval I/O Size

Figure 13: Applications with all types of characteristics experience improvement in I/O performance with GIFT.

throttling power, and hence, better opportunities to fill the bandwidth gap. However, this also causes slight reduction in I/O time improvement (2% points). Next, Fig. 11(c) shows the impact of parameter N (the length of the receding window) on GIFT performance. Increasing N does not impact I/O time but it improves effective system bandwidth slightly due to better stability from one decision instance to the next. Overall, GIFT does better than POFS across a wide range of N values.

Studying GIFT's characteristics over time, Fig. 12(a) shows that GIFT collects a large number of samples as time progresses for both, default parameter configuration ($\tau = 0.8, B_{thres} = 0.1$) and extreme cases ($\tau = 0.1, B_{thres} = 0.1$ and $\tau = 0.8, B_{thres} = 0.8$). The sample collection continues in order to adjust to application characteristics and learn about new applications. Fig. 12(b) also shows that the number of outstanding node-hours is quite low at all times due to effectiveness of GIFT's redemption rate thresholds – therefore, indicating that only a small system regret budget is needed. Fig. 12(b) also shows that even under a pessimistic parameter selection ($\tau = 0.1, B_{thres} = 0.1$, low redemption rate threshold for applications to be considered throttle-friendly), GIFT needs a low number of outstanding node-hours at all times (less than 20 hours at any instance, although the corresponding 2% improvement in system throughput translates to a gain of more than 5,800 node-hours). Even if outstanding node-hours are awarded daily, the system regret budget needs to be only 360 node-hours over 24 days, much less than the 5,800 node-hours gained with 2% system throughput improvement.

GIFT provides performance improvement for applications of different characteristics, under high I/O contention, and device bandwidth (SSD vs. HDD). We performed simulation based exploration to understand how GIFT



(a) App I/O Time (b) App Runtime (c) Effective I/O B/w

Figure 14: GIFT performs better than POFS at all contention levels.

performs when key application characteristics are varied: number of appearances of an application, number of I/O intervals, and the size of I/O per I/O interval per MPI process (rank). We found (Fig. 13 (a)-(c)) that GIFT continues to provide a significant improvement in application I/O performance as we vary the number of appearances of an application, number of I/O intervals, and the per-interval I/O size across a wide range. Our results (Fig. 14) also show that GIFT's performance benefits actually improve as we increase the contention level from 100-applications set to 1,000-applications set; this is expected because a higher-level of contention increases the chances for GIFT to exercise throttle and reward. For 1,000 application-set GIFT improves mean application I/O time by up to 16%, mean application run time by up to 7%, and effective system bandwidth by up to 30%. Finally, although GIFT does not rely on specific storage device characteristics to provide benefits, we studied the effect of device bandwidth (e.g., SSD vs. HDD) on the limits of GIFT performance. As expected, we did not find the GIFT performance improvements to be sensitive to the underlying storage device.

GIFT implementation is low-overhead and scalable on a real system. GIFT has two sources of overhead: computation and communication. MDS incurs the computation overhead due to solving an LP optimization problem. Communication overhead is incurred due to message exchanges between the OSTs and MDS. To obtain pessimistic estimates on the GIFT implementation overhead on a real system, we increase the number of OSTs from 32 to 200 and increase the application set size to 1,000 – amplifying the degree of GIFT overheads. We measured that the CPU overhead on the MDS increased from 1 ms to 5 ms which is negligible compared to decision instance interval (10 seconds); GIFT produces similar results with similar decision instance interval lengths, however choosing too small interval (e.g., 1 second) can make overhead effects visible and choosing very large interval (e.g., 10 minute) can lead missed opportunities for throttle-and-reward. The volume of messages between the MDS and the OSTs is also minimal (less than 4 MB over two days) and occurs on a non-critical network path. In our real system experiments, we measured that overall GIFT's implementation imposes a negligible overhead on I/O performance even under pessimistic scenarios (less than 0.01%).

5 Discussion

Relationship between I/O bandwidth improvements and system throughput. We note that GIFT does not actively manipulate the I/O bandwidth allocation to directly improve the system throughput. It is trivial to improve the system throughput - for example, by allocating more I/O bandwidth share to short-running jobs which can significantly increase the system throughput at the expense of fairness. Nevertheless, as our results show, GIFT is able to improve the overall system throughput. This is because GIFT eliminatea I/O bandwidth inefficiencies by increasing the I/O bandwidth toward synchronous progress which reduces the overall I/O time and run time of applications. Reducing the overall run time of applications by judiciously utilizing the available I/O bandwidth, in turn, leads to completion of more jobs per unit time (i.e., system throughput increases).

Why traditional notions of measuring fairness alone may be not be adequate for assessing the effectiveness of GIFT. A conventional notion of fairness measures the amount of equal opportunity among all participants. In the case of GIFT, this translates to providing equal bandwidth to all jobs concurrently performing I/O on the same OST (i.e., POFS). However, this does not lead to effective equal bandwidth division since jobs may not be able to leverage the full I/O bandwidth due to non-synchronous I/O progress. While, GIFT does not enforce this fair opportunity at every decision instance, it does enforce it as a constraint in the long run. Thus, GIFT enforces fair opportunity as a constraint.

Another conventional notion of fairness measures the amount of equal performance among all participants. For example, calculating the difference between maximum and minimum performances, or the standard deviation of performances, or Jain's Fairness Index [27]. Fairness can be viewed at as all jobs having equal I/O performance. In practice, this is difficult to enforce and impractical to achieve in a diverse and dynamic I/O environment of an HPC storage system. Job I/O performance depends on a variety of job-specific aspects which are not in control of GIFT (GIFT only performs time-divided bandwidth allocation) such as number of OSTs across which a file is stripped, size of I/O, type and pattern of I/O, I/O interface (POSIX, MPIIO, STDIO), etc. Thus, while GIFT enforces equal opportunity in terms of bandwidth (resource) allocation as hard constraint, it cannot enforce overall equal I/O performance.

In the case of GIFT, one could argue that fairness can be defined as all applications having equal improvement as compared to POFS. However, this definition is not meaningful since POFS already performs instantaneous fair allocation, thus, I/O performance with POFS is fair and attempting to achieve "fair improvement from a fair performance" does not have practical value for end users. Therefore, as discussed in Sec. 4, GIFT's fairness is better quantified by focusing on the applications which achieves worse performance than POFS. If the improvement over POFS is positive, then GIFT is considered fair for such beneficiary applications, but the improvement over POFS among such beneficiary applications is not equal. This is because GIFT rewards certain applications opportunistically by increasing their I/O bandwidth if it helps reduce the overall bandwidth waste. Finally, GIFT compensates unfairness in one type of resource allocation by allocating another type of resource - this feature makes GIFT fairness fundamentally different than traditional notions.

6 Related Work

Many prior works have focused on identifying the root causes of contention and characterizing the I/O bottlenecks [2, 10, 12, 22, 28, 31, 34, 38, 39, 46, 47, 63, 68, 81, 82, 85]. These works do not propose mitigation techniques. Studies focusing on application-level techniques [13, 35, 42, 56, 61, 89, 91, 92], such as CALCioM [14], rely on application modifications and cooperation for coordinating I/O transactions among applications. Client-side solutions, which coordinate I/O requests to and from the client-attached burst buffers or requests handlers [6, 25, 29, 36, 37, 76–78], end up underutilizing the back-end bandwidth due to the lack of a storage-system view. In general, client-side techniques are complementary to GIFT and can be used to further enhance application performance. On the other hand, server-side solutions aim to efficiently schedule the I/O requests from the server nodes to the disk targets [17, 20, 21, 50, 64, 69, 70, 72, 74, 90]. For example, IOrchestrator [88] uses spacial locality of I/O requests to unfairly prioritize the most disk efficient requests. Note that none of these studies consider the distributed and synchronous I/O behavior of HPC applications. This paper introduced, GIFT, a new I/O bandwidth allocation approach to ensure synchronous I/O progress for HPC application while maximizing I/O throughput and ensuring fairness.

7 Conclusion

Improving effective system I/O bandwidth, providing fairness among applications, and ensuring synchronous I/O progress are three major challenges in parallel storage systems, but no existing approaches have considered them as a joint problem. GIFT identifies and solves this new problem using a throttle-and-reward approach - yielding significant improvements (17% in mean effective system I/O bandwidth and 10% in the mean application I/O time). GIFT is available at `https://github.com/GoodwillComputingLab/GIFT`.

Acknowledgment. We are thankful to our shepherd (André Brinkmann), Phil Carns, Robert Ross, and anonymous reviewers for their constructive feedback. This work is supported in part by NSF Awards 1910601 and 1753840, Northeastern University, and Massachusetts Green High Performance Computing Center (MGHPCC).

References

[1] *Stampede2 User Guide*, 2018 (accessed January 10, 2019). https://portal.tacc.utexas.edu/user-guides/stampede2.

[2] Gonzalo Pedro Rodrigo Alvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 521–526. IEEE, 2016.

[3] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Parallel and Distributed Processing Symposium (IPDPS), 2009 IEEE International*, pages 1–12. IEEE, 2009.

[4] Katie Antypas, BA Austin, TL Butler, RA Gerber, Cary Whitney, Nick Wright, Woo-Sun Yang, and Zhengji Zhao. NERSC Workload Analysis on Hopper. Technical report, Technical report, LBNL Report, 2013.

[5] Brian Austin, Tina Butler, Richard Gerber, Cary Whitney, Nicholas Wright, Woo-Sun Yang, and Zhengji Zhao. Hopper Workload Analysis. 2014.

[6] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.

[7] John Bent, Sorin Faibish, Jim Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon Woodring. Jitter-Free Co-Processing on a Prototype Exascale Storage Stack. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2012.

[8] 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*, page 21. ACM, 2009.

[9] Suren Byna, A Uselton, D Knaak Prabhat, and Y He. Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper. In *Cray user group meeting*, 2013.

[10] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *FAST*, pages 329–344, 2017.

[11] Franck Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *IJHPCA*, 23(3):212–226, 2009.

[12] Christopher S Daley, Devarshi Ghoshal, Glenn K Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas J Wright. Performance Characterization of Scientific Workflows for the Optimal use of Burst Buffers. *Future Generation Computer Systems*, 2017.

[13] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-Free I/O. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 155–163. IEEE, 2012.

[14] Matthieu Dorier, Gabriel Antoniu, Rob 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 IEEE 28th International*, pages 155–164. IEEE, 2014.

[15] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for Peta-scale Systems: A Look into the Future of Practical Rollback-Recovery. *TDSC 2004*, 1(2):97–108, 2004.

[16] Thomas Furlani. XDMoD Value Analytics. 2018.

[17] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC Applications under Congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1013–1022. IEEE, 2015.

[18] Richard Gerber, James Hack, Katherine Riley, Katie Antypas, Richard Coffey, Eli Dart, Tjerk Straatsma, Jack Wells, Deborah Bard, Sudip Dosanjh, et al. Crosscut Report: Exascale Requirements Reviews, March 9–10, 2017–Tysons Corner, Virginia. An Office of Science Review Sponsored by: Advanced Scientific Computing Research, Basic Energy Sciences, Biological and Environmental Research, Fusion Energy Sciences, High Energy Physics, Nuclear Physics. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States); Argonne . . . , 2018.

[19] Richard A Gerber and Harvey Wasserman. Large Scale Computing and Storage Requirements for High Energy Physics. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 20102.

[20] Ajay Gulati, Arif Merchant, and Peter J Varman. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *ACM SIGMETRICS*

Performance Evaluation Review, volume 35, pages 13–24. ACM, 2007.

[21] Ajay Gulati, Arif Merchant, and Peter J Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 437–450. USENIX Association, 2010.

[22] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015.

[23] Jun He, Duy Nguyen, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *FAST*, volume 15, pages 119–133, 2015.

[24] John Hearns, Marc A Kaplan, and Egonle Bo. Limit / fair share of gpfs bandwidth, Jan 2018.

[25] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. 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*, pages 69–80. ACM, 2016.

[26] M Heroux and S Hammond. MiniFE: Finite Element Solver.

[27] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput Fairness Index: An Explanation. In *ATM Forum contribution*, volume 99, 1999.

[28] Ye Jin, Xiaosong Ma, Mingliang Liu, Qing Liu, Jeremy Logan, Norbert Podhorszki, Jong Youl Choi, and Scott Klasky. Combining Phase Identification and Statistic Modeling for Automated Parallel Benchmark Generation. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):309–320, 2015.

[29] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance Differentiation for Storage Systems using Adaptive Control. *ACM Transactions on Storage (TOS)*, 1(4):457–480, 2005.

[30] Seong Jo Kim. Parallel I/O Profiling and Optimization in HPC Systems. 2014.

[31] Youngjae Kim and Raghul Gunasekaran. Understanding I/O Workload Characteristics of a Peta-scale Storage System. *The Journal of Supercomputing*, 71(3):761–780, 2015.

[32] Michelle Koo, Wucherl Yoo, and Alex Sim. I/O Performance Analysis Framework on Measurement Data from Scientific Clusters. 2015.

[33] Douglas Kothe and Ricky Kendall. Computational Science Requirements for Leadership Computing. *Oak Ridge National Laboratory, Technical Report*, 2007.

[34] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O Performance Challenges at Leadership Scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 40. ACM, 2009.

[35] Han Deok Lee, Young Jin Nam, Kyong Jo Jung, Seok Gan Jung, and Chanik Park. Regulating I/O Performance of Shared Storage with a Control Theoretical Approach. In *MSST*, pages 105–117, 2004.

[36] 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), 2015 31st Symposium on*, pages 1–16. IEEE, 2015.

[37] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[38] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *FAST*, volume 14, pages 213–228, 2014.

[39] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 819–829. IEEE, 2016.

[40] Glenn K Lockwood, Wucherl Yoo, Suren Byna, Nicholas J Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. UMAMI: A Recipe for Generating Meaningful Metrics Through Holistic I/O Performance Analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 55–60. ACM, 2017.

[41] William Loewe, T McLarty, and C Morrone. IOR Benchmark, 2012.

[42] 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–12. IEEE, 2010.

[43] Robin Lougee-Heimer. The Common Optimization INterface for Operations Research: Promoting Open-source Software in the Operations Research Community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[44] Uri Lublin and Dror G Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[45] Robert Lucas. Top Ten Exascale Research Challenges. In *DOE ASCAC Subcommittee Report*, 2014.

[46] 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*, pages 33–44. ACM, 2015.

[47] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M Wild. Analysis and Correlation of Application I/O Performance and System-Wide I/O Activity. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on*, pages 1–10. IEEE, 2017.

[48] George S Markomanolis, Bilel Hadri, Rooh Khurram, and Saber Feki. Scientific Applications Performance Evaluation on Burst Buffer. In *International Conference on High Performance Computing*, pages 701–711. Springer, 2017.

[49] Deirdre N McCloskey, Stephen T Ziliak, et al. The Standard Error of Regressions. *Journal of economic literature*, 34(1):97–114, 1996.

[50] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 245–254. ACM, 2011.

[51] Jamaludin Mohd-Yusof, S Swaminarayan, and TC Germann. Co-Design for Molecular Dynamics: An Exascale Proxy Application, 2013.

[52] Andrey Ovsyannikov, Melissa Romanus, Brian Van Straalen, Gunther H Weber, and David Trebotich. Scientific Workflows at Satawarp-Apeed: Accelerated Data-Intensive Science using NERSC's Burst Buffer. In *Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on*, pages 1–6. IEEE, 2016.

[53] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[54] Torben Kling Petersen. HPC Storage Current Status and Futures.

[55] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, et al. LPCC: Hierarchical Persistent Client Caching for Lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 88. ACM, 2019.

[56] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. 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*, page 6. ACM, 2017.

[57] Robert Ross, Robert Ross, Gary Grider, Gary Grider, Evan Felix, Evan Felix, Mark Gary, Mark Gary, Scott Klasky, Scott Klasky, et al. Storage Systems and Input/Output to Support Extreme Scale Science. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[58] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[59] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[60] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[61] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI*, volume 12, pages 349–362. USENIX, 2012.

[62] Nikolay A Simakov, Joseph P White, Robert L DeLeon, Steven M Gallo, Matthew D Jones, Jeffrey T Palmer, Benjamin Plessinger, and Thomas R Furlani. A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD. *arXiv preprint arXiv:1801.04306*, 2018.

[63] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular HPC I/O Characterization with Darshan. In *Extreme-Scale Programming Tools (ESPT), Workshop on*, pages 9–17. IEEE, 2016.

[64] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. Server-Side I/O Coordination for Parallel File Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2011.

[65] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE, 2002.

[66] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S Mehringer, Eric Wernert, H Tufo, D Panda, et al. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 15. ACM, 2017.

[67] Miklos Szeredi. FUSE: Filesystem in Userspace. `https://fuse.sourceforge.net/`, 2005. Online (accessed January 10, 2019).

[68] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It* IS* Rocket Science. In *HotOS*, volume 13, pages 1–5, 2011.

[69] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathrn Mohror, and Adam Moody. IO-Cop: Managing Concurrent Accesses to Shared Parallel File System. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 52–60. IEEE, 2014.

[70] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST*, volume 7, pages 5–5, 2007.

[71] Edward Walker. The Real Cost of a CPU Hour. *Computer*, (4):35–41, 2009.

[72] Chien-Min Wang, Tse-Chen Yeh, and Guo-Fu Tseng. Provision of Storage QoS in Distributed File Systems for Clouds. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 189–198. IEEE, 2012.

[73] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving Large-scale Storage System Performance via Topology-Aware and Balanced Data Placement. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 656–663. IEEE, 2014.

[74] Hui Wang and Peter J Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *FAST*, volume 14, pages 229–242, 2014.

[75] Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Controlled Contention: Balancing Contention and Reservation in Multicore Application Scheduling. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 946–955. IEEE, 2015.

[76] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-buffer File System For Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Press, 2016.

[77] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. Trio: Burst Buffer Based I/O Orchestration. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 194–203. IEEE, 2015.

[78] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A High-Performance Burst Buffer System for Scientific Applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014.

[79] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[80] Li Xi and Zeng Lingfang. LIME: A Framework for Lustre Global QoS Management. *Lustre Administrator and Developer Workshop*, 2018.

[81] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing Output Bottlenecks in a Supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[82] Bing Xie, Yezhou Huang, Jeffrey S Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting Output Performance of a Petascale Supercomputer. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 181–192. ACM, 2017.

[83] Cong Xu, Shane Snyder, Vishwanath Venkatesan, Philip Carns, Omkar Kulkarni, Suren Byna, Roberto Sisneros, and Kalyana Chadalavada. DXT: Darshan eXtended Tracing. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[84] Fan Yang and Andrew A Chien. Extreme Scaling of Supercomputing with Stranded Power: Costs and Capabilities. *arXiv preprint arXiv:1607.02133*, 2016.

[85] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the Root Causes of Cross-application I/O Interference in HPC Storage Systems. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 750–759. IEEE, 2016.

[86] L Zeng, J Kaiser, A Brinkmann, T Süß, L Xi, Q Yingjin, and S Ihara. Providing QoS-Mechanisms for Lustre through Centralized Control Applying the TBF-NRS. *Lustre User Group*, 2017.

[87] Joe Zerr and Randal Baker. SNAP. `http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/snap/`, 2018 (accessed January 10, 2019).

[88] 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*, pages 1–11. IEEE Computer Society, 2010.

[89] Xuechen Zhang, Kei Davis, and Song Jiang. Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 330–341. IEEE, 2012.

[90] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-Aware Batch Scheduling for Petascale Computing Systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 254–263. IEEE, 2015.

[91] Timothy Zhu, Michael A Kozuch, and Mor Harchol-Balter. WorkloadCompactor: Reducing Datacenter Cost while Providing Tail Latency SLO Guarantees. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 598–610. ACM, 2017.

[92] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.