ASCAR: Automating Contention Management for High-Performance Storage Systems

Yan Li, Xiaoyuan Lu, Ethan L. Miller, Darrell D. E. Long
Storage Systems Research Center, University of California, Santa Cruz
{yanli,xyuanlu,elm,darrell}@cs.ucsc.edu

***Abstract*—High-performance parallel storage systems, such as those used by supercomputers and data centers, can suffer from performance degradation when a large number of clients are contending for limited resources, like bandwidth. These contentions lower the efficiency of the system and cause unwanted speed variances. We present the Automatic Storage Contention Alleviation and Reduction system (ASCAR), a storage traffic management system for improving the bandwidth utilization and fairness of resource allocation. ASCAR regulates I/O traffic from the clients using a rule based algorithm that controls the congestion window and rate limit. The rule-based client controllers are fast responding to burst I/O because no runtime coordination between clients or with a central coordinator is needed; they are also autonomous so the system has no scale-out bottleneck. Finding optimal rules can be a challenging task that requires expertise and numerous experiments. ASCAR includes a SHAred-nothing Rule Producer (SHARP) that produces rules in an unsupervised manner by systematically exploring the solution space of possible rule designs and evaluating the target workload under the candidate rule sets. Evaluation shows that our ASCAR prototype can improve the throughput of all tested workloads – some by as much as 35%. ASCAR improves the throughput of a NASA NPB BTIO checkpoint workload by 33.5% and reduces its speed variance by 55.4% at the same time. The optimization time and controller overhead are unrelated to the scale of the system; thus, it has the potential to support future large-scale systems that can have millions of clients and thousands of servers. As a pure client-side solution, ASCAR needs no change to either the hardware or server software.**

I. Introduction

If there is only one thing that all storage systems share, from exascale cloud storage to high-performance computing (HPC) storage systems, it is the ever increasing demand for higher performance, which is often satisfied by adopting higher parallelism: I/O operations are cut into smaller requests and distributed to different servers concurrently in order to achieve high throughput. However, this design also causes the clients to contend for limited resources, such as bandwidth and disk access time. When I/O requests from many clients are mingled together, they form a sequence of random I/O requests that can adversely affect the efficiency of network and storage systems. Under heavy workloads, these contentions lower the efficiency of the system and cause unwanted speed variances over time and between clients. A storage system without contention management is like a traffic system without traffic lights. What we need is a high-performance, lightweight network and storage contention management mechanism, which can work with many different storage system architectures, and is autonomous so that the human cost on system optimization can be kept to minimum.

Today's traffic management and contention control in distributed large-scale storage systems are not optimal. Most solutions need models of the system and related devices for establishing control thresholds, either through direct input from the system administrator or periodic negotiation between nodes. To find meaningful parameters that can fully utilize the system's potential, system designers and administrators often have to spend too much time benchmarking and tweaking the system; the situation can be much worse if the system changes often.

Our Automatic Storage Contention Alleviation and Reduction (ASCAR) system is designed for increasing the bandwidth utilization and reducing speed variance. ASCAR is a delay-based general-purpose client-side I/O rate control system. It uses traffic controllers on storage clients, which control the upper speed limit and the congestion window size of each connection. The latter determines the number of I/O requests that can be outstanding for each connection. These controllers follow a set of traffic rules without the need to communicate with each other or a control center, even when handling highly dynamic workloads. This shared-nothing design improves response speed and scalability. We treat the storage servers as black boxes and design the traffic controller to be simple, requiring no change to the hardware or server software. This simple design makes ASCAR fit for many different environments, such as data centers, and both High Performance Computing (HPC) and enterprise systems.

The effectiveness of a rule-based traffic control system largely depends on the quality of the rule sets. Traffic control rules designate what actions to take in different congestion states. Manually tweaking the rules is onerous and inefficient. We design a SHAred-nothing Rule Producer (SHARP) to produce and optimize traffic rules for a specific workload running on a specific system. SHARP works in an unsupervised manner and systematically explores the solution space of possible designs. In the preparation phase, ASCAR first extracts key behaviors from the target workload and generates a short signature workload. Starting from one fixed initial rule, SHARP runs the signature workload, measures its performance, and records the number of times each rule is triggered. SHARP optimizes the hottest rule in the set by using heuristics to generate candidate rules and evaluating them using the signature workload on the real storage system. After finding the optimal

978-1-4673-7619-8/15/$31.00 © 2015 IEEE

parameters for one rule, SHARP expands the rule set by dividing the congestion state space of the hottest rule at the observed mean values of the congestion indicators into disjoint new rules, then restarts the optimization process with the new rule set. By repeating this process, SHARP can produce complex traffic rules that are best optimized for a given workload running on a given system. The time complexity of this process is unrelated to the size of the underlying storage system, and is linear to the complexity of the rule set, which only depends on the complexity of the control strategy.

ASCAR stores the best discovered rule sets for different workloads in a database. When a new workload enters the system, ASCAR measures its performance characteristics and compares them with the known workloads in the database to find the most similar known workload, whose control rule set is then chosen for controlling this new workload. We have conducted a series of experiments to understand the measurement of similarity between workloads in terms of applying traffic control rules.

The evaluation shows that ASCAR can effectively increase the throughput and lower the speed variance, at the same time (Table IV) for all the evaluated workloads. It increases the bandwidths of the BTIO Class B and C checkpoint workloads [43] by 29% and 35% respectively. For the random write workload, the bandwidth is increased by 25%. Since ASCAR increases the bandwidth utilization, it cannot optimize those workloads whose bandwidth is near the hardware's limit. Current evaluation results show that ASCAR works best on workloads that exhibit high contention or speed fluctuation.

The time needed for SHARP to generate rules varies from 7 to 36 hours. The long optimization process only needs to be run once and can benefit all similar subsequent workloads until the system's configuration is modified. The optimization does not show overfitting; the generated best rule sets work equally well on different workloads as long as they exhibit the same I/O pattern. Analyses of the best rules show that they increase the bandwidth utilization by choosing the best combination of congestion indicators and control parameters, and applying primarily two control strategies: adaptive I/O rate limiting and slow-start, fast-fallback.

The contributions of this paper include:

- a shared-nothing scalable storage traffic control framework that features lightweight client-side traffic controllers and automatic traffic rule generation,
- the SHARP algorithm for optimizing rule-based control systems in an unsupervised manner on real systems when mathematical model or simulator is hard to establish,
- evaluations of a wide range of workloads,
- the ASCAR prototype that works out-of-the-box with the Lustre 2.4 file system [27]. Our prototype only requires changes on the client-side, and can be easily evaluated in a production environment. We release it as an open-source project to promote further research on this topic.

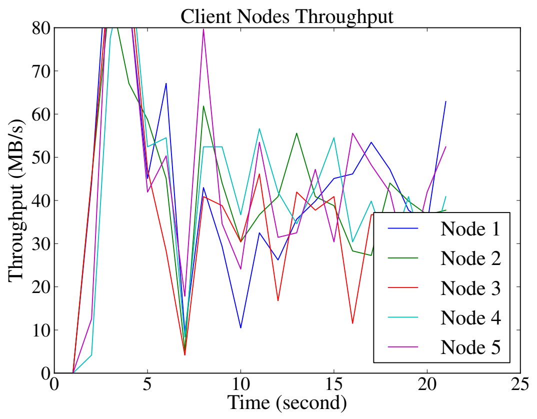
II. Background

Storage traffic management studies methods for maximizing certain properties of a storage system, such as I/O throughput and latency. Common methods include optimizing data placement, traffic routing, and congestion management. Data placement and routing algorithms are developed to minimize the distance between data providers/consumers and storage devices, and to spread the traffic evenly among nodes and switches to balance the load [33]. But even with the optimal placement and routing configuration, limited resources, such as network bandwidth and disk access time, can still be contended when the system is under heavy workload. Contention for network bandwidth may lead to network congestion, causing packet loss, timeout, or random disconnection. Severe contention in storage devices lowers cache efficiency, increases seek time for disk drives, and causes write amplification for flash drives. Failing to keep resource contention under control would lower bandwidth utilization and exacerbate speed fluctuation. In this paper, "congestion" includes severe contention in the network, server, and devices.

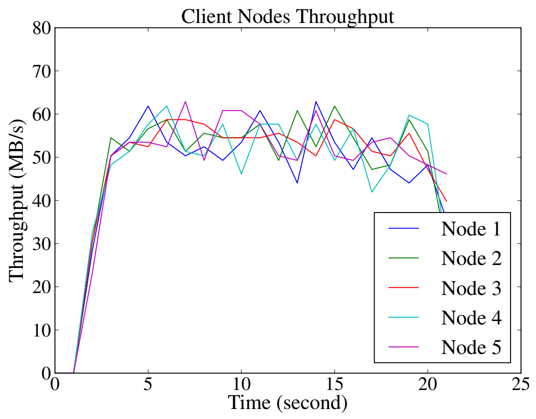
Distributed storage systems Distributed high-performance storage systems, such as Lustre [27], GPFS [31], and Ceph [41], are designed to run on a large number of servers to achieve the level of high throughput and low latency that are impossible to get from one server. Application's I/O requests can vary widely in size, thus modern file system clients aggregate small I/O requests and split large I/O requests so that they can be handled more efficiently, usually by equally distributing the workload across multiple servers. In Lustre, one file system-level request is issued by using one Remote Procedure Call (RPC). The Lustre client always tries to issue multiple RPCs to each server to increase throughput, and this also enables better optimization, like re-grouping and re-ordering at the server side. The number of these concurrent RPCs is controlled by a congestion window, which limits how many requests can be outstanding at any time.

The end-to-end data path between a client and a storage device usually involves many network switches and servers. Contentions may occur when data paths cross each other, e.g., when they need to use the same switch, the same server, or the same storage device. Network congestion occurs when there are more packets in the switch's queue than it can handle. Similarly, server and device congestion occurs when there are more requests than the server or device can process in a timely and efficient manner. Slowing down the incoming requests is an efficient way for relieving congestion.

Traffic control in distributed storage systems Applications can have different requirements for traffic control. HPC checkpoint workloads, for example, issue sequential writes from each node, which dumps about the same amount of data to the storage concurrently. They need not only high aggregated throughput but also even bandwidth distribution among nodes; if some nodes write faster they still have to wait for the slow nodes to finish writing because all nodes must finish the I/O before the next cycle of computation can



(a) Without ASCAR



(b) With ASCAR

Fig. 1: Comparing a random write workload when running with and without ASCAR control. The benchmark that runs without ASCAR (left) shows high temporal and spatial bandwidth fluctuation (average throughput is 46 MB/s). When running with ASCAR (right), the lines are flatter, and the average throughput is higher at 52 MB/s.

start. Some other applications favor proportional bandwidth allocation over overall aggregated throughput. One example is that the virtual machines from different customers running in the cloud need good performance isolation. ASCAR uses a customizable objective function so that different requirements can be properly handled.

Bernstein et al. have proven that distributed traffic control is a NEXP-hard decentralized partially observable Markov decision process (DEC-POMDP) [3], where outcomes are partly random (due to the complexity of network and nonlinearity of storage devices) and partly under the control of the traffic controllers. Information of the entire history of observations up to this point, including the times of all outgoing I/O requests and their finish times, and the times of all future I/O requests, are needed to generate an optimal traffic control scheme for maximizing an objective function. Most practical traffic control solutions use a certain method of approximation.

A traffic control strategy that maximizes system efficiency should always be work-conserving. Work-conserving traffic control allocates unused resources as soon as they become available to the next consumer and does not waste usable resources. For example, let the total usable amount of a certain resource be m. When there are 12 consumers, a fair work-conserving allocator would allocate $m/12$ units of the resource to each consumer. When the number of consumers reduces to 7, each consumer's share of resource would be increased to $m/7$. This feature is essential because letting usable resources sit idle is wasteful and undesirable. Work-conserving bandwidth allocation is hard for distributed systems because the peak performance of the system for running a specific workload is not easily known. It is affected by many factors, such as the states of network switches, servers and storage devices. It is also affected by the workload itself: a sequential read workload can achieve a different throughput than a random write workload. So far, researchers have yet to discover a practical model to calculate the theoretical peak performance given a complex mixed read/write workload running on a complex storage systems, and still have to rely on benchmarking the workload on the real system.

The current dominant method for work-conserving resource allocation in these systems is best-effort, of which the basic principle is round-robin scheduling for requests at the same priority level and placing a premium on requests of higher priority levels. This strategy is simple, easy to implement, and behaves relatively well with small scale storage systems. However, this simple policy does not monitor the congestion states and cannot achieve optimal resource allocation in a resource-constrained environment. The throughput of a random write workload, as shown in Figure 1a, exhibits high temporal and spatial speed variance. Facing these challenges, an application's best choice is, unfortunately, to send out I/O requests as fast as possible, i.e., to mob the servers, in order to grab more shares of the resources. Under this situation, congestion easily occurs at many levels in the system, hurting the overall efficiency.

Delay-based congestion control In network congestion management, delay-based strategy detects congestion by monitoring the packet Round Trip Time (RTT) and following preset rules to tweak the congestion window. The congestion window designates how many packets are allowed to be on-the-fly before allowing new packets to be sent. The traffic control rules contains multiple control parameters, including the thresholds of RTT [15] (or the current RTT to the lowest RTT ratio [13]), speed of increasing/decreasing the congestion window, and rate limits. Good parameters can help reduce congestion and increase overall system performances, while bad parameters can only make the system worse. When being used in wide area computer networks, like the Internet, the control algorithm's parameters are often pre-tweaked by system designers. When applying delay-based congestion control to storage systems, there are several salient differences. First, one storage system is often very different from another, and they cannot share the same control strategy. Second, unlike in computer networks, where the pressure on the system can be easily measured by the number of bits to send, different I/O patterns can cause radically different pressure on the system due to the statefulness of storage devices. Third, network designers use simulators heavily before deploying a real network, but unfortunately there is no good simulator for simulating a large and complex storage cluster. These differences indicate that new methods must be developed for storage systems before we can apply the delay-based congestion control strategy, not least of which is the way to discover the optimal parameters for each storage system and each I/O workload.

To discover the optimal parameters, existing methods either require the system designers or administrators to carry out a huge amount of work to benchmark the system in order to gather system performance characteristics, or require the clients and servers to communication with each other frequently to measure the performance and negotiate the parameters. These cross-client communication normally grows at $O(n^2)$ and has no way to scale to support more than a few hundreds of clients. Also, network synchronization cannot react fast enough to handle burst or highly dynamic workloads.

Winstein and Balakrishnan described a method, Remy, for producing network QoS rules using simulation [42]. Remy produces good rule sets but requires the model of the network. Remy's method cannot be applied directly to storage systems, because precise modeling of storage systems is very difficult, and no storage simulator can do performance simulation for a complex storage cluster at the precision level required by Remy.

Design goals Overall, our design goals of a scalable storage traffic management solution are:

- to handle the congestions in all the network, servers and disk layers,
- to regulate highly dynamic workloads,
- to scale to support very large storage systems,
- not to incur overhead on network or server,
- not to require change on the system hardware or server software. (History suggests that solutions that depend on proprietary or customized hardware are often expensive, hard to maintain, and face more resistance in deployment.)

III. System Design

ASCAR uses a delay-based congestion control strategy that is similar to those that are being used in computer networks. Traffic controllers derive the congestion state along an end-to-end data path by measuring the process time (PT) of each I/O request, which is the duration between the time a client sends one request and the time it receives the acknowledgment from the server after the request is processed. Controllers then adjust the congestion windows according to a set of rules without the need to communicate with either a central scheduler/regulator or other clients. This design makes it possible to support future large-scale storage systems that may have millions of clients. Figure 2 shows the core components of ASCAR, including the ASCAR Rule Manager (ARM) and traffic controllers. The ARM is a small daemon that runs on a management node and governs the discovery, storage, and deployment of traffic rules. Traffic controllers run on storage clients and regulate data streams between clients and servers by following rule sets.

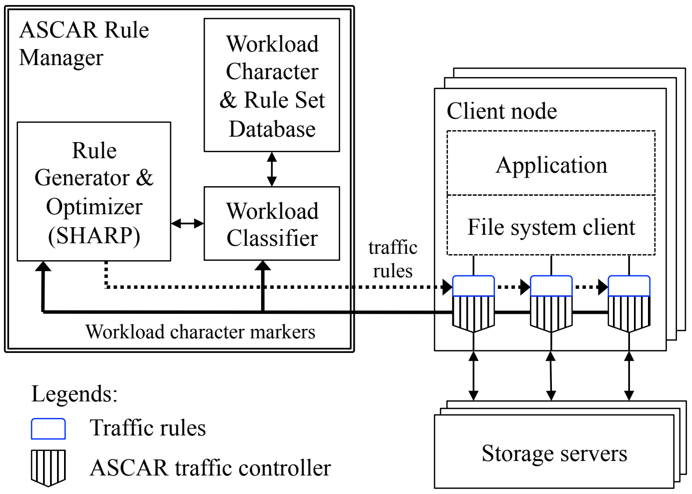


Fig. 2: A parallel storage system with ASCAR

A. *ASCAR Rule Manager*

The ARM only needs to communicate with the traffic controllers when there is a need to change the rule set, for instance when a new workload starts and a rule set needs to be deployed, or when the user wants to further optimize the rule set. Most of the time, the controllers will follow the deployed rule set to regulate the traffic. Therefore the ARM node needs neither high computation power nor fast network. It can be placed on the slower management network of a real cluster or co-located with the cluster's management node. ARM contains SHARP, the rule producer, who takes a workload and an objective function as input and produces traffic rules that guide how the traffic controllers work. The details of SHARP will be introduced in the following sections.

B. *Traffic controller*

ASCAR's controller is a software component that runs on each storage client and regulates one data stream between one client and server pair. Since one controller only handles the traffic between one client and one server, multiple controllers are needed when one client talks to multiple servers at the same time. For the system shown in Figure 2, each client has three traffic controllers because each client needs to communicate with three servers. By using one controller for each data stream, a client can apply different congestion windows and rate limits to different steams, which may have different congestion states. Ideally, a client should monitor the congestion state of each individual storage device, rather than each server. But in practice, information about storage devices is often not exposed to clients, and adding more complexity to either the I/O protocol or server software is what we want to avoid.

Overall, putting traffic controllers on the client side has the following benefits:

- the ability to monitor contention that occurs at any place on the data path, including network, server, and devices,
- the ability to instantly slow down an outgoing I/O stream when congestion is detected (for comparison, server-side traffic controllers cannot instantly tell the clients to slow down when the network is already congested),
- each data path's congestion state can be monitored without being affected by other unrelated data paths,
- easy deployment without the need to change server software.

C. Detecting congestion

As discussed in the Background section, distributed traffic control is a NEXP-hard DEC-POMDP problem. Therefore, as a compromise, we have to seek for good congestion indicators that can reflect the real-time contention state of a connection and, at the same time, are easy to track and calculate. Remy [42], a QoS system for networks, shows three good congestion indicators for network: exponentially weighted moving average (EWMA) of gaps between acknowledgments (ACKs) arriving from the receiver, EWMA of the gap between TCP sender timestamps embedded in ACKs, and current RTT (round-trip delay time) to minimum RTT ratio. We discover that, with the following adaptation, they are also good congestion indicators for storage systems. Instead of using TCP ACKs, we use the stream of replies from servers, which are sent out when servers finish processing I/O requests. We replace RTT with the Process Time (PT), which is the time needed for a server to process one I/O request. The congestion indicators used by ASCAR are summarized in Table I. Each traffic controller keeps a record of these indicator variables and updates them when a reply is received.

TABLE I: Congestion indicators used by ASCAR.

Variable	Description
ack_ewma	exponentially weighted moving average of gaps between server replies
send_ewma	exponentially weighted moving average of gaps between the original sent times of the corresponding requests of the replies received by the client
pt_ratio	current Process Time / shortest Process Time seen so far

D. Traffic rules

A management node (can be co-located with the cluster's management node) deploys a traffic rule set to all traffic controllers before starting a workload. For multiple workloads running together, different rule sets will be deployed to clients running different workloads.

Let tuple C be the congestion state, we have $C = \langle \text{ack\_ewma}, \text{send\_ewma}, \text{pt\_ratio} \rangle$. All possible values of the congestion state variables form the congestion state space $\mathrm{K} : \{\langle \text{ack\_ewma}, \text{send\_ewma}, \text{pt\_ratio} \rangle : \ \text{ack\_ewma}, \text{send\_ewma} \in [0, \infty),\ \text{pt\_ratio} \in [1, \infty)\}$. An action, A, defines how a traffic controller controls the congestion window (cwnd) and speed rate of outgoing I/O requests. The size of the congestion window determines how many I/O requests can be allowed in flight. Many modern distributed storage systems support tunable congestion window, like the max_rpcs_in_flight (MRIF) in Lustre, which limits the maximum number of concurrent requests in flight from a client to the paired server. We use a tuple of two variables to express an action $A = \langle m, b, \tau \rangle$, in which m and b controls the congestion window using equation $\text{cwnd} = m \times \text{cwnd} + b$, and τ is the minimum gap (in milliseconds) between two successive outgoing I/O requests. A rule set, S, contains one or more rules and maps the entire congestion state space K to actions: $\mathrm{S} = \{C \to A : C \in \mathrm{K}, \sum C = \mathrm{K}\}$. After receiving a reply, the traffic controller updates the congestion indicators, searches the rule set to find the corresponding action, and performs the action. The controller also keeps a record of how many times each rule is used and the mean value of each congestion indicator, which will be used by SHARP to refine the rules. Since we observe that there is a delay between changing the congestion window and seeing the action's effect, we limit the number of changes to the congestion window ASCAR can apply to two per second. With this restriction, ASCAR still works well with highly dynamic workloads that have sub-second burst I/O.

In Lustre 2.4, large I/O operations are broken down to small fixed-size requests (1 MB by default). This makes the job of traffic controlling easier, since the number of I/O requests is proportional to the amount of I/O work that needs to be done, and the controllers do not have to calculate the sizes of requests when they limit the I/O rate. For other systems, if I/O requests can be of different sizes, the traffic controllers must take that into consideration.

The functions described above are relatively easy to implement in a modern distributed file system. Our ASCAR prototype is implemented in the Lustre client kernel module; it tweaks max_rpcs_in_flight to control the congestion window and adds delays in the I/O request sending function to control the rate limit. ASCAR's controller introduces two kinds of overhead: the calculation overhead for updating congestion indicators and the control overhead when regulating outgoing requests. The overhead grows linearly as the number of the I/O requests increases but is still negligible comparing to other tasks of the file system on a modern computer. Since none of them is affected by the scale of the whole storage system, ASCAR can linearly scale out to support very large storage system.

One limitation of our current design is that we have yet to implement metadata traffic control in ASCAR even though it can be done in a similar manner. We plan to evaluate metadata heavy workloads in the future. Another limitation is that each client must stick to the rule set to achieve the global optimum. A rogue or selfish client can take advantage of this system by always sending a request without restraint. Since the traffic controller can be implemented in the kernel, this requirement can be easily met in controlled environments where all clients are managed, like in HPC or enterprise systems, but might be a potential issue for wider deployments with unmanaged clients.

E. Objective function

An objective function describes the goal of the optimization. It generates a score that reflects the performance of a workload. The score is used to judge the merit of a rule set. An objective function uses average throughput (*tp*) and speed variance (*var*) as defined below:

$$\begin{aligned} tp &= \frac{\sum_{c=0}^{N} \sum_{t=0}^{T} throughput(c,t)}{N \times T} \\ var &= \text{stddev}\,(throughput(c,t)) \end{aligned}$$

N is the number of clients, T is the length of the benchmark in seconds, *throughput*(*client*, *time*) is the measured throughput of *client* at *time*. *var* covers both spatial and temporal throughput variance.

There are several possible ways for combining the throughput and speed variance as a single score. The most straightforward method is using *tp* as the score when *var* falls within a limited range. To avoid using an arbitrary number as the range limit, we use the coefficient of variation (CV), which is calculated as $k = var/tp$.

$$score(tp, var) = \begin{cases} tp & \text{if } var/tp < k_0 \\ 0 & \text{otherwise.} \end{cases}$$

The scores of this method do not reflect the speed variance. To reflect the variance in the score, we can multiply *var* by a factor α and add it to *tp*:

$$score(tp, var) = tp - \alpha \times var. \tag{1}$$

α controls the weight of stableness versus efficiency.

We have also evaluated objective functions of other forms, such as $U_\alpha(x) = (x^{1-\alpha})/(1 - x)$ (the alpha-fairness metric traditionally used to evaluate allocations of bandwidth on shared links [35]), $(1-\alpha)tp+\alpha/var$, $a\times tp^{\alpha} - b\times var$, and $a\times\log_\alpha(tp) - b \times var$. But they either failed to provide an effective balance between throughput and variance, or were too sensitive to rounding errors. We have not incorporated I/O latency into the objective functions yet, because the applications we evaluate are not very sensitive to high latency. That is planned as a future work.

A real application's I/O workload can be as long as several hours or days, and ASCAR needs to benchmark the workload for many times in order to test different rule sets. For that reason, ASCAR uses shorter signature workloads, which usually run for around 20 to 30 seconds, instead of the real workload. The signature workload needs to cover the *dominant* I/O pattern from the application. A checkpoint signature workload, for instance, should be doing sequential writes concurrently from all clients. ASCAR uses a workload monitor to collect features from an application and generates the signature workload accordingly. The feature set is described in Section III-G.

Because the signature workload is significantly shorter than the real workload, a problem we encountered during the evaluation of ASCAR is that the signature workload's benchmark result might not be stable on a large cluster, especially when the benchmark is too short. Longer I/O workloads generally yield more stable performance results because fluctuations can be amortized over time, but they can also make the optimization process too long. We use the following heuristic in SHARP to find the shortest signature workload. Starting from 10 seconds, SHARP gradually increases the signature workload's run time and runs it for 10 times to determine the speed variance, until the CV is lowered to less than 0.1. This method currently works well enough, and in the future we plan to implement more complex methods for extracting workload's key features, such as those described in [32, 38]. Our experiments show that a good signature workload should have a CV of less than 0.1 and should last about 20 to 60 seconds.

F. Producing a traffic rule set

SHAred-nothing Rule Producer (SHARP) finds the optimal rule set that can maximize the value of the objective function given a specific workload running on a specific storage system. SHARP benchmarks the signature workload on the real system to test the effect of candidate rule sets during the generation process.

On the high level, a SHARP search process consists of a series of epochs, and an epoch can include one or more rounds. Within each epoch, SHARP only optimizes the hottest rule. Conceptually, each candidate rule set can be thought of as a point in the whole possible rule space. For each round, SHARP picks points (candidate rule sets) from the rule space and tests them using the signature workload; more points are picked around the start value of the hottest rule's action variables. In other words, the further we move away from the start value of the hottest rule's action, the fewer candidate rules we try. This is to fine tune the action's variables at their current values, and, at the same time, to avoid being trapped locally. The best rule set from this round will then be used as the start value for next round.

When the best rule set from this round is not better than the best result of the previous round, we have reached the best possible action for this hottest rule. SHARP then splits the hottest rule's congestion state space at the observed mean value of each congestion indicator, and moves on to next epoch.

Algorithm 1 gives a detailed description on how SHARP produces a traffic rule set. SHARP takes two inputs: *benchmark*(*ruleset*) and *objFunc*. Function *benchmark*(*ruleset*) deploys *ruleset* to all clients, runs the benchmark, and returns the test result; *objFunc* is the objective function that calculates a score for the result. Algorithm 1 keeps refining the rule set and never stops. (In practice, it can be stopped after running for the desired optimization time, usually around 10 hours, depending on the length of the benchmark and available machine time.)

There are several implications about using a real storage system instead of a simulator to test candidate rules. First, we cannot run multiple benchmarks concurrently on a real system as we can do with simulators, so the search algorithm has to evaluate candidate rules sequentially; not being able to parallelize this process makes the search process lengthy.

Algorithm 1 The traffic rule set producing algorithm.

```
1: ▷ Producing traffic rule set for benchmark using objective function objFunc
2: procedure ProduceTrafficRuleSets(benchmark, objFunc)
3:     ▷ Begin with one rule that maps the whole congestion state space to one initial action.
4:     $ruleset \leftarrow \langle[0,\infty),[0,\infty),[1,\infty)\rangle \rightarrow \langle 1,0,20000\rangle$
5:     $epoch \leftarrow 0$
6:     ▷ Within each epoch, the number of rules in the ruleset remains unchanged. The optimization focuses on optimizing the hottest
         rule because it has the biggest impact on the overall performance.
7:     while True do
8:         $epochBestScore \leftarrow 0$
9:         ▷ Optimizing the hottest rule within ruleset by generating and evaluating candidate actions for it. The one that yields the
             highest score will be chosen.
10:        $round \leftarrow 0$
11:        repeat
12:            ▷ Construct candidate actions by increasing/decreasing each action variable geometrically from the current value and
                 calculating Cartesian products, e.g., $\{m \pm 0.01, m \pm 0.01k_b, m \pm 0.01k_b^2, m \pm 0.01k_b^3, \cdots\} \times \{b \pm 0.01, b \pm 0.01k_b, b \pm$
                 $0.01k_b^2, b \pm 0.01k_b^3, \cdots\} \times \{\tau \pm 0.01, \tau \pm 0.01k_\tau, \tau \pm 0.01k_\tau^2, \tau \pm 0.01k_\tau^3, \cdots\}$.
13:            $newRulesets \leftarrow$ GenerateCandidateRulesets($ruleset$)
14:            $roundBestScore \leftarrow 0$
15:            for all $r$ in $newRulesets$ do
16:                $result \leftarrow benchmark(r)$
17:                $score \leftarrow objFunc(result)$
18:                if $score > roundBestScore$ then
19:                    $roundBestScore \leftarrow score$
20:                    $ruleset \leftarrow r$
21:                end if
22:            end for
23:            if $roundBestScore > epochBestScore$ then
24:                $epochBestScore \leftarrow roundBestScore$
25:                $epochBestRuleset \leftarrow ruleset$
26:            end if
27:            $round \leftarrow round + 1$
28:        until $roundBestScore < epochBestScore$
29:        ▷ Split the hottest rule's congestion state space at the observed mean value of each congestion indicator into $2^d$ hypercubes,
             where $d$ is the number of congestion indicators.
30:        $ruleset \leftarrow$ SplitRule($epochBestRuleset$)
31:        $epoch \leftarrow epoch + 1$
32:    end while
33: end procedure
```

To mitigate this issue, the signature benchmark needs to be as short as possible, and the search process must be highly efficient. Second, the benchmark results of some workloads fluctuate between runs. To make sure we pick the best rule set in each round, we benchmark the top candidate rule sets at least three times at the end of each round to make sure their results are stable and reproducible. SHARP also detects rule sets that generate high variant results and adaptively runs these specific rule sets more times to rule out the outliers. With these methods, we can use a relatively short signature workload to accelerate the search process and, at the same time, mitigate the problem of volatile results of some benchmarks.

Line 12 of Algorithm 1 is SHARP's rule refining process, which searches for the optimal action for a rule. Due to the complexity of the storage and network stack, the mapping from action space to performance results is a multi-dimensional non-monotonic discrete function. In searching for local optima for this kind of functions, random-restart hill climbing search [30] is a viable option, but it can also be slow and complex. Remy [42] describes a successive approximation method that evaluates candidate actions which are generated from the Cartesian product of action variables. These variables are geometrically increased/decreased from their current values, then uses the optimum rule in this round as the start point for next round, until no better rule can be found. This method tries to find local optima and evaluate other alternative locations at the same time to avoid being "trapped" locally. But Remy uses a fixed common ratio (search step size) for all action variables, and normally needs one or two CPU-weeks to run, which is impractical when running on a real storage system. Based on this method, we use the following heuristic to calculate the step size for different action parameters from a tunable time constraint:

$$step\_size = \sqrt[step]{\frac{var\_upper\_limit - var\_lower\_limit}{delta\_gran}}.$$

delta_gran is delta granularity that determines the granularity to change variables to avoid meaningless small changes in the first few steps. After evaluating a wide range of settings using many workloads, we discover that the following parameters, as shown in Table II on the next page, work best for ASCAR (we use 30 as the upper limit of Lustre's max_rpcs_in_flight).

TABLE II: List of search settings for each action variable.

Act var	Lower limit	Upper limit	Delta gran	Search step
m	0.3	2	0.05	4
b	-2.3	2.3	0.3	4
τ	0	70,000	500	6

Another heuristic SHARP uses to shorten the search time is that in one epoch, SHARP only focuses on the hottest rule, while Remy's search tries to improve all rules, which is too long to run with limited gain in storage systems. This does not mean that the hottest rule attracts all the attention, because it will eventually be broken down to smaller rules until it is no longer the hottest rule. After that, other rules will get the chance to be optimized.

There is no fixed time requirement for this whole optimization process. The longer it runs, the more complex rule sets it produces. In our evaluation, most rule sets contain no more than 29 rules, and we see diminishing returns for going beyond that. Given a 3-dimensional congestion state space, four epochs are needed to evolve one initial rule to 29 rules. The time needed in each epoch depends on how long it takes to find the local optimal parameters, which is proportional to the length of the signature workload and the number of combinations of parameters to evaluate. In our evaluation, most useful rule sets are discovered within 12 hours, which is a big improvement from the several weeks needed by Remy's method. From the discussion above, we can see that the speed of this optimization process is proportional to the length of the signature workload and the complexity of the rule sets, and is unrelated to the size of the underlying storage system. The time needed for this optimization can be fine tuned to meet different requirements of workloads.

Twelve hours is a long time for running an offline training/optimizing system. But it can be justified in many situations. First, many HPC applications generate repetitive workloads, like checkpoints or sequential read/write. These workloads are generated over and over, so spending a couple of hours in exchange for a long-term double-digit percentage increase in performance is reasonable. Second, when the cluster and storage system are under construction or expansion, it is often necessary to find the best parameters for tuning the system for a wide range of workloads; ASCAR can accelerate this process by automating the evaluation and rule discovery process.

G. Bootstrapping ASCAR using common workload set

A real world storage system usually needs to handle a variety of different workloads. We call the set of workloads that a storage system needs to process its *common workload set*. ASCAR uses a database to store the discovered rule sets. An ASCAR-enabled storage system can bootstrap this database by finding the optimal rule set for each of the common workloads.

But the database can only cover a limited set of workloads. For each new workload coming into the system, we need to decide which traffic rule set suits it best. For that purpose, we extract a set of features from the workload and store them in the database. Each record in the database is a mapping from a feature set of a workload to the best rule set for it. When a new workload comes in, ASCAR extract its features and searches the database to find the most similar workload, whose corresponding rule set will then be deployed to optimize this new workload.

Since the similarity of workloads are determined by comparing their features, choosing the correct feature set is crucial for this task. There is little known work which studies the similarity of workloads that can benefit from the same traffic control rules, therefore we have carried out a series of experiments to understand this issue. We pick a workload with known features and let ASCAR generate a good rule set for it that can improve its performance. We then tweak the workload's features, one at a time, and measure whether the generated rule set can still improve its performance (certain cases where two or more features are changed are also evaluated). As an example, we can use a workload that issues random read requests at 1 MB each and sequential write at 8 MB each, with a 10:1 read to write ratio and a known rule set that can increase its throughput by 20%. Based on this workload, we can tweak each of its features. First we generate workloads using 1+1 MB, 1+2 MB, and 1+4 MB read requests and measure the rule set's effectiveness. Then we change the write requests and use 8 ± 1 MB, 8 ± 2 MB, and 8 ± 4 MB, etc., and measure the rule set's effectiveness.

From the contention management point of view, the feature set should express the workload's pressure on the storage system, its randomness, and dynamics (how many ups and downs the workloads have). The more dynamic a workload is, the more gaps between requests it has, which allows for using more aggressive congestion window control. Our experiments, as described in section IV-F below, show that the most important features are:

- type (read, write, create, delete),
- request size,
- positional gap,
- temporal gap.

The positional gap describes the gap between the locations requested by two consecutive I/O operations; it is zero for sequential workloads and non-zero for other workloads. The temporal gap describes the delay between outgoing requests. A real workload can have multiple types of operations combined together, and each type can have different values for request size, positional gap, and temporal gap. A random read/write workload, for instance, can contain a read thread that performs non-stop random reads using 1 MB requests and a write thread that performs a sequential write workload using 50 MB request size with an average temporal gap of 50 ms. The requests we discuss here are file system-level requests between file system clients and servers, which can be different from application level I/O requests. Because of the disk cache, file system-level requests are usually a subset of application level I/O requests.

We did an information gain feature selection [5] using the results of the experiments, which are shown in Figure 4 in the Evaluation section. It shows the following correlations between changed workload features and the effectiveness of an existing rule set:

- a strong correlation with the read to write ratio,
- a strong correlation with the read size for small random reads (when the average read request size is smaller than 40 MB in this test system),
- a weak correlation with read request size when read is sequential,
- a very weak correlation with write size no matter they are random or sequential.

Basically, when comparing the similarity of workloads in terms of traffic control, the most important features are read to write ratio and read size when the reads are random; the other features can be safely ignored. Therefore, in order to cover most possible workloads, we can bootstrap the system's workload/rule set database using common workloads from the combinations of the following feature sets:

- read to write ratio: 2, 4, 6, 8, and >10;
- read: sequential, >40 MB random read, 30~40 MB random, 20~30 MB random, 10~20 MB random, 5~10 MB random, 1~5 MB random.

IV. Implementation and evaluation

A. Our ASCAR prototype

We implemented a prototype of ASCAR for Lustre 2.4.0 running on Linux. The traffic controller is implemented in Lustre's client kernel modules. SHARP is implemented as a user-space program. Table III summarizes the changes of our prototype system against Lustre 2.4.0.

TABLE III: List of ASCAR prototype components (LOC is Lines of source Code changed or added).

File	LOC	Changes
`include/ascar.h`	179	Traffic controller
`osc/osc_request.c`	169	Traffic controller
`osc/qos_rules.c`	116	Traffic rule set parser
`ascar_sharp.sh`	374	SHARP main program
`osc/lproc_osc.c`	110	The `procfs` interface
`gen_candidate_rules.py`	166	Implementation of GenerateCandidateRulesets()
`split_rule.py`	145	Implementation of SplitRule()
`ascar-tests/` (dir)	396	Test cases

The prototype's control client is implemented within the Lustre file system client in the kernel space, thus the computation cost for tracking the congestion indicators is very low. In fact, we did not observe any measurable CPU overhead on the client side. Memory-wise, the data structure for tracking the congestion indicators consumes only 48 bytes per server. Therefore, even supporting very large file systems with thousands of concurrent server connections would not require too much extra memory.

B. Hardware and workloads used for evaluation

The purpose of this evaluation is to understand ASCAR's effectiveness on a variety of workloads. We measured the changes on throughput, bandwidth allocation fairness between clients, and the optimizing time needed. We covered common HPC workloads, such as sequential write (checkpoint) and sequential read (analytical/big data), as well as random read/write workloads as a reference for a wider range of applications. We also included NASA NBP BTIO [43], a realistic checkpoint I/O benchmark derived from a real application.

The evaluation system contains five dedicated servers and five dedicated clients. Our client and server nodes use the same hardware configuration: Intel Xeon CPU E3-1230 V2 @ 3.30 GHz, 16 GB RAM, one Intel 330 SSD for the OS. Each node has one 1 Gb network connection. For the Lustre cluster, each storage server node uses one 7200 RPM HGST Travelstar Z7K500 hard drive, whose raw I/O performance is measured at 113 MB/s for sequential read and 106 MB/s for sequential write. The Lustre cluster has one dedicated metadata node and four storage nodes, which match the default stripe count four. The Lustre file system uses default settings: 1 MB I/O size, 1 MB stripe size, stripe count four. No workload is memory intensive, so all server and clients nodes have plenty of memory for buffering and running worker threads. The cache policies of read and write are both Lustre default – write cache is write-through; the server replies the completion of a write when the data hit the disk. It is worth noting that the evaluation system's network bandwidth is relatively small when compared to modern supercomputers, but its measured aggregated network bandwidth (~500 MB/s) to disk bandwidth ratio (~500 MB/s) is 1:1, which is on par with modern supercomputers [6]. Thus, the configuration of the evaluation system is sufficient for studying the effectiveness and potential of ASCAR on increasing bandwidth utilization and bandwidth sharing fairness in a resource-constrained environment.

We use IOR [20] for generating those sequential workloads. The I/O size of IOR is set to 1 MB, which is common among HPC systems [19]. It also matches Lustre's default I/O size. The configuration of the IOR sequential read/write workloads is: MPI I/O, 5 client nodes, 3 GB in total data volume, and each process has its own dedicated file (unshared). We generate those random workloads using FileBench [36]. The I/O size is set to 1 MB, and we patched FileBench to do direct I/O on Lustre to avoid being affected by local cache. (The patch is needed because FileBench's memory buffer is not aligned and cannot do direct I/O on Lustre.) For the random read and random write workloads, each client accesses a separate 5 GB file (unshared). In order to understand the effect of using a shared file, we also ran the random read/write workload using a shared file (all clients share the same file). The read to write ratio (r:w ratio) is fixed at 1:1 unless otherwise noted. The FileBench fileserver workload creates 1,000 files with a mean file size of 5 MB and mean directory width of 20; there are 50 threads doing create, write, read, append, and delete operations, with a mean I/O size of 1 MB and mean append size of 5 MB. The BTIO Class B workload uses four clients and writes 1.7 GB in total. The BTIO Class C workload uses 25 clients and writes 6.8 GB in total.

TABLE IV: Summary of workloads and performance improvements. Each result is the average of 10 test runs. "TP" is the throughput average (in MB/s) of all clients. "Var" is the standard deviation of throughput (in MB/s) over time and between clients. Those percentages in parens show the increases of TP and Var over non-ASCAR baseline. We used two objective functions: those "(max TP)" traffic rules were generated using an objective function that favors higher throughput, and "(α = 1)" rule sets were generated using an objective function based on Equation 1. "Run time" is the duration of SHARP's optimizing. BTIO is recorded differently because it does burst I/O. We use BTIO's own speed results as TP and speed variance of 10 runs as Var. Since calculating temporal speed variance for a burst-I/O workload is meaningless, we cannot use Equation 1-based objective functions. Instead, the α = 1 columns of BTIO workloads show the results of the rule sets with the least speed variance.

Workload	w/o ASCAR (baseline)			ASCAR (max TP)		ASCAR (TP/Var)		Run time (h)
	BW	Var	Var/BW	Throughput	Variance	Throughput	Variance	
Sequential write	421	18	4%	428 (+2%)	15 (-18%)	382 (-9%)	8 (-54%)	12
Sequential read	445	16	4%	452 (+2%)	14 (-11%)	451 (+1%)	14 (-11%)	9
Random write	219	20	9%	273 (+25%)	22 (+11%)	257 (+17%)	9 (-55%)	19
Random read	238	9	4%	244 (+3%)	4 (-48%)	241 (+1%)	7 (-17%)	7
Random read/write (shared)	48	4	9%	50 (+5%)	4 (0%)	49 (+3%)	4 (-3%)	5
FileBench fileserver	358	25	7%	414 (+16%)	25 (+1%)	412 (+15%)	24 (-4%)	24
BTIO (Class B)	116	7	6%	150 (+29%)	8 (+14%)	146 (+26%)	3 (-57%)	36
BTIO (Class C)	119	3	3%	160 (+35%)	4 (+15%)	159 (+33%)	1 (-55%)	23

C. Evaluation results

The performance improvements of ASCAR are listed in Table IV. For each workload, we run ASCAR using two different objective functions: one uses only *tp* as the score, the other uses Equation 1 with α = 1. With the first function, ASCAR maximizes the throughput without considering speed variance; with the second function, ASCAR takes a balanced approach toward throughput and variance. It can be seen that with the "max TP" objective function, ASCAR improves the throughput for all test workloads, some have the added benefit of lower variance. With the "α = 1" objective function, ASCAR reduces speed variance for all workloads. For HPC systems that favor high throughput, lowering speed variance might not be an important goal; the results generated using α = 1 are included to demonstrate the effect of using different objective functions.

The effect of ASCAR varies between workloads. Since ASCAR is a delay-based congestion control system, it works best with the kind of congestions that can be alleviated by applying self-control to clients. The experimental results show that ASCAR is generally good at improving throughput for write-heavy workloads (25% to 35% increases), but is not as effective on read-heavy workload (2% to 5% increases). Figure 1 shows the effect of ASCAR on a random write workload for lowering speed variance and increasing I/O throughput at the same time.

Sequential write (checkpoint) is a common workload in HPC systems and needs high throughput for parallel writes. Modern distributed file systems, like Lustre, generally handle this kind of simple, sequential write workloads well enough so that the space for further optimization is limited. But ASCAR not only decreases speed variance by 17.9%, it also increases throughput by 1.8% at the same time. The sequential read workload's baseline throughput is 445 MB/s, which is already near the bandwidth limit of the hard drives. Thus it sees only 2% increase in throughput by using ASCAR.

ASCAR increases the throughput of the random read workload by 3%. According to Lustre's manual, random read is considered one of the worst cases for Lustre because the reads from clients cannot be reordered like the write requests and have to be processed in the order they came in, which usually requires a lot of seeks. So the random read throughput is relatively low even when only one client is issuing random read. The random read/write workload using a shared file is very slow on Lustre due to the contention on locks; the access regions of clients are all overlapped, and locks are needed for every I/O request. ASCAR increases its throughput by 5%. Lock contention here is causing clients to spend much time waiting. ASCAR cannot optimize these workloads much, because they exhibit the kind of congestion that cannot be controlled by exercising self-restraint. For these workloads, ASCAR can still be used to lower speed variance if needed (-17% and -3%).

The BTIO benchmark is challenging because of its burst writes. On our test system, BTIO Class B contains 200 checkpoints, each of which lasts for only 0.2 seconds; BTIO Class C contains 200 checkpoints of 0.9 seconds each. No traditional traffic control solution can handle workloads that change this fast, because most of them require slow communication between clients. ASCAR's controllers are built into the file system client and require no network communication, so they are highly responsive. The best rule sets discovered by SHARP can increase the throughput of BTIO Class B by 29.9% and BTIO Class C by 34.8%.

D. How the rules affect the workload

In order to understand how the traffic rules help increasing the bandwidth utilization, we looked into the detail of the rules and measure how ack_ewma, PT ratio, I/O throughput, congestion window (max_rpcs_in_flight), and τ are affected

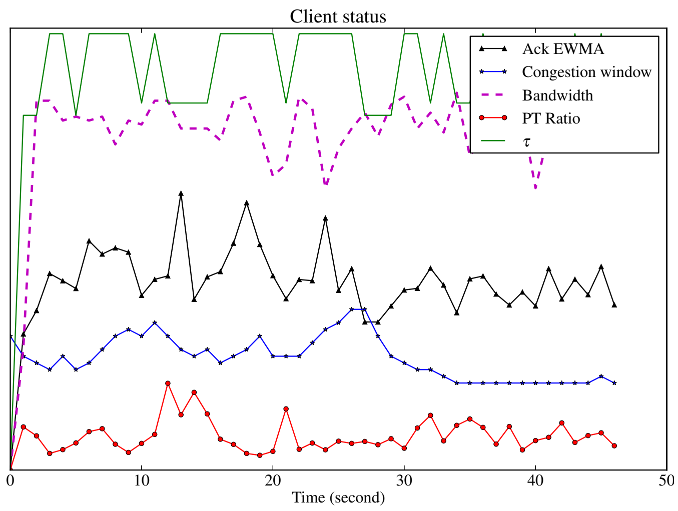


Fig. 3: When running the checkpoint workload, one client's ack/send ewma, PT ratio, congestion window, and throughput change over time.

TABLE V: Excerpt of the best traffic rule set for the checkpoint workload. Twice a second, the ack_ewma and PT ratio are measured and are used to trigger a matching rule. The rule's *m* is multiplied to the congestion window, *b* is added to the congestion window, and τ is used to control the rate limit for the outgoing I/O stream. The "Times" column shows the times each rule is trigger. Average ack_ewma and average PT ratio are the average values of ack_ewma and PT ratio when the rule is triggered.

ack ewma	PT ratio	m	b	τ	Times	Avg. ack ewma	Avg. PT ratio
[41, 48)	[2.4, 4.5)	1	-1.7	33	3011	45	3.2
[48, ∞)	[0, 4.5)	1	0.9	40	7426	60	2.6

by the traffic rules. Figure 3 shows the states of one I/O connection during a checkpoint workload when it is controlled by ASCAR. This ASCAR rule set contains 15 rules. Two of the most often triggered rules account for 51% of all rule triggers and are shown in Table V. The first rule is triggered when the ack_ewma and send_ewma are low (the avg. ack ewma is 45) but PT ratio is high (avg. PT ratio is 3.2). This rule shrinks the congestion window by 1.7 and lowers τ. This means that the I/O requests will be sent faster, but fewer are allowed to be outstanding. The second rule is triggered when ack_ewma and send_ewma are high (average at 60) but PT ratio is low (average at 2.6), and this rule increases the congestion window by 0.9 and increases τ. This means a lower sending rate but more requests are allowed to be outstanding. PT ratio is a more real-time reflection of the current congestion state, while ack_ewma and send_ewma show the trend of the congestion state because history values are taken into calculation. In other words, under these rules, if the connection is not congested (low EWMA values) but the situation is getting worse (high PT ratio), the client would send requests quickly to take advantage of the current low latency, but would also restrain from sending too many requests until the previous requests are finished to prevent further worsening the connection. On the contrary, if the connection is congested (high EWMA values) but the situation is improving (low PT ratio), the client would slowly send requests to prevent clogging the connection, but would also send more requests before hearing back from the server since we already know that the condition is getting better. From these analyses we can see that the machine generated rules are reasonable. It used to be difficult to pick up the correct values for the rate limits and congestion window change speeds without doing many experiments. Now SHARP can automate this process.

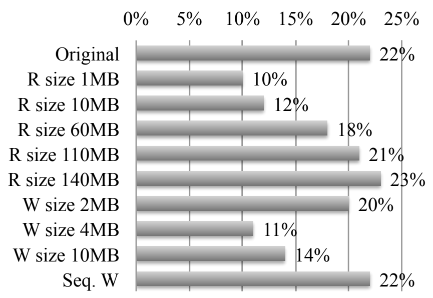
We dig deeper into the layers between the application and storage devices to find out where the performance improvement comes from. Different workloads have different bottlenecks. We observe that when the bandwidth utilization is increased, the measured hard drive access time is also decreased. The access time measures the average time needed for the drive to process one fixed size request. When I/O streams from different clients are interlaced, extra seeks between I/O requests lead to high access time. This slow-start, fast-fallback policy can be used to mitigate these extra seeks. In practice, the access time of disk drives depends on how the I/O requests interlace and the characters of the drives, such as the seek time, rotational latency, and re-ordering of requests. Designing the optimal traffic rules requires knowledge of these properties and how the I/O requests are interlaced with other requests. Using an automated optimizer, like SHARP, frees the designer from the work of measuring and understanding the internals of each involved device.

Since the workloads we picked for this evaluation are distinctive from each other, their related traffic rules are also different. This implies that the optimization process should be run at least once for each different category of workloads. It is not needed for workloads within the same category. Most checkpoint workloads exhibit a similar sequential write I/O pattern and should be able to share the same best set of traffic rules.

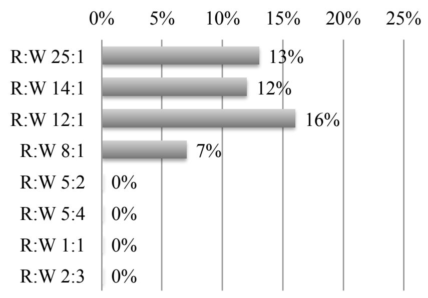
E. Search efficiency

Unlike Remy [42], which can use a network simulator to evaluate candidate rule sets, ASCAR has to use the real storage system, which is often busy and cannot be occupied exclusively for a very long time, therefore search efficiency is very important for ASCAR. We have introduced several heuristics in SHARP to improve the search efficiency over Remy's search method. In one epoch, SHARP only optimizes the hottest rule instead of trying to improve all rules. Also, SHARP is more aggressive on splitting rules when a local optimal rule set is discovered.

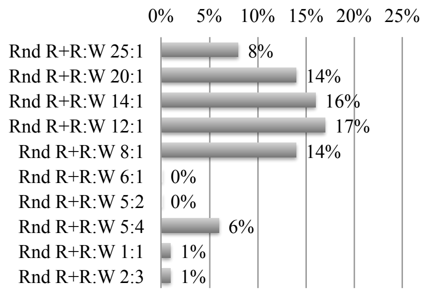
We compared the efficiency of different traffic control rule search methods. The efficiency of a search method is measured by how much the best discovered rule set can improve the I/O performance after a fixed search time. A highly efficient search method discovers rule sets that better improve the I/O



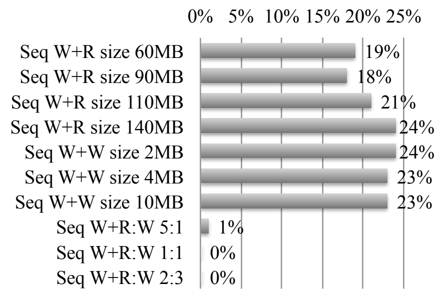
(a) Changed read request size



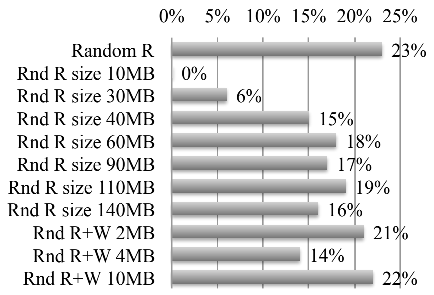
(b) Changed read to write (r:w) ratio



(c) Changed r:w ratio and read from sequential to random



(d) Changed write from random to sequential, combined with changed read size, write size, and r:w ratio



(e) Changed read from sequential to random, combined with changed read and write request size

Fig. 4: A rule set's effect on changed workloads. The original workload is a sequential read (100 MB per request) plus random write (1 MB per request). The best rule set discovered by ASCAR increases its throughput by 22%. Each bar here shows the same rule set's effect on throughput when we changed one or two parameters of the workload.

performance in a shorter time. The results are shown in Figure 5. ASCAR generates better rule sets in a shorter time than Remy's search method.

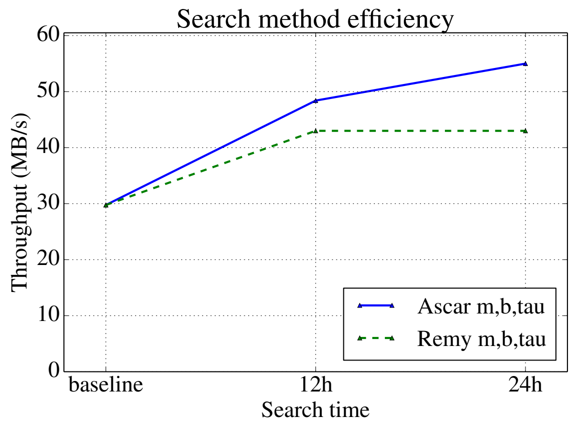


Fig. 5: Comparing the search efficiency of different methods on one workload (the same workload as we used in section IV-F).

F. The effectiveness of rules on changed workloads

This section of evaluation focuses on measuring the effectiveness of a rule set after the workload is changed. This evaluation is important for determining whether an ASCAR system is stable. The ASCAR system is stable if a small change in the workload does not cause much change to the effectiveness of an existing traffic rule set on increasing the workload's performance; in other words, we need to know whether the rule sets generated by ASCAR are overfitted to the specific training workloads. Studying the effectiveness of a rule set on changed workloads also helps us to understand how to calculate the similarity of different workloads in terms of selecting traffic control rules. When a new workload starts, ASCAR needs to search its workload/rule set database for the most similar known workload that already has a generated rule set.

Due to the space limitation, we present the result of tweaking only one workload in Figure 4. It is a fairly typical result. The workload is a sequential read workload using 100 MB requests plus random write using 1 MB requests. We have the following observations:

- changing the r:w ratio has the biggest impact on the rule set's effectiveness,
- changing the read size does not change the effectiveness unless the read size is very small (<= 10 MB),
- changing the write size to a larger value also affects the rule set's effectiveness, but as the write size becomes larger than 10 MB, the rule set becomes useful again.

The rules we derived from these observations are described above in section III-G.

V. Related Work

There are mainly two categories of methods for storage traffic management. The first category focuses on removing contention during the design phase, usually by optimizing the routing of data path or how applications access data, such as ADIOS [22].

The second category focuses on managing contention at the runtime after the system is put into production, and includes scheduling-based methods and delay-based methods. Many early systems work on a single-server [7, 25, 34], or for a specific kind of workloads, such as media recording and streaming [23, 46, 47] or virtual machines [10].

One step further are the later systems that support Distributed Rate Limiting (DRL), which limits the aggregated rate (or throughput) of distributed agents by some preset criteria. They can be roughly classified to four categories:

- Controllers work from the client side [12, 16, 18].
- Controllers work from the server side [10, 14, 24, 40, 45].
- Controller works from a proxy layer between the clients and servers [16, 21, 49].
- A mix of the above three methods [4, 50].

All scheduling-based and delay-based systems mentioned above face several challenges. Some of them require frequent communication or synchronization between rate limiters to function [12, 24, 40, 49, 50]. They have the following disadvantages: a centralized controller is a single-point of failure and a single-point of contention that does not scale out; those systems cannot handle highly dynamic workloads because remote synchronization and controlling is often slow; the volume of periodic communication and synchronization between clients often becomes prohibitively large when the system scales beyond a few hundreds of clients; they impose extra burden on the network and do not work well when the network latency is high due to congestion. Server-sided and proxy DRL solutions, by nature, cannot handle network congestion well since they cannot limit client-side sending.

The next issue is capability discovery – finding the optimal I/O bandwidth for a specific workload running on a specific system. This optimal value depends on nearly every aspect of the workload and the system; we already know that many features, like the I/O request size, randomness, network topology, etc., can all affect this value. This optimal value is important because it is the basis for allocating bandwidth to clients. Current traffic control solutions either assume a certain fixed value as the top performance of the system [14, 40, 44], use a time-sharing scheduler [10, 12, 17, 24, 29, 34, 45, 49], or employ a simple control-theoretic heuristic: slow-start, fast fallback [37, 44, 50]. Some of them require that the storage system's performance conforms to a mathematical model [18, 39]. Manually determining the capabilities of devices and servers is difficult because of the reason we mentioned above: the optimal bandwidth changes by workload. In practice, these solutions only shift the burden of determining the optimal bandwidth to the administrators.

With all these mechanisms, the system administrators still face the problem of determining the best parameters. This task can be daunting because a file system or a traffic control solution may have hundreds of parameters to tune. For the slow-start, fast-fallback method described above, if the start is too slow or the falling back is too fast, the system's capacity is wasted; if the speed increasing is too fast or the falling back is not fast enough, the system becomes unstable.

Remy [42] produces network QoS rules given parameter ranges of the target network. It requires a descriptive model of the target network and uses a simulator to generate test environments to evaluate workloads. In the process, Remy generates candidate rules by increasing/decreasing all variables of the busiest action by a geometric progression with a fixed common ratio. ASCAR does not need a model of the storage system, which can be very difficult to get. Instead, ASCAR uses the real storage system. Using the real system can lead to more accurate results because there's no simulator that can accurately simulate a large and complex distributed storage system. SHARP improves the QoS rule production process with several heuristics that greatly reduce the number of rules to evaluate. Hippodrome [2] automatically changes and evaluates the system design to prevent device overload. Unlike ASCAR, which only needs controllers on each clients, Hippodrome requires intrusive and radical changes to the whole system.

We view the following work on data placement and provisioning as complementary to our work. Improved data placement can further reduce network congestion. HPC storage systems can explore data locality by placing data near to its consumer, as shown by Shipman et al. [33], or by rebalancing data placement when a hotspot is detected [21, 48]. Other related data placement optimization works include Pesto [11] and BASIL [9] for virtual machine deployment. An alternative to complex traffic management mechanism is to provide higher than necessary bandwidth by generously over-provisioning, where the total system capacity is based on peak workload estimates. Alvarez et al. [1] proposed a framework for calculating the resource demand to avoid too much waste. Another problem is converting high-level SLA or application time requirements to a precise QoS I/O bound. Zhang et al. proposed machine learning methods for this purpose [51].

VI. Conclusions and Future Work

ASCAR successfully increases the bandwidth utilization for all the workloads we have evaluated, and can decrease speed variance in many cases at the same time. These improvements come with no need to change either hardware or server software. And the whole optimization process is unsupervised, requiring no human knowledge of either the workload or the system. ASCAR can be safely evaluated in production environments and makes little assumption of the storage system and is applicable to a wider range of different systems.

ASCAR is designed to be scalable, and there is no performance bottleneck in ASCAR. The time and space complexity of all the algorithms used in ASCAR are not related to the number of the nodes in the system, so, theoretically, ASCAR can scale to support millions of nodes. Even though we do not have resources to validate this yet, some of ASCAR's design ideas have already been proven to be highly scalable in managing computer network congestion [42].

There are several limitations of the current generation of ASCAR. First, the long offline studying/optimizing process, while reasonable for repetitive workloads, can be improved by using an online optimizing system. Second, the user may need to run the optimizer for each distinct workload. Third, we are assuming that all the clients stick to the rules and there is no rogue player. This should not be a problem for HPC systems, but need to be addressed when deploying in other uncontrolled environments, like in the cloud.

We are actively working on a more complex and adaptive online SHARP optimizer to address some of these issues. The online version of SHARP will be able to tweak and measure the effectiveness of traffic rules online while the workloads are running. An online optimizer would be a big improvement over the current offline SHARP optimizer, because the online optimizer would not need to occupy the system (or a part of it) exclusively for a long time. As a first step, we are working on optimizing repetitive workloads, like the checkpoint workloads in HPC systems. This repetition gives us the opportunity to try different traffic rules for each checkpoint, gradually polishing the rules and generating better rules. By analyzing the performance of existent traffic rule set, a new rule set will be designed and tested during the next checkpoint run. We are also working on evaluating ASCAR on a larger scale, including how to deal with multiple workloads. We plan to cover more categories of workloads, such as disk rebalancing and error recovery workloads in the future. We are also studying the effectiveness of supporting heterogeneous storage devices, which can be a mixture of hard drives, SSDs, and full in-memory storage, such as [26, 28]. Since ASCAR uses a separate controller for each connection, we believe it can handle mixed devices well if certain design conditions can be met.

The SHARP algorithm is efficient at producing rules for complex rule- or policy-based control systems. A future research topic is evaluating this algorithm with other similar problems, such as data placement, device power management, load balancing, and data re-routing in large-scale storage systems. We are also actively working on improving the algorithm in the following aspects:

- To use only a part of the storage system to produce traffic rule sets that can be used for the whole storage system. This will be useful when using the whole storage system is too expensive for traffic rules generation.
- The current traffic rule set producing process only optimizes the hottest rule in one epoch. This can be improved by using more complex searching processes, such as simulated annealing, random-restart hill climbing [30], and Greedy Randomized Adaptive Search Procedures (GRASP) [8].

In some use cases, bounded throughput is still needed in addition to proportional allocation [24]. We are exploring methods to design customizable traffic rules for this purpose, and we will also investigate the possibility of combining ASCAR with other traffic solutions to provide more functionalities.

Lustre is mainly used by the HPC community, but the mechanism of ASCAR is not limited to HPC and should be able to be applied to other environments running other systems. We are seeking collaborators who want to provide a test environment and to bring ASCAR to other systems.

We will publish the source code of our prototype and raw experimental data at http://www.ssrc.ucsc.edu/ascar.html. The prototype works out-of-the-box on Lustre clients and can be safely evaluated in a production environment.

Acknowledgments

This research was supported in part by the National Science Foundation under awards IIP-1266400, CCF-1219163, CNS-1018928, the Department of Energy under award DE-FC02-10ER26017/DESC0005417, Symantec Graduate Fellowship, and industrial members of the Center for Research in Storage Systems. We would like to thank the sponsors of the Storage Systems Research Center (SSRC), including Avago Technologies, Center for Information Technology Research in the Interest of Society (CITRIS of UC Santa Cruz), Department of Energy/Office of Science, EMC, Hewlett Packard Laboratories, Intel Corporation, National Science Foundation, NetApp, Sandisk, Seagate Technology, Symantec, and Toshiba for their generous support. We also thank the faculty and students of SSRC, especially Christina Strong, Lincoln Thurlow, and Andy Hospodor for their help, guidance, and valuable comments.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, Nov. 2001. ISSN 0734-2071. doi: 10.1145/502912.502915.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[3] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4):819–840, Nov. 2002. ISSN 0364-765X. doi: 10.1287/moor.27.4.819.297.

[4] D. Bigelow, S. Iyer, T. Kaldewey, R. Pineiro, A. Povzner, S. A. Brandt, R. Golding, T. Wong, and C. Maltzahn. End-to-end performance management for scalable distributed storage. In *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW '07)*, Reno, NV, 2007.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1st edition, 2007.

[6] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading hpc i/o performance using a scientific-application

derived benchmark. In *Proceedings of SC07*, pages 1–12, Nov 2007. doi: 10.1145/1362622.1362636.

[7] A. Elnably, H. Wang, A. Gulati, and P. Varman. Efficient QoS for Multi-Tiered Storage Systems. In *Proceedings of the 4th Workshop on Hot Topics in Storage and File Systems (HotStorage '12)*, Berkeley, CA, 2012. USENIX.

[8] T. A. Feo and M. G. C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2):109–133, 1995. ISSN 0925-5001. doi: 10.1007/BF01096763.

[9] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing Across Storage Devices. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2010. USENIX Association.

[10] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*, pages 19:1–19:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038935.

[12] A. Gulati, G. Shanmuganathan, X. Zhang, and P. Varman. Demand Based Hierarchical QoS Using Storage Resource Pools. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[13] D. A. Hayes and G. Armitage. Revisiting tcp congestion control using delay gradients. In *networking11p2*, NETWORKING'11, pages 328–341, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20797-6.

[14] S. Ihara. New Quality of Service policy for lustre based on the lustre network request scheduler (NRS). In *Lustre Admins and Developers Workshop 2013*, Sept. 2013.

[15] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM Computer Communication Review*, 19(5):56–71, Oct. 1989. ISSN 0146-4833. doi: 10.1145/74681.74686.

[16] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the 2004 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005694.

[17] T. Kaldewey, T. Wong, R. Golding, A. Povzner, C. Maltzahn, and S. Brandt. Virtualizing disk performance. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, Apr. 2008.

[18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4): 457–480, 2005.

[19] Y. Kim, R. Gunasekaran, G. Shipman, D. Dillow, Z. Zhang, and B. Settlemyer. Workload characterization of a leadership class storage cluster. In *Proceedings of the 5th Petascale Data Storage Workshop (PDSW '10)*, pages 1–5, Nov 2010. doi: 10.1109/PDSW.2010.5668066.

[20] Lawrence Livermore National Laboratory. IOR software. http://www.llnl.gov/icc/lc/siop/downloads/download.html, 2003.

[21] Q. Liu, N. Podhorszki, J. Logan, and S. Klasky. Runtime I/O Re-Routing + Throttling on HPC Storage. In *Proceedings of the 5th Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, Berkeley, CA, 2013. USENIX.

[22] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-156-9. doi: 10.1145/1383529.1383533.

[23] D. D. E. Long and M. N. Thakur. Scheduling real-time disk transfers for continuous media applications. In *Proceedings of the 12th IEEE Symposium on Mass Storage Systems*, pages 227–232, Monterey, Apr. 1993. IEEE.

[24] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in Large Disk Arrays. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*, pages 245–254, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998638.

[25] M. P. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 57–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043563.

[26] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[27] Open Scalable File Systems, Inc. The Lustre® file system. http://www.opensfs.org/, 2014.

[28] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *Operating Systems Review*, 43(4):92–105, Dec. 2009.

[29] A. Povzner, D. Sawyer, and S. Brandt. Horizon: Efficient Deadline-driven Disk I/O Management for Distributed Storage Systems. In *Proceedings of the 19th IEEE International Symposium on High Performance Distributed*

Computing (HPDC '10), pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851478.

[30] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition, 2009. ISBN 0136042597.

[31] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[32] H. Shan and J. Shalf. Using IOR to Analyze the I/O performance for HPC Platforms. In *Cray User Group 2007 Proceedings*, 2007.

[33] G. M. Shipman, D. Dillow, S. Oral, F. Wang, D. Fuller, J. Hill, and Z. Zhang. Lessons learned in deploying the world's largest scale Lustre file system. In *Cray User Group 2010 Proceedings*, 2010.

[34] D. Skourtis, S. Kato, and S. Brandt. QBox: Guaranteeing I/O Performance on Black Box Storage Systems. In *Proceedings of the 21th IEEE International Symposium on High Performance Distributed Computing (HPDC '12)*, pages 73–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0805-2. doi: 10.1145/2287076.2287087.

[35] R. Srikant. *The Mathematics of Internet Congestion Control*. Systems & Control: Foundations & Applications. Birkhäuser Boston, 2004. ISBN 9780817632274.

[36] SUN Microsystems and File system and Storage Lab (FSL) at Stony Brook University. FileBench. http://filebench.sourceforge.net/, 2014.

[37] A. S. Tanenbau. *Computer Networks (5th Edition)*. Prentice Hall, 2010.

[38] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual Machine Workloads: The Case for New NAS Benchmarks. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 2013. ISBN 978-1-931971-99-7.

[39] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 588–595, 2004. doi: 10.1109/MASCOT.2004.1348316.

[40] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2007. USENIX Association.

[41] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[42] K. Winstein and H. Balakrishnan. TCP ex Machina: computer-generated congestion control. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '13)*, pages 123–134, Hong Kong, 2013. ISBN 9781450320566.

[43] P. Wong and R. F. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing (NAS) Division, 2003.

[44] J. C. Wu and S. A. Brandt. The design and implementation of AQuA: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, College Park, MD, May 2006.

[45] J. C. Wu and S. A. Brandt. Providing Quality of Service Support in Object-Based File System. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, Sept. 2007.

[46] J. C. Wu, S. Banachowski, and S. A. Brandt. Automated QoS support for multimedia disk access. In *Proceedings of Multimedia Computing and Networking Conference 2005 (MMCN '05)*, pages 103–107, San Jose, CA, Jan. 2005.

[47] J. C. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk scheduling for multimedia systerms and servers. In *Proceedings fo the ACM International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '05)*, pages 189–194, Stevenson, WA, June 2005. ACM.

[48] J. C. Wu, B. Hong, and S. A. Brandt. Ensuring performance in activity-based file relocation. In *Proceedings of the 26th IEEE International Performance Conference on Computers and Communication (IPCCC '07)*, pages 75–84, 2007. doi: 10.1109/PCCC.2007.358881.

[49] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. Figueiredo, and S. Seelam. vPFS: Bandwidth Virtualization of Parallel Storage Systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[50] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Transactions on Storage*, 2(3):283–308, Aug. 2006. ISSN 1553-3077. doi: 10.1145/1168910.1168913.

[51] X. Zhang, K. Davis, and S. Jiang. QoS Support for End Users of I/O-intensive Applications Using Shared Storage Systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, pages 18:1–18:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063408.