Title
ASCAR: Fully Automatic Storage Contention Management System

Permalink
https://escholarship.org/uc/item/64w438xg

Author
Li, Yan

Publication Date
2017

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ASCAR: FULLY AUTOMATIC STORAGE CONTENTION MANAGEMENT SYSTEM

A dissertation submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Yan Li

March 2017

The Dissertation of Yan Li
is approved:

______________________________
Professor Darrell D. E. Long, Chairman

______________________________
Professor Ethan L. Miller

______________________________
Professor Ahmed Amer

______________________________
Tyrus Miller
Vice Provost and Dean of Graduate Studies
# Table of Contents

List of Figures ........................................ v
List of Tables ........................................ vii
Abstract ........................................... viii
Dedication ........................................... x
Acknowledgments ...................................... xi

1 Introduction ..................................... 1

2 Background ..................................... 10
   2.1 Congestion Control at Scale .......................... 10
   2.2 Dynamic Parameter Tuning ............................... 17

3 SHARP: Rule-based Contention Control .............. 23
   3.1 System Design ........................................ 23
       3.1.1 ASCAR Rule Manager ............................... 24
       3.1.2 Traffic Controller .................................. 25
       3.1.3 Detecting congestion ............................... 26
       3.1.4 Traffic rules ...................................... 27
       3.1.5 Objective function ................................. 29
       3.1.6 Producing a traffic rule set ...................... 30
       3.1.7 Bootstrapping ASCAR using common workload set 35
   3.2 Implementation and Evaluation ...................... 38
       3.2.1 Our ASCAR prototype ............................... 38
       3.2.2 Hardware and workloads used for evaluation ...... 39
       3.2.3 Evaluation results ................................. 40
       3.2.4 How the rules affect the workload ................ 44
       3.2.5 Search efficiency .................................. 46
       3.2.6 The effectiveness of rules on changed workloads 47
List of Figures

2.1 Comparison of a random write workload when running with and without ASCAR control ........................................... 11
3.1 A parallel storage system with ASCAR ................................................................. 24
3.2 One client’s ack/send EWMA, PT ratio, congestion window, and throughput when running the checkpoint workload ................................................. 44
3.3 Comparison of the efficiency of different search methods ................................................................. 47
3.4 A rule set’s effect on changed workloads ........................................................................ 49
4.1 A sample write workload to illustrate the terms used in Pilot ........................................... 59
4.2 Throughput of a multi-node random read write workload showing the setup phase, the warm-up phase caused by caching effect, and the cool-down phase caused by shutting down of I/O threads, ..................................................... 61
4.3 Sample sequence of work amounts for the first seven rounds automatically calculated by Pilot ........................................................................... 65
4.4 Interleaved execution of benchmarks can help to lower temporal correlation between rounds .................................................................................... 70
4.5 The actual and Pilot predicted time for writing certain amount of data sequentially ........................................................................... 77
5.1 CAPES architecture ........................................................................................................... 80
5.2 Throughput of different congestion window sizes in StorageGameSim ..................................................... 91
5.3 StorageGameSim converge time under different noise levels ......................................................... 94
5.4 Overview of random read write workloads evaluated with CAPES ............................................. 97
5.5 Overview of Filebench file server and sequential write workload evaluated with CAPES .............................................. 99
5.6 Filebench file server workload throughput with and without CAPES tuning ............................................. 100
5.7 Prediction error during the training process ............................................. 101
5.8 Training’s impact on the workload performance ............................................. 102
List of Tables

3.1 Congestion indicators used by ASCAR . . . . . . . . . . . . . . . . . . 26
3.2 List of search settings for each action variable . . . . . . . . . . . . . 34
3.3 List of ASCAR prototype components and lines of source code . . . . 38
3.4 Summary of workload performance improvements by ASCAR . . . . . 43
3.5 Excerpt of the best traffic rule set for the checkpoint workload . . . . 45

4.1 The time needed to reach the desired width of CI (10% of result mean) using various methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

5.1 Successful rate for converge test . . . . . . . . . . . . . . . . . . . . . 93
5.2 List of hyperparameters and their values used in CAPES evaluation . . 96
5.3 List of technical measurements of the CAPES evaluation . . . . . . . . 103
Abstract

ASCAR: Fully Automatic Storage Contention Management System

by

Yan Li

High-performance parallel storage systems, such as those used for high-performance computing and data centers, can suffer from performance degradation when a large number of clients are contending for limited resources, like bandwidth. This kind of contention is common among any storage systems that have to serve a large number of users or applications, and can lower the performance of the system and cause unpredictable speed variances. The performance degradation can cause significant resource waste for large storage systems.

This thesis describes 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 is a fully autonomous software system. It requires no change to the hardware or system design, and integrates well with existing systems. On the high level, ASCAR measures the system’s and workload’s running states and tunes one or more parameters in order to push a user designated performance metric higher. The metric can be any measurable properties of the system or the workloads, such as I/O throughput, latency, or application runtime.

ASCAR includes two sets of algorithms for different tuning requirements. The first method is rule-based. Each client’s control agent regulates the traffic independently according to a preloaded rule set. 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 the SHAred-nothing Rule Producer (SHARP) that produces and refines control rules iteratively without the need of human supervision. SHARP systematically explores the solution space of possible rule designs and evaluating the target workload under the candidate rule sets.

The second method uses a neural network-based reinforcement learning method called Q-learning to perform continual analyzing of the states of the system and workload, and to tune the values of the traffic control parameters. This method is named CAPES, Computer Automated Performance Enhancement System. Deep Q-Learning (DQL) is an unsupervised machine learning method that requires no prior knowledge of the system or workloads, does not need existing dataset for training, and performs well on diverse input data featuring long delays between action and reward. Most complex storage systems show such a property: there is usually a long delay between setting a traffic control parameter and the change in traffic metrics. A multilayered deep neural network is chosen as DQL’s value function, and experience replay is used to mitigate overfitting.

SHARP and CAPES are synergistic and cover different tuning requirements. SHARP is best for relatively stable workloads, requires no runtime communication between agents, and therefore can easily scale to support very large storage systems. CAPES is best for tuning unpredictable workloads and requires communication between monitoring and control agents.

Evaluation of SHARP and CAPES are done on the Lustre parallel file system. Lustre distributes I/O requests to many servers in parallel in order to reach high performance, and can multiply the number of application I/O requests, causing contention throughout the system. SHARP and CAPES are both effective at improving the throughput of the test workloads during the evaluation, some by as much as 45%.
Dedicated to my loving parents, 李进 and 刘亚丽.
Acknowledgments

I would like to express my special appreciation and thanks to Professor Darrell Long, my dissertation advisor and academic father, for his guidance and contribution to this work, and for his friendship during my study at the University of California, Santa Cruz. He is not only my mentor in computer science, but also in mathematics, physics, history, language, culture, politics, and many other aspects.

I am deeply indebted to Professor Ethan Miller, Professor Ahmed Amer, and Brent Gorda for their guidance, support, and encouragement. I could not have finished my research and dissertation without their help. I wish to thank Pamela and Don Urfer, who gave me the warmest welcome and helped me to settle down when I first set foot upon this land that is 6004 miles away from where I was born. I also wish to thank Andy Hospodor, Cynthia McCarley, and every student in the Storage Systems Research Center, we had a great time together. You guys rock!

My grandparents, parents, friends, and my other family are a very important part of my life, and through their love and support have contributed significantly to this work. My grandfather, Weixin Li, raised me up under the notion that every man should get a PhD. My father, Jin Li, sparked my curiosity in science and led me to this fantastic world of computer and electronics. My mother, Yali Liu, has always been supportive of my whimsical decisions to pursue whatever dreams I have. My wife, Chao, provided support and encouragement through the many years we have been together; I am very fortunate to have her.

Several members of the Storage Systems Research Center have contributed to this work. This dissertation includes material of the following previously published papers. It is my pleasure to acknowledge my coauthors of these papers.
• Yan Li, Xiaoyuan Lu, Ethan L. Miller, Darrell D. E. Long. “ASCAR: Automating Contention Management for High-Performance Storage Systems.” *Proceedings of the Thirty-first International Conference on Massive Storage Systems and Technologies* (MSST 2015), May 30–June 5, 2015, Santa Clara, CA, USA. © 2015 IEEE. With permissions from Xiaoyuan Lu. Ethan Miller and Darrell Long directed and supervised the research which formed the basis for this paper. I have done the ideation, design, implementation, evaluation, and drafting of the paper with Xiaoyuan’s help on Section V.F (The effectiveness of rules on changed workloads).


• Unpublished work: Yan Li, Kenneth Chang, Oceane Bel, Ethan L. Miller, Darrell D. E. Long. “CAPES: Unsupervised System Performance Tuning Using Neural Network-Based Deep Reinforcement Learning.” Under review. With permissions from Kenneth Chang and Oceane Bel. Ethan Miller and Darrell Long directed and supervised the research which formed the basis for this paper. I have done the ideation, design, implementation, evaluation, and drafting of the paper with Kenneth Chang and Oceane Bel’s help on design, implementation, and paper writing.
This dissertation was completed with the aid of the National Science Foundation under awards IIP-1266400, CCF-1219163, CNS-1018928, CNS-1528179, by the Department of Energy under award DE-FC02-10ER26017/DESC0005417, by a Symantec Graduate Fellowship, by a grant from Intel Corporation, and by industrial members of the Center for Research in Storage Systems.
Chapter 1

Introduction

Recent advance in computing technology has put mobile computing devices into everybody’s pocket, and people are spending more and more time using these devices. Mobile devices and applications such as social network and smart personal assistant usually rely on cloud computing platforms to provide the computation and storage that the mobile devices can not provide. This trend has resulted in a quick and fast expansion of data centers across the globe. IDC forecasts worldwide cloud IT infrastructure market to grow 24% year over year in 2015, driven by public cloud datacenter expansion [29].

Data center is very costly to build and operate. The Department of Energy reported in 2010, the energy cost during the data center equipments’ life time exceeded the cost of the equipments, and by 2020, the carbon footprint of data centers will exceed the airline industry [13]. Because the US is targeting a 17% reduction in greenhouse gas emission by 2020, it is critical to increase the efficiency of data center operations in order to offset the carbon footprint incurred by the rapid expansion of cloud computing. As pointed out by the United Nation’s Framework Convention on Climate Change (UNFCC), almost one-half of the necessary climate mitigation will need to come from improved energy efficiency [69].
For a large data center, a 1% increase in system throughput can easily save the operator hundreds of thousands of dollars per hour. Yet when it comes to performance, there are a lot of ad hoc solutions and workarounds, but little science and even less theory. The performance tuning technologies people use in today’s large scale storage systems are similar to what people used in smaller systems. They require human expertise and time, and can be both costly and ineffective when handling data center-scale storage systems.

This thesis focuses on contention control (also called “congestion control” in other published material). Large scale storage systems usually have high intrinsic 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. Contention control deals with how to manage this kind of contention to increase system efficiency.

A storage system without contention management is like a traffic system without traffic lights. 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. 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.

**Overall Design**  This thesis describes the Automatic Storage Contention Alleviation and Reduction (ASCAR) system. ASCAR is designed for increasing the bandwidth utilization and reducing speed variance. It is a delay-based general-purpose client-side I/O rate control system.

ASCAR includes the following major components:

- **Controllers**: resides on the target system’s nodes, monitors the states of the system and workloads, and tweak parameters such as the upper speed limit and the congestion window size of each connection according to rules or instructions received.
- **An automatic performance measuring component named Pilot.**
- **SHAred-nothing Rule Producer (SHARP)**: generates control rules for control agents.
- **Computer Automated Performance Enhancement System (CAPES)**: neural network-based reinforcement learning controller.

**Controllers**  An ASCAR controller can work in two modes:

- **Rule Control Mode**: follows a preloaded rule set generated by SHARP.
- **Agent Mode**: works as agents for the CAPES Control Node by reporting system performance indicators and performing traffic control according to instructions.
In the Rule Control Mode, the 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. In the Agent Mode, a controller works with CAPES and can conceptually be thought as a Monitoring Agent plus a Control Agent. A Monitoring Agent gathers Performance Indicators from the target system’s nodes and sends them to the CAPES Control Node. Performance Indicators are system measurements that are related to the system’s operating status. The CAPES Control Node sends back Actions to the controller’s Control Agent, and the Control Agents makes the appropriate changes on the target nodes when actions are received.

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.

**Automatic Performance Measurement (Pilot)** ASCAR is designed to be a fully autonomous system, it has to have the ability to automatically execute performance benchmark and measurement, and compare measurement results. Because decisions about what traffic control policy to use would be based on performance measurement results, we need these results to be statistically valid. This is challenging because many statistical methods have rigid prerequisites that must be met. Also, because ASCAR needs to evaluate the performance of thousands of candidate parameters, the shorter each measurement step is the shorter the whole optimization process is.

The performance measurement component of ASCAR is designed to get statistical valid results as quickly as possible. We identified a collection of analytical methods that are necessary to generate results that meet the statistical requirements, and designed heuristic methods to automate and accelerate them. These methods include algorithms to
measure and correct the autocorrelation of samples, to use $t$-distribution to estimate the optimal test duration from existing samples, to detect the duration of the warm-up and cool-down phases, to detect shifts of mean in samples, and to use $t$-test to estimate the optimal test duration for comparing benchmark results. In addition to these heuristics, we also propose two new algorithms: a simple and easy-to-use linear performance model that makes allowance for warm-up and cool-down phases using only total work amount and workload duration, and a method to detect and measure the system’s performance bottleneck while keeping the overhead within an acceptable range.

This automatic measurement component is packaged as a standalone software framework called *Pilot* to make it easier for reuse in other projects. It can automate many performance evaluation tasks based on real-time analytical results, and can help people who have insufficient statistics knowledge to get good performance results. Pilot can also free experienced researchers from painstakingly executing benchmarks so they can get scientifically correct results using the shortest possible time. Pilot is a light-weight framework written in C++ with an interface for integrating into existing workload software.

### SHAred-nothing Rule Producer (SHARP)

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, and SHAred-nothing Rule Producer (SHARP) is designed to produce and optimize traffic rules for a specific workload running on a specific system autonomously. 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 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.

A prototype of ASCAR controller and SHARP has been developed on the Lustre parallel file system [52] and evaluated. The evaluation shows that rules generated by SHARP can effectively increase the throughput and lower the speed variance, at the same time (Table 3.4) for all the evaluated workloads. It increases the bandwidths of the BTIO Class B and C checkpoint workloads [76] 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 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.

**CAPES: Deep Reinforcement Learning-Based Parameter Tuning** While SHARP is effective at finding the optimal traffic control rules to improve peak time performance, it requires a relatively long training process, requires the workloads to be relatively stable, and needs retraining after the system or the workloads change. CAPES (Computer Automated Performance Enhancement System) is designed to attack these issues from a different angle. CAPES is based on Deep Reinforcement Learning and uses a neural network core. It does not require a special training process and can run continuously to handle changes in the system and workload. It is also designed to do parameter tuning in general in addition to storage traffic control.

Parameter tuning is about finding the optimal values of the system’s parameters in order to make the system behave in a certain way, such as running a certain workload more efficiently. Almost all complex systems provide a set of tunable parameters so that they can be customized to handle different workloads without changing the source code,
and a well tuned system can usually perform much better than using default parameters. Parameter tuning is considered a challenging job and is usually done by domain experts. It is challenging firstly because the consequences of adjusting a system’s parameter can be influenced by factors such as system hardware, operating system, and running workloads. The correlation between these variables makes predicting the effects of changing a certain parameter difficult at best, at worst humans may not have a clue what might happen if something were to change. This issue arises from the fact that human experts derive predictions from past optimization tasks, and these experiences are hard to quantify or automate. Second, the delay between an action and the resulting change in performance makes it even harder to correlate the relationship between system input and output (a credit assignment problem). Third, the available parameter space is huge, often including thousands of parameters. Humans can only evaluate a few commonly accepted parameter values and tend to use those same values across many parameters for simplicity, leaving a larger, more diverse, parameter space unexplored. Fourth, assigning humans to monitor dynamic workloads is simply too costly in practice.

In machine learning practice, we can approach this problem as a game where the goal is to find an appropriate setting that will render the system more efficient. By observing some indicators in the system, the player can maximize a cumulative performance reward such as energy usage, operations per second, or data transfer throughput. Recent advancements in machine learning showed that Deep Reinforcement Learning (DRL) can perform unsupervised learning well on diverse data featuring long delays between action and reward [49]. Many techniques are being used in tandem to make this possible, such as Deep Q-learning and experience replay [41].
A prototype of CAPES has been developed on Lustre and evaluated. The evaluation shows that it can increase the throughput of Lustre by up to 36% under heavy workloads. The training time was around 40 to 70 hours, which can be done during the system’s daily operation. This is a significant increase in efficiency, especially for large and expensive system installations.

In comparison to earlier automatic tuning systems, CAPES has several advantages:

- It requires no prior knowledge of the target system.
- It requires little change to the target system and little downtime for setting up.
- It can run continuously to handle dynamically changing workloads.
- It can dynamically chooses optimal values for parameters that used to be set statically.
- Training experiences from different sessions can be saved and later merged to train new models.
Chapter 2

Background

2.1 Congestion Control at Scale

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 [62]. 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 thesis, “congestion” includes severe contention in the network, server, and devices.
Figure 2.1: Comparison of a random write workload when running with and without ASCAR control. The benchmark that runs without ASCAR (a) shows high temporal and spatial throughput fluctuation, with an average throughput at 46 MB/s. When running with ASCAR (b), the average throughput is higher at 52 MB/s, and the lines are flatter, which means fairer allocation of bandwidth between clients and more consistent application performance.
**Distributed storage systems** Distributed high-performance storage systems, such as Lustre [52], GPFS [61], and Ceph [73], 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 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). NEXP (also called
NEXPTIME) is the set of decision problems that can be solved by a non-deterministic
turing machine using time $2^{O(n)}$. DEC-POMDP is NEXP-hard, which means that
it is at least as hard as any NEXP-complete problem. In DEC-POMDP, 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-conser-
ving. 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 2.1a, 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 [30] (or the current RTT to the lowest RTT ratio [25]), 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 communicate 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.

Fine tuning a computer system is considered to be a form of art that requires extensive knowledge and experience, thus this service is often limited to a large installations that can afford such a service. Often, for a service provider, there are simply not enough domain experts that understand all the quirks of a product. Most users have to fall back to following an inflexible, untailored performance tuning guide. Even for experts, a lengthy trial-and-error process is needed to obtain enough understanding about the customer’s workloads, and this process can last weeks to months.
Yet even with the best experts, it is almost impossible to achieve optimal performance. Using static values are not ideal for handling dynamic workloads. Depending on how the system is used, workloads can change over time or be cyclical, but they rarely stay stable. The best a human expert can do is to come up with several sets of optimal setting values for several typical workloads at a coarse grade, and the system will pick a set of values when a certain condition is detected. It is prohibitively expensive and time consuming to find optimal values for all possible workloads. In addition to this difficulty, it would be very hard to exhaustively test all possible combinations of parameter values even for a static workload, so the common practices is to only test a few known good sets of values. Furthermore, modern systems may behave differently under peak traffic. When systems are pushed to the limit, the efficiency of many components can drop rapidly. This phenomenon is usually called “congestion collapse”, and it is a common curse among network and storage researchers.

In all, it is clear that the existing, human driven, “tune-benchmark-tune cycle” does not fit the evolving nature and scalability of new technology. An automated hands free solution to performance tuning is ever more important in an era of widespread high performance computing.

Winstein and Balakrishnan described a method, Remy, for producing network QoS rules using simulation [75]. 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.)

2.2 Dynamic Parameter Tuning

In addition to the traffic control problem, the CAPES control system as described in Chapter 5 is a more general system that can be used to tune a wider range of parameters.

The Challenge of Parameter Tuning  Parameter tuning focuses on finding optimal parameter values that can make a target system or application perform better than its previous state. Better performance may be defined as lower energy consumption, higher throughput, or lower latency.

Automated parameter tuning faces several challenges in practice:
1. No precise model has been successfully constructed for modern computer systems because of their complexity.
2. Workloads are often dynamic and affect each other’s features.
3. The parameter space can be large, and sweeping through the entire space would be prohibitively slow.
4. It has to be responsive: when the workloads change, the system should also adjust quickly.
5. Some distributed systems can be large, such as a data center or an exascale supercomputer, so the tuning system has to be extremely scalable.
6. Tuning for multiple objectives should be possible, such as tuning for throughput and latency at the same time.

A simple approach to automatic parameter tuning is to construct a static tuner, which evaluates a few good sets of candidate parameter values using a user supplied workload. This is usually done after the computer system has been installed and before it is handed over to the customer. However, the effectiveness of such an offline system is highly limited, and it cannot handle any dynamically changing workloads.

A dynamic approach can continuously observes the system and make changes to the parameter values according to certain policies. Theoretically, most automatic dynamic parameter tuning can be constructed as a Decentralized Partially Observable Markov Decision Process (DEC-POMDP), which is NEXP-hard like the congestion control problem [6]. A perfect tuning system would need access to information of the entire history of observations, including the start times of all interaction requests, its finish times, and the times of all future requests; this is simply impossible to construct in practice. Therefore, all existing dynamic tuning approaches have adopted some sort of approximation in their policies.

Designing these policies is inherently difficult because there is no known method that can perfectly establish the cause and effect between an action and the subsequent changes in system status. First, the delay between applying a modification and its consequences makes it difficult to say whether a performance increase was due to a recent modification, or the effects of a modification done several minutes prior. Second, the delays could vary in length, and the length itself could be affected by the system, the workload, and many other factors. Third, any improvements measured could be the effects of several modifications, taken in a specific order, which could be difficult to trace back. This is commonly known as the “Credit Assignment Problem” [67].


**Hyperparameter**  In the machine learning community, the parameters of the machine learning algorithms are referred to as hyperparameters [8] to distinguish them from the parameters of the target system. Hyperparameters can greatly affect the efficiency of the machine learning algorithm and need to be chosen carefully. Common methods for hyperparameter optimization include Bayesian optimization, random search, and gradient-based optimization.

**Q-learning**  Reinforcement Learning (RL) [67] is a branch of machine learning; it is concerned with how an agent ought to take actions within an environment in order to maximize a certain reward. Reinforcement learning has seen successes in many areas, from robotics to automation. An environment can be modeled as a stochastic finite state machine with inputs (actions) and outputs (observation and reward). The interaction between the environment and the agent is usually constructed as a Markov Decision Process (MDP) because the outcomes are partly random and partly under the control of the agent.

In the context of CAPES, we treat the target system to tune as the environment. The tuning module is added to the environment as an agent, which observes the state of the target system. Actions are then calculated by the tuning module and issued to the target system to carry out. We use the output of an object function, whose input is the target system performance, as the reward. Using an objective function provides the flexibility to tune the system for one or more objectives. The agent’s goal is to find a policy to maximize the expected sum of all future rewards. Let $r_t$ be the reward we expect at time $t$, the expected sum of future rewards at time $t$ can be expressed as:

$$R_t = \sum_{i=t}^{n} r_i$$
Because of the stochastic nature of the process, the further into the future we predict, the less precise the prediction becomes. Thus, it is better to discount future rewards so they are less important than immediate rewards. Also the workload could change, making current modifications possibly ineffective. Let $\gamma$ be the discount rate, the new expected sum of rewards with discount is:

$$R_t = \sum_{i=t}^{n} \gamma^{i-t} r_i$$

Let $s_t$ be the system’s state and $a_t$ be an action at time $t$. Assuming we continue optimally from time $t$, we define an action-value function $Q(s, a)$ that represents the best expected sum of reward we can get. We define $\pi$ as the policy that dictates what action to choose in each future decision step. The purpose of Q-learning is then to find the policy $\pi$ (or trainable parameters of $\pi$) that can maximize $Q$:

$$Q(s_t, a_t) = \max_\pi R_{t+1}$$

If function $Q$ exists, the decision making step can be expressed as:

$$\pi(s) = \max_a Q(s, a)$$

Bellman proved that $Q$ can be solve iteratively:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

This is the Bellman equation [67] and is one solution to the credit assignment problem because iteratively solving this equation does not require the delay between an action and an reward to be known.
Solving the Q-function is the core task of Q-learning. Because the state space is usually prohibitively large, generalizing from known experience is important, and a nonlinear action-value function approximator is often used to express the Q-function, such as a neural network. Recent advances in deep neural networks (DNN) \cite{5,49} have shown that it can effectively learn concepts directly from raw sensory data. DNN employs several layers of neurons to build up progressively more abstract representations of the data. When DNN is being used to approximate the $Q$ function it is often called a $Q$-network, and its weights are often referred to as $\theta$.

However, reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator is being used. Many techniques have been developed to solve these challenges. Of these, experience replay is one of the most important methods.

**Experience Replay**  The transitions of states, actions, and rewards received from each step can be kept in a database so that we can replay these experiences later in different orders to break temporal correlation introduced by traditional training process that uses sequential system status. During a training step $i$, we take a random sample uniformly from the database and pack them into a minibatch ($D$). Thus, the goal of the training step is to adjust $\theta_i$ to reduce the loss $L_i$ (here we use average mean-square error) in the Bellman equation for all samples within the minibatch ($E_D$ means that the loss is calculated as a mean across all samples in minibatch $D$):

$$L_i(\theta_i) = E_D[(r_i + \gamma \max_{a'} Q(s', a'; \theta^-_i) - Q(s, a; \theta_i))^2]$$ (2.1)

This kind of training is generally referred to as experience replay \cite{41}. Because we are doing unsupervised learning, we use another $Q$-network of parameter $\theta^-_i$ to approximate the optimal target value, and this network is called the target $Q$-network.
We can use two methods to get the weights of the target network: getting the weights from a previous iterator or using a slowly updated $Q$-network (however, the update rate is limited to a small value).

Experience replay is important to preventing overfitting of a neural network and to provide generalization. Limiting the update rate of target $Q$-network has proven to increase the stability and efficiency of the training process [49].

The Exploration-Exploitation Trade-off  Because DNN-based reinforcement learning is often used in environments that have prohibitively large state spaces and when the models are too hard to construct, a Monte Carlo approximation is often used, where we only update the $Q$ function for states that the agent has actually visited in the environment, usually using a temporal difference method. This means that for unseen states the $Q$ function can only extrapolate from known experiences. Thus, it is important for the agent to “experience” as many states as possible during the training process (exploration). During the initial exploration process, we usually generate random actions to explore as many states as possible.

However, in a complex environment, performing totally random actions usually would not take the agent far. A standard practice is to initially use a mix of random and DNN-calculated actions, and gradually increase the chance of taking a calculated action further into the training process.
Chapter 3

SHARP: Rule-based Contention Control

The strength of rule-based contention control is responsiveness and scalability. Each client makes traffic control decisions independently and autonomously based on pre-defined rules. It is responsive to changes in I/O traffic because no communication is needed in the decision making process. This also helps it to scale to support a large number of clients thanks to the absence of communication burden.

3.1 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
Figure 3.1: A parallel storage system with ASCAR

support future large-scale storage systems that may have millions of clients. Figure 3.1 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.

3.1.1 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.

3.1.2 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 3.1, 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.
3.1.3 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 [75], 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 3.1. Each traffic controller keeps a record of these indicator variables and updates them when a reply is received.

Table 3.1: Congestion indicators used by ASCAR

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack_ewma</td>
<td>Exponentially weighted moving average of gaps between server replies.</td>
</tr>
<tr>
<td>send_ewma</td>
<td>Exponentially weighted moving average of gaps between the original sent times of the corresponding requests of the replies received by the client.</td>
</tr>
</tbody>
</table>
3.1.4 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 $C = \langle \text{ack}_{\text{ewma}}, \text{send}_{\text{ewma}}, \text{pt}_{\text{ratio}} \rangle$. All possible values of the congestion state variables form the congestion state space $K : \{\langle \text{ack}_{\text{ewma}}, \text{send}_{\text{ewma}}, \text{pt}_{\text{ratio}} \rangle : \text{ack}_{\text{ewma}}, \text{send}_{\text{ewma}} \in [0, \infty), \text{pt}_{\text{ratio}} \in [1, \infty)\}$. 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 $\tau$ 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: $S = \{C \to A : C \in K\}$. 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.
3.1.5 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 that uses the mean \((tp)\) and standard deviation \((stdev)\) of throughput can be defined as:

\[
\begin{align*}
\text{tp} & = \frac{\sum_{c=0}^{N} \sum_{t=0}^{T} \text{throughput}(c, t)}{N \times T} \\
stdev & = \text{var} (\text{throughput}(c, t))
\end{align*}
\]

\(N\) is the number of clients, \(T\) is the length of the benchmark in seconds, \(\text{throughput} (\text{client}, \text{time})\) is the measured throughput of \(\text{client}\) at \(\text{time}\). Both spatial and temporal throughput variance are reflected in \(stdev\).

There are several possible ways for combining the mean and standard deviation of the throughput as a single score. The most straightforward method is using \(tp\) as the score when \(stdev\) 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 \(k = \frac{stdev}{tp}\).

\[
\text{score}(tp, stdev) = \begin{cases} 
\text{tp} & \text{if } \frac{stdev}{tp} < k_0 \\
0 & \text{otherwise}
\end{cases}
\]

The scores of this method do not reflect the speed variance. To reflect the variance in the score, we can multiply \(stdev\) by a factor \(\alpha\) and add it to \(tp\):

\[
\text{score}(tp, stdev) = tp - \alpha \times stdev.
\] (3.1)

\(\alpha\) controls the weight of stableness versus efficiency.
We have also evaluated objective functions of other forms, such as

\[ U_\alpha(x) = \frac{x^{1-\alpha}}{1-x}, \]

\[ \text{score} = (1 - \alpha)tp + \frac{\alpha}{stdev}, \]

\[ \text{score} = a \times tp^\alpha - b \times stdev, \]

\[ \text{score} = a \times \log_\alpha(tp) - b \times stdev. \]

\( U_\alpha(x) \) is the alpha-fairness metric traditionally used to evaluate allocations of bandwidth on shared links [64]. They either failed to provide an effective balance between throughput and standard deviation, 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 as long as it needs to calculate a tight enough confidence interval that can be used to compare the effectiveness of rule sets. Since the number of candidate rule sets is quite large, we need to make the benchmark as short as possible for each benchmark session. The method we used is described in detail in Chapter 4.

### 3.1.6 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.
**Algorithm 1** The traffic rule set producing algorithm.

1: \[\triangleright \text{Producing traffic rule set for benchmark using objective function } \text{objFunc}\]

2: \textbf{procedure} \text{ProduceTrafficRuleSets}(\text{benchmark}, \text{objFunc})

3: \[\triangleright \text{Begin with one rule that maps the whole congestion state space to one initial action.}\]

4: \text{ruleset} ← \langle (0, \infty), [0, \infty), [1, \infty) \rangle \rightarrow \langle 1, 0, 20000 \rangle

5: \text{epoch} ← 0

6: \[\triangleright \text{Within each epoch, the number of rules in the ruleset remains unchanged. The } \text{optimization focuses on optimizing the hottest rule because it has the biggest impact on the overall performance.}\]

7: \textbf{while} True \textbf{do}

8: \text{epochBestScore} ← 0

9: \[\triangleright \text{Optimizing the hottest rule within ruleset by generating and evaluating candidate actions for it. The one that yields the highest score will be chosen.}\]

10: \text{round} ← 0

11: \textbf{repeat}

12: \[\triangleright \text{Construct candidate actions by increasing/decreasing each action variable geometrically from the current value and calculating Cartesian products, e.g., } \{m \pm g_{m}k_{m}(i = 0..s_{m})\} \times \{b \pm g_{b}k_{b}(i = 0..s_{b})\} \times \{\tau \pm g_{\tau}k_{\tau}(i = 0..s_{\tau})\}. \text{\textit{g} is step granularity, \textit{k} is step size, and \textit{s} is search steps for each variable. See Table 3.2 for details.}\]

13: \text{newRulesets} ← \text{GenerateCandidateRulesets(ruleset)}

14: \text{roundBestScore} ← 0

15: \textbf{for all } \text{r} \text{ in newRulesets} \textbf{do}

16: \text{result} ← \text{benchmark}(\text{r})

17: \text{score} ← \text{objFunc(result)}

18: \textbf{if} score \textbf{> roundBestScore} \textbf{then}

19: \text{roundBestScore} ← \text{score}

20: \text{ruleset} ← \text{r}

21: \textbf{end if}

22: \textbf{end for}

23: \textbf{if} \text{roundBestScore} \textbf{> epochBestScore} \textbf{then}

24: \text{epochBestScore} ← \text{roundBestScore}

25: \text{epochBestRuleset} ← \text{ruleset}

26: \textbf{end if}

27: \text{round} ← \text{round} + 1

28: \textbf{until} \text{roundBestScore} \textbf{< epochBestScore}

29: \[\triangleright \text{Split the hottest rule’s congestion state space at the observed mean value of each congestion indicator into } 2^{d} \text{ hypercubes, where } d \text{ is the number of congestion indicators.}\]

30: \text{ruleset} ← \text{SplitRule(epochBestRuleset)}

31: \text{epoch} ← \text{epoch} + 1

32: \textbf{end while}

33: \textbf{end procedure}
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: \textit{benchmark(ruleset)} and \textit{objFunc}. Function \textit{benchmark(ruleset)} deploys \textit{ruleset} to all clients, runs the benchmark, and returns the test result; \textit{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. 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 [59] is a viable option, but it can also be slow and complex. Remy [75] 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:

\[
\text{step\_size}(k) = \text{step} \cdot \sqrt{\frac{\text{var\_upper\_limit} - \text{var\_lower\_limit}}{\text{delta\_gran}}}.
\]
\textit{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 3.2, work best for ASCAR.

\begin{table}[h]
\centering
\caption{List of search settings for each action variable}
\begin{tabular}{lcccc}
\hline
Act var & Lower limit & Upper limit & Delta gran (g) & Search step (s) \\
\hline
\textit{m} & 0.3 & 2 & 0.05 & 4 \\
\textit{b} & \text{-2.3} & 2.3 & 0.3 & 4 \\
\textit{\tau} & 0 & 70,000 & 500 & 6 \\
\hline
\end{tabular}
\end{table}

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.

3.1.7 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 §3.2.6 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 [8] using the results of the experiments, which are shown in Figure 3.4 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: 0.1, 2, 4, 6, 8, and > 10;
• read: sequential, > 40 MB random read, 30 to 40 MB random, 20 to 30 MB random, 10 to 20 MB random, 5 to 10 MB random, 1 to 5 MB random.

3.2 Implementation and Evaluation

We implemented a prototype of ASCAR for Lustre 2.4 and 2.9 running on Linux and evaluated a wide range of workloads. Lustre 2.9 is the latest version as of March 2017.

3.2.1 Our ASCAR prototype

ASCAR’s traffic controller is implemented in Lustre’s client kernel modules. SHARP is implemented as a user-space program. Table 3.3 summarizes the changes of our prototype system against Lustre 2.4.0.

Table 3.3: List of ASCAR prototype components and lines of source code changed or added.

<table>
<thead>
<tr>
<th>File</th>
<th>LOC</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/ascar.h</td>
<td>179</td>
<td>Traffic controller</td>
</tr>
<tr>
<td>osc/osc_request.c</td>
<td>169</td>
<td>Traffic controller</td>
</tr>
<tr>
<td>osc/qos_rules.c</td>
<td>116</td>
<td>Traffic rule set parser</td>
</tr>
<tr>
<td>ascar_sharp.sh</td>
<td>374</td>
<td>SHARP main program</td>
</tr>
<tr>
<td>osc/lproc_osc.c</td>
<td>110</td>
<td>The procfs interface</td>
</tr>
<tr>
<td>gen_candidate_</td>
<td>166</td>
<td>Implementation of</td>
</tr>
<tr>
<td>rules.py</td>
<td></td>
<td>GenerateCandidateRulesets()</td>
</tr>
<tr>
<td>split_rule.py</td>
<td>145</td>
<td>Implementation of SplitRule()</td>
</tr>
<tr>
<td>ascar-tests/ (dir)</td>
<td>396</td>
<td>Test cases</td>
</tr>
</tbody>
</table>

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.

### 3.2.2 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 [76], 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 [9]. 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 [39] for generating those sequential workloads. The I/O size of IOR is set to 1 MB, which is common among HPC systems [37]. 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 [66]. 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.

3.2.3 Evaluation results

The performance improvements of ASCAR are listed in Table 3.4. For each workload, we run ASCAR using two different objective functions: one uses only $tp$ as the score, the other uses Equation 3.1 with $\alpha = 1$. 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 “$\alpha = 1$” 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 $\alpha = 1$ 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 2.1 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 the standard deviation of speed 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%.
Table 3.4: Summary of workload performance improvements by ASCAR

<table>
<thead>
<tr>
<th>Workload</th>
<th>w/o ASCAR (baseline)</th>
<th>ASCAR (max TP)</th>
<th>ASCAR (α = 1)</th>
<th>Run time (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>Stdev</td>
<td>Stdev/TP</td>
<td>Throughput</td>
</tr>
<tr>
<td>Sequential write</td>
<td>421</td>
<td>18</td>
<td>4%</td>
<td>428 (+2%)</td>
</tr>
<tr>
<td>Sequential read</td>
<td>445</td>
<td>16</td>
<td>4%</td>
<td>452 (+2%)</td>
</tr>
<tr>
<td>Random write</td>
<td>219</td>
<td>20</td>
<td>9%</td>
<td>273 (+25%)</td>
</tr>
<tr>
<td>Random read</td>
<td>238</td>
<td>9</td>
<td>4%</td>
<td>244 (+3%)</td>
</tr>
<tr>
<td>Random read/write (shared)</td>
<td>48</td>
<td>4</td>
<td>9%</td>
<td>50 (+5%)</td>
</tr>
<tr>
<td>FileBench fileserver</td>
<td>358</td>
<td>25</td>
<td>7%</td>
<td>414 (+16%)</td>
</tr>
<tr>
<td>BTIO (Class B)</td>
<td>116</td>
<td>7</td>
<td>6%</td>
<td>150 (+29%)</td>
</tr>
<tr>
<td>BTIO (Class C)</td>
<td>119</td>
<td>3</td>
<td>3%</td>
<td>160 (+35%)</td>
</tr>
</tbody>
</table>

Each result is the average of 10 test runs. “TP” is the throughput average (in MB/s) of all clients. “Stdev” is the standard deviation of throughput (in MB/s) over time and between clients. Those percentages in parenthesis show the increases of throughput and standard deviation 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 3.1. “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 Stdev. Since calculating temporal speed variance for a burst-I/O workload is meaningless, we cannot use Equation 3.1-based objective functions. Instead, the α = 1 columns of BTIO workloads show the results of the rule sets with the least speed variance.
3.2.4 How the rules affect the workload

![Graph showing client status over time with various metrics like Ack EWMA, Congestion window, Bandwidth, PT Ratio, and τ.]

**Figure 3.2:** One client’s ack/send EWMA, PT ratio, congestion window, and throughput when running the checkpoint 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 by the traffic rules. Figure 3.2 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 3.5. The first rule is triggered when the ack_ewma and send_ewma are low (the mean of ack ewma is 45) but PT ratio (the ratio of current Process Time to the best historical Process Time) is high (the mean of PT ratio is 3.2). This rule shrinks
Table 3.5: Excerpt of the best traffic rule set for the checkpoint workload

<table>
<thead>
<tr>
<th>ack_ewma</th>
<th>PT ratio</th>
<th>m</th>
<th>b</th>
<th>τ</th>
<th>Times</th>
<th>Avg. ack_ewma</th>
<th>Avg. PT ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>[41, 48)</td>
<td>[2.4, 4.5)</td>
<td>1</td>
<td>-1.7</td>
<td>33</td>
<td>3011</td>
<td>45</td>
<td>3.2</td>
</tr>
<tr>
<td>[48, ∞)</td>
<td>[0, 4.5)</td>
<td>1</td>
<td>0.9</td>
<td>40</td>
<td>7426</td>
<td>60</td>
<td>2.6</td>
</tr>
</tbody>
</table>

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.

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. A human expert usually...
needs to spend a long time doing benchmark to pick up the optimal values for the rate limits and congestion window change speeds, and SHARP is designed to 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.

### 3.2.5 Search efficiency

Unlike Remy [75], 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 performance in a shorter time. The results are shown in Figure 3.3. ASCAR generates better rule sets in a shorter time than Remy’s search method.

![Search method efficiency](image)

**Figure 3.3:** Comparison of the efficiency of different search methods (on the same workload as we used in § 3.2.6).

### 3.2.6 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 3.4. 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 § 3.1.7.
Figure 3.4: 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.
3.3 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 [44]. These methods require carefully planning from the beginning. ASCAR focuses on mitigation congestion during runtime without having to modify the design of an existing system.

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 [15, 48, 63], or for a specific kind of workloads, such as media recording and streaming [45, 78, 79] or virtual machines [20]. ASCAR is designed to be a distributed contention management system that can scale to support modern High-Performance Computers and data centers.

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 [22, 33, 36].
- Controllers work from the server side [20, 28, 47, 72, 80].
- Controller works from a proxy layer between the clients and servers [33, 43, 83].
- A mix of the above three methods [7, 85].

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 [22, 47, 72, 83, 85]. 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. ASCAR’s autonomous client controller does not require communication during congestion control and therefore has a better scalability.

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 [28, 72, 81], use a time-sharing scheduler [20, 22, 34, 47, 55, 63, 80, 83], or employ a simple control-theoretic heuristic: slow-start, fast fallback [68, 81, 85]. Some of them require that the storage system’s performance conforms to a mathematical model [36, 71]. 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. ASCAR’s SHARP is designed to solve the above two problems by automatically generating control rules and systematically exploring the parameter space.

Remy [75] 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 [3] 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. [62], or by re-balancing data placement when a hotspot is detected [43, 82]. Other related data placement optimization works include Pesto [21] and BASIL [19] 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. [2] 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 [86].
Chapter 4

Automatic Performance Benchmark

ASCAR needs to benchmark an application or workload for as long as it needs to calculate a tight enough confidence interval that can be used to compare the effectiveness of rule sets. Since the number of candidate rule sets is quiet large, we need to make the benchmark as short as possible for each benchmark session. All this have to be done automatically without human supervision. After a comprehensive study of existing statistical methods, we feel the need to design an automatic system for this purpose. This subsystem of ASCAR is useful for many other computer performance research and engineering tasks, and is described in detail in this chapter.

4.1 The Challenges of Performance Measurement

4.1.1 Performance measurement and benchmarking

*Performance measurement* is concerned with how to measure the performance of a running program or system, while *benchmarking* is more about issuing a certain workload on the system in order to measure the performance. High quality performance measurement and benchmarking are very important for almost everyone who uses an
electronic or computer system, from researchers to consumers. For instance, performance evaluation is critical for computer science researchers to understand the performance of newly designed algorithms. System designers run benchmarks and measure application performance before design decisions can be made. System administrators need performance data to find the most suitable products to procure, to detect hardware and software configuration problems, and to verify if the performance meets the requirements of applications or Service-Level Agreement. Yet not everyone has received rigorous training in statistics and computer performance evaluation. That is one of the reasons why we can find many incomplete or irreplicable benchmark results, even in peer-reviewed scientific publications. A widely reported study published in *Science* [53] found that 60% of the psychology experiments could not be replicated. Computer science cannot afford to be complacent. Hoefle and Belli analyzed 95 papers from HPC-related conferences and discovered that most papers are flawed in experimental design or analyses [27].

Performance measurement and benchmarking are similar but not identical. We usually can control how to run benchmarks, but measurement often needs to be done on applications or systems over which we have little control. Our following discussion applies to both benchmarking and measurement in general, and we use these two terms interchangeably unless otherwise stated.

Benchmarks must meet several requirements to be useful. Firstly, the measurement results must reflect the performance property one plans to measure (accuracy). Secondly, the measurement must be a good approximation of the real value with quantified error (precision). Thirdly, multiple runs of the same benchmark under the same condition should generate reasonably similar results (repeatability). And finally, if the results are meant for publication, they also must include enough hardware and software information so that other people can compare the results from a similar environment (comparability) or replicate the result (replicability).
People usually use these terms, especially accuracy and precision, to mean many different things. Here we give our definition of these terms. It is assumed that the readers have basic statistics knowledge and understand basic concepts such as mean, variance, and sample size. Boudec wrote a good reference book for readers who want to know more about these statistical concepts [10].

**Accuracy** reflects whether the results actually measure what the user wants to measure. A benchmark usually involves many components of the system. When we need to measure a certain property of the system, such as I/O bandwidth, the benchmark needs to be designed in such a way that no other components, such as CPU or RAM, are limiting the measured performance. This requires carefully designing the experiment [23], measuring the usage of related components while the benchmark is running, and checking which component is limiting the overall performance.

**Precision** is related to accuracy but is a different concept. Precision is the difference between the measured value and the real value the user needs to measure. In statistical terms, precision is the difference between a sample parameter and its corresponding population parameter. Precision can be described by confidence interval (CI). The CI of a sample parameter describes the range of possible population parameter at certain likelihood. For instance, if the CI of a throughput mean ($\mu$) is $C$ at the 95% confidence level, we know that there is a 95% chance that the real system’s throughput is within interval $[\mu - \frac{C}{2}, \mu + \frac{C}{2}]$. In practice, CIs are typically stated at the 90% or 95% confidence level. We can see that the common practice of presenting a certain performance parameter using only one number, such as saying the write performance of a disk drive is 100 MB/s, is misleading.
Repeatability is critical to a valid performance measurement because the goal of most performance benchmark is to predict the performance of future workloads, which exactly means that we want the measurement results to be repeatable. In addition to the accuracy and precision, errors in the measurement can have a negative impact on repeatability. There are two kinds of errors: the systematic error and random error. Systematic error means that the user is not measuring what he or she plans to measure. For instance, the method of benchmarking is wrong or the system has some hidden bottleneck that prevents the property to be measured from reaching its maximum value. In these cases, even though the results may look correct, it may well be not repeatable in another environment. Random errors are affected by “noise” outside our control, and can result in non-repeatable measurements if the sample size is not large enough or samples are not independent and identically distributed (i.i.d.).

Basically, if any published number is derived from fewer than 20 samples, or is presented as a mean without variance or confidence interval (CI), the authors are likely doing it wrong. The following problems can often be found in published results:

- **Imprecise**: the result may not be a good approximation of the “real” value; often caused by failing to consider the width of CI and not collecting enough samples,
- **Inaccurate**: the result may not reflect what you need to measure; often caused by hidden bottleneck in the system,
- **Ignoring the overhead**: not measuring or documenting the measurement and instrumentation overhead,
- **Presenting improvements or comparisons without providing the p-value**, making it impossible to know how reliable the improvements are.
We will discuss these properties in detail, and one can see that scientifically performing a benchmark demands significant knowledge of the computer system and statistics. We cannot expect all administrators, engineers, and consumers to have received rigorous training in both computer science and statistics. It is not news that many vendors publish misleading, if not utterly wrong, benchmark results to promote their products. Many peer-reviewed research publications also suffer from poor understanding or execution of performance measurement [27].

4.1.2 Getting results fast

The old wisdom for running benchmark is to run it for as long as one can tolerate and hope the law of large numbers can win over all errors. Yet time is a scarce resource. People usually do not allocate enough time for performance evaluation or tuning, yet few newly designed or deployed systems can meet the expected performance without a length tuning process. A large part of the tuning process is spent on running customer’s benchmark workloads for model construction or testing candidate parameters. Designers are usually given only a few days for these tasks and have to rush the results by cutting corners, which often leads to unreliable benchmark results and sub-optimal system tuning decisions.

The old wisdom is no longer suitable for today’s fast changing world, where we simply no longer have a lot of time to run benchmark. We have heard field support engineers complaining[^1] that they are usually only given one or two days after the installation of a new computer cluster or distributed storage system to prove that the system can deliver whatever performance promised by the salesperson, very often using the customer’s own benchmark programs. Modern distributed systems can have hundreds, if not thousands, of parameters to tune, and the performance engineer needs to run an

[^1]: Private conversation with an engineer from a major HPC provider.
unfamiliar benchmark repeatedly and try different parameters. Apparently the shorter the benchmark is the more parameters can be tested, thus resulting in better system tuning results.

Existing analytical software packages, such as R [57], are either too big or slow for run-time analysis, are hard to integrate with applications (for instance, R is a large GPL-licensed software package), or require the user to write complex scripts to use all its functions.

In all, we realize that we need an easy-to-use software tool that can guide the user and help to automate most of the analytical tasks of computer performance evaluation. We are not introducing new statistical method in this thesis. Instead we focus on two tasks: finding the most suitable and practical methods for computer performance evaluation, and design heuristics methods to automate and accelerate them.

4.2 Pilot Algorithms

Terms in statistics can be confusing, or worse, have subtle differences when being used by different people or publication. Many statistical methods can also have subtle differences when being introduced by different text books for different purposes. We first define the terms we will use, then introduce the statistical methods that we will be automating in Pilot, grouped by the purpose we want to achieve. Our focus is not the actual detail of each statistical method, which can be found in text books or the publication we cite. Instead, we focus on the requirement, cautions, and checks that must be done for these methods to be valid, and how they can be automated. These “small” details are often being ignored in practice.

We define the following terms. Also see a sample in Fig. 4.1.
**Performance index (PI)** is one property we want to measure. For instance, the throughput of a storage device is a PI, and its latency is another PI.

**Session** is the context for doing one measurement. We can measure multiple PIs in one session. One session can include multiple **rounds** of benchmarks, and each round can have a different length.

**Work amount** is the amount of work involved in one round of benchmark. The work amount is related to the length of the workload. For instance, in a sequential write I/O workload round, we write 500 MB data using 1 MB writes, the work amount of this round is 500.

**Work unit** is a smallest unit of work amount that we can get a measurement from. Using the above sample, if the I/O size is 1 MB, we can measure the time of each I/O syscall and calculate the throughput of each I/O operation. Here the work unit is 1 MB, and we have 500 work units in that workload round. Not all workloads should be divided into units. Pilot expects the work unit to be reasonably homogeneous. So, for instance, reading one 1 MB from different locations of a device can be thought as homogeneous because the difference in performance is small and mostly normally distributed. But shifting from sequential I/O to random is not homogeneous because that would result in huge difference in I/O performance. In general, the user should only divide the workload into units when

![Performance Index (PI): write throughput](image)

**Figure 4.1:** A sample write workload to illustrate the terms used in Pilot. This workload consists of two rounds. Each round has a work amount of 500 MB.
one expects them to have similar performance. If not the user should not use the work unit-related analytical methods of Pilot and should stick with readings (see next term) only. We leave heterogeneous work units as a future work.

**Reading** is a measurement of a PI of a round. Each benchmark round generates one reading for each PI at the end of the round. In the sample above, when PI is the throughput of the device, we can get one throughput reading for each round.

**Unit reading (UR)** is a measurement of a PI of a work unit. In the sample above, we would have 500 throughput Unit readings for Round 1 because it contains 500 work units, and these 500 unit readings would be the throughput of each 1 MB I/O operation.

**Work-per-second (WPS)** is the calculated speed at which the workload consumes work. This is usually the desired PI for many simple workloads.

Some workload cannot be meaningfully separated into homogeneous work units, such as booting up a system and randomly reading files of different sizes. We get only readings from this kind of workloads.

### 4.2.1 Warm-up and Cool-down Phase Detection

Performance results are often used to predict the run time of future workloads, and it is a common practice to use one number to express the performance of a PI. For example, people usually say “the write throughput of this device is $X$”. Using only one number assumes that the device’s performance follows a linear model. Linear models ($work\ amount = duration \times speed$) are simple, but using only one number can only state the device’s stable performance and is not adequate when the performance of the PI can be significantly affected by a long warm-up or cool-down phase.
Most computer devices require a setup or warm-up phase before its performance can reach a stable level, like shown in Fig. 4.2. If not properly accounted for, these warm-up phases can have a negative impact on the precision of the measurement. A common practice is to run the workload for a long time and hope the effect of the warm-up phase can be amortized. However, when the duration of the warm-up phase is not known, there is no way to know the actual impact on the precision (see the samples in §4.4). We describe two methods to address different kinds of workloads.

![Throughput of a multi-node random read write workload showing the setup phase, the warm-up phase caused by caching effect, and the cool-down phase caused by shutting down of I/O threads.](image)

**Figure 4.2:** Throughput of a multi-node random read write workload showing the setup phase, the warm-up phase caused by caching effect, and the cool-down phase caused by shutting down of I/O threads.

We consider the following phases of a workload:

1. The setup phase, including the steps that do not consume work amount, such as allocating memory, initializing variables, and opening files, etc.
2. The warm-up phase when the system starts to perform work but has not yet reached stable performance;
3. The stable phase where the work amount is being consumed at a stable rate;
4. The cool-down phase when the system’s performance starts to drop before finishing all the work (this is usually observed in multi-threaded workloads when some but not all threads finish the allocated work and the number of active threads starts to drop at the end of the workload).

We call them collectively the non-stable phases. When the workload has multiple rounds, each round may or may not have its own non-stable phases, and when they have, the duration can be different. We consider two cases, the first is when the benchmark can provide unit readings, the second is for workloads that cannot provide unit readings.

**Workloads that can provide unit reading** If the benchmark workload can provide unit readings, which is the measurement of each work unit, we can calculate the shift in UR mean and use these change-points to separate the URs into phases. Multiple change-point detection is a challenging research question, especially when we cannot make any assumption about the distribution of the error or the underlying process. The method we use also has to be fast to calculate and should support online update.

After evaluating many change-point detection methods, we found that the E-Divisive with Medians (EDM) [46], which is a new method published by Matteson and James in 2014, best fits our requirements. EDM is non-parametric (works on mean and variance) and robust (performs well for data drawn from a wide range of distributions, especially non-normal distributions). EDM’s initial calculation is $O(n \log n)$ and can do update in $O(\log n)$ time.

EDM outputs a list of all the change-points in the time series. It is common to see many change-points at the start and end of the workload. These change-points divide the test data into multiple segments. Pilot uses a heuristic method to determine which
segment is the stable segment: it has to be the longest segment and also dominate the test data (containing more than 50% of the samples). This method can effectively remove any number of non-stable phases at the beginning and the end.

**Workloads that cannot provide unit reading** Some workload cannot be meaningfully separated into units. In these cases, we designed the following Work-per-second (WPS) Linear Regression Method to detect and remove the non-stable phases from the results of these workloads. A linear regression model works best when:

1. The work amount of the workload is adjustable,
2. There is a linear relationship between the work amount and the duration of the workload,
3. The duration of the setup, warm-up, and cool-down phases are relatively stable across rounds.

It is not necessary to check these conditions beforehand. We will know that one or more of them are false if the result of the WPS method has a very wide CI or a high prediction error. The WPS method also applies autocorrelation detection and subsession analysis, which make it more tolerant of the inconsistency in measurements.

Let $w$ be the work amount, $t$ be the total duration of the workload, $t_{su}$ be the duration of the setup phase, $t_{wu}$ be the duration of the warm-up phase, $t_s$ be the duration of the stable phase, $t_{cd}$ be the duration of the cool-down phase, $w_{wu}$ be the work amount consumed by the warm-up phase, $w_s$ be the work amount consumed by the stable phase, and $w_{cd}$ be the work amount consumed by the cool-down phase. We have (note that the
setup phase of a workload does not consume work amount):

\[ t = t_{su} + t_{wu} + t_s + t_{cd} \] (4.1)

\[ w = w_{wu} + w_s + w_{cd} \] (4.2)

\( v_s \) is the stable system performance we need to measure. By definition, it can be calculated from the work amount of the stable phase divided by the duration of the stable phase:

\[ v_s = \frac{w_s}{t_s} \] (4.3)

Combining equation (4.1), (4.2), and (4.3), we can have

\[
\begin{align*}
\frac{\alpha}{v_s} + \frac{1}{v_s} w &= \left( t_{su} + t_{wu} + \frac{w - w_{wu} - w_{cd}}{v_s} + t_{cd} \right) \\
&= \left( t_{su} + t_{wu} + t_{cd} - \frac{w_{wu} + w_{cd}}{v_s} \right) + \frac{1}{v_s} w \\
&= \alpha + \frac{1}{v_s} w.
\end{align*}
\] (4.4)

Equation (4.4) is the model we use in Pilot. Given enough number of \((w, t)\) pairs, we can use regression to estimate the value of \(\alpha\) and \(v_s\). The current implementation of Pilot uses the Ordinary Least Square estimator [24] for its simplicity, and other estimators can be added when necessary. We need the samples to be i.i.d. in order to calculate the CI of \(v_s\) using the t-distribution. We use subsession analysis, which calculates the autocorrelation coefficient of input samples and merges adjacent correlated samples to create fewer but less correlated samples, before running the regression estimator (see § 4.2.2).

In addition to the requirements we talked about earlier, linear regression requires that the following conditions be met:

1. The differences between the work amounts of rounds are sufficiently large,
2. The sample size is sufficiently large.

We designed Pilot to keep running the workload at various length and for many rounds until the desired width of the CI is reached. Because we cannot know the total number of rounds that are needed at the beginning, we designed the following algorithm to generate different work amount for each round: let \((a, b)\) be the valid range for the work amount, we pick the midpoint of the interval as the work amount for the first round \((a + \frac{b-a}{2})\). This midpoint divides the interval into two smaller intervals of equal length. We then use the midpoints of these intervals for future rounds. Repeating this process can give us a sequence of unequal numbers that can be used as the work amounts. Fig. 4.3 gives a the first few numbers in this sequence as a sample.

![Figure 4.3: Sample sequence of work amounts for the first seven rounds automatically calculated by Pilot. Round 1 (Rd.1) is the midpoint of \(a\) and \(b\); Round 2 is the midpoint of \(a\) and Round 1; Round 3 is the midpoint of Round 1 and \(b\); Round 8 would be at the midpoint of \(a\) and Round 4.](image)

Pilot takes \(a\) and \(b\) from user input. In practice the user usually will likely set \(a\) to 0. This could cause the problem that some rounds are too short. Very short rounds are usually meaningless because they could be dominated by the non-stable phases. Pilot checks the duration after running each round, and if it finds that the previous round is shorter than a preset lower bound, the result will be stored but not used in analysis. Pilot doubles the work amount of the previous round until the round duration is longer than the lower bound, and will update \(a\) to that work amount.

In practice, the algorithm as described above has another drawback that the work amount of the first few rounds may be very large if \(b\) is a large number. For instance, if the user wants to understand the throughput of a device and uses \((0, \text{device size})\) for the
valid parameter range, the first few rounds can be very long, and it would take a long
time before the user can see the benchmark result. It is important for Pilot to give the
user a quick (albeit rough) estimation of the result before spending a long time refining it.
We use the following heuristic method in Pilot to solve this problem. Say that we know
in round 1 that the time needed for finishing work amount \( a \) is \( t_1 = s \) seconds, and for
each new round we want it to be \( k \) seconds longer than the previous round. This means
that the nth round would be \( t_n = s + (n - 1)k \) seconds long. Therefore, the total duration
(\( t \)) of the \( n \) rounds would be:

\[
t = \sum_{i=1}^{n} t_n = \frac{1}{2}k(n - 1)n + ns.
\]

Now if we want to get the initial result in \( t \) seconds, we can calculate \( k \):

\[
k = \frac{2t - 2sn}{n^2 - n}\tag{4.5}
\]

Pilot uses equation (4.5) to calculate the initial slice size where \( t \) is a tunable parameter
with a preset value 60 seconds. The number of rounds, \( n \), should be greater than 50 in
most cases [12] for the central limit theorem to take effect.

Another problem is that the work amount derived from this algorithm may be shorter
than \( \alpha \) (sum of the work amount of all non-stable phases). The method we use in Pilot to
handle this issue is that we calculate the value of \( \alpha \) after each round, and use the new
value of \( \alpha \) to update \( a \). We also remove all results from previous rounds whose work
amount is smaller than the newly calculated \( \alpha \).
4.2.2 Auto-correlation Detection and Mitigation

A benchmark session needs to be long enough so that we can collect enough samples to calculate the CI at the desired confidence level. The more samples we have, the narrower the CI can be made. However, a crucial issue that is often overlooked in many published benchmark results is the autocorrelation among samples. Autocorrelation is the cross-correlation of a sequence of measurements with itself at different points in time. Conceptually, a high autocorrelation means that previous data points can be used to predict future data points, and that would invalidate the calculation of CI no matter how large the sample size is. Most measurements in computer systems are autocorrelated because of the stateful nature of computer systems. For instance, most computer systems have one or more schedulers, which allocate time slice to jobs. The measured performance of such jobs would be highly correlated when they are taken within a single time slice, and would change significantly between time slices if the duration of a measurement unit is not significantly longer than the size of a time slice. The autocorrelation in the samples must be properly handled before we can go on to the next step to calculate the sample’s CI.

Autocorrelation is measured by the autocorrelation coefficient of a sequence, which is calculated as the covariance between measurements from the same sequence as

\[ R(\tau) = \frac{\text{E}[(X_t - \mu)(X_{t+\tau} - \mu)]}{\sigma^2}, \]

where \( \tau \) is the time-lag. The autocorrelation coefficient is a number in range \([-1, 1]\). where \(-1\) means the sample data are reversely correlated and \(1\) means the data is autocorrelated. In statistics, \([-0.1, 0.1]\) is deemed to be a valid range for declaring the sample data has negligible autocorrelation \([16]\).
Subsession analysis \[16\] is a statistical method for handling autocorrelation in sample data. \(n\)-subsession analysis models the test data and combines every \(n\) samples into a new sample. Pilot calculates the autocorrelation coefficient of measurement data after performing data sanitizing, such as non-stable phases removal, and gradually increases \(n\) until the autocorrelation coefficient is reduced to within the desired range.

4.2.3 Deciding Optimal Session Length

On a high level, a benchmark session comprises many rounds. We calculate the CI after collecting new data from each round. The session ends when the CIs of all PIs reach the target. Because each round can have non-stable phases that are not contributing samples, we should maximize the length of each round and minimize the number of rounds. This also has the extra benefit of including those work units that are far from the beginning of the initial work amount.

But we cannot begin the first round using the maximum work amount, because in many cases the maximum work amount can be very large. This is typical in storage benchmarks where the limit can be the total size of a storage device, and it would take several dozens of hours to finish one round that fully writes a device. In network benchmarks, we can even set the maximum work amount to unlimited because these workloads can keep running forever. If we start the first round of workload with the full work amount, we risk letting the user wait too long a time before showing the first result. Therefore, we begin the benchmark session with a few short trial rounds to learn the duration to work unit ratio.

Workloads that provide unit readings  
We treat each unit reading as one measurement. One workload round can usually provide hundreds or thousands of measurements, making it faster to reach the required sample size. For instance, if a sequential write workload
writes 500 MB data using 1 MB I/O, we can get 500 throughput measurements if the workload saves the duration of each write. Pilot sends the unit reading results through the non-stable phases removal, performs auto-correlation reduction, and uses the rest of the unit readings to calculate the CI. Pilot keeps running new rounds of the workload until the desired width of the CI is reached.

**Workloads that cannot provide unit reading** These workloads are handled using the WPS method (§4.2.1), which also performs non-stable phases detection and removal, subsession analysis, and decides the optimal number of rounds to achieve the desired width of CI.

4.2.4 Comparing Results

The need to compare benchmark results using the shortest possible time is our very first motivation for designing Pilot. Not only useful for system design and tuning, a handy tool for comparing benchmark results can help many tasks in systems research, such as choosing the fastest data structure for storing certain data, finding performance regressions in software development, and deciding the best parameters for storage or network communication. Without using the correct statistics method at runtime, most of these decisions were done in an ad hoc manner, either prematurely with too little data, or blindly wasting time to gather more than necessary data.

Suppose we have $n$ workloads and the results of them are comparable, which means that they are using the same unit of measurement and are of the same scale. Now we need to rank (order) these workloads according to their results. In some cases we need to run these workloads to get new results, in other cases the comparison involves old benchmark results. For old results we need three values: the mean, subsession sample size, and subsession variance (we need to use the subsession analysis, as shown in §4.2.2).
Sequential execution of workloads:

<table>
<thead>
<tr>
<th>App 1</th>
<th></th>
<th></th>
<th>App 2</th>
</tr>
</thead>
</table>

Interleaved execution of workloads for lowering temporal correlation:

<table>
<thead>
<tr>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
</tr>
</thead>
</table>

**Figure 4.4:** Interleaved execution of benchmarks can help to lower temporal correlation between rounds.

to reduce autocorrelation between samples because i.i.d. is a hard requirement for all the analyses we use in this section). For the rest of this section all samples are subsession samples that have low autocorrelation.

Unlike the algorithm in §4.2.3, which executes multiple rounds of a workload guided by heuristics until a desired width of CI is reached, the algorithm for comparing results interleaves the execution of workload rounds to reduce temporal correlation between workload rounds, as shown in Fig. 4.4. For instance, Pilot executes the workloads in the following order: Workload 1 Round 1, Workload 2 Round 1, Workload 3 Round 1, . . . , Workload n Round 1; then go back to run Workload 1 Round 2, Workload 2 Round 2, etc.

There are two cases when we consider comparing two benchmark results. The first case is when their CIs are not overlapped. In this case we can be sure that one is greater than the other at the confidence level used to calculate the CIs. The second is when the CIs overlap. In this case we use the Welch’s unequal variance \( t \)-test [74] (an adaptation of Student’s \( t \)-test [65]) to compare the benchmark results, A and B. Welch’s \( t \)-test is more reliable when the two samples have unequal variance and unequal sample sizes, which are true for most system benchmarks. This test can effectively tell us the probability of rejecting a hypothesis and the required sample size. Here the null hypothesis (the hypothesis we want to reject) is that there is no statistical significant difference between result A and B (\( A = B \)). We compute the probability (\( p \)-value) of getting results A and B if the null hypothesis is true. Let \( \bar{x} \) be the mean of the result, \( \sigma^2 \) be the variance, and \( n \)
be the sample size. We can calculate the $p$-value:

$$t = \frac{\bar{x}_A - \bar{x}_B}{\sqrt{\frac{\sigma^2_A}{n_A} + \frac{\sigma^2_B}{n_B}}}$$  \hspace{1cm} (4.6)$$

$$\nu = \left| \frac{\left(\frac{\sigma^2_A}{n_A} + \frac{\sigma^2_B}{n_B}\right)^2}{\frac{\sigma^4_A}{n_A^2(n_A-1)} + \frac{\sigma^4_B}{n_B^2(n_B-1)}} \right|$$  \hspace{1cm} (4.7)$$

$$p = 2 \, \text{cdf}(t, \nu)$$  \hspace{1cm} (4.8)$$

Equation (4.7) is the Welch-Satterthwaite equation for calculating the degree of freedom ($\nu$), and $\text{cdf}(t, n)$ is the Student’s $t$-distribution with $\nu$ degrees of freedom. We multiply it by 2 to calculate the two-tailed distribution.

The comparing result algorithm runs until:

1. There are enough data for calculating the CIs,
2. Each adjacent CI pair is either non-overlap or their $p$-value of the null hypothesis ($A = B$) is less than the predefined threshold (usually 0.01),
3. Every CI is tighter than the required width (this step is optional but recommended because a narrower CI makes it easier to compare with new results in the future).

Pilot needs to decide the work amount for running each round of the workload. The value has to be chosen in such a way that we can minimize the number of rounds needed to reduce the impact of the overhead of starting a round. Mathematically the optimal subsession sample size can be calculated using a variation of equation (4.6).
4.3 The Pilot Framework

Our analysis of recent publications in the systems research field shows three common types of benchmarking. The first is to evaluate the performance of a piece of source code, which is usually relatively short, requires little to no setup, and does not have a dedicated supporting benchmark framework involved. Samples of such cases are comparing the performance of two hash algorithms and comparing the performance of two ways of iterating over a large matrix. The second kind of benchmarks includes most workloads that are specifically designed for performance measurement. They are usually complex, requires a relatively long setup process, needs a pre-configured benchmark framework, and can have a long duration (hours to days). The third kind of benchmark includes those that are done quickly and on spot, often when the user needs to have a quick estimate of the performance of a certain piece of hardware or software, such as doing a short benchmark to decide which external hard drive is faster or downloading a file from the Internet to test the speed of the Wi-Fi. In these cases, the precision requirement of the measurement is not paramount, but they have to be done quickly and usually involve using a command line program (like dd [18] or cURL [56]).

Pilot is designed to work with all three kinds of benchmarks. Pilot takes the requirements of the benchmark as the input, applies the algorithms as described in the previous section to execute the benchmark, and generates a detailed report of the benchmark results. The default workload benchmark declaration includes the following tasks:

1. The number of PIs, and for each PI:
   (a) name and unit,
   (b) desired width of CI (default to 10% of mean),
   (c) confidence level (default to 95%),
   (d) desired autocorrelation coefficient (default to $[-0.1, 0.1]$).
2. The valid range of the work amount,
3. Non-stable phases removal (default: EDM and WPS),
4. Bottleneck detection (default: enabled),
5. Overhead detection (default: enabled),

For the first kind of benchmark, Pilot can be easily linked into an existing code base by adding a few lines of code, very much like the way a unit test framework is used. We support C/C++ first because they are the language in which most performance critical code is written. For the second kind of benchmark, Pilot provides an extensive list of library functions for integrating with the existing benchmark framework. The developer of the benchmark can choose to either let Pilot decide the number of rounds and the work amount for each round (the simplest way) or to manually control the executing of the workload and use Pilot’s analytical functions as a guide to the execution (more flexible). For the third kind of benchmark, Pilot can run quick and short benchmark jobs by controlling a workload program through a command-line interface.

4.4 Evaluation

Pilot is designed to make running benchmarks easier. That cannot be measured unless we do a user study. Unfortunately, we do not have the resource for that, nor could we measure how much time can be saved by using Pilot, because that depends on how people run benchmarks before using Pilot. Instead, we evaluate the characteristics of the algorithms in order to understand them better. We present two evaluations: time to reach the desired width of CI and prediction error of the results.
4.4.1 Time to reach desired CI

Reaching the desired width of CI means that the precision requirement is achieved and is often the major goal of a measurement. Understanding the time to reach (TTR) desired CI can help us plan and design benchmark tasks, and identify problems in current benchmarking practices. We compare the TTR of three methods: *UR data without non-stable phase removal*, *UR with EDM non-stable phase removal*, and *WPS method*. The desired width of CI is set to 10% of mean. All methods include data sanitization, such as short-round detection and using subsession analysis to mitigate autocorrelation.

The result is shown in Table 4.1. We can see that the TTR varies greatly in different workloads. TTR is not only affected by the required sample size but also the autocorrelation in the samples (high autocorrelation requires merging more samples), which in turn can be affected by different non-stable phases removal methods. TTR varies greatly even when using the same method. This highlights the importance of using runtime analysis to decide the optimal sample size on-the-fly and the inadequacy of using a fixed benchmark duration. Using a fixed duration of benchmark leads to imprecise results for workloads that require a long duration and a waste of time for workloads that only require a short duration. Since ASCAR is designed to be an automatic system for finding the optimal traffic control rules, it is critic for us to reach the designed confidence interval using as short time as possible.
<table>
<thead>
<tr>
<th>Workload</th>
<th>UR w/o non-stable removal</th>
<th>UR with EDM non-stable removal</th>
<th>WPS method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Result CI, $\sigma^a$ TTR CI$^b$, $\sigma$ SS CI$^c$, $\sigma$</td>
<td>Result CI, $\sigma$ TTR CI, $\sigma$ SS CI, $\sigma$</td>
<td>Result CI, $\sigma$ TTR CI, $\sigma$</td>
</tr>
<tr>
<td>Seq. write (flash)</td>
<td>707–741, 6 12–15, 10 6–9, 14</td>
<td>719–719, 6 13–15, 5 12–16, 14</td>
<td>664–681, – 267–389, 58</td>
</tr>
<tr>
<td>Send data over Wi–Fi</td>
<td>26–27, 0.3 33.79–45.47, 20 1–2, 1</td>
<td>20–22, 0.4 50–106, 289 1–1, 1</td>
<td>21–26, – 8120–14345, 3012</td>
</tr>
</tbody>
</table>

$^a$ Result CI is the average measured CI of the results from respective method. $\sigma$ is the average measured standard deviation. The unit for Result CI and $\sigma$ is MB/s. All CIs are calculated at 95% confidence level.

$^b$ TTR CI shows the CI of time-to-reach. The unit for TTR is a second.

$^c$ SS is the subsession size (how many adjacent samples are merged to reduce autocorrelation). All methods include autocorrelation removal.
We can also see that the EDM method converges to the desired CI using the shortest time. Without non-stable phases removal, the URs from the hard disk have very high autocorrelation, and Pilot needs to merge almost a thousand adjacent samples to reduce the autocorrelation to below 0.1. If we remove the non-stable phases first as in the EDM method, the autocorrelation becomes much lower and also the time needed to reach the desired sample size is shorter; EDM can be seen as a way to sanitize the samples. The WPS method does not require UR data and can only get one sample per round, thus it requires a longer time to reach the required sample size. We can see that sometimes the WPS method needs more than 8000 seconds. Therefore, it should only be used when UR data is not available. The results of all three methods also demonstrate the importance of detecting and handling autocorrelation. The high numbers of SS mean that many workloads have inherently very high autocorrelation between samples and it is wrong to apply almost any statistical method if the results are not i.i.d.

The result matches our expectation. The UR with EDM method should be the first choice for workloads that can provide UR data, and the WPS method can be used for other workloads. Whenever possible, benchmark workloads should be divide into measurable small units to increase the sample size and reduce the TTR.

4.4.2 Predicting future workloads

One major purpose for measuring performance is to use it to predict the time needed for running future workloads. It can be seen from Table 4.1 that different methods generate slightly different results. This evaluation is designed to see which results can best predict future durations. We measure the actual durations of sequentially writing different amounts of data and compare them with the predicted durations that are calculated using the values from Table 4.1. We use the center of the CIs in the calculation. The result is shown in Fig. 4.5.
Figure 4.5: The actual and Pilot predicted time for writing certain amount of data sequentially. Three methods are used to predict the run time: UR without non-stable phases removal, UR with EDM non-stable phase removal, and the WPS method. Closer to the actual duration is better. The slope of the actual duration line changes at around 2000 MB, after which the effect of write cache starts to become less significant. The CI for each data point is very tight so we omit them in the figure for clarity.

The slope of the actual duration line becomes steeper after around 2000 MB. Our test machine’s Linux OS caches most of the writes before 2000 MB in memory so the time needed is short. After that the writes become slower because the system has to actually write the data to the disk. Therefore, the write performance after the first 2000 MB better reflects the real disk’s write performance. It can be seen that the EDM line’s slope best matches the slope of the post-2000 MB segment of the actual duration line. This means that UR with EDM non-stable phases removal is best for calculating the stable performance. Without non-stable phases removal, the slope of the green line is negatively affected by the huge write cache of the system. The WPS method also gets the correct slope before and after 2000 MB so it can also find the correct stable performance when UR is not available.
It should be noted that due to limited space we are only presenting the results of one HDD, and the prediction errors also depend on how the workload works. Our current conclusion is that the EDM method is best for getting the stable performance and also has the shortest TTR. The WPS method can effectively detect the non-stable phases even when only readings are available.

4.5 Related Work

Boudec’s book [10] extensively covers the statistics knowledge that is needed for computer performance evaluation. Hoefle and Belli analyzed 95 papers from HPC-related conferences with a focus on experimental design and how to present the results to make them more interpretable [27]. Jimenez et al.’s work on using OS-level container can be adopted along with Pilot to get reproducible performance evaluation results [32].

Auto-pilot [77] provides a script language for controlling the execution of benchmarks along with other mechanisms for timing and analyzing results. Auto-pilot offers functions to calculate CI and stop the execution when the required width is reached but lacks other functions such as handling unit readings separately, reducing auto-correlation, detecting non-stable phases, or generating work amount dynamically. Some parts of it are written in Perl and cannot be easily integrated into C programs for doing high performance run-time analysis. ministat [35] is a tool for comparing benchmark results at a certain confidence level and drawing the results using ASCII art.
Chapter 5

CAPES: Deep Reinforcement Learning-Based Parameter Tuning

In previous chapters, we focused on the autonomous rule-based traffic control method, which requires no communication between autonomous clients and can scale to support very large systems, but does have the weakness that training could take a long time. If we allow some communication between clients with a control controller, we would be able to apply many other machine learning methods. CAPES is such a system that relies on periodically collecting a small amount of state information from each client, and applies Deep Reinforcement Learning to congestion control. The communication volume is very small. CAPES is fully automatic, unsupervised, and non-intrusive as ASCAR, and has the extra benefit of being able to do the training and tuning process online, doing away with ASCAR’s length offline training process.
5.1 Design and Algorithm

Figure 5.1 shows the architecture of CAPES and how it is deployed to optimize a target system. CAPES assumes little of the target system; it only requires an interface to periodically extract states of the system and a way to change parameter values. Figure 5.1 illustrates one possible way to use CAPES with a target system that contains several server nodes and application nodes. Each of these nodes will have a Monitoring Agent and a Control Agent running on them.
The Monitoring Agents gather Performance Indicators and Rewards from the target system’s nodes and send them to the Interface Daemon. Performance Indicators are system measurements that are related to the system’s operating status (§5.1.1). Rewards vary based on current tuning efforts, and reflect the successfulness of the current tuning (§5.1.2). The Interface Daemon (§5.1.3) relays the incoming Performance Indicators into the Replay DB. The DRL Engine reads the Performance Indicators from the Replay DB to do training steps (§5.1.4). At a fixed interval, the DRL Daemon sends back an Action via the Interface Daemon, which will broadcast the action to the Action’s targeted Control Agents. These actions are also stored within the Replay DB, as part of Experience Replay. Finally, the Control Agents makes the appropriate changes on the target nodes when actions are received (§5.1.7).

In a production environment, the Interface Daemon, Replay DB, and DRL Engine can run on one or more dedicated nodes, depending on the volume of the data, to prevent interference with other nodes in the target system. However, for efficiency, the DRL Engine needs to run on a node utilizing a GPU for faster Deep Neural Network computation. In addition, the node that the Replay DB runs on should have plenty of RAM, ideally to keep the whole database in memory.

5.1.1 Choosing Performance Indicators

Performance Indicators are important for tuning the system because DNN relies on analyzing them to understand how the system is running. We should include system states that are related to the metric we wish to tune. This is a feature selection problem, which was deemed to be one of the most important steps for successfully applying almost any machine learning algorithm. However, advances in deep neural networks (DNN) has largely made this step less important because DNN is good at picking out useful data among noisy, raw inputs.
Therefore, we can be quite liberal on choosing performance indicators; any system status that are likely related to the performance of the system should be included. Both raw system statuses and secondary system statuses, derived from raw system status, can be included. Samples of raw system status include number of CPUs, CPU utilization, free memory, separate read/write I/O rate of each storage device, and buffer size. Samples of secondary system status could be the total number of active threads, which needs to be calculated by counting the number of threads that are running.

All inputs to the Deep Neural Network should be converted into floating point numbers. This is easy for integers. Enumerative values, such as which I/O scheduler the system is using and what power status the system is in, should be converted into numerical values, such as 0, 1, or 2.

System statuses that are accumulative in nature should generally be excluded unless they are known to be related to system performance. Such accumulative statuses include system uptime, total send/received bytes of network, total read/write bytes of I/O devices, etc. The rates of change of these statuses can be useful for indicating a system’s operational status, but the total sums are generally not useful. For instance, it is unlikely that the system’s optimal settings should rely on how many bytes the system has received since it started up.

Date and time should also be included if the workload is known to be cyclical, such as many enterprise workloads, however we should not include it as a single representation. Instead, it is easier for the Deep Neural Network to understand if we include the month, day of the week, hour, and minute as separate performance indicators. By doing this, the Deep Neural Network can discover any relationship between change in the workload and the hour that it changed.
On the downside, picking too many performance indicators may result in increased computational cost. We can safely overlook this concern because the throughput of a modern desktop GPU can offset the increased costs given that we do not grossly overload the system (see Table 5.3 for the measured training speed in the evaluation).

5.1.2 Reward

Reward plays an important role and guides the direction of the whole tuning process. After performing each action on the target system, CAPES measures an immediate reward. For instance, after changing the congestion window size, we can measure the change of transmission throughput at the next second to use it as the reward. Only the immediate reward is necessary because there is no need to worry about the delay between an action and a reward since the Q-function will ultimately converge to the optimal oracle-like function after iterative trainings, according to Bellman’s proof [67].

In order for Bellman’s equation,

\[
Q(s, a) = r + \gamma \max_{a'} Q(s', a'),
\]

to converge, \(Q\) cannot be infinite. This is especially important for system performance tuning because our tuning does not have an ending point in time and would have an infinite number of future rewards. The traditional method is to introduce a discount, \(\gamma\), for future rewards. We have designed another method that uses a differential value for reward.

Using the previous sample, let \(p_t\) be the system’s perceived performance (such as throughput) at time \(t\). The reward after performing action \(a_t\) should be:

\[
r_t = p_{t-1} - p_t
\]
$p_t$ is called the cumulative reward. This can be easily calculated by retrieving the last two statuses from the Replay DB and doing the calculation.

Since the system can be tuned for different objectives, it is common to use a more complex objective function to calculate $p$ from several measurements, as shown in §3.1.5. A transaction processing system would need high throughput, while an interactive system may need lower latency and better fairness.

System statuses, such as throughput and latency, are naturally noisy because of the complex interaction of various components of the system; even for the most stable workload running on the simplest system, it is unlikely to get identical throughput readings for two consecutive measurements. Noise has negative impacts on the training process because it disrupts the link between reward and feedback. For instance, good actions may lead to bad rewards because of noise in the measurement. Therefore, noise makes applying DRL to performance tuning more challenging than to other deterministic systems such as Atari games [49]. In the evaluation section, we first need to evaluate whether this kind of noise would prevent the system from converge, and second, evaluate whether we can use techniques such as averaging to mitigate the noise in the system (§5.2.2).

### 5.1.3 Monitoring Agents and The Interface Daemon

A Monitoring Agent runs on each node that needs to be monitored. At a predesignated sampling frequency, it collects Performance Indicators and sends them to the Interface Daemon for processing. We call each of these actions a sampling tick. In order to minimize both CPU utilization and network communication, we use a differential communication protocol designed to only send out a performance indicator when its data is different from the value of the previous sampling tick. In addition, all network
communications are compressed. If the target system uses different networks for data and control, the monitoring agents uses the control network to communicate with the Interface Daemon.

The Interface Daemon is a lightweight daemon that receives incoming messages from all Monitoring Agents. It also receives suggested actions from the DRL Engine, and broadcasts them to the Control Agents. Introducing the Interface Daemon into the system has several benefits. First, it decouples the network communication code from other parts of the system code. Second, it is the only component that needs to write to the Replay DB (the DRL Engine only needs to read from it), greatly reducing the overhead of locking the Replay DB. Third, this enables independent control of the Monitoring Agent and the DRL Engine so we can choose to do solely monitoring or training on demand.

### 5.1.4 Modeling and Training the Deep Neural Network

Mathematically, the purpose of the training step is to minimize the prediction error for the training data. The Deep Reinforcement Learning (DRL) Engine retrieves uniformly random observations from the Replay DB and feeds them randomly into the deep Q-network for training (experience replay). Because the performance indicators of one sampling tick cannot reflect the moving trend of these indicators, it is common to use a stack of multiple, consecutive, snapshots in the DNN training process \[^{49}\]. Let \( d_{i,j} \) be the total performance indicators of node \( i \) at time \( j \), \( N \) be total number of nodes, and
$S$ be the number of sampling ticks. We construct the observation at time $t$ as a matrix:

$$s_t = \begin{bmatrix}
    d_{1,t-S+1} & d_{2,t-S+1} & \cdots & d_{N,t-S+1} \\
    d_{1,t-S+2} & d_{2,t-S+2} & \cdots & d_{N,t-S+2} \\
    \vdots & \vdots & \ddots & \vdots \\
    d_{1,t} & d_{2,t} & \cdots & d_{N,t}
\end{bmatrix}$$

A set of random observations from the Replay DB are packed together as one minibatch and fed to the DNN trainer. The batching minimizes data movement overhead between the main memory and GPU memory, and can be processed using matrix-matrix multiplies, which is especially efficient in the context of GPU processing.

The Q function can be parameterized using a neural network in many ways that differ in terms of the number, size, and type of hidden layers, and how the Q-value (e.g. the predicted reward) for candidate actions are calculated. There are primarily two methods for calculating the Q-values: the first type maps an observation-action pair to scalar estimates, the second type maps an observation to an array of Q-values of each action. The first type requires a separate forward pass to compute the Q-value of each candidate action, resulting in a cost that scales linearly with the number of actions. The main advantage of the second type is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. We have chosen the second type for CAPES.

Because the observations are float numbers that represent system statuses and are usually not related by locality (adjacent numbers in observation are not necessarily related), we choose to use a multi-layered perceptron (MLP) network to construct the DNN. MLP is a mature method that can learn to classify any linearly separable and non-separable set of inputs. It can represent boolean functions, such as AND, OR, NOT, and XOR, and can allow a user to get approximate solutions for complex problems.
In CAPES, we use a standard two-hidden-layer MLP with a hyperbolic tangent (tanh) nonlinear activation function. The two hidden layers are of the same size of the input array. The final output layer is a fully-connected linear layer with a single output for each valid action.

We use the RMSProp method [26] for reducing the error of the DNN as calculated from Equation 2.1. RMSProp is a minibatched version of resilient backpropagation [58]. The DRL Engine is a separate process, and always runs during the training step using different random minibatches. For each minibatch, we update the target network’s $\theta_i^−$ using $\theta_i$:

$$\theta_i^− = \theta_i^− \times (1 - \alpha) + \theta_i \times \alpha,$$

where $\alpha$ is the target network update rate.

### 5.1.5  Replay Database

One training step needs the transition of system status from $t$ to $t + 1$, the action performed, and the reward after performing the action: $d_t = (s_t, s_{t+1}, a_t, r_t)$. In CAPES, we store system status and actions in two tables that are indexed by $t$ in the Replay Database. The algorithm CAPES uses to construct a minibatch for training is shown in Algorithm 2.

CAPES checks that the Replay DB contains enough data for each sampled timestamp. Because of the noisy nature of the system, it is common for certain nodes to have missing data. DNN can well tolerate a certain degree of missing data.
Algorithm 2 Constructing a minibatch of size \( n \) from data in the Replay DB.

1: procedure \textsc{ConstructMinibatch}(n)
2: \hspace{1em} samplesNeeded \leftarrow n
3: \hspace{1em} while True do
4: \hspace{2em} Uniformly generate \( \text{samplesNeeded} \) timestamps
5: \hspace{2em} for each timestamp \( t_i \) do
6: \hspace{3em} if Replay DB contains enough data at \( t_i \) then
7: \hspace{4em} Get \( s_t, s_{t+1}, a_t \) from Replay DB
8: \hspace{4em} \( r_i \leftarrow \text{CalcReward}(s_t, s_{t+1}) \)
9: \hspace{4em} \( D^+ = (s_t, s_{t+1}, a_t, r_t) \)
10: \hspace{2em} end if
11: \hspace{2em} end for
12: \hspace{1em} if \( D \) has \( n \) samples then return \( D \)
13: \hspace{1em} end if
14: \hspace{1em} samplesNeeded \leftarrow n - \text{len}(D)
15: \hspace{1em} end while
16: end procedure

5.1.6 Exploration Period

As we have stated in the background section, it is important for the agent to experience as many states as possible during the training process. The initial training period uses a standard \( \epsilon \)-greedy policy, in which the tuning agent takes the estimated optimal action with probability \( 1 - \epsilon \), and randomly picks an action for the other cases. We let \( \epsilon \) to anneal linearly from 1.0 to 0.05 during the training period. Additionally, the Interface Daemon has a controlling program that has access to the scheduling of the workload. Whenever a new workload is started on the system, the Interface Daemon notifies the DRL Engine to bump up \( \epsilon \) to 0.2 (so 20% random action) so that the tuning agent can do some exploration while avoiding local maximums.

5.1.7 Performing Actions

Actions dictate what a target system’s parameters should be, and CAPES can tune many parameters at the same time. At a fixed rate (every action tick), CAPES decides on an action that either increases or decreases one parameter by a step size. The valid range
and tuning step size are customizable for each target system. For instance, one can say
that we need to tune the I/O size, which has a valid range from 1 KB to 256 KB, and
a tuning step size of 1 KB. We also include a NULL action that performs no action if
we wish. The DNN can choose to do the NULL action if it sees no need to change any
parameter. Thus, the total number of actions we are training the DNN for is:

\[ 2 \times \text{number\_of\_tunable\_parameters} + 1 \]

The same observation data format is used in both training and action steps. The
DRL Engine always uses the observation of the current \( t \) to calculate the candidate
action. Before broadcast, the Interface Daemon will call an Action checker to rule out
egregiously bad actions. This step is optional, and we have not used it in our evaluations,
but if there are known bad parameter values, they can be shielded from the target system.
The Interface Daemon then determines which Action Message should be sent to which
Control Agent. A Control Agent will listen for inbound Action Messages from the
Interface Daemon and will change the system parameters accordingly.

The architecture shown in Figure 5.1 is not meant to limit how the system can be
deployed. For instance, if the sampling of performance indicators is already implemented
and the data is stored on a central monitoring node, CAPES can tap into that information
without the need to deploy Monitoring Agents.
5.2 Implementation and Evaluation

We chose the Lustre file system as the target system because it is a high performance
distributed file system that can distribute I/O requests from every node to many servers
in parallel. It can also generate a huge amount of I/O to stress the system. The purpose
of the evaluation is to test whether CAPES can improve the throughput of the workload
during peak times and to understand its effectiveness on a variety of workloads.

5.2.1 Implementation

We implemented a CAPES prototype to evaluate this design. The majority of the
system is written in Python, with the DNN implemented using Google TensorFlow \[1\].
We carefully profiled all code and optimized all hotspots to ensure minimal resource use
of the Monitoring and Control Agent, and to maximize the training speed. The Replay
DB is a SQLite database using Write-Ahead-Logging for optimal concurrent write/read
performance. The whole system has about 6,000 lines of code.

Each Lustre client maintains one Object Storage Client (OSC) for a server it talks
to. We have four servers, and are using stripe count four so each client has four OSCs.
Each OSC’s Performance Indicators are calculated independently. We use the same
Performance Indicators for detecting congestion as used earlier in §3.1.3:

1. \texttt{max\_rpcs\_in\_flight}: Lustre congestion window size.
2. Ack EWMA: exponentially weighted moving average (EWMA) of gaps between
server replies.
3. Send EWMA: EWMA of gaps between the original sent times of the corresponding
requests of the replies received by the client.
   Process Time is the time needed by the server to process one I/O request.
5.2.2 Converge Test

Before starting to evaluate CAPES on a real system, we need to make sure that the algorithm can actually converge. Because a real system has too many uncertainties that are out of our control, we designed a simulator, StorageGameSim, for this converge test. We have to make it clear that this simulator is designed and implemented only for this single test and is never meant to be realistic, and could not be used to train neural networks for tuning real systems.

StorageGameSim simulates a storage system that has two control parameters, the congestion window size ($m$) and I/O rate limit ($\tau$). Every second, StorageGameSim calculates a measured throughput solely decided by $m$ and $\tau$. The throughput is first calculated from the congestion window size using a relation shown in Figure 5.2 and then capped by $\tau$, which is similar to the $\tau$ we used in the real system. The throughput is affected by the congestion window size in such a way that it peaks at a certain window size (here when $m = 50$), and gets lower when the congestion window size moves away from the ideal size.

![Figure 5.2: Throughput of different congestion window sizes in StorageGameSim](image)

Figure 5.2: Throughput of different congestion window sizes in StorageGameSim
During this test, StorageGameSim is controlled by the DQL Daemon, and is considered succeeded (converged) when the throughput has been maintained within ideal_throughput $\pm 10\% \times$ ideal_throughput for more than 20 seconds without break. We measure how many (simulated) seconds has passed before the DQL Daemon successfully “wins” the game.

We also introduced two mechanisms into StorageGameSim to make it more realistic and more challenging for the DQL controller.

- A Gaussian noise is added to the final calculated throughput before returning it to the DQL controller. The standard deviation of the noise is set to be a certain percentage of the throughput, and we have evaluated several different levels of noise as listed in Table 5.1: 1%, 10%, and 50%. The standard deviation of real measured throughput on our system is between 10% and 50%, as a reference.

- A delay between change in congestion window and corresponding change in throughput. In other words, the throughput is calculated using the congestion window three seconds ago. This is to test if the DQL controller can successfully attribute rewards to the correct action when delay exists.

The converge test can also be used as a way of doing hyperparameter search, in which different hyperparameters are evaluated and tested to see their effect on converge time. The most important setting is how the reward should be calculated. As stated in the reward section (§ 5.1.2), we need to evaluate different reward calculation methods to see how they fare when facing different levels of noise. Simple averaging and a more complex form, exponentially weighted moving average (EWMA), are common methods for filtering out the noise from measurements, and we have designed three methods for calculating rewards. Let $p_t$ be the throughput measured at time $t$, the reward $r_t$ is calculated as:
• Differences of throughput: \( r_t = p_t - p_{t-1} \).

• Differences of throughput moving average (arithmetic mean): \( r_t = \text{avg}(p_{t-m-1}, \cdots, p_{t-1}) - \text{avg}(p_{t-m}, \cdots, p_t) \). \( m \) is the average window size.

• Differences of throughput EWMA: \( r_t = \text{ewma}_t - \text{ewma}_{t-1} \).

Table 5.1: Successful rate for converge test

<table>
<thead>
<tr>
<th>Noise percentage level</th>
<th>Reward type</th>
<th>Diff.</th>
<th>Diff. avg</th>
<th>Diff. EWMA</th>
<th>Sample size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>100%</td>
<td>100%</td>
<td>97%</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td>76%</td>
<td>100%</td>
<td>94%</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>94%</td>
<td>56%</td>
<td>82%</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1 shows the percentage of tests that successfully converged to and hold the ideal congestion window size within a 1-hour time limit. Any tests that lasted longer than one hour are considered failed. Among those successful tests, Figure 5.3 shows the converge time.
Figure 5.3: StorageGameSim converge time under different noise levels. Error bars show 95% confidence intervals. Noise is measured by the ratio of standard deviation to throughput mean.

Figure 5.3 shows that the higher the noise level, the more training steps DQL needed to converge. Taking the success rate as shown in Table 5.1 into consideration, the “diff. method” has the highest success rate in high noise environment and also the shortest converge time. It is the method that we use in the following experiments.

5.2.3 System Setup

The evaluation system contains four dedicated servers and four dedicated clients. All nodes use the same hardware: an Intel Xeon CPU E3-1230 V2 @ 3.30 GHz, 16 GB RAM, and one Intel 330 SSD for the OS. The network is gigabit ethernet with measured peak aggregated throughput of ∼500 MB/s. Each storage server node uses one 7200 RPM HGST Travelstar Z7K500 hard drive, of which raw I/O performance is measured at 113 MB/s for sequential read and 106 MB/s for sequential write. We used Lustre’s default
stripe count four and 1 MB stripe size. 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 a write completion when data hits the disk. We specifically picked this storage and network hardware so the whole system has a 1:1 network to storage bandwidth ratio, matching other larger supercomputers [9], in order to study and test CAPES on a system that mimics typical real world resource-constrained environments. The storage workloads were generated using Filebench [66] running on all clients in parallel.

The Monitoring and Control Agents only run on Lustre clients, and we do not tune anything on the server node for this prototype. All other components of CAPES run on another dedicated node. Our CAPES node has an Intel Xeon CPU E5-2637 @ 3.00 GHz, 128 GB RAM, an SSD RAID, and one nVidia GTX 1070 GPU.

It is worth noting that the whole evaluation system is not located on an isolated network due to the IT requirements of our department, and we have observed network traffic interference from time to time, such as the routine network scanning of the IT department and machine status queries from the cluster monitoring system. We did not isolate the whole system because we consider this kind of noise as beneficial to the evaluation, because more noise makes the training and tuning process challenging, and a tuning system works only within a perfect environment is not pragmatically interesting.

The hyperparameters used in the evaluation are listed in Table 5.2. We chose those values using informal trials with minimal effort, so it is conceivable that better tuning results and/or a shorter training duration can be achieved by using better values. It is within our future work to perform a systematic grid search on these hyperparameters.
Table 5.2: List of hyperparameters and their values used in CAPES evaluation

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action tick length</td>
<td>1</td>
<td>One action is performed every second.</td>
</tr>
<tr>
<td>$\epsilon$ initial value</td>
<td>1</td>
<td>Initial value of $\epsilon$ (100% actions are random at the beginning of training).</td>
</tr>
<tr>
<td>$\epsilon$ final value</td>
<td>0.05</td>
<td>Final value of $\epsilon$ (5% actions are random after the training process).</td>
</tr>
<tr>
<td>discount rate ($\gamma$)</td>
<td>0.99</td>
<td>The discount rate as used in Equation 2.1</td>
</tr>
<tr>
<td>initial exploration period</td>
<td>20h</td>
<td>The duration of which the initial value of $\epsilon$ is linearly annealed to the final value.</td>
</tr>
<tr>
<td>minibatch size</td>
<td>32</td>
<td>Number of observations over which each stochastic gradient descent update is performed.</td>
</tr>
<tr>
<td>missing entry tolerance</td>
<td>20%</td>
<td>For each observation, as much as 20% missing data is tolerated.</td>
</tr>
<tr>
<td>number of hidden layers</td>
<td>2</td>
<td>The number of hidden layers beside the input and output layers. The size of the hidden layers is the same as the input.</td>
</tr>
<tr>
<td>RMSProp learning rate</td>
<td>0.001</td>
<td>The learning rate of RMSProp.</td>
</tr>
<tr>
<td>RMSProp decay</td>
<td>0.9</td>
<td>Discounting factor for the history/coming gradient.</td>
</tr>
<tr>
<td>sampling tick length</td>
<td>1 s</td>
<td>One sample is taken every second.</td>
</tr>
<tr>
<td>sampling ticks per observation</td>
<td>10</td>
<td>The number of sampling ticks to be included in one Observation. This effectively pack 10 seconds information leading to $t$ into one observation.</td>
</tr>
<tr>
<td>target network update rate ($\alpha$)</td>
<td>0.01</td>
<td>For each minibatch, the target network’s $\theta_i^-$ is updated as $\theta_i^- = \theta_i^- * (1 - \alpha) + \theta_i * \alpha$.</td>
</tr>
</tbody>
</table>

5.2.4 Evaluation Workloads and Performance Increase

We evaluated the following synthetic workloads:

- Random read and write with various read to write ratios: 9 : 1, 4 : 1, 1 : 1, 1 : 4, 1 : 9;
- Filebench file server; and
- Filebench five-stream concurrent sequential write.
**Figure 5.4:** Overview of random read write workloads evaluated with CAPES. Throughput before, after 12 hours training, and after 24 hours training are shown. Error bars show 95% confidence intervals.

**Random read and write workloads** In these random read and write workloads, each client has five threads doing the same random read and write with a fixed ratio. We have evaluated various different read to write ratios to mimic a broad range of real applications. We conducted training processes of 12 and 24 hours with the goal of optimizing the aggregated read/write throughput. After training, we evaluated the effects of CAPES’s tuning.
It can be seen in Figure 5.4 that CAPES works best with workloads that are dominated by writes; it increased the performance of the workload with 1:9 read to write ratio by 45%. CAPES did not show obvious effect on read-heavy workloads. This is expected because tuning the number of allowed outstanding I/O requests of Lustre does have a bigger impact on write than read. The evaluation used storage servers that have hard disk drives as the underlying storage device, which need to spend a majority of I/O time doing seek for random reads and would not be affected much by the number of outstanding read requests. In contrast, outstanding random write requests can be merged and handled more efficiently, thus tuning the number of allowed outstanding write requests has a bigger impact on the efficiency of the merge, and in turn the performance.

We also measured the performance results after different training duration. This can help us understand how long the training session needs to be. We can see that training for 24 hours had slightly better results than training for 12 hours only for read-heavy workloads, and had little effect on other workloads. This is likely due to that changing the congest window size has a non-obvious effect on the read performance, and that small changes in the read performance cannot be easily discerned from noise. Therefore, it is understandable that the training would need a longer duration to converge.
**Figure 5.5:** Overview of Filebench file server and sequential write workload evaluated with CAPES. Throughput before and after CAPES tuning are shown. Error bars show 95% confidence intervals.

**Filebench file server workload**  In addition to the random read write workloads, we have also evaluated the Filebench file server and a sequential write workload, as shown in Figure 5.5. Filebench file server is a synthetic workload and mimics a busy file server, which is one of the most common and important workloads among data centers and enterprise storage servers. It simulates a typical heavy-loaded file server I/O pattern and includes both read, write, and metadata operation. It loops through the following I/O operations using a prepopulated set of files:

1. Create a file and write the file to 100 MB,
2. Open another file and append random sized data (mean at 100 MB),
3. Open a randomly picked file and read 100 MB,
4. Delete a random file, and
5. Stat a random file.
We ran two instances of the file server workload on each node and have eight instances in total. These nodes generate enough traffic to saturate the server nodes. The sequential write workload has five sequential write threads on each client. Each thread does sequential write with 1 MB write size.

In order to test whether there is overfitting and the trained DNN is general enough to handle the workload when some system settings are changed for the file server workload, we conducted three benchmark sessions in total. Each session includes measuring the baseline throughput and tuned throughput, each taking two hours each. Therefore, the total benchmark duration for each session is four hours.

![Filebench file server workload throughput with and without CAPES tuning.](image)

**Figure 5.6:** Filebench file server workload throughput with and without CAPES tuning. Error bars show the confidence interval at 95% confidence level.

Some previous parameter optimization and congestion control systems suffered from overfitting, and were sensitive to changes in the file system, especially changes of the on disk data location, allocation of files among servers, and available free space. To test if the trained DNN also suffers from this problem, we spread out the three benchmark sessions over two weeks and added numerous unrelated heavy I/Os between the sessions. The results are shown in Figure 5.6 and show that CAPES has achieved increases in
throughput between 13% and 36%. Rigorous statistical checks have been done using the Pilot tool (Chapter 4): throughput was measured every second; autocorrelation of the samples are checked to ensure they are i.i.d. and not temporally correlated; confidence intervals are calculated at 95% confidence level.

5.2.5 Training Efficiency

Figure 5.7 shows how the predicted error changes over time during the whole training process. The prediction error show the difference between the DNN’s predicted performance and the real performance. It is an important metric of training efficiency: the lower prediction error it gets, the better the DNN can know which action to take to get a desired performance boost.

![Prediction error over time](image)

**Figure 5.7**: Prediction error during the training process

We can see that the prediction error decreases steadily as the training session continues. The disturbance in the first half of the training process may be caused by network interference, other I/O requests on the same file system, or some other reasons that we could not identify, because the evaluation cluster is not wholly isolated.
### 5.2.6 Training Session’s Impact on the Workload

The training session includes carrying out random actions on the target system, therefore it is important to understand the training’s impact on the target system’s performance. Because we used an \( \epsilon \)-greedy policy that anneals from 100% random action to 5% action, the DNN should be able to “mitigate” the impact of the bad random actions when it has a chance to deliver a calculated action, except for the beginning of the training session. Figure 5.8 confirms this speculation and shows that the overall throughput of the whole 70-hour training session is comparable to the three baseline throughputs we measured at three different times.

![Training’s Impact on Performance](image)

**Figure 5.8:** Training’s impact on the workload performance. Throughput without and with CAPES training are shown. Error bars show the confidence interval at 95% confidence level.

### 5.2.7 Other Measurements

We provide other related measurements we have collected during the evaluation process in Table 5.3. They are useful for understanding the computational cost and for planning to build a trainer for a larger system. It can be seen that the messages sent out
by the Monitoring Agents utilizes a small amount of network traffic, and the size of the
Replay DB handleable by a modern computer. Using a modern GPU can also achieve
about 10x the training performance in comparison to running it on a modern CPU.

Note that these measurements were from an in-house cluster that had eight Monitoring
Agents. Training times should grow linearly when the number of nodes increases. Using
a fast GPU is critical for finishing the training step as quickly as possible, especially
when the target system has a large number of nodes. TensorFlow is designed to be
efficient for distributed machine learning, therefore when necessary multiple GPUs (in
SLI mode), or multiple training nodes should be adopted.

**Table 5.3:** List of technical measurements of the CAPES evaluation (8 Monitoring
Agents in Total)

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration of training step (CPU)</td>
<td>(\approx 0.1) s</td>
<td>One training step of a 32-observation minibatch on CPU.</td>
</tr>
<tr>
<td>duration of training step (GPU)</td>
<td>(\approx 0.01) s</td>
<td>One training step of a 32-observation minibatch on GPU.</td>
</tr>
<tr>
<td>number of records of the Replay DB</td>
<td>250k</td>
<td>One record per second. 70 hours in total.</td>
</tr>
<tr>
<td>size of the DNN model</td>
<td>84 MB</td>
<td>The size of the deep neural network in memory.</td>
</tr>
<tr>
<td>total size of the Replay DB on disk</td>
<td>0.5 GB</td>
<td>The size of the SQLite database on disk (no compression).</td>
</tr>
<tr>
<td>total size of the Replay DB in memory</td>
<td>1.5 GB</td>
<td>The size of the whole Replay DB in memory when being used by the training session.</td>
</tr>
<tr>
<td>performance indicators per client</td>
<td>44</td>
<td>Every client collects this many performance indicators per second (float numbers).</td>
</tr>
<tr>
<td>observation size</td>
<td>1760</td>
<td>One observation contains this many float numbers.</td>
</tr>
<tr>
<td>average message size per client</td>
<td>(\approx 186) B</td>
<td>Every second one client sends out about this many bytes to the Interface Daemon. This is the compressed size of all 44 performance indicators.</td>
</tr>
</tbody>
</table>
5.3 Related Work

Parameter optimization is a challenging research question. The optimal values of parameters can be affected by every aspect of the workloads and the system, such as the I/O request size, randomness, and network topology. Different software versions can also have different quirks, causing their performance to vary. Existing solutions can be classified by whether a model is required and whether the tuning is a one-time process or a continuous process that can be used in production.

Feedback control theory is commonly used in model-based approaches and are often combined with slow-start, fast fallback heuristics [14, 68, 85]. There are other more complex models as well [36, 71]. Model-based approaches work well when the system and workloads are relatively simple and well understood. Most of these solutions still require the administrator to choose values for critical parameters. For instance, if the start is too slow or the falling back is too fast, the system’s capacity is wasted; if the speed increases too fast or the falling back is not fast enough, the system becomes unstable under peak workloads.

Model-less, general purpose approaches usually treat the target system as a black box with knobs and adopt a certain search algorithm, such as hill climbing or evolutionary algorithms [31, 60, 75]. These search-based solutions are often designed as a one-time process to find the optimal parameter values for a certain workload running on a certain system. The search process usually requires a simulator, a small test system, or the target system to be in a controlled environment where the user workload can be repeated again and again, testing different parameter values. Most of these search methods are a one-time process: if the status of the target system or workloads do not match what the optimizer saw during the bootstrap tuning process, it would fail to improve the system. This inflexibility limited their use in real world environments. There are also domain specific solutions that tunes the parameters of a certain application [4, 17, 70].
The efficiency of search-based algorithms depends on the size of the parameter space, and many of them suffer from overfitting because search algorithms do not provide generalization; when the system or workload changes, the search process needs to be redone. Zhang et al. proposed a method that used neural network to accelerate a traditional search method and to add a certain degree of generalization [84]. [11] is an early attempt at using neural network-based reinforcement learning to tune a single server. CAPES is a more complete system that works on a larger scale, and has taken advantage of the recent rapid progress of deep learning techniques.

There are other optimization solutions that change the architecture of the system automatically, like Hippodrome [3]. They require intrusive and radical modifications to the whole system. There are also tools such as [87] that can manage parameters of a large number of nodes. CAPES can work in tandem with such systems to achieve more comprehensive coverage of performance optimization in addition to parameter tuning.
Chapter 6

Conclusion and Future Work

Storage systems that are not properly tuned are very common among small and large organizations, who usually lack the resources or expertise to carry out the tuning task, or have only done the tuning when the system is being installed. Since we have demonstrated that we can achieve a more than 30% increase in throughput for many workloads in this research, we can know that currently there is a huge waste of resources out there. With more and more data are being migrated to data centers, any increase in performance and reduction in cost can go a long way to address the rising human and energy cost of the rapidly expanding computing industry.

This thesis introduced the Automatic Storage Contention Alleviation and Reduction system (ASCAR), a fully autonomous storage traffic contention management system. Two control methods are introduced, evaluated, and compared: a rule-based method (SHARP) and a Deep Q-Learning-based method (CAPES). We have built prototype systems to validate these ideas and demonstrated their performance. 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.

6.1 Automatic Congestion Control

The main purpose of this research is to demonstrate that automatic congestion control can effectively increase a system’s peak time performance without any human input. Because ASCAR and CAPES only requires adding monitoring and tuning function to the clients, which already exist in many existing systems, they can be easily applied to many small and large systems with little cost. They can help many organizations and data centers to optimize their systems’ performance and reduce cost.

**SHARP** When in rule-based control mode, 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 SHARP 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 [75].

There are several limitations of the rule-based control mode of ASCAR. First, the long offline studying/optimizing process, while can be justified for repetitive workloads, can be impractical for unpredictable workloads. 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. CAPES attacks these problem using a different solution.
CAPES  CAPES is capable of finding optimal values for the congestion window size and I/O rate limit of a distributed storage system in a noisy environment. The optimal tuning reduces peak time congestion and increases overall throughput by up to 45% for write-heavy workloads and around 13% for a mixed read/write/metadata workload. It does not have obvious effects on read-heavy workloads. The neural network-based tuner reached peak performance mostly within 12 hours of training, and for read-heavy or very noise workloads it might need longer training time.

Compared to manual parameter tuning, CAPES is superior in that it does not require to be supervised, it does not require prior knowledge of the system, it can always run during normal operations, and it can dynamically change parameters. We maintain that automated tuning system could play an important role for future complex distributed systems, such as data centers and supercomputers, to both reduce management costs and increase performance.

CAPES’s design is general purpose and does not assume anything except that a target system has parameters that can be tuned during run time. With an early prototype, we have demonstrated that it can tune a Lustre file system with minimal human intervention.

DNN-based reinforcement learning does have a disadvantage in that it can be difficult to explain how the trained model works. Usually this is not a compelling problem for performance tuning problems, but can be problematic if the target system is mission critical and suboptimal actions need to be absolutely avoided. That is why we introduced the action checker component (see Figure 5.1).

The problems that ASCAR and CAPES can solve have some overlap. They can both to used to ease the congestion in the system and improve performance, yet they have the following distinctions:

- ASCAR focuses on throttling traffics while CAPES can tune almost any parameter.
- ASCAR requires a special training phase while CAPES does not.
• ASCAR can generate rules that require no data sharing between clients, which is especially good for scalability, while the current design of CAPES requires constant communication between Agents and the Engine.

6.2 Automatic Benchmark and Performance Measurement

This thesis also described the performance measurement component of ASCAR named Pilot that can be used separately. Manually monitoring and performing benchmark analysis are demanding and error prone. Pilot includes a series of algorithms and heuristics to automate this process and generate reports that are scientifically and statistically valid using as short time as possible. Evaluation shows that these methods are non-parametric and robust, and can shorten the time needed for running benchmark. Pilot can be downloaded and compiled separately as an easy-to-use open source framework, which is under active development and may contain more heuristics methods for running benchmarks than can be described in this thesis. We hope it can increase the quality of performance evaluation in computer systems research and reduce people’s effort.

6.3 Future Work

There are three categories of future work that can be done to expand the research as described in this thesis: theoretical work, implementation work, and application work.
6.3.1 Theoretical

Automatic computer performance tuning is an emerging technology that need to further explored, especially with the increasing human and management cost of modern data centers. Recent advances in the deep learning technology have already achieved very impressive results on many challenging problems such as computer gaming playing and data center management [49]. New deep learning techniques are being invented on an almost daily basis and sometimes can greatly increase the training efficiency. These new techniques, such as batch normalization and continuous Deep Q learning [40], need be systematically evaluated and added to CAPES to make it more intelligent and generate better results.

Another research opportunity is to combine ASCAR SHARP’s rule generation and the design of neural networks from CAPES to design a new system that uses Deep Reinforcement Learning to tweak the rules and the parameters in the rules. By doing this we can get an ideal system: a rule-based shared-nothing system that does not require a separate training process.

For Pilot, we can develop more functions, such as a distributed mode that can measure and make decisions based on measurements from many nodes and analytic functions for detecting slow shift for long term monitoring. The development will be handled in a community-friendly manner and we welcome outside contributions.

6.3.2 Implementation

Improvements can be done for the Lustre-specific evaluation prototype. For instance, we can collect information from server nodes in addition to client nodes. We can also tune more parameters in addition to the congestion window size and a hard rate limit; DNN is known to be quite effective at handling 20 or more candidate actions [49], which
maps to at least 10 tunable parameters. We would also look into using a systematic approach to hyperparameter optimization, such as using grid search, to make the machine learning processes in ASCAR and CAPES more efficient.

6.3.3 Application

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, such as Ceph [73] and OpenStack [https://www.openstack.org/].

In some use cases, bounded throughput is still needed in addition to proportional allocation [47]. 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 functions.

Full in-memory storage is gaining popularity for performance-critical application, like multi-stage checkpoint [38, 42, 50] or high-speed database (like RAMCloud [54] and memcache [51]). Network congestion is often severer in these systems due to the high bandwidth provided by RAM. We would like to expand and experiment ASCAR with these systems in the future.

We published the source code of our prototype and raw experimental data at [http://www.ssrc.ucsc.edu/ascar.html](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. Pilot’s source code, precompiled packages, tutorials, API documentation, mailing list, issue tracker, and wiki can be found at [https://ascar.io/pilot](https://ascar.io/pilot).
Bibliography


[74] B. L. Welch. “The generalization of Student’s problem when several different population variances are involved”. In: Biometrika 34.1-2 (1947), pp. 28–35.


