3-25-2021

Enabling Distributed Applications Optimization in Cloud Environment

Pinchao Liu
pliu002@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

ENABLING DISTRIBUTED APPLICATIONS OPTIMIZATION IN CLOUD ENVIRONMENT

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Pinchao Liu

2021

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Pinchao Liu, and entitled Enabling Distributed Applications Optimization in Cloud Environment, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

S. S. Iyengar

Jason Liu

Deng Pan

Gang Quan

Liting Hu, Major Professor

Date of Defense: March 25, 2021

The dissertation of Pinchao Liu is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2021

DEDICATION

I dedicate this dissertation to my dad, Dianqing Liu, who encouraged me to pursue this degree but was never able to see me finish, and to my mother and wife, for their unconditional love, support, and understanding during this process.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor, Dr. Liting Hu, for her unlimited support and guidance during my Ph.D. program. Dr. Hu guided me into new research areas and taught me a lot of research experience. She encouraged me to create novel ideas, guided me in preparing research draft, and instantly introduced cutting-edge topics to me. She always encouraged me to attend the top-tier conferences and supported me to take the opportunity to communicate with other researchers and scientists. Her unlimited support helps me to be succeed in my Ph.D. career and create many fantastic research works. She guided me to join the cycle of systems research in computer science and trained me to be a young scientist.

Second, I would like to thank my committees and professors at Florida International University. Many thanks to my committees, Dr. S. S. Iyengar, Dr. Jason Liu, Dr. Wei Zeng, and Dr. Gang Quan, for your great encouragement and support during my research work and guided me to deeply explore unsolved problems.

Third, I would like to express my thanks to my mentors during my intern at Facebook, Dr. Hui Cheng and Dr. Vipul Bansal. Thank you for your guidance and support in the fall of 2020, which gave me a taste of frontier research across the industry and academia.

Forth, I would like to thank my friends at FIU and Miami, who give me warm supports and brought colorful life during this special journey. Many thanks to Hailu Xu and Boyuan Guan, my teammates in Elves research Lab. Many thanks to Qian Zhong, Yekun Xu, Wei Ren, Yuyang Zhou, Xiaoyu Dong, Shaogang Yu, Wang Tang, Tianyi Wang, Huifeng Zhang, and to all my friends that I encountered in this beautiful city. Thank you all for being special parts during this journey.

Finally, from the bottom of my heart, I would like to express my gratitude to my parents and family. My father, Dianqing Liu, who supported me to pursue a higher degree and always encouraged and trusted me. My mother, Xifeng Wang, who gives me priceless

and infinite love and support, which helps me to be here. My wife, Na Jia, who gives her understanding to let me finish my degree and supports my family in my hometown. Their love overwhelms everything I have and is so invaluable that it exceeds anything in the world.

ABSTRACT OF THE DISSERTATION

ENABLING DISTRIBUTED APPLICATIONS OPTIMIZATION

IN CLOUD ENVIRONMENT

by

Pinchao Liu

Florida International University, 2021

Miami, Florida

Professor Liting Hu, Major Professor

The past few years have seen dramatic growth in the popularity of public clouds, such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Container-as-a-Service (CaaS). In both commercial and scientific fields, quick environment setup and application deployment become a mandatory requirement. As a result, more and more organizations choose cloud environments instead of setting up the environment by themselves from scratch. The cloud computing resources such as server engines, orchestration, and the underlying server resources are served to the users as a service from a cloud provider. Most of the applications that run in public clouds are the distributed applications, also called multi-tier applications, which require a set of servers, a service ensemble, that cooperate and communicate to jointly provide a certain service or accomplish a task. Moreover, a few research efforts are conducting in providing an overall solution for distributed applications optimization in the public cloud.

In this dissertation, we present three systems that enable distributed applications optimization: (1) the first part introduces DocMan, a toolset for detecting containerized application's dependencies in CaaS clouds, (2) the second part introduces a system to deal with hot/cold blocks in distributed applications, (3) the third part introduces a system named FP4S, a novel fragment-based parallel state recovery mechanism that can handle many simultaneous failures for a large number of concurrently running stream applications.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
1.1 Motivation 1
1.1.1 Dependency Detection of Distributed Applications 3
1.1.2 Storage Optimization of Distributed Applications 3
1.1.3 States Recovery of Distributed Applications 4
1.2 Summary and Roadmap 4

2. BACKGROUND 6
2.1 Distributed Applications' Dependency Detection Works 6
2.1.1 Injecting Code to Obtain Runtime Information 6
2.1.2 Based on Communication Pattern 6
2.1.3 Network Packets Inference 7
2.2 Distributed Applications' Storage Optimization Works 7
2.2.1 Replica Allocation and Management Based Systems 8
2.2.2 Skewed Popularity Based Systems 9
2.2.3 Caching Based Systems 10
2.3 Distributed Applications' States Recovery Works 11
2.3.1 Stateful date Processing Systems 11
2.3.2 Failure Recovery in Data Processing Systems 12

3. CORRELATION DETECTION OF DISTRIBUTED SYSTEMS 15
3.1 DocMan System Design 21
3.1.1 Data Collection 21
3.1.2 Distance Identification 23
3.1.3 Hierarchical Clustering 25
3.1.4 Resource Usage Pattern Prediction 26
3.1.5 DocMan Integrates with Kubernetes 27
3.1.6 Rationale 28
3.1.7 Testbed 29
3.1.8 Workload and Metrics 29
3.1.9 DocMan's Functionality Evaluation 31
3.1.10 Discussion 40
3.2 Summary 41

4. ADAPTIVE REPLICATION OF HOT/COLD BLOCKS OF DISTRIBUTED SYSTEMS 43
4.1 Introduction 43
4.2 Architecture Design 47
4.2.1 Data Generation 47
4.2.2 Dynamic Model to Copy Hot Blocks to MemCached 47

4.2.3 Caching with MemCached . . . 48
4.2.4 Evaluation Metrics . . . 50
4.3 Experimental Results . . . 51
4.3.1 Dynamic Hot and Cold Block Detection . . . 51
4.3.2 Vanilla Model to Copy Hot Blocks to MemCached . . . 52
4.3.3 Performance Analysis . . . 55
4.4 Summary . . . 58

5. STATE RECOVERY OF STREAM PROCESSING DISTRIBUTED SYSTEMS 60
5.1 Introduction . . . 60
5.2 System Design and Implementation . . . 63
5.2.1 Overview . . . 64
5.2.2 DHT-based Ring Overlay . . . 65
5.2.3 Fragmented Parallel State Recovery . . . 67
5.2.4 FP4S API . . . 69
5.3 Adaptivity Analysis . . . 70
5.3.1 Adaptive Parameter Tuning . . . 70
5.3.2 Analysis . . . 71
5.3.3 Instrumentation requirements . . . 73
5.4 Evaluation . . . 75
5.4.1 Setup . . . 75
5.4.2 FP4S vs Checkpointing Recovery . . . 77
5.4.3 Fragmented Parallel Recovery Algorithm . . . 79
5.4.4 Load Balance . . . 82
5.4.5 Overhead Analysis . . . 83
5.5 Summary . . . 84

6. CONCLUSION AND FUTURE WORK . . . 86
6.1 Conclusion . . . 86
6.2 Lessons Learned . . . 87
6.3 Broader Impact . . . 88
6.4 Future Work . . . 88

BIBLIOGRAPHY . . . 90

VITA . . . 102

LIST OF TABLES

TABLE PAGE

3.1 Compare With Other Monitoring Tools 38

5.1 FP4S API 70

5.2 Real-world application's dataset. 76

LIST OF FIGURES

FIGURE PAGE

1.1 Optimization stack implemented in this dissertation. Using the cloud environment and distributed applications as the foundation, we build optimization systems from different perspectives, such as applications placement, storage optimization, and failure recovery, all of which can be intermixed implemented into distributed applications. 2

3.1 A containers cluster use case. 17

3.2 Network topology of the shared uplinks with Top-of-Rack crashes, leading to a decline in the performance of many containers. 19

3.3 Example of a multi-tier RUBiS application showing correlation on server usage. The different sets of RUBiS applications have their own resources usage trend. 21

3.4 3D plot of correlation matrix between containers including iPerf containers, RUBiS containers, and Hadoop containers. 24

3.5 Integrate the DocMan component into Kubernetes. 28

3.6 Hierarchical tree generated by clustering algorithm. 30

3.7 ROC curve shows the accuracy of DocMan black box method. The overall accuracy area is 0.93, which is considered as excellent level. 32

3.8 Before and after containers mapping. 33

3.9 Throughput comparison for RUBiS before arrangement and after arrangement. It shows after arrangement the throughput increased averagely 1.93 times than before arrangement. 34

3.10 Latency comparison of Hadoop application and Spark application before arrangement and after arrangement. 35

3.11 Resource usage comparison between the actual value and the prediction value. 36

3.12 Microsoft Azure trace prediction for a CPU intensive task. (VM id: YANkWPlG) 37

3.13 The average running time of generating the prediction model and each model's mean squared error for different epoch set up. 38

3.14 Compare LSTM different epoch results with default prediction method. . . . 38

3.15 Overhead comparison between DocMan, Wireshark and tcpdump. 39

4.1 MemCached server integration with HDFS to dynamically replicate popular blocks. 45

4.2 Read count of HDFS data blocks . . . 51

4.3 After hit count is over the predefined threshold, the block is defined as Hot and MemCached records it. After a specific period, once the hit count stops increasing, MemCached will reset the records and release the occupied memory. . . . 54

4.4 Word count execution time. . . . 55

4.5 Grep execution time. . . . 56

4.6 I/O throughput for word count and grep. . . . 57

4.7 CPU usage comparison. . . . 58

4.8 Memory usage comparison. . . . 58

5.1 Contrast of stateless stream processing and stateful stream processing. . . . 61

5.2 FP4S system design. . . . 64

5.3 The routing process is cooperatively fulfilled by the routing table and the leaf set. . . . 66

5.4 The fragment-based parallel state recovery process. . . . 68

5.5 Probability of successful recovery. . . . 71

5.6 Maximum limit for HDFS rate vs Recompute rate c in FP4S. . . . 74

5.7 State recovery time for different input state sizes. . . . 77

5.8 State saving time for different input state sizes. . . . 78

5.9 Total failure recovery time by varing # concurrently running stream applications. . . . 79

5.10 Adjust raw fragment (m) parameter. . . . 80

5.11 Normal probability plot of the state size across all DHT nodes. . . . 80

5.12 Adjust parity fragment (k) parameter. . . . 81

5.13 Adjust unavailable block number (e). . . . 81

5.14 Heatmap of the state size across all DHT nodes. . . . 82

5.15 The overhead analysis of the FP4S-enabled Storm at runtime. . . . 83

CHAPTER 1

INTRODUCTION

In this chapter, we first describe the motivations behind our work. Then, we define the problems we are going to address in this thesis. Finally, we outline the road map of this dissertation, which we describe in detail in the later chapters.

1.1 Motivation

Cloud computing services cover a vast range of options now, from the basics of storage, networking, and processing power through to natural language processing and artificial intelligence as well as standard office applications. Pretty much any service that doesn't require you to be physically close to the computer hardware that you are using can now be delivered via the cloud. Public cloud is the classic cloud computing model, where users can access a large pool of computing power over the internet (whether that is IaaS, PaaS, or SaaS).

One of its significant benefits is the ability to rapidly scale a service. The cloud computing suppliers have vast amounts of computing power, which they share out between a large number of customers – the 'multi-tenant' architecture. Their huge scale means they have enough spare capacity that they can easily cope if any particular customer needs more resources, which is why it is often used for less-sensitive applications that demand a varying amount of resources. Around one-third of enterprise IT spending will be on hosting and cloud services in 2021 [Ran20], indicating a growing reliance on external sources of infrastructure, application, management, and security services.

Distributed application architecture is one of the most popular architectural patterns today. It moderates the increasing complexity of modern applications. It also makes it easier to work in a more agile manner. That's important when we consider the dominance

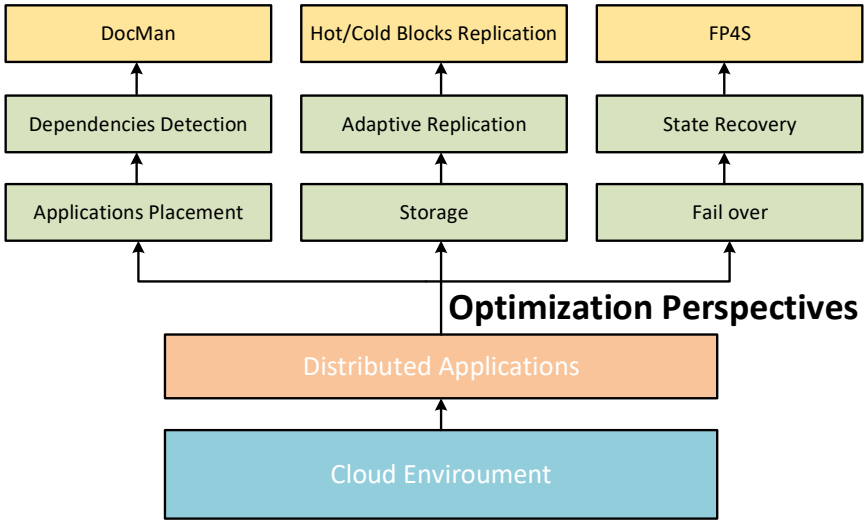


Figure 1.1: Optimization stack implemented in this dissertation. Using the cloud environment and distributed applications as the foundation, we build optimization systems from different perspectives, such as applications placement, storage optimization, and failure recovery, all of which can be intermixed implemented into distributed applications.

of data processing and other similar usages today. Sometimes it is called tiered architecture, or multiple layer architecture. A multi-layered software architecture consists of various layers, each of which corresponds to a different service or integration. Because each layer is separate, making changes to each layer is easier than having to tackle the entire architecture. The solutions to enable the distributed applications optimization in public cloud become a must-have requirement. However, former studies rarely provide an overall solution to enable the optimization. As figure 1.1 shows, we do the system optimization from three perspectives. They all cater to the distributed applications and are based on cloud environment. These optimization solutions can be implemented standalone or together as a whole solution. The system DocMan focuses on detecting the applications dependencies which the information can be used for applications placement. The system of hot/cold block replication deals with the skewed storage utilization. The system FP4S used to solve the failure recovery problem to recover states of applications.

1.1.1 Dependency Detection of Distributed Applications

To be aware of distributed applications' dependencies is the first step to do further optimization. The novelty of this part of work lies in that, instead of using an intrusive approach to detect containerized application's dependencies (e.g., via intercepting packet [BDIM04, HSG+12, MPZ10] or injecting code [CKF+02, EPPB11, TTZ+09]), our designed system, DocMan, uses a ***non-intrusive*** approach via capturing the CPU, memory and I/O logs for detecting dependencies. The rationale behind our approach is that we observed that real-world containerized multi-tier applications exhibited strong correlation among their resource usage statistics.

1.1.2 Storage Optimization of Distributed Applications

With the advancement of ever-growing online services, distributed Big Data storage i.e. Hadoop, Dryad gained much more attention than ever and the fundamental requirements like fault tolerance and data availability have become the concern for these platforms. Data replication policies in Big Data applications are shifting towards dynamic approaches based on the popularity of files. Formulation of dynamic replication factor paved the way for solving the issues generated by existing data contention in hotspots and ensuring timely data availability. But from the empirical observations, it can be deduced that the popularity of files is temporal rather than perpetual in nature and, after a certain period, content's popularity ceases most of the time which introduces the I/O bottleneck of updating replication in the disk.

To handle such temporal skewed popularity of contents, we propose a dynamic data replication toolset using the power of in-memory processing by integrating MemCached server into Hadoop for getting improved performance. We compare the proposed algorithm with the traditional infrastructure and vanilla memory algorithm, as the evidence

from the experimental results, the proposed design performs better i.e throughput and execution period.

1.1.3 States Recovery of Distributed Applications

Most distributed applications are by nature long-running. They run in highly dynamic distributed environments or clouds where many application operators may leave or fail at the same time. Most of them are stateful, in which operators need to store and maintain large-sized state in memory, resulting in expensive time and space costs to recover them. The state-of-the-art information processing systems offer failure recovery mainly through three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery, which are either slow, resource-expensive or fail to handle many simultaneous failures.

We present FP4S, a novel fragment-based parallel state recovery mechanism that can handle many simultaneous failures for a large number of concurrently running applications. The novelty of FP4S is that we organize all the application's operators into a distributed hash table (DHT) based consistent ring to associate each operator with a unique set of neighbors. Then we divide each operator's in-memory state into many fragments and periodically save them in each node's neighbors, ensuring that different sets of available fragments can reconstruct lost state in parallel. This approach makes this failure recovery mechanism extremely scalable, and allows it to tolerate many simultaneous operator failures.

1.2 Summary and Roadmap

The rest of this dissertation is organized as follows. We introduce the background details in Chapter 2, then we describe the DocMan system in Chapter 3. We next show the design

and details of storage optimization in Chapter 4. The details of design and implementation of FP4S system in Chapter 5. Finally, we conclude this dissertation and describe the future work in Chapter 6.

CHAPTER 2

BACKGROUND

2.1 Distributed Applications' Dependency Detection Works

Based on our study, there are three different categories: Injecting code to obtain runtime information, based on communication pattern between cluster service provider and tenants to detect the dependencies, and optimizing cluster applications based on network packets inference.

2.1.1 Injecting Code to Obtain Runtime Information

Code injection technologies inject some extra code into the source code of target programs to capture runtime information. For example, Fay [EPPB11] uses runtime instrumentation and distributed aggregation to get software execution traces. vPath [TTZ+09] provides path discovery by monitoring thread and network activities and reasoning about their causality. It is implemented in a virtual machine monitor, making it agnostic of the overlying middleware or application but it requires changes to the VMM code and the guest OS. Pinpoint [CKF+02] injects end-to-end request IDs to track requests. It uses clustering and statistical techniques to correlate the failures of requests to the components that caused the failures. Code injection can achieve more accurate results for detecting dependencies. However, it requires changing the program infrastructure and source code of target applications, which we aim to avoid.

2.1.2 Based on Communication Pattern

Cloudtalk [AIR17] lets users describe their tasks to the cloud and help them make appropriate choices for task placement. BtrPlace [HLM13] is used as a planning tool to limit the

number of applications or to predict the need to acquire new servers, meanwhile, provides a high-level scripting language, allowing service users to describe requirements. However, they did not consider the natural dependencies between tasks, and require users to have domain knowledge to provide accurate resources usage description. Coflow [CS12] defines a set of APIs to enable cluster applications to optimize the network usage. But it is limited to a predefined set of communication patterns and only supports network information.

2.1.3 Network Packets Inference

Meng et al. [MPZ10] propose to initialize the applications in data center based on their network traffic information. They design a two-tier approximate algorithm to find the traffic patterns and adjust the network architectures to improve the cluster performance. Magpie [BDIM04] captures events from OS kernel, middleware, and application components and calculates the time correlation of these events. DocMan not only focuses on network metric but also focuses on other metrics, which enable more considerable factors for the provider to optimize cluster performance.

2.2 Distributed Applications' Storage Optimization Works

Distributed applications gained popularity for its high performance and reliability but the default replication management, such as in HDFS, is not sufficient to handle the congestion that arises from a high density of user requests. Especially the adaptability of data popularity needs to be handled with a time-based dynamic replication policy for improving the system performance. As a result, adaptive replica management model is needed to ensure improved performance along with fault tolerance and studied from various perspectives.

2.2.1 Replica Allocation and Management Based Systems

Data locality is an important factor to improve the data availability in time. To reach better data locality an algorithm DARE (Distributed Adaptive Data Replication) [ALC11] is proposed where probabilistic sampling and competitive aging algorithm is used in each node to produce solutions of replica allocation and replica placement. Comparing to DARE, the system we propose, that leverages the memory's higher I/O speed to obtain a better replica allocation and placement result. Besides, Dare is adaptive in workload changes. The experimental studies based on two mixed workload traces from Facebook show that the data locality improves with different schedulers in Hadoop.

CDRM (Cost-Effective Dynamic Replication Management Scheme) [W$^+$10] is another model that can calculate and maintain a minimal number of replica for a given availability requirement. It can adapt to the changes of environment in terms of data node failure and workload changes and maintains a rational number of replica. However, to use CDRM model, it lacks an uniform algorithm to deal with replica number and locality. To use the model we present, it is able to dynamically and uniformly adjust replica locations. It replaces replica based on capacity and blocking probability of data nodes and dynamically redistributes workloads by the adjustment of replica number and their placement location. CDRM is cost effective and performs better in comparison with default replication management of HDFS.

An offline replica allocation algorithm named MORM (Multi-objective Optimized Replication Management) [LZC14] was proposed which looks for near optimal solutions by balancing different optimization factors. However, it only based on existing DFS design to balance the trade-offs. The system we propose, not only consider to optimize the replica, but also add memory component to optimize the overall performance.

EDAS (Efficient Data Access Scheme) [AO15] is an algorithm for HDFS data replication where user can get quick access of replica data by the decision of access node

according to the load of the node. The idea of EDAS is actually based on the historical data access record from HDFS metadata and anti-blocking probability selection method. These systems are based on different strategies for replica allocation and management but there is no use of popularity concept. In the system, popularity is an important concern and on top of that we cache more popular files in MemCached which should allow us to increase data availability.

2.2.2 Skewed Popularity Based Systems

ERMS (Elastic Replication Management System) [C$^+$12] is an active storage model for HDFS. It dynamically increases the number of hot files and reduces the number of cold files by tracking real time data type. The work is based on probability or constant data type but we use real time data popularity in Hot/Cold block detection mechanism. ERMS is also feasible to manage extra replicas of hot data by using a replica placement strategy. Based on their experiments it is clearly shown that ERMS successfully improves the performance of HDFS and reduce storage overhead.

Predictive analysis is another approach to dynamically replicate the data file where the utilization of each data can be predicted by probability theory. Then using this utilization information, popular files can be replicated and non-popular files can be deleted. This process improves the availability of data files compared to the default scheme [B$^+$15].

HDFS-DRM [LMC15] devised a design to solve the hot issues in HDFS based on cloud storage where it makes use of dynamic adjustment mechanism and deletes duplicate node selection mechanism. It increases or decreases the number of replicas by tracking the record of reading requests from users of each data file in a certain amount of time. However, the optimization is in the disk storage level, the disk I/O bandwidth will finally become the bottleneck for the overall application performance.

Scarlett [A+11], a replication system for MapReduce, was proposed where data replication is done based on their popularity to avoid contentions and improve data locality. Scarlett decreases rack network traffic of jobs by increasing the replication factor of a file according to its popularity information. They also represent a few experimental statistics of data popularity based on job logs analysis from a large production cluster supported by Microsoft's Bing. After using Scarlett in two MapReduce frameworks, Hadoop and Dryad, results showed speed up in job execution effectively using limited extra storages and network resources.

The system is also based on time variant skewed popularity where we detect Hot and Cold blocks by trace data analysis. Most of the prior works are based on probability or constant data type but we use real time data popularity in Hot/Cold block detection mechanism. Definitely real time data set is more reliable as the popularity changes over time.

2.2.3 Caching Based Systems

In order to reduce I/O bottlenecks of HDFS, MEM-HDFS [I+14] was demonstrated using MemCached as a caching system. The main focus is to provide intelligent caching and HDFS data blocks replication with consideration of different deployment strategies for the local and remote MemCached servers. MEM-HDFS implementation resulted in increased throughput and reduced job generation time.

HDFS native cache system, Centralized Cache, which is an explicit caching mechanism that allows users to specify paths to be cached by HDFS. The NameNode will communicate with DataNodes that have the desired blocks on disk, and instruct them to cache the blocks in off-heap caches [cen].

In MEM-HDFS or Centralized Cache, they make use of only caching concept while in the system we merge caching and popularity concept together. After detecting the Hot blocks we cache them in MemCached that gives more flexibility in reducing data contention.

2.3 Distributed Applications' States Recovery Works

Designing a state recovery mechanism for distributed stateful processing systems in cloud is non-trivial, and existing failure recovery techniques for data processing do not achieve the necessary scalability and efficiency. We summarize existing stateful stream processing systems as follows.

2.3.1 Stateful date Processing Systems

Many industrial date processing systems either do not support state (Heron [KBF+15], S4 [NRNK10], early version of Storm [sto]), or rely on in-memory data structures such as hash tables and hash table variants to store state. For example, Muppet [LLP+12] and Trident [ai] (an extension of Storm) store state via hash tables. Spark Streaming [spa] enables state computation via RDDs [ZCD+12] which are inherent hashmaps. Some other systems such as Millwheel [ABB+13], and Dataflow [ABC+15] choose to separate state from the application logic and have state centralized in a remote storage [ABB+16, CCD+03, ACÇ+03] (e.g., a database management system, HDFS or GFS) shared among applications, along with periodically checkpointing state for fault tolerance. A few other systems such as Kafka [ae], Samza [af, NPP+17], Spark Streaming [spa], and Flink [aa, CEF+17] use a combination of "soft state" stored in in-memory data structures along with "hard state" persisted in on-disk data store (e.g., RocksDB [aq], LevelDB [am]).

Scaling to large distributed states and recovering from failures in such systems is quite expensive, because when a single node fails, the distributed states for all dependent nodes must be reset to the last checkpoint, and computation must resume from that point, costing a lot of extra time and space to accomplish recovery. Moreover, these systems rely on a single master for handling failures and stragglers, exhibiting significant overhead from centralization bottlenecks.

2.3.2 Failure Recovery in Data Processing Systems

Existing stream processing systems offer failure recovery mainly through the use of three approaches: replication recovery, checkpointing recovery, and DStream-based lineage recovery.

Replication recovery. In the process of replication recovery, there is a completely separate set of hot failover nodes, which processes the same stream in parallel with the primary set of nodes. The input records are sent to both. When there is a failure or multiple failures in the primary nodes, the system automatically switches over to the secondary set of nodes and the system can continue processing with very little or no disruption. The replication recovery has been widely used in systems such as Flux [SHB04] and Borealis [BBMS05]. The failover is fast, and it can handle multiple concurrent failures. However, the replication recovery costs twice the hardware.

Checkpointing recovery. In the process of checkpointing recovery, each of the nodes in the pipeline has a buffer in memory, which retains a backup of the records that it has forwarded to the downstream nodes since the last checkpoint. All nodes periodically checkpoint their states to a remote storage such as HDFS or GFS. A standby set of nodes is maintained in the system. If any of the primary node fails, the standby node will retrieve the latest checkpoint from the persistent storage, and its upstream node essentially

replays the backup records serially to this failover node to recreate the lost state. The checkpointing recovery has been widely used in systems such as TimeStream [QHS+13], Trident [ai], Drizzle [VPO+17] and Multi-level Checkpointing from LLNL [MBMDS10]. It avoids the 2× hardware cost. However, the failover is much slower than the replication recovery because it has to retrieve the checkpointed state from the disk and replay the buffered data on the last state to recompute the new state. Drizzle [VPO+17] introduced group scheduling and pre-scheduling to avoid the centralized scheduling bottleneck. However, it used batch processing model and focus on scheduling tasks for one application while FP4S uses a record-at-a-time processing model and focus on many concurrently running jobs.

DStream-based lineage recovery. To achieve both fast recovery and small hardware overhead, the DStream-based lineage recovery was proposed, which is used in systems such as Apache Spark based systems [aa, CEF+17, ZDL+13, SGH+16]. The most recent state is stored in each node's memory using a data structure called Resilient Distributed Dataset (RDD) [ZCD+12], together with the lineage graph, that is, the graph of deterministic operators used to build RDDs. When nodes fail in the system, instead of preparing nodes for failover, DStream will re-run the lost tasks in parallel on other reliable nodes in the cluster using the lineage graph. Then these tasks can be parallelized to recompute the lost states. However, the entire recovery processing is linear, that is, the lost tasks need to be executed or computed strictly in line with the original lineage graph on other nodes. As such, the recovery process may be slow when the lineage graph is long and incur multiple data uploads through the network, consuming a critical resource in geo-distributed network settings.

To our best knowledge, the very few research projects that are broadly relevant to state management solutions are [TAB13, HLL16, an, ad], which either point out the criticality of making state explicit [TAB13, HLL16] or develop mechanisms for reprocessing

state [an, ad], but propose no effective solutions for fast state recovery for concurrently running stream applications.

CHAPTER 3

CORRELATION DETECTION OF DISTRIBUTED SYSTEMS

Nowadays, there has been a quick growth in the popularity of Container-as-a-Service (CaaS) clouds. CaaS is a new form of container-based virtualization in which container engines, orchestration and the underlying server resources are served to the users as a service from a cloud provider. In CaaS cloud, the customers can manage containers using the service provider‘s routines or web portal interface and pay for the required CaaS resources, such as hardware capacities, load balancing, and security level. The purchase of CaaS resources proceeds as follows: (1) the provider provides to the customers with a variety of pre-configured, templated container images containing various operating systems and pre-installed libraries and software; (2) the customer estimates her application needs, selects container images and resource needed for her application such as the desired vCPU and memory capacities and outbound network bandwidth; also specifies the operating systems to use and virtual machine instances (the norm is running containers inside the VMs obtained from the cloud provider); (2) the customer finalizes the contracts with the cloud provider for some period of use, e.g., a one-month term; and (3) the provider launches the containers and manages them.

With the interest in CaaS clouds skyrocketing among customers, cloud providers are capitalizing on the opportunity through hosting container management services. For example, Microsoft rolled out its Azure Container Service (ACS) [micb]. Google developed the Kubernetes container orchestration tool for Google Container Engine [goo]. Amazon launched EC2 Container Service (ECS) [Ama]. The list is not long, but they attract more and more customers. According to the RightScale 2018 report [0], Microsoft Azure Container Service has reached adoption of 20% (with a strong growth of 82%); Google Container Engine has reached adoption of 14% (with a strong growth of 75%); and Amazon container service (ECS/EKS) has reached adoption of 44% (with 26% growth rate).

Unfortunately, while the progress that towards container management services in CaaS clouds is remarkable, the performance optimization of the containerized applications has been largely neglected. Left unchecked, it will pose a threat to customer experiences of CaaS clouds and cause customers to get much less than what they paid for, which eventually waste significant computing resources and power for cloud providers.

Many applications, such as RUBiS [rub], Hadoop MapReduce [map], Storm [sto] are distributed applications in which two or more components cooperate and communicate to jointly provide a certain service or accomplish a job. A typical distributed application, for instance, contains a presentation tier for basic user interface and application access services, an application processing tier for processing the core business or application logic, and a data access tier for accessing data, and finally a data tier for holding and managing data. Containers are similar to virtual machines (VMs), but much more lightweight, i.e., less resource and time-consuming. By comparison, application containerization allows customers to launch individual applications without the need to rent an entire VM. It does this by "virtualizing" an operating system and giving containers access to a single operating system kernel, each container comprising the files, setting, dependencies and libraries required for the application to run on an OS. Most of the containerized applications that run in CaaS clouds are the distributed applications, also called multi-tier applications, which require a set of containers, a container ensemble, that cooperate and communicate to jointly provide a certain service or accomplish a task.

The containerized applications that run on public CaaS clouds could have been optimized by combining the application's inter-container dependency knowledge with infrastructure's server load, network topology or links knowledge to choose the optimized endpoint physical servers to place containers [CKS13, CZM+11, CZS14, HKZ+11, PSLJ11]. For example, when the network I/O dependencies are known, the containers that have the strong network I/O dependencies can be placed geographically close to each other, e.g.,

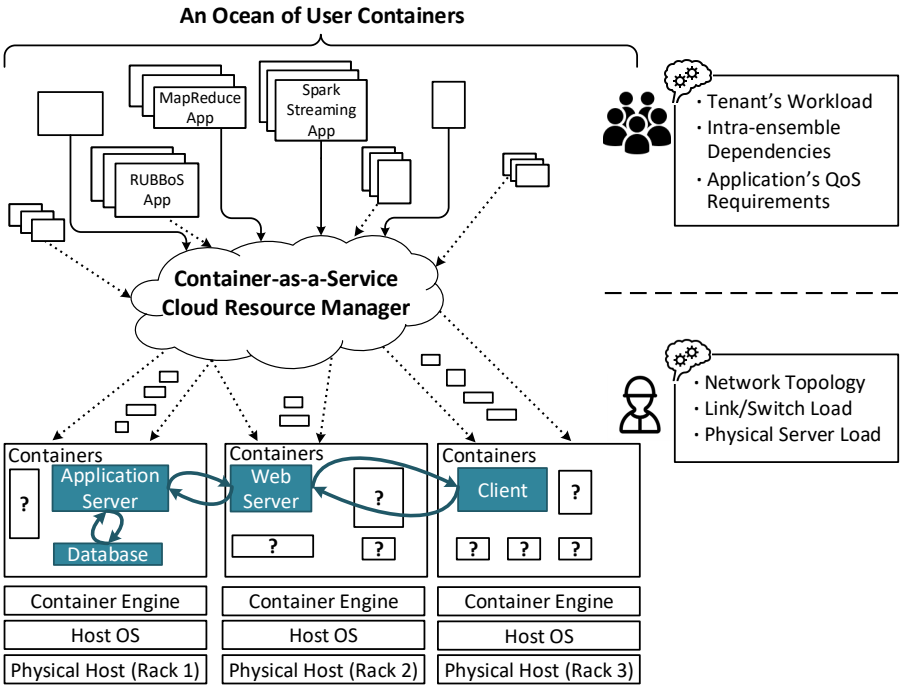


Figure 3.1: A containers cluster use case.

within the same rack or server. In that case, their communicating messages do not need to traverse long bottlenecked network paths such as Top-of-Rack switches, L2 switches and etc, significantly reducing the latency for containerized applications. It is also a win-win for the provider. The oversubscribed uplink bandwidth will be saved and link/switch load will be reduced accordingly, the provider can save significant power in the data center network (DSN) [AFRR+10, FPR+10, MYAFM10] by aggregating the traffic into fewer network devices and links and putting the others into sleep. Another example is that, when the computational dependencies are known, the containers that have strong computational dependencies can be placed slightly remote to each other, e.g., within the same rack but on different servers. In that case, the resource interferences of co-located containers can be reduced, which will be particularly beneficial to web applications by addressing the long tail latency problem and big data processing applications by improving their throughputs.

The above optimizations and others have been designed for, and used in, private clouds where the private cloud providers run both the applications and the infrastructure and have full control over them. Unfortunately, these optimizations are almost impossible to implement in public CaaS clouds today due to the "double-blind" opacity between the customers (tenants) and the cloud providers. Thus, public cloud providers are understandably reluctant to share with customers about the underlying network and storage topology and its current load on links and physical servers. Also, customers are reluctant to share with providers about the running workload, containerized application's intra-ensemble dependencies, or the containerized application's QoS requirements.

Figure 3.1 demonstrates the scenario of a real-life use case. The cloud provider uses the container orchestration tool of Kubernetes [kub] for the containerized applications deployment and management. It will ask customers to declare minimum and maximum server resources for their containers, and then slot their containers into where ever the server they fit. Usually, it uses automatic resource binpack algorithms, and without the consideration of the dependencies among containers. In such a case, catastrophic consequences can occur since the cross rack requests. Such as client's posts arrive at the front end container running the web server, then forward to the application server container, which in turn query data from the database container. While container resource requirements are met, the containers collaboration process will use more network resources than placement with dependency considerations.

In this case, catastrophic consequences may happen, such as the client requests arrive at the container running the web server front end and are then forwarded to one of the containers running application servers, which in turn may request data from a backend container hosting a database. Although the containers resource requirements are fulfilled, the containers cooperation process will use far more network resources than a placement with the dependencies consideration.

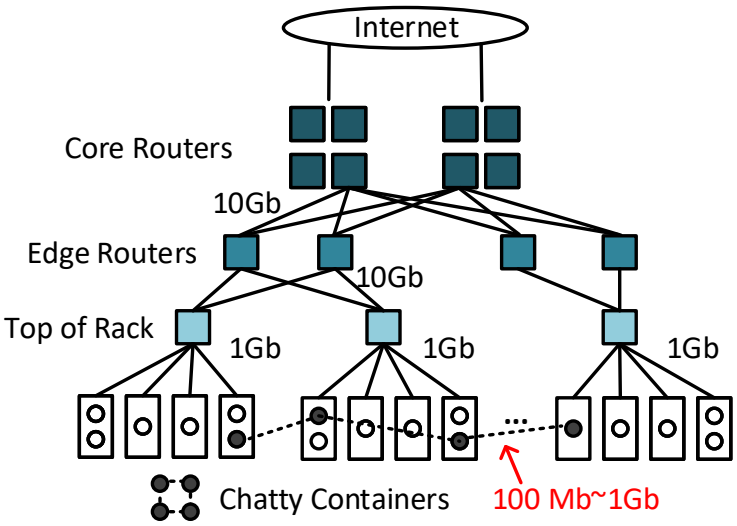


Figure 3.2: Network topology of the shared uplinks with Top-of-Rack crashes, leading to a decline in the performance of many containers.

Figure 3.2 shows the network topology of Amazon EC2' useast-1d datacenter inferred by CloudTalk[AIR17]. Rack uplink bandwidth is typically oversubscribed, which leads to significant rack-level congestion [FPR+10, AIR17]. In the public cloud platforms such as AWS, the norm is running containers inside VMs obtained from the cloud providers. These VMs are initially launched with only the OS and no application-specific software. Because VMs within the different racks need to share the oversubscribed uplinks, this will cause a worst case of end-to-end available bandwidth for containers as low as 100 Mbps (the value ranges from 100 Mbps to 1 Gbps as measured by [GHJ+09].

In order to effectively address the aforementioned issues, we present DocMan, a novel container toolset that is designed to optimize resources usage in CaaS clouds. Be specifically, DocMan recognizes the potential container ensembles by using resource utilization matrix. It assesses the "chattiness" degree among the containers, then enables optimized container placement strategy to alleviate the bi-section bandwidth saturated condition. Last but not least, it uses deep learning technology to dynamically provision the running containers' resources.

DocMan has the following unique properties:

- ***Transparency and privacy-preserving*** — as a non-intrusive tool, DocMan does not require any code modification to the containers or applications.
- ***Lightweight*** — unlike the network sniffer tools that bring heavy burdens to the management software, DocMan has negligible CPU and memory overheads.
- ***Actionable*** — insights derived from running DocMan can help management software better co-locate container ensembles on the underlying hosts and provision more reasonable computation resources.

Discovering container ensembles and their inter-container dependencies are quite challenging. A naive method that continuously gathers statistics about all communicating containers is prohibitively expensive. First, it would require introspection of all packets sent and received by the containers; this would induce notable CPU overheads and additional per packet latency of the tens of microseconds. Second, additional memory resources would be required to maintain statistics for every pair of IP addresses.

DocMan uses a three-step approach. Firstly, it acquires container-level statistics commonly available in container management systems (e.g., Kubernetes), such as the total numbers of packet in/out over time. Secondly, it computes the correlation coefficients among these statistics. Finally, it divides the corresponding containers into subsets (also called ensembles) using correlation values and a hierarchical clustering algorithm [War63].

DocMan has been designed and implemented. We evaluated its effectiveness on a testbed that has 24 physical machines deploying with 144 containers. These containers perform a variety of batch workloads, Internet services, business tasks, and streaming tasks. Experimental results demonstrate that DocMan is able to recognize container instances with 92.0% accuracy as well as predict the resource usage pattern with low error

Figure 3.3: Example of a multi-tier RUBiS application showing correlation on server usage. The different sets of RUBiS applications have their own resources usage trend.

rate (root mean square error <0.3) during the limited epochs (<200). The applications performance is improved in terms of throughput and latency by using the proposed placement strategy. Specifically, we observed that the application throughput of the RUBiS instance increased by 1.93x, the application throughput of the Hadoop MapReduce instance increased by 33.4%, and the Spark Streaming instance improved by 7.9%. For the intensive CPU Microsoft Azure trace, it can get potentially 14.9% resource saving.

3.1 DocMan System Design

This section illustrates DocMan's design, which includes (1) monitoring of basic resource usage statistics of containers; (2) computing the correlation coefficients among these statistics to identify their distances; and (3) using hierarchical clustering algorithms to detect potential container ensembles.

3.1.1 Data Collection

The first step is to capture the following universally available system-level metrics about each container:

CPU: CPU usage per second in percentage terms (%).

Memory: Memory usage per second (MB).

I/O: Packets transmitted per second (KByte/sec).

These periodic measurements result in three time series signals per container. Before explaining the actual analysis being applied, we illustrate the utility of taking these measurements with a simple example.

Multi-tier applications (e.g., RUBiS [rub]) typically use a request-response architecture, in which a client container sends a request to the front end (e.g., Apache), which assigns the work to an appropriate server (e.g., Tomcat) running the application logic. The application logic services the request by querying the backend (e.g., a database server like MySQL) to produce the necessary output, and sending the response back to the client. Therefore, given the nature of multi-tier applications, we can expect correlations between the "*CPU*", "*Memory*" and "I/O" statistics for interacting containers. For example, an instant rise of CPU usage or packet flow rate in one container directly or indirectly triggers activities in other containers, thereby creating the correlations among these statistics.

To more clearly show the correlations among these statistics, we did an experiment on two RUBiS instances with 2000 clients and 500 clients workloads, respectively. Each instance has three containers, i.e., client, Tomcat server and MySql database. The container-level metrics of CPU, memory and I/O for these containers are continuously collected and compared in Figure 3.3, where the system load is the normalized value of CPU, memory plus I/O with the same weight. The results show that the containers that collaborate together to accomplish the same task within the same instance tend to have similar trends of system load, while the containers that are responsible for different tasks do not exhibit correlations. The same logic can be applied to other applications as well, such as big data analytics application like Hadoop MapReduce applications [map], Storm applications [sto], Spark Streaming applications [spa], which are popular in CaaS clouds. For

example, MapReduce adopts a Partition/Aggregate pattern which scale out by partitioning tasks into many sub-tasks and assigning them to worker containers (possibly at multiple layers). These worker containers are expected to work together to accomplish the task.

There are several design tradeoffs about data collection:

- *Metrics selection.* Although "I/O" (PacketIn and PacketOut) is a more intuitive metric to do correlation analysis among communicating containers, we also include "CPU" and "memory" as complementary metrics because they will provide extra information in certain scenarios. For example, if a bunch of containers do many-to-many but infrequent data exchanges, it is difficult to detect them using only I/O metric, especially when the interval between data exchanges is larger than the sampling window. In such cases, other metrics can capture the missing signals.
- *Sampling window.* Larger sampling window does not always means more accurate result. For example, if the large window accidentally covers the idle period when the container ensemble has no inter-communications. The final result may not be as accurate as smaller windows. On the other hand, if the sampling window is too small, it may fail to catch up the useful information timely. Therefore, we need to find the appropriate sampling window for each representative application.

3.1.2 Distance Identification

Each container's log can be organized as a vector of an array (α, β, γ). α is the *CPU* usage record; β is the *Memory* usage record; and γ is the I/O usage record, respectively. Then, the all containers generate a vector matrix for the cloud service provider to analyze. We choose the Pearson product-moment correlation coefficient (PMCC) [PMC] to measure the degree of correlation, giving a value between -1 and +1 inclusive.

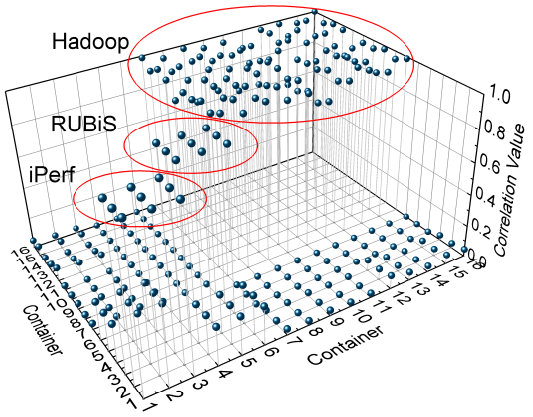


Figure 3.4: 3D plot of correlation matrix between containers including iPerf containers, RUBiS containers, and Hadoop containers.

The mathematical logic is as below: there are two vectors X and Y to demonstrate any two vectors which are generated by the containers. Each element is an array of resources usage record in the vector. $X = X_1, X_2, \ldots, X_n$, $Y = Y_1, Y_2, \ldots, Y_n$, then the correlation between X and Y is:

$$corr = \frac{\sum_{i=1}^{n} \left(\begin{pmatrix} X_{\alpha_i} \\ X_{\beta_i} \\ X_{\gamma_i} \end{pmatrix} - \begin{pmatrix} X'_{\alpha_i} \\ X'_{\beta_i} \\ X'_{\gamma_i} \end{pmatrix} \right) \left(\begin{pmatrix} Y_{\alpha_i} \\ Y_{\beta_i} \\ Y_{\gamma_i} \end{pmatrix} - \begin{pmatrix} Y'_{\alpha_i} \\ Y'_{\beta_i} \\ Y'_{\gamma_i} \end{pmatrix} \right)}{\sqrt{\sum_{i=1}^{n} \left(\begin{pmatrix} X_{\alpha_i} \\ X_{\beta_i} \\ X_{\gamma_i} \end{pmatrix} - \begin{pmatrix} X'_{\alpha_i} \\ X'_{\beta_i} \\ X'_{\gamma_i} \end{pmatrix} \right)^2} \sqrt{\sum_{i=1}^{n} \left(\begin{pmatrix} Y_{\alpha_i} \\ Y_{\beta_i} \\ Y_{\gamma_i} \end{pmatrix} - \begin{pmatrix} Y'_{\alpha_i} \\ Y'_{\beta_i} \\ Y'_{\gamma_i} \end{pmatrix} \right)^2}} \tag{3.1}$$

The calculation result *corr* is between -1 and 1. If it is 0, it means they have no relationship. Near -1 means that they are negatively related with each other and near 1 means that they are positively related with each other. Since the negative relationship is meaningless in practice, we update all the negative numbers to 0 in the output matrix.

Figure 3.4 shows the result that is performed by Python package numpy[num] calculation. It illustrates that the containers are performing same tasks, which are having a higher value near to 1. Such as the value between container 0, 1, and 2, the respective

values are 1.00, 0.99, and 0.99. In the 3D chart, the values are at the top part. However, comparing to other containers, they have a lower value which is near to zero, that means they do not have a direct relationship. Such as 1 and 15, the value is only 0.03. For 2 and 14, the value is only 0.04. These values are at bottom part in the chart.

We further define the concept of distance to describe the strength of dependencies between two containers:

$$Distance(X_i, Y_i) = \begin{cases} \frac{1}{corr_i}, & corr_i > 0 \\ \infty, & corr_i = 0 \end{cases} \tag{3.2}$$

The distance value matrix will be used as input information for the next step hierarchical clustering.

3.1.3 Hierarchical Clustering

Clustering[clu] is the process of making a group of abstract objects into classes of similar objects. A cluster of data objects can be treated as one group. Two commonly used clustering algorithms are hierarchical clustering and *k*-means clustering. DocMan uses hierarchical clustering for the following reasons:

- Hierarchical clustering does not require the number of clusters in advance.
- It works well with both globular and non-globular clusters, while *k*-means fails to handle non-globular data.
- *k*-means clustering is sensitive to initial centroids. If the user does not have adequate knowledge about the data set, this may lead to the erroneous results.

The process of hierarchical clustering is as follows:

- Step 1: initially assign each container to a cluster, so that there are *N* initial clusters for *N* containers.

- Step 2: find the closest (most similar) pair of clusters and merge them into a single cluster.
- Step 3: compute distances (similarities) between the new cluster and each of the old clusters.
- Step 4: repeat Step 2 and Step 3 until all items are clustered into a single cluster of size N.

Concerning Step 4, of course, there is no point in having all N items grouped into a single cluster, but doing so results in the construction of the complete hierarchical tree, which can be used to obtain k clusters by just cutting its $k-1$ longest links. K can be based on the number of racks in the datacenter, or it can be chosen to make the inter-cluster distance less than a certain threshold.

3.1.4 Resource Usage Pattern Prediction

After the ensembles of the correlated containers are identified, the system uses deep learning techniques to predict their resource usage patterns. The fundamental behind our approach lies in that, many cloud jobs are recurrent. It has been demonstrated in related literature that over 60% of the jobs in real-world large enterprise clusters are recurrent [**?**][**?**].

Inspired by the observation that runtime metrics in a resource usage log are a sequence of events produced by the execution of a highly possible recurrent job (and hence can be viewed as a structured language), we use a long short-term memory (LSTM) recurrent neuron network for the pattern prediction over resource usage traces.

A Recurrent Neural Network (RNN) is an artificial neural network that uses a loop to forward the output of the last state to current input, thus keeping track of history for making predictions. However, RNN can only have a short memory of a few terms. To overcome this limitation, the long short-term memory (LSTM) networks are proposed

which can remember long-term dependencies over sequences. LSTMs have demonstrated success in various tasks such as machine translation [SVL14], sentiment analysis [DL15], and medical self-diagnosis [LSD+16]. If the prediction results underestimates the actual resource requirement, the application might suffer performance issue. So a provision margin concept is introduced, it adds a buffer zone to the predicted value, which will help to guarantee the required resource utilization meanwhile save the resource comparing to hard-coded over-provisioning.

Under-estimating the number of future requests results in extra delays when allocating cloud resources to clients due to the need for waking up machines upon arrival of any unpredicted request(s). In order to reduce the occurrences of such cases, a safety margin can be added to the number of predicted requests to accommodate such variations. The cost of this safety margin is that some PMs will need to be kept idle even though they may or may not be needed. We propose to use a dynamic approach for selecting the appropriate safety margin value, where the value depends on the accuracy of predictors: it increases when the predictions deviate much from the actual number of requests and decreases otherwise.

3.1.5 DocMan Integrates with Kubernetes

DocMan can help container management software (such as Kubernetes) better co-locate container ensembles on the underlying hosts. DocMan collects logs from every physical node by adding one log collector component per node. It doesn't require any changes to the applications running on the node.

Figure 3.5 shows the infrastructure of integrating DocMan into Kubernetes. The DocMan Log Collector streams application logs to its own standard out. The DocMan logging agent is a separate node that in charge of running the algorithm to cluster the containers.

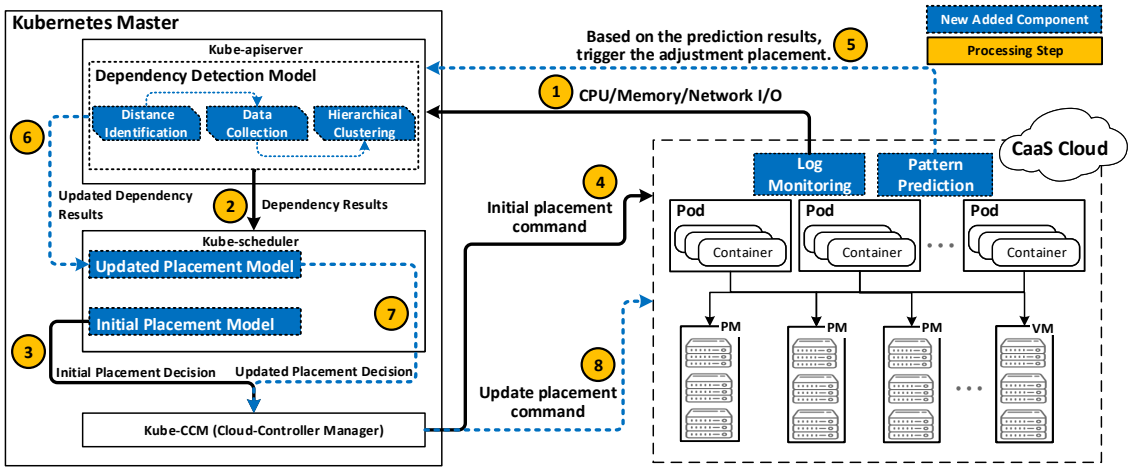


Figure 3.5: Integrate the DocMan component into Kubernetes.

The logic behind redirecting logs is minimal, so it's hardly a significant overhead. The standard output can be handled by existing component of Kubelet, that gives the feedback about containers dependency to the service manager.

3.1.6 Rationale

Lightweight property. Containers are regarded as a lightweight replacement of virtual machines (VMs). Because of the negligible runtime overhead, containers have a much higher deployment density per physical host than VMs. Although VMs are still widely used in the Infrastructure as a Service (IaaS) space, we see Linux containers dominating the Platform as a Service (PaaS) landscape. The DocMan design keeps the lightweight property by using a black-box approach with minimized runtime overhead.

Micro-services architecture. The need for management and orchestration systems capable of scheduling, deploying, updating and scaling the containerized applications is crucial. This is especially true with the rise of the micro-services architecture [mica], where several software components composing an application. The DocMan's design

tries to explore and clarify the dependencies within these components, acting as an important tool for micro-services architecture.

The container clustering solutions. To the best of our knowledge, DocMan is the first tool to detect the dependencies within containerized applications. DocMan leverages the management utilities' logs, e.g., from Kubernetes, as input to mine useful information.

3.1.7 Testbed

The experiments are performed on 24 servers with dual-socket and dual-core. The 24 servers are equally distributed across 4 edge switches. The switches are connected with each other, with an over-subscription ratio of at most 4:1. The servers run Docker version 17.06 on Ubuntu 16.04. The machine learning platform is TensorFlow 2.0, which is a library for numerical computations using data flow graphs. It supports a wide spectrum of state-of-the-art deep learning approaches, which includes a class of RNN that has found practical applications is Long Short-Term Memory (LSTM) .

Experiments use 3 instances of RUBiS (9 containers), 6 instances of Hadoop MapReduce (60 containers), 3 instances of Sparks Streaming (30 containers), 3 instances of iPerf (9 containers), 3 instances of Redis (9 containers), and 3 instances of Storm (27 containers) resulting in a total of 144 containers cluster running a variety of business, batch workloads and internet services.

3.1.8 Workload and Metrics

Apache Hadoop MapReduce(v2.7.1) [had] is a programming model for process vast amounts of data in parallel on large clusters. We use 6 instances of 10 Hadoop MapReduce containers, one of them is master, the rest are slaves. The MapReduce application `Pentomino`

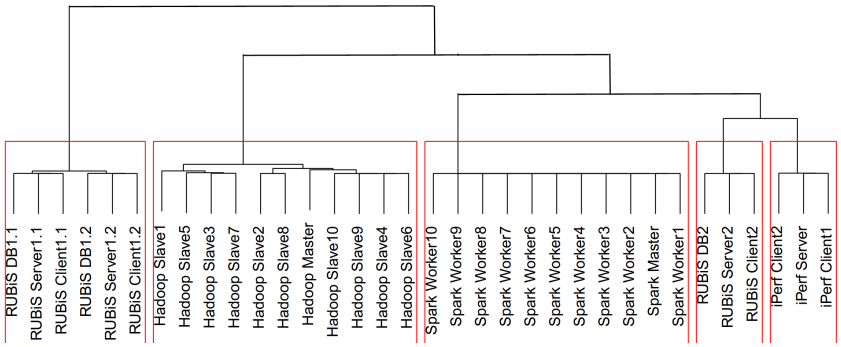


Figure 3.6: Hierarchical tree generated by clustering algorithm.

is deployed to Hadoop for LSTM model testing. The applications of `Word Count` and `Secondary Sort` are deployed into Hadoop instances for performance testing purpose.

Apache Spark Streaming(v1.4) [spa] is a big data analytics system. We use three instances of 10 Spark Streaming containers for master nodes and worker nodes. The master node starts the master process and the built-in standalone cluster. Each worker is responsible for launching the executor process. The Spark machine learning application `Word2Vec` with 10GB input data and the MapReduce applications of `Word Count`, `Secondary Sort` are deployed into Spark instances for performance testing purpose.

iPerf3 [ipe] is a tool for network performance measurement and adjustment. It is an important cross-platform tool that generates standardized performance measurements for any network. We use three instances of iPerf containers, where every instance having one server and two clients.

RUBiS(v1.4.2) [rub] is an auction site benchmark modeled after eBay.com. A PHP version of RUBiS is used, which has an Apache web server front end and a Tomcat application server connecting with a MySQL backend database. The benchmark is produced by adopting a web simulation client. The client performs 2000 rounds basic website activities such as register a new account, sell the item, browse item, view homepage, bit the item and so on. All containers within each instance are interactive and will have the strong network I/O dependencies.

Apache Storm(v1.1.3) [sto] is a distributed real-time computation system. It makes easy to process unbounded streams of data. We set three instances of 9 Storm containers, three of them are ZooKeepers, one of the containers is Nimbus, and the rest of them are Supervisors. The tasks of `Reach`, `RollingTopWords` and `Topologies of WordCount` are deployed onto the instances respectively.

Redis(v3.2) [red] is an open source data store structure implementing a distributed, in-memory key-value database with optional durability. We use three instances of 3 Redis containers, which one of them is the server and two of them are clients.

3.1.9 DocMan's Functionality Evaluation

We evaluate DocMan's functionality from following four perspectives:

- The accuracy rate for recognizing the correct set of the container ensembles.
- The benefits of the findings for improving container applications performance through the comparison of the throughput and latency between initial containers placement and final containers placement.
- The potential resource saving by using the deep learning prediction.
- The runtime overhead induced by the DocMan toolset.

DocMan Dependency Detection Accuracy Rate

Figure 3.6 shows the hierarchical trees constructed by using the decreasing dependencies strength. We firstly determine a $N*N$ correlation strength matrix, and then run the hierarchical clustering algorithm over the distance matrix. It shows the calculated dependencies between the containers being observed, which are generated by R [r]. It also demonstrates that DocMan can effectively recognize most of the underlying containers dependencies in

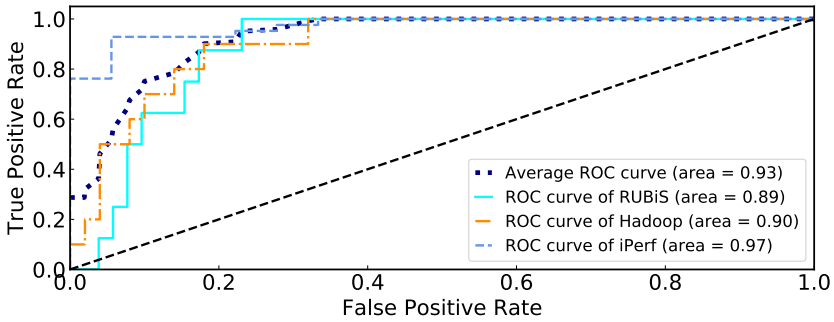


Figure 3.7: ROC curve shows the accuracy of DocMan black box method. The overall accuracy area is 0.93, which is considered as excellent level.

the testbed. Meanwhile, DocMan finding container ensembles process does not require any code injecting or modification to the containers or the applications to obtain these results.

Figure 3.7 shows the ROC curve results of the classifiers, which is a graphical plot that illustrates a binary classifier system's diagnostic ability as its discrimination threshold is varied. In our cases, it is more complicated than the binary classifier since multiple applications can be co-existing in the same cluster. We depict three classifiers accuracy results of RUBiS, Hadoop MapReduce and iPerf, and the overall average accuracy line, which is calculated based on the separate accuracy results. The average accuracy is 92.0% and the overall underline average area of ROC is 0.93, which 1 means a perfect result. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test. The result shown in the Figure 3.7 is considered as an excellent one.

DocMan's Potential Benefits

After determining the container ensembles, we next evaluate the benefits of using this obtained information as inputs for improving container placement on the hosts.

Figure 3.8 shows the mapping of applications to host and rack before and after the arrangement driven by DocMan's insights. We set the bandwidth between the different

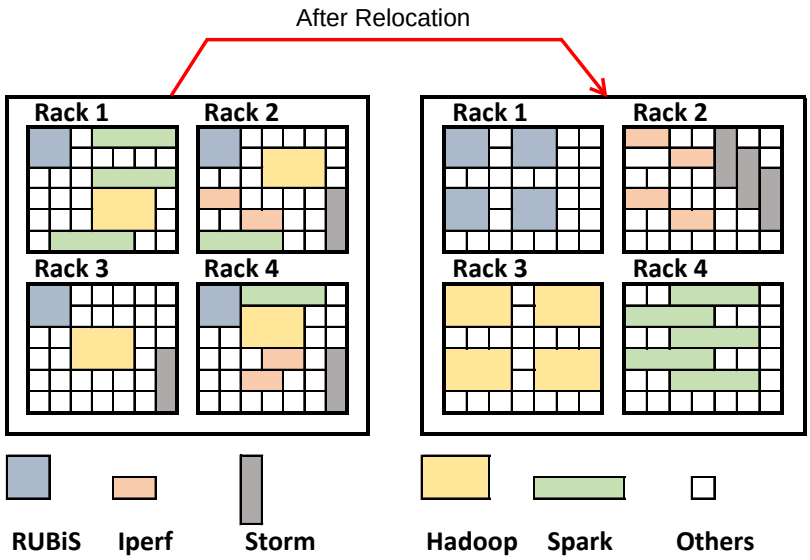
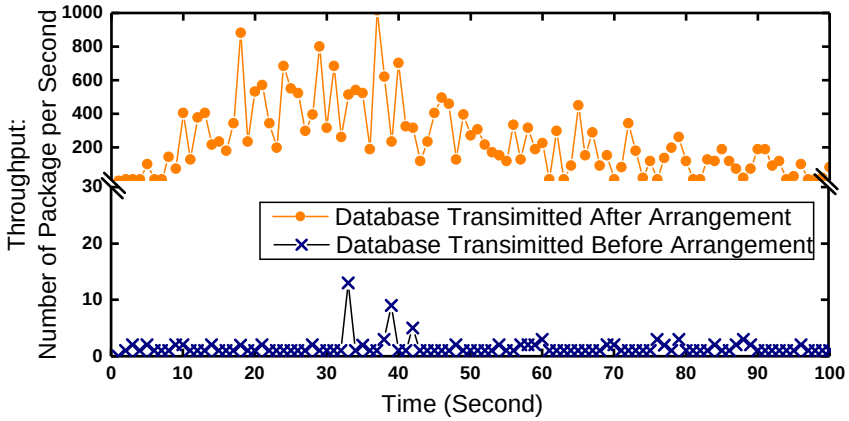


Figure 3.8: Before and after containers mapping.

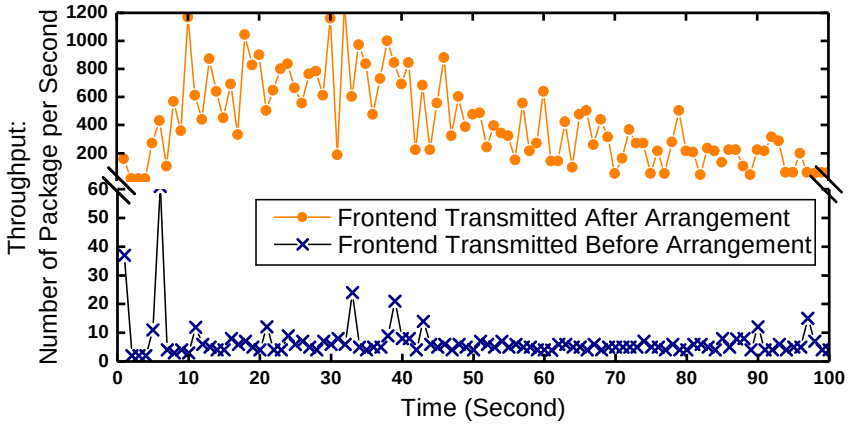
racks to a 10Mbps limited dual segment to simulate the saturation of the upper network traffic. Based on the internal dependencies of the collection, we put the same instance application in a rack to avoid saturated network traffic and then prove that the new placement strategy can optimize applications performance. Here, the assumption is that the performance bottlenecks are not located in CPU or memory.

Figure 3.9 presents the comparison results for RUBiS instance before and after arrangement. There are three different RUBiS running periods statistic data from 2000 rounds basic website activities. The periods are up ramp, runtime session, down ramp, and wrap them up as the final overall statistic. It shows that the average throughput after the arrangement has increased by an average of 1.93x because the new placement effectively avoids saturated networks.

Figure 3.10 presents the enhanced performance of Hadoop MapReduce and Spark instances. It shows that the performance for Hadoop MapReduce is averagely increased by 33.4%, and performance for Spark is averagely increased by 7.9%. In the experiment, the `Word Count` data source is generated by a program that randomly selects words from



(a) Throughput comparison for RUBiS database transmitted package numbers.

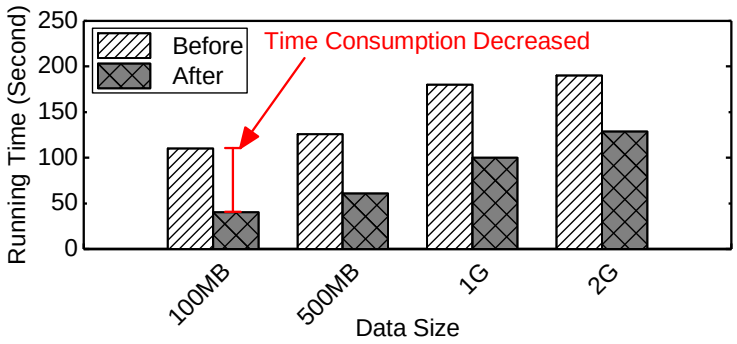


(b) Throughput comparison for RUBiS frontend transmitted package numbers.

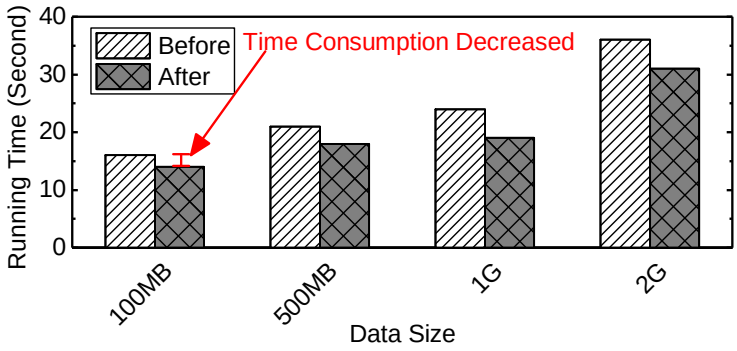
Figure 3.9: Throughput comparison for RUBiS before arrangement and after arrangement. It shows after arrangement the throughput increased averagely 1.93 times than before arrangement.

the dictionary file and then counts the number of occurrences of each word in the given input set. The `Secondary Sort` problem involves ordering values associated with keys in the decreasing phase.

These experimental results show that dependency perception enables container placement to gain performance, demonstrating that the information obtained using the DocMan toolset can help improve container placement and migration operations.



(a) Latency comparison of Hadoop Secondary Sort application.



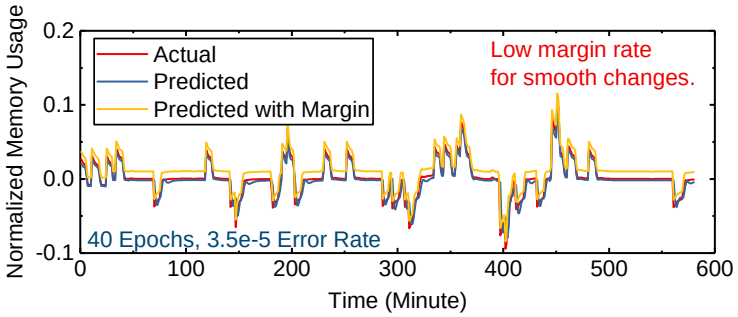
(b) Latency comparison of Spark Word Count application.

Figure 3.10: Latency comparison of Hadoop application and Spark application before arrangement and after arrangement.

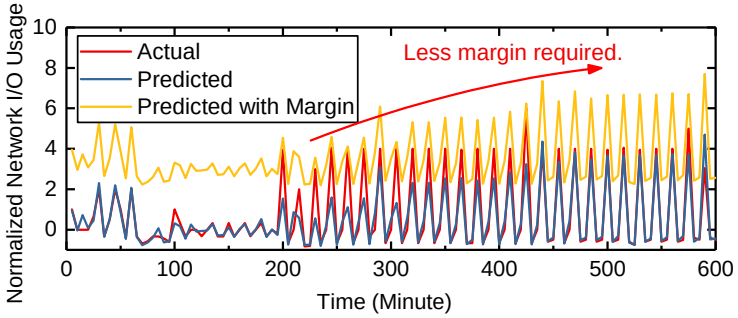
DocMan's Prediction of Containers Workload Pattern

To further leverage the results from DocMan, we set the deep learning neuron network to predict the resource usage. We set up the model as 6 layers, which include 3 LSTM layers, 2 dropout layers, and 1 dense layer. The dropout layer is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks [HSK+12].

Figure 3.11a and Figure 3.11b show the comparison result of resources usage predicted value with the true value for Spark machine learning task `Word2Vec` and Hadoop MapReduce task `Pentomino`. Predicting a single time point resource usage value is not



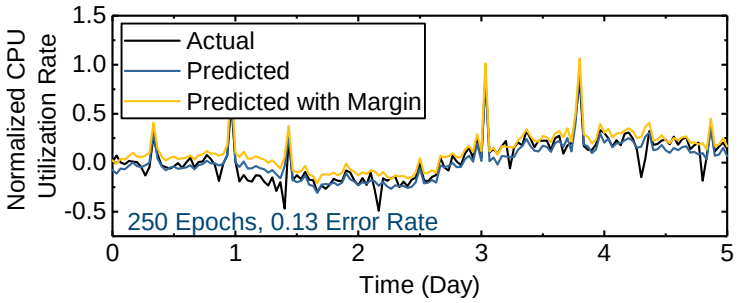
(a) Spark Word2Vec memory usage prediction. The memory usage changes smoothly, within limited epoch times, error rate got converged and only require a limited prediction margin.



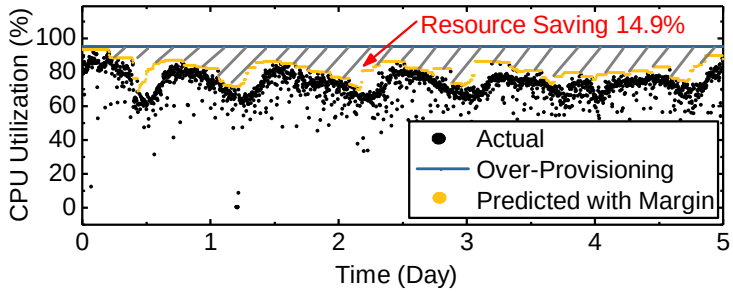
(b) Hadoop Pentomino network I/O usage prediction. The network I/O usage changes suddenly around the time point 200. The prediction model learns the unexpected change gradually, less prediction margin required along with the accuracy increasing.

Figure 3.11: Resource usage comparison between the actual value and the prediction value.

much meaningful since that will not give a chance for the system to automatically or manually adjust the resource arrangement. We set 50 steps as a prediction period for the normalized value. Figure 3.11a shows a smooth changing about memory usage, the prediction error rate converged to 0.0001 after 40 epochs training. Meanwhile, 10% prediction margin will cover all the actual value forming as a sealed envelop. Figure 3.11b shows the resource usage with a dramatic (increased more than 200%) change during its running time. The prediction margin needs to set as 100% to cover all the actual value. However, along with the learning process, the error rate decreased quickly. After 100 time period prediction, the required margin decreased to less than 25%.



(a) Comparison result of the CPU utilization true value and the prediction value.



(b) Comparison result of the hard coded resource over-provisioning with the predicted value with reasonable margin.

Figure 3.12: Microsoft Azure trace prediction for a CPU intensive task. (VM id: YANkW-PlG)

Figure 3.12 shows the prediction value about a CPU intensive server in Microsoft Azure from its traces [CBM+17]. For this CPU utilization trace, we use the prediction result showing in Figure 3.12a to compare it with hard-coded over-provisioning, there is 14.9% resource saving potentially. Figure 3.12b shows its CPU average utilization is over 70% and has the regular tidal pattern. If the utilization keeps in idle status or keeps in spike status, there is not much saving space for hard-coded over-provisioning.

Figure 3.13 shows the relationship between running time and prediction error rate when increasing training epoch. The running time increases linearly with the epoch increment. The error rate decreases when epoch increasing at the beginning, however, after it reached some point, it will gradually stop decrease, even increase in the figure of 200 epoch, which is caused by model over-fitting.

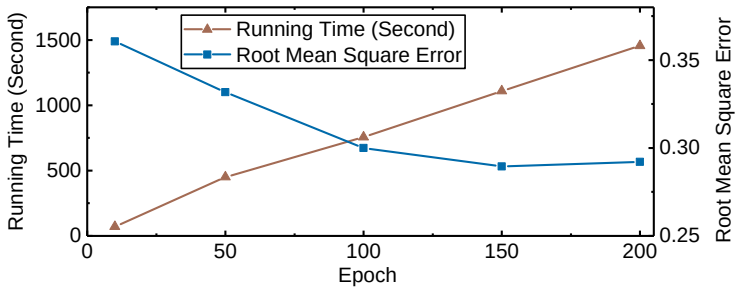


Figure 3.13: The average running time of generating the prediction model and each model's mean squared error for different epoch set up.

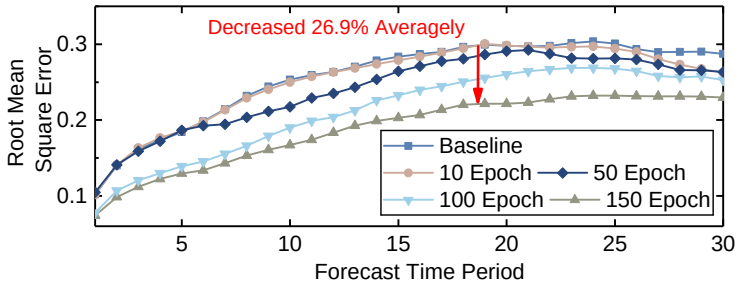
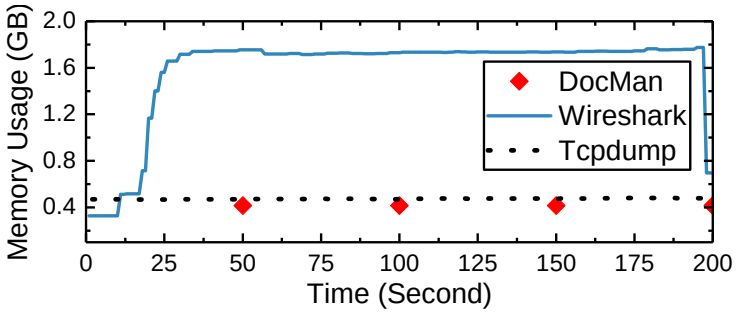


Figure 3.14: Compare LSTM different epoch results with default prediction method.

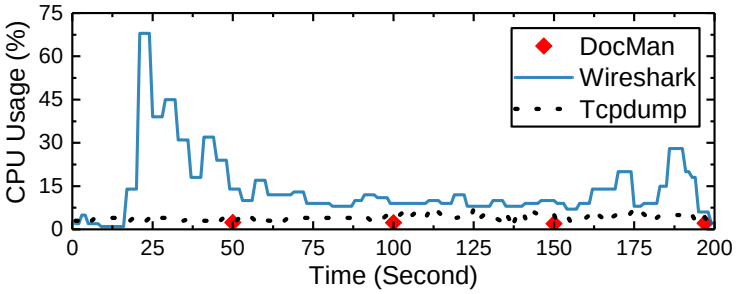
Table 3.1: Compare With Other Monitoring Tools

Monitor Tool	**Code Base**	**Memory Cost**	**CPU Peak**	**CPU Average**
DocMan	**3100 Lines**	**0.55GB**	**8.8%**	**7.8%**
Wireshark	30MB	1.20GB	68.0%	14.3%
tcpdump	931KB	0.05GB	10.0%	7.5%

Figure 3.14 shows the result of using LSTM to do the prediction compared with the baseline prediction. The epoch from 10 to 150, all the results are better than the baseline result, the best one decreases the error rate about 26.9%, that will provide a more meaningful hint to the resource management tool or cloud service provider.



(a) Memory usage.



(b) CPU usage.

Figure 3.15: Overhead comparison between DocMan, Wireshark and tcpdump.

DocMan's Overhead Analysis

Table 3.1 shows the comparison of DocMan and other two popular network monitoring tools, Wireshark and tcpdump. It demonstrates the attributes of the DocMan toolset's low overhead and lightweight feature.

To analyze the overhead of DocMan toolset, the first phase of correlation detection has no extra overhead because the resource utilization information is existing in the container management systems such as Kubernetes. The second phase of distance identification and clustering has a complexity of $O(N^2)$, where N represents the number of the containers. The third phase of offline prediction model training will not introduce new overhead for the online containers.

Figure 3.15 shows the overhead of DocMan toolset comparing with Wireshark and tcpdump. Wireshark is a widely-used protocol and network package analyzer. Tcpdump is a command line tool that analyzes common network packet. Figure 3.15a shows that

Wireshark uses 5 times more memory than tcpdump or DocMan. For DocMan clustering step, it runs on every 50 seconds periodically. The memory additional usage is stable and as low as 20MB. Figure 3.15b shows the CPU usage overhead. To launch Wireshark, it incurs a CPU burst. It can potentially impact other running container applications because it costs high CPU computation within a short period. Although after it goes to the stable status, it also consumes more CPU resource than DocMan. The CPU usage for tcpdump is similar to DocMan. However, DocMan can provide more insights to container service providers than a simple package analyzer.

3.1.10 Discussion

In this section, we discuss the open questions and current limitations of our prototype DocMan.

Potential drawback of the hierarchical clustering algorithm. Hierarchical clustering results may suffer from the chain-effect. The Single-link method merges the clusters whose two nearest neighbors have the smallest distance, which makes this approach sensitive to noise. Once a point would be assigned to a cluster, it will not be considered by joining into other clusters, which may lead to noisy points interruption. The experiments demonstrate the hierarchical clustering algorithm's effectiveness, but it is better to have other clustering approaches compared and evaluated if the noisy point can cause inaccurate results.

A finer granularity containers dependency level detection. DocMan focuses on the non-intrusive property to keep the container image isolation. Hence, we try to leverage the existing available server logs in management utilities. If we remove the constraint, then we can get a deep insight into containers relationships, such as sequential dependency and complementary dependency by using the container application level logs.

Bi-section network resource performance bottleneck heuristic. The heuristic may not always the root cause of performance latency, especially when the instance of contains does not have frequent network communications. In our evaluation, the time-sensitive instance got tremendous performance improvement such as RUBiS compared to Hadoop. However, even though the performance improvement is not noticeable, DocMan provides useful insights for the CaaS management tool to make future sophisticated decisions on container configuration, deployment, and placement.

Incorporate with incremental learning mechanism. The server resource log input data is continuously generated which can be used to extend the existing model's knowledge i.e. to further train the model. It represents a dynamic technique that can be applied when training data becomes available gradually over time or its size is out of system memory limits. Our static offline training model is expected to gain more accuracy by adapted to facilitate with incremental learning mechanism.

3.2 Summary

In this part of work, we study the dependency detection problem in a public CaaS cloud environment. First, we identify that there exists hidden dependencies between containers that belong to the same application by monitoring their resource usage statistics at runtime. Second, we design a black-box toolset called DocMan to detect these dependencies with negligible overhead. Third, we evaluate the accuracy of DocMan with real-world containerized applications.

DocMan's methods are fully implemented, but additional work is required for using it to continuously detect and manage containers at cloud-scale. For example, we need to filter out background traffic noises (i.e., heartbeat packets), since such traffic might otherwise be interpreted as intra-ensemble communications. We also plan to leverage

DocMan's insights to guide the container placement, thus improving containerized application's performance and amortizing the expenses related to their debugging and maintenance.

CHAPTER 4

ADAPTIVE REPLICATION OF HOT/COLD BLOCKS OF DISTRIBUTED SYSTEMS

4.1 Introduction

With the advent of cloud computation and development of infrastructure as a utility, we witness the emergence of Big Data industries based on data-intensive services i.e. Social Networking, Online Services especially over the last decade imposing computational, data and network traffic on the host servers. From the traditional centered file system, the industry moved towards the distributed scalable file system utilizing MapReduce framework i.e. HDFS [B$^+$08], GFS [DG08], Dyrad [IBY$^+$07] to cope up with the need of processing millions of user requests every second. Data replication mechanism has facilitated the way of fault tolerance and data availability in the distributed systems. Hadoop [had] is one of the state-of-the-art open source platforms to handle large scale data-intensive applications. Even though HDFS provides scalable and efficient data processing along with fault-tolerance [S$^+$10], data access and data movement overheads due to high disk I/O are primary bottlenecks of its current architectural design [SRC10, LW14].

Data replication is a widely used mechanism in the distributed systems to reduce the overall bandwidth consumption, response time and increase data availability. With the advancement and development of various technologies, new data replication and replica management approaches, both static and dynamic, have been proposed to achieve adaptiveness and better performance [W$^+$10].

Data replication and placement in Hadoop are uniform where the load balancing and the data locality for optimization are mostly handled by the applications. Trace driven log data analysis and experiments of different popular websites such as Yahoo! and Microsoft's Bing depict the existence of skewed popularity of different data and hotspots in

clusters [KC13, A$^+$11]. Consequently, keeping the uniform replication for every file without considering their popularity leads to performance overhead incurring data contention in the nodes denoted as hotspots where the popular data file resides. These drawbacks of HDFS architecture have lead to different dynamic approaches for the replication management. Based on the predictive analysis while keeping redundancy of data storage, dynamic adjustment of replication and replacement has been adapted, and improved algorithms have been introduced to effectively alleviate hotspots and data contention in the existing design [A$^+$11, B$^+$15, ALC11, VRMB11, AO15]. User access histories analysis, probabilistic prediction of data utilization have been included to effectively figure out the Hot and Cold blocks triggering the replication management [LMC15, A$^+$10, DL12]. Improvement in data availability has been achieved with replication of popular data in more locations than the default one. Replica placement also has been considered in these dynamic approaches [SCW$^+$12].

In fact, memory access I/O bandwidth is much higher than that of disk access bandwidth while processing user requests in the clusters. Conventional way of HDFS supports saving and loading each block from the disk resulting in I/O overhead incurring significant performance issues. Moreover, existing HDFS design cannot take the full advantage of the high-performance networks efficiently due to the high latency disk access. Combining in-memory I/O processing, HDFS can potentially overcome the issues.

On the other hand, in-memory storage systems allow applications to cache the results of the queries across the cluster nodes resulting in improved performance in SQL online query processing [I$^+$14]. So this introduces the possibility of adapting in-memory processing power in the existing design of distributed Big Data storage systems, and motivating to explore the answers of following questions:

- To improve I/O performance, can we leverage in-memory processing and caching concepts in the traditional HDFS?

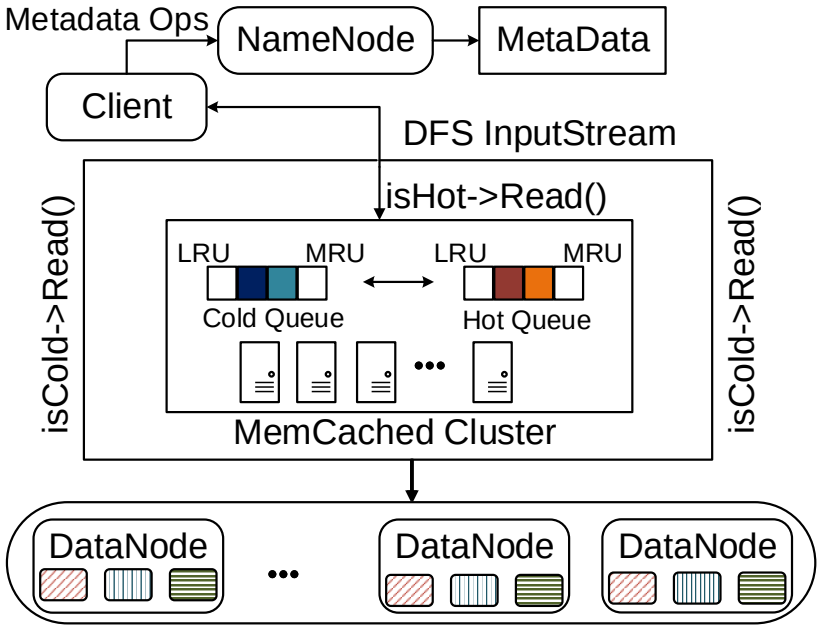


Figure 4.1: MemCached server integration with HDFS to dynamically replicate popular blocks.

- How much overall performance improvement can be achieved with dynamic replication factor for Hot blocks while keeping the existing default replication for Cold blocks?
- Can data contention be alleviated or reduced for DataNodes of HDFS by using distributed in-memory processing?

MemCached [Fit04] is a cost-efficient, high-performance distributed memory caching system to reduce the disk I/O access. It is designed as an in-memory key-value store to speed up the data access in the dynamic real-time applications. Leveraging MemCached, popular Social Networking site Facebook obtained improved performance while providing almost real time communication for millions of users [N$^+$13].

Integration of MemCached servers in the existing design can either be co-located with the DataNodes or on the separate nodes in the cluster. Placing MemCached servers in a separate location other than the DataNodes will result in reduction of contention. Uti-

lizing MemCached as the main replication block for popular Hot blocks, to reduce the load of reading blocks from HDFS and quickly access to the data, can improve overall performance while maintaining the default replication for fault-tolerance. Overheads of adding and deleting replicas in hard disks consume more computational power and each time the popular file is accessed, it needs to be read from disk according to traditional model. Whereas using MemCached caching capabilities, popular temporal data can be loaded in memory for the faster access and updated based on the timely access to add more replicas when required or deleted when they lose the popularity. The data popularity can be modulated by implementing an efficient dynamic replication algorithm for MemCached. This is more effective and efficient than the conventional approaches of changing the replication factor frequently in HDFS, as it significantly mitigates the disk I/O bottleneck and increasing instant data availability. So in a nutshell, integration of MemCached sever along with HDFS, can be exploited to answer all the above mentioned questions and guarantee high I/O throughput, performance gain for specific Hot blocks and finally reduced data contention in hotspots. Figure 4.1 shows the basic ideas to integrate these components together to fulfill the expected requirements. The different steps involved in accomplishing the proposed design can be outlined as

1. Configuration of MemCached server on separate nodes in cluster with HDFS for the further experiments.
2. Implementation of effective and dynamic data replication algorithms using MemCached as a caching layer.
3. Evaluation of the proposed system's performance to compare with existing system.

4.2 Architecture Design

Although different studies have been conducted to improve the dynamic replication management, there exist different trade-off among the various design choices. Therefore, we propose a toolset integrating MemCached as a caching layer to handle data popularity for dynamic replication management with new proposed algorithm.

4.2.1 Data Generation

BigDataBench [big] is used as a benchmark for the work and generated synthetic text data using its big data generation tools named BDGS (Big Data Generator Suite). BDGS module generates data in three steps:

- Application-specific and real-world data selection.
- Generation models construction, parameters and configuration derivation from data.
- Provide extensive workload testing.

After data generation, we integrate multiple workloads in BigDataBench to process the data set. The Wikipedia entries are used as the dataset and are processed to two different workloads `Word Count` and `grep`. Different sizes of data files are generated as input into HDFS.

4.2.2 Dynamic Model to Copy Hot Blocks to MemCached

HDFS plays the role as data saving layer, it is usually shared by multiple upper level applications. That means it will be hard to expect all these applications using the consistent way to access data from HDFS. Using the predefined Vanilla model will not be able to cater to variety access patterns to the data.

To more effectively use the limited space in memory, we propose the WLRU-MRU collaborative dynamic replacement algorithm. Its main idea is that each accessed data block will have a higher possibility to be accessed again soon that will have a much higher weight in the priority queue. But along with the time, the possibility to access it again will be decreased much more quickly until it is not worth to stay in the memory. However, during the decay process, if a data block is being hit by any application, it will be granted more life time, and back to the original decay speed.

Algorithm 1 shows the procedure of how to move the data block in Hot queue and Cold queue in MemCached by combining LRU and MRU algorithm to collaboratively do the replacement. LRU makes up for the deficiency of the LRU by introducing the concept of two checks. MRU replacement policy as the most recently used data block will be evicted, when a block of data is missing. The time prediction of MRU is higher than LRU. As for subsequent call for the Hot blocks get the data from MemCached server, read I/O performance will be improved as the fact is that memory I/O is much more faster than disk I/O.

4.2.3 Caching with MemCached

The proposed design leverages a caching mechanism which can improve the system performance and perform task based on time sequence. Since MemCached is a key-value in-memory storage, it is customized to store the detected Hot data blocks over a certain period of time. All DataNodes will be configured with local MemCached which will ensure data locality. When specific block becomes Hot, for the first time we will have to set the contents in MemCached as key-value pair. For all the subsequent calls, it will be read from MemCached rather than DataNode till the blocks remain Hot.

Algorithm 1: Dynamic collaborative replacement algorithm

```
   Input: Label M, Label N
 1 if (M==0) //datablock is not in MemCached. then
 2     if (N==0) //tag hit is 0 then
 3         LRU; //call LRU
 4         Replace(bottom); //replace the data at bottom
 5         MoveDown(other data); //other data move down in turn
 6     if (N==1) //tag hit is 1 then
 7         MRU;
 8         //call MRU
 9         Replace(top); //replace the data at top
10         MoveUp(other data); //other data move up in turn
11 if (M==1) //datablock is in MemCached. then
12     if (N==0) //tag hit is 0 then
13         MRU; //call MRU
14         minHeap.put(<hitData.id, hitData.weight++>);
15         MoveUp(other data); //other data move up in turn
16     if (N==1) //tag hit is 1 then
17         LRU; //call LRU
18         maxHeap.put(<hitData.id, hitData.weight++>);
19         MoveDown(other data); //other data move down in turn
```

Advantages of MemCached over other approaches are that it can handle high memory load for it's distributed characteristics, responds quickly and finally provides accurate expiration times. Since the main focus is replication depending on time, MemCached has the advantages over other in-memory key-value storage and it perfectly suits in the design choice.

4.2.4 Evaluation Metrics

To compare the performance of the proposed model, we need the benchmark performance which was gathered from the initial log analysis of Hadoop without any modification and the one with vanilla MemCached integration, which using a simple threshold value to control the Hot/Cold blocks. Job execution time, I/O throughput, CPU usage and memory usage are considered as the performance metric for all the workloads. All of the metrics will be recomputed using modified replication management scheme in Hadoop and will be used to evaluate the overall performance.

To evaluate the proposed design, we analyze the same performance metrics taken initially and expecting to get a performance improvement in all the metrics.

Even though we are expecting several improvements, there might be some overhead involved as follows:

- In case of cache miss, there will be extra overhead in the I/O operation since an additional layer is involved.
- Since the Hot blocks will get change over time, updating the MemCached will be another overhead.
- Due to the limited size of caching, we have to design an approximate policy for the blocks replacement in MemCached.

4.3 Experimental Results

The test is performed on the server that has two Intel processors, 16GB of memory and 6TB hard drives. The different sizes (from 100GB to 500GB) of data is generated with BigDataBench BDGS module and, performed two different operations on these synthetic data. The experimental results includes i) identification of Hot and Cold blocks in HDFS (Hadoop 2.8.2 [had]), and ii) Exploration of the initial performance metrics of the system from log analysis.

4.3.1 Dynamic Hot and Cold Block Detection

To detect which blocks are hot and which are cold in HDFS, first we generated a 60 min stream of block access using randomization. Then, we analyzed the Global map to get the number of the read count for each block. In the settings, 16 data blocks are stored in the Hadoop cluster, each block contained 524MB data generated by BDGS.

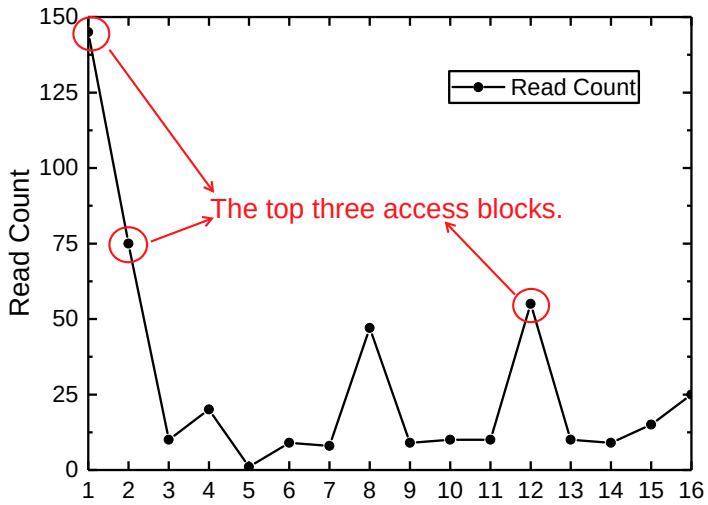


Figure 4.2: Read count of HDFS data blocks

Figure 4.2 shows the read count history of each 16 data blocks of the HDFS. Here, data block 1, 2, and 12 are the top three blocks accessed during the time period. Meanwhile,

data block 5 is the least accessed block. Global key-value store is used, so every node in the Hadoop cluster can access the status of Hot and Cold Blocks.

To detect the Hot and Cold block dynamically, it is required to find the trace of number of block reads in the real time. And from the trace we find out that, DFSClient actually sends a read or write request directly to DataNode, where as NameNode only provides DFSCLient with the list of located DataBlocks which is then used to read the blocks. Since DFSClient reads the file directly from DataNode, if to modify the NameNode to detect Hot and Cold blocks, we would have to send the packets over network from DFSClient to NameNode every time for a successful read which would result in extra netword overhead. So DFSInpustream which reads the data from DataNode seems to be the optimal place to inject the modification. Also we use HashMap, Global key-value store, so every node in the Hadoop cluster can access the status of Hot and Cold Blocks.

Using any other kinds of storage for keeping record of Hot/Cold status, like file will result in more I/O overhead and might hurt the improvement gain. So using the in-memory key-value store for Hot and Cold block detection is leveraged for the higher performance. This will enable the function to analyze the HDFS file access in real time and detect the Hot blocks in the system.

4.3.2 Vanilla Model to Copy Hot Blocks to MemCached

In this vanilla model, MemCached is used as a caching system to alleviate the loads of user requests based on a predefined threshold value. Algorithm 2 describes the Hot block detection mechanism. When DFSClient requests for a specific data block, NameNode looks for all the available DataNodes options and return the LocatedBlocks as a list to the client. Then the client processes to choose the best node and passes the request to DFSInputStream to handle packet transfer from DataNode. DFSInputStream uses BlockReader

and PacketReceiver sequentially to read the packets from the stream. Then checking the access pattern in DFSInputStream read function to update the access counts in a map as key-value. When the access count for any block reaches the predefined threshold, the system will apply caching techniques.

We compare the proposed design with this vanilla model and the default HDFS infrastructure. Since MemCached has the expiration time, if a block becomes Cold after certain period, it will be automatically deleted which let the vanilla model also serves the purpose of simple dynamically changing replication.

Figure 4.3 shows the process of moving in the Hot block and moving it out when it becomes Cold. This process does not require any complicated calculation, but it is required to define the Hot block threshold and the expire time in advance. These setups require the experience of using the HDFS applications, meanwhile, the applications accessing data blocks method tends to be consistent during the life cycles. The log analysis process (CPU uses, memory uses, job execution time) will be performed again after copying Hot files in MemCached and then compare the results with the previous one.

All data blocks are not accessed uniformly in HDFS. Depending on the popularity, different blocks could be accessed more frequently than the others making the residing DataNodes hot. So, detecting the Hot and Cold blocks dynamically using the access history is one of the main challenges. Dynamic detection steps include populating the file access stream with a randomized scheme over a certain period of time to represent the real life scenario. During the streaming time, we analyze the file access requests from the DFSClient (Distributed File System Client) and the most frequently accessed file will be identified as Hot blocks for the future steps.

Algorithm 2: Dynamic Hot and Cold block detection

Input: BlockID and SequenceNo of CurrentLocatedBlock that is being read from DFSClient, Map with (BlockID,AccessCount), threshold

Output: Set status of BlockID

```
1  key = CurrentLocatedBlock.BlockID + SequenceNo;
2  if (key exists in Map ) then
3      AccessCount = Map.get(key) + 1 ;
4      Map.put(key, AccessCount, timer) ;
5  else
6      Map.put(key, 1, timer) ;
7  end
8  AccessCount = Map.get(key);
9  if AccessCount > threshold then
10     status(BlockID) = hot;
11     Set BlockID content in MemCached ;
12 else
13     status(BlockID) = cold;
14     Normal operation;
15 end
```

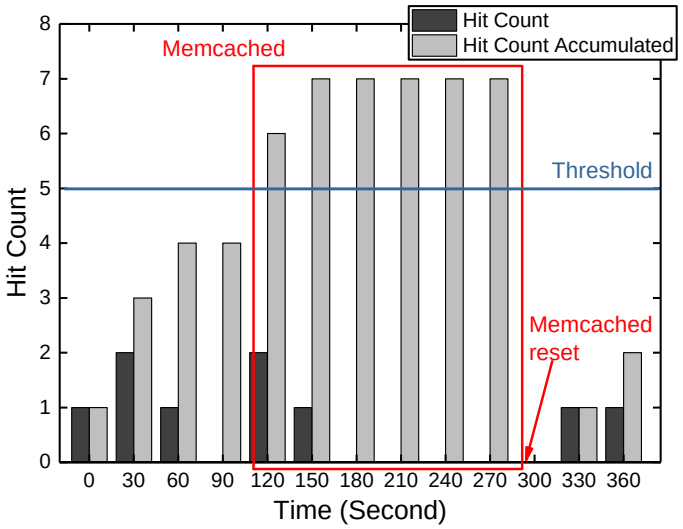


Figure 4.3: After hit count is over the predefined threshold, the block is defined as Hot and MemCached records it. After a specific period, once the hit count stops increasing, MemCached will reset the records and release the occupied memory.

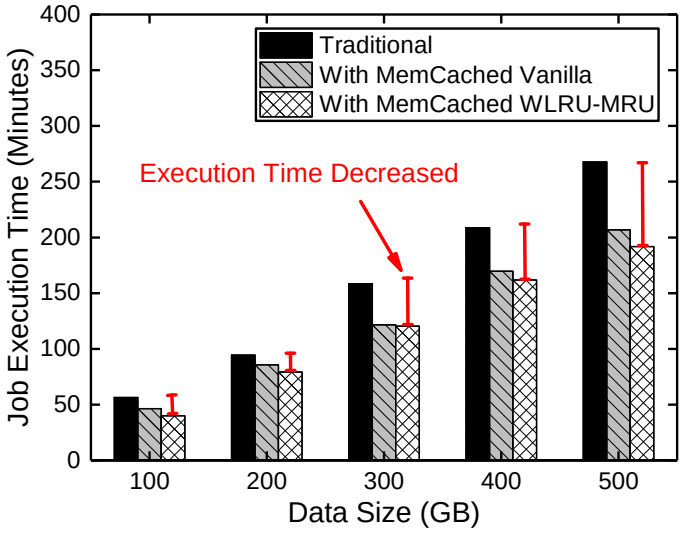


Figure 4.4: Word count execution time.

4.3.3 Performance Analysis

To gather the current system performance metrics, we used two workloads from BDGS: `Word Count` and `Grep`. The workloads are applied on different sizes of data stored in HDFS. `Word Count` data source is generated by the program which randomly picks words from a dictionary file, and then counts the number of occurrences of each word in the given input set. `Grep` extracts matching strings from text files and counts how many time they occurred.

Figure **??** shows the jobs execution time comparison results between original HDFS framework and the proposed dynamical model design. Figure 4.4 is about `Word Count` execution time. It shows after applied new design, the execution time decreased, especially for the bigger data size input, the decreased time is much more remarkable. For 400GB and 500GB file size input, the execution time decreases 36% and 29% respectively comparing to the default setup, even 10% and 17% comparing to the vanilla MemCached integration. Figure 4.5 is about `Grep` execution time. Although there is no clear trend

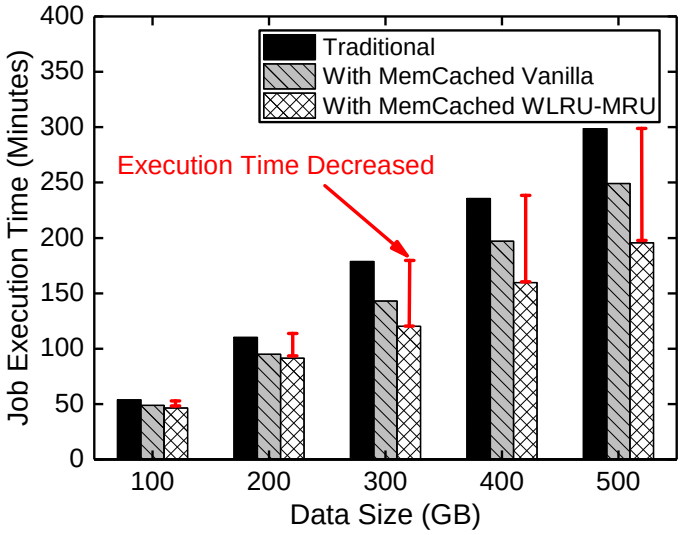
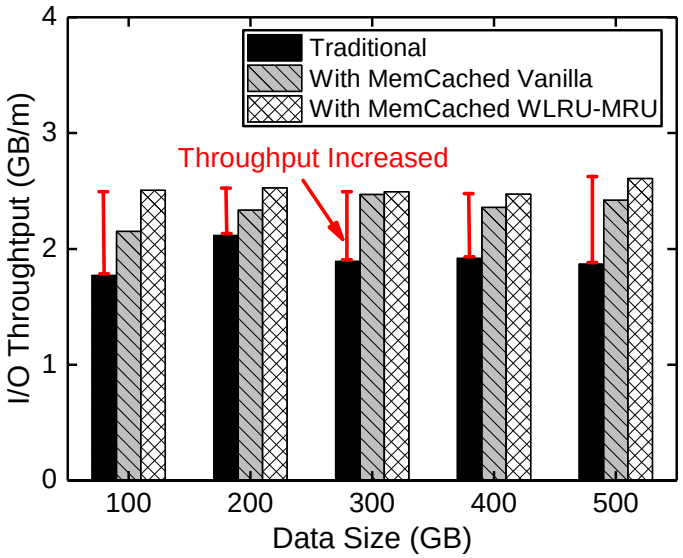


Figure 4.5: Grep execution time.

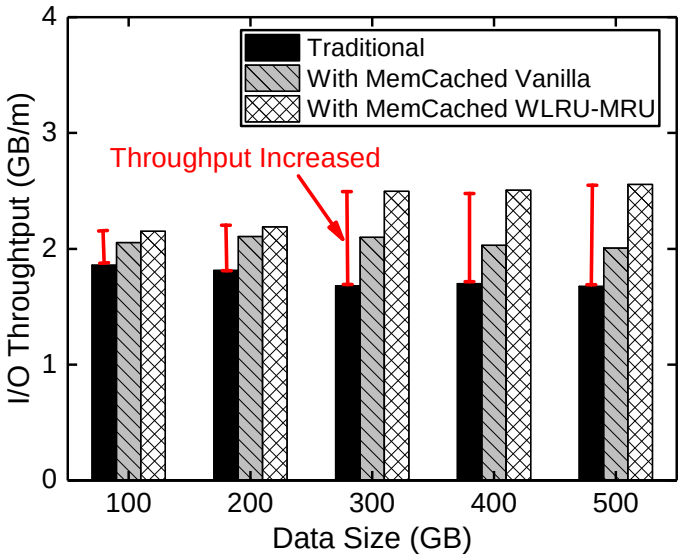
of the impact for execution time along with data size increasing, the performance still increased after integrating with MemCached.

Figure 4.6 shows the evaluation of read performance in terms of throughput. The throughput factor is not much sensitive with the changes of input data size. Worth to note is, that after implementing MemCached, both `Word Count` and `Grep` got increased throughput, since Hot block data are available to be retrieved from memory which far quickly than retrieving data from the disk. Since the experiments are performed on synthetic datasets and randomizing the file access pattern to simulate the real life scenario, we need to figure out optimal threshold of time window in MemCached to set after which the data will expire.

For the overhead analysis, Figure 4.7 shows that 60s expiration window is the optimal one in the testing cases and CPU time is the lowest for this configuration which is even lower than the original default HDFS setup. The reason can be explained as the I/O wait time is negligible because of the in-memory caching. That means deployed MemCached would cost some CPU resource, but the saved I/O wait time can neutralize the impact. Figure 4.8 shows that, after implementing MemCached into HDFS, the memory cost is



(a) Word count I/O throughput.



(b) Grep I/O throughput.

Figure 4.6: I/O throughput for word count and grep.

increased, that because these memory cost is majorly used to save Hot block dynamically. This is the trade-off for the system performance increase. So to implement the proposed design is based on the circumstance assumption that the nodes are not struggling in memory usage and there are spare memory spaces to use for performance increase.

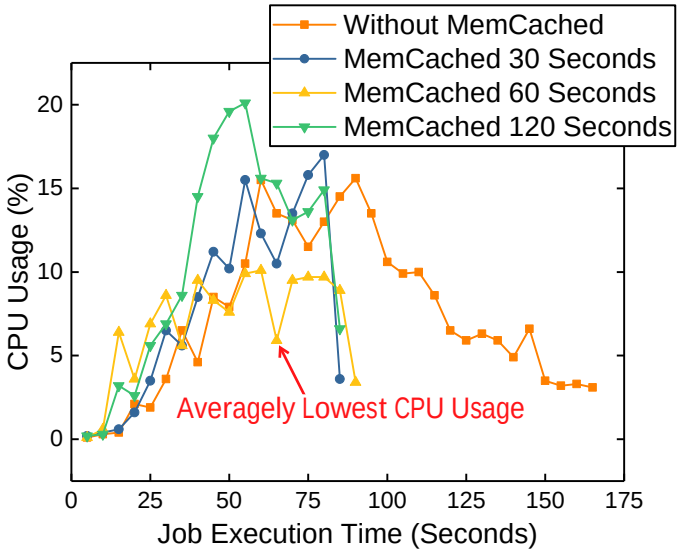


Figure 4.7: CPU usage comparison.

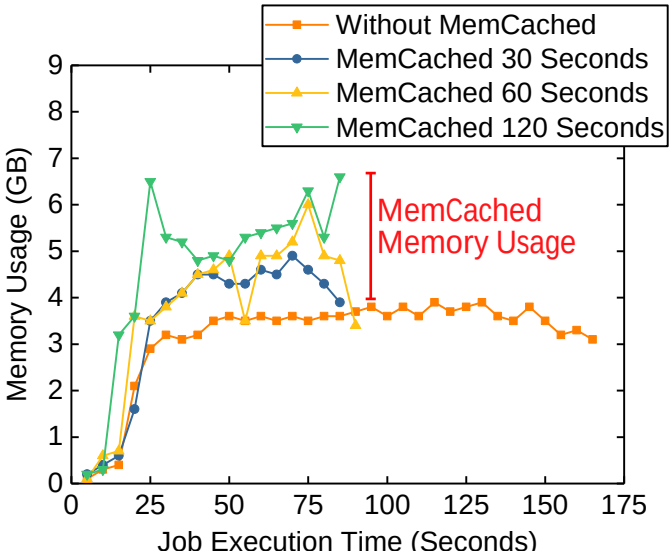


Figure 4.8: Memory usage comparison.

4.4 Summary

In this paper, we analyze the possibility to improve the HDFS data blocks replication mechanism. The caching strategy MemCached is leveraged in the design to effectively replicate the popular blocks in memory. The experimental results show that the proposed design is able to improve the HDFS based applications' performance from different per-

spectives and has the potential to overcome the critical issues of traditional Big Data storage systems.

CHAPTER 5

STATE RECOVERY OF STREAM PROCESSING DISTRIBUTED SYSTEMS

5.1 Introduction

Data stream processing technology has become a critical building block of many real-time applications, such as making business decisions from marketing streams, identifying spam campaigns from social network streams, predicting tornados and storms from radar streams, and analyzing genomes in different labs and countries to track the sources of a potential epidemic. Over the last decade, a boom of stream processing systems has been developed including Storm [sto], Trident [ai], Spark Streaming [spa], Borealis [AAB+05], TimeStream [QHS+13], S4 [NRNK10], etc.

A driving need is that today many stream applications need to store and update the large-sized application state along with their processing, and process live data streams in a timely fashion from massive and distributed data sets. This poses a significant challenge to the failure recovery mechanism of state-of-the-art stream processing systems. This is because (1) stream operators are by nature long-running in which failures and stragglers are inevitable and very difficult to predict; (2) a large number of stream applications may run concurrently on the same platform, and many distributed operators may fail simultaneously; and (3) large distributed states must be restored efficiently after node failures.

A stream is an unbounded sequence of tuples (e.g., online social network's microblog streams) generated continuously in time. A stream processing system creates a logical topology of stream processing operators, connected in a directed acyclic graph (DAG), processes the tuples of a stream as they flow through the DAG, and outputs the results in a short time. DAGs can be implemented via many patterns, such as the partition/aggregate pattern which scales out by partitioning tasks into many sub-tasks (e.g.,

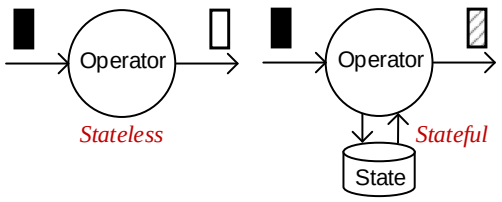


Figure 5.1: Contrast of stateless stream processing and stateful stream processing.

Dryad [IBY+07]), sequential/dependent pattern in which streams are processed sequentially and subsequent streams depend on the results of previous ones (e.g., Storm [sto]), and hybrid pattern with sequential/dependent and partition/aggregate (e.g., Spark Streaming [spa], Naiad [MMI+13]). Figure 5.1 shows the contrast of stateless stream processing vs stateful stream processing. Input records are shown as black bars. A stateless operator transforms each input record one at a time and outputs each result based solely on that last record (white bar). A stateful operator maintains the value of state for some of the records processed so far (in local memory or remote storage) and updates it with each new input, such that the output (the bar with the pattern) reflects results that take into account both historical records and the new input. State recovery is the process of recovering application states when one or many operators fail or lose their states.

Application developers are facing significant challenges in handling many simultaneous failures for a large number of concurrently running stream applications.

The first challenge is "*how to scale recovery with the size of the state, the number of simultaneous failures and the number of concurrently running stream applications on a shared platform?*" Existing studies [sto, ai, spa, NRNK10, ab, ae, af, ABC+15, LLP+12, KBF+15] mostly inherit MapReduce's "single master/many workers" architecture, where the central master is responsible for all scheduling activities. As such, they are limited to a fixed computation model, e.g., asynchronous stream processing like Storm [sto], synchronous mini-batch processing like Spark[ag], etc. Note that the recovery operation is a critical consumer of time and space. It must quickly recover all failure operators'

lost states on failover nodes (if any) without blocking the normal processing of stream applications. As a result, it is difficult (or even impossible) for the centralized master to manage state recovery of a large number of concurrently running applications due to the inherent centralized bottlenecks.

The second challenge is "*how can we handle many simultaneous failures while achieving fast recovery and imposing low hardware cost?*" State-of-the-art stream processing systems offer failure recovery mainly through three approaches: replication recovery [SHB04, BBMS05], checkpointing recovery [sto, ai, QHS+13] and DStream-based lineage recovery [aa, CEF+17, ZDL+13, SGH+16], which are either slow, resource-expensive or fail to handle many simultaneous failures. Replication recovery adds significant hardware cost because multiple copies must concurrently run on distinct nodes for failover. Checkpointing recovery is known to be prohibitively expensive, and users in many domains disable it as a result [MMI+13, ABB+16, PD10, PLGC15, GXD+14]. DStream-based lineage recovery is slow when the lineage graph is long and falls short in handling multiple simultaneously failures.

We present FP4S, a novel fragment-based parallel state recovery mechanism to address the challenges listed above: to efficiently handle many simultaneous failures for a large number of concurrently running stream applications in a fast, scalable, and lightweight manner.

FP4S operates as follows: (1) we first organize all the application's operators into a distributed hash table (DHT) based consistent ring [RD01] to provide each operator with a unique set of neighbors; (2) afterward, we divide each operator's in-memory state into many fragments using erasure codes [RS60]. Erasure codes operate by converting a data object into a larger set of code blocks such that any sufficiently large subset of the generated code blocks can be used to reconstruct the original data object; and (3) finally, we periodically checkpoint each node's state in its neighbors, ensuring that different sets

of available fragments can be used to reconstruct failed state in parallel. By doing that, this failure recovery mechanism is extremely scalable to the size of the lost state, significantly reduces the failure recovery time and can tolerate many simultaneous operator failures.

We apply FP4S on Apache Storm and evaluate it using large-scale experiments with real-world datasets. Experimental results demonstrate the scalability, efficiency, and fast failure recovery of FP4S. When compared to the state-of-the-art solutions (Apache Storm [sto]), FP4S reduces in 37.8% the state recovery latency and reduces more than half of the hardware costs. It can scale to many simultaneous failures and successfully recover the states when up to 66.6% of states fail or get lost.

Contributions. We make the following technical contributions:

- We propose a decentralized architecture using a DHT-based consistent ring and erasure codes to recover the distributed states for numerous concurrently running stream applications. To the best of our knowledge, FP4S is the first work to use a fully decentralized architecture for state recovery.
- We implement the FP4S prototype on the state-of-the-art stream processing system Storm and demonstrate its portability to many other stream processing systems.
- We make a comprehensive evaluation of the scalability, fast recovery and robustness of FP4S on a large cluster using real-world stream application's datasets.

5.2 System Design and Implementation

In this section, we describe the basic workflow of FP4S, introduce each component, show how stream applications' distributed states are recovered by the FP4S-enabled stream processing system, and explain the performance, scalability and flexibility benefits of using FP4S.

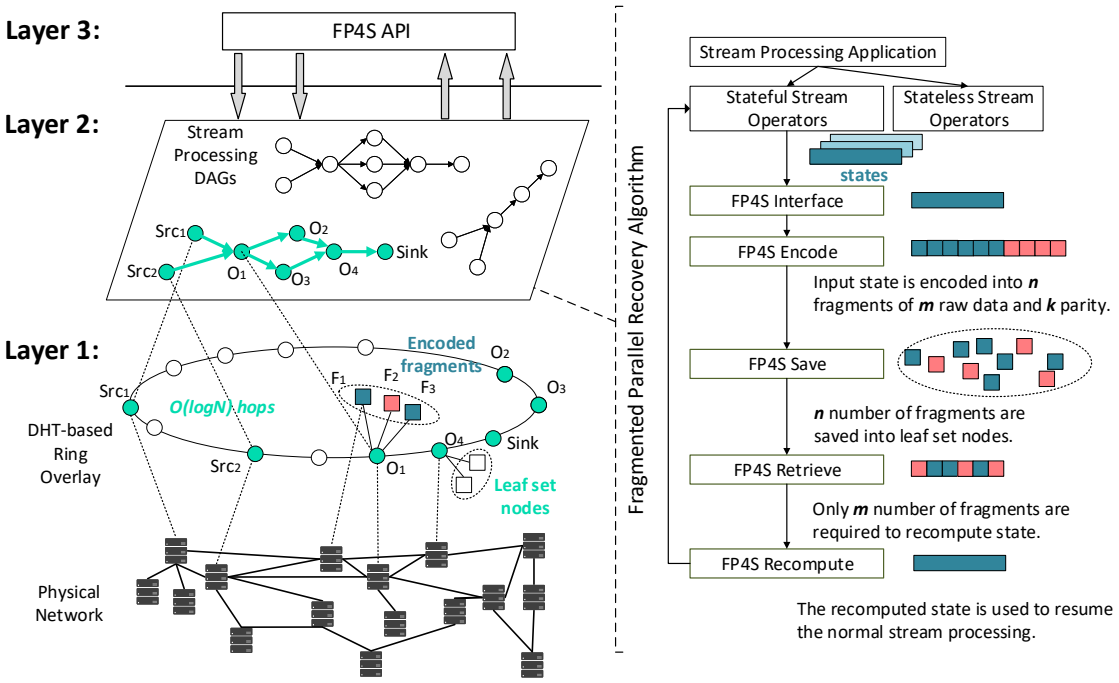


Figure 5.2: FP4S system design.

5.2.1 Overview

The FP4S aims to achieve the following goals:

- *Resource efficient.* Avoid the replication hardware overhead.
- *Fast recovery.* Avoid the slow recovery of retrieving state from disk and replaying the data input that hurts the service quality of stream applications.
- *Resilient to multiple failures.* The mechanism needs to handle multiple simultaneous failures due to the much higher node dynamics in large clusters.

As show in Figure 5.2, the FP4S system consists of three layers: *The DHT-based consistent ring overlay*, *the fragmented parallel state recovery mechanism*, and *the high-level FP4S interfaces* that are exposed to the stream processing systems (e.g., Storm [sto], Spark Streaming [spa], Heron [KBF+15]) for implementing the state recovery for stream applications.

- **Layer 1: DHT-based ring overlay.** Each data center server is installed with one or many in-situ stream operators, also called "nodes" in this study. We organize these potentially hundreds of thousands of nodes into a distributed hash table (DHT) based ring overlay (e.g., Pastry [RD01], Chord [SMK+01]) which is commonly used in Bitcoin [N+08], BitTorrent [Coh03], and FAROO [ak]. This overlay is self-organizing and self-repairing. To do that, each node needs to maintain two data structures: a routing table and a leaf set, in which the routing table is used for looking for the state (within $log(N)$ hops) and the leaf set nodes are used for recovering the application state if one or more nodes fail.
- **Layer 2: fragmented parallel state recovery.** Periodically, the state in each node's memory is divided into m identically-sized blocks, which are encoded into n blocks, where $n > m$. The n blocks of the state are replicated to n nodes from the original node's leaf set nodes in parallel, guaranteeing that the original state can be reconstructed from any m blocks.
- **Layer 3: high-level interfaces to stream processing systems.** The high-level FP4S programming API (Table 5.1) is exposed to the state-of-the-art stream processing systems and programmers for implementing the parallel state recovery policies for concurrently running stream applications, e.g., Storm [sto], Spark [spa], and Flink [aa].

5.2.2 DHT-based Ring Overlay

FP4S leverages DHT-based consistent overlay [RD01, SMK+01] to support parallel recovery of distributed states for a large number of concurrently running stream applications. In this DHT-based consistent ring overlay (e.g., Pastry [RD01], Chord [SMK+01]), each node is equal to the other nodes, and they have the same rights and duties. The

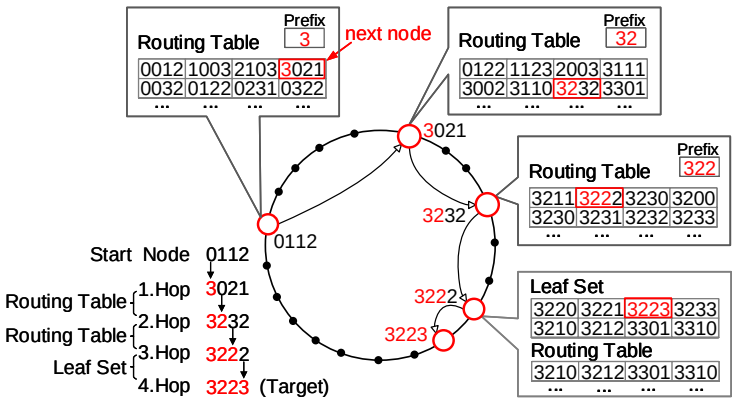


Figure 5.3: The routing process is cooperatively fulfilled by the routing table and the leaf set.

primary purpose of this model is to enable all nodes to work collaboratively to deliver a specific service. For example, in BitTorrent [Coh03], if someone downloads some file, the file is downloaded to her computer in parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others that ask for it.

Similar to BitTorrent in which many machines work collaboratively to undertake the task of downloading files and uploading files, we enable distributed stream operators to work collaboratively to undertake the original centralized master's failure recovery task. First, each stream operator maintains an in-memory buffer to store the application state. Instead of storing states at a remote storage, these distributed stream operators store the states for each other. Second, these distributed stream operators (nodes) are self-organized into a DHT-based overlay. Each node is randomly assigned a unique `NodeId` in a large circular `NodeId` space. `NodeIds` are used to identify the nodes and route stream data. It is guaranteed that any data can be routed to a node whose `NodeId` is numerically closest to the destination node within $O(logN)$ hops. To do that, each node maintains two data structures: a routing table and a leaf set.

1) *Routing table:* The routing table consists of physical node characteristics (`NodeId`, `IP`) organized in rows by the length of common prefix. When routing a message, each node forwards it to the node in the routing table with the longest prefix in common with the destination `NodeId`. Figure 5.3 shows this routing process of Pastry's DHT [RD01]. At each routing step, given a key, Pastry routes messages to the node whose `NodeId` is numerically closest to the key. The node first checks if the key falls in the range of the `NodeIds`' leaf set. If so, the message is directly forwarded to that node. If not, the message is forwarded to another node in the routing table whose `NodeId` shares a common prefix with the key by at least one more digit (see Figure 5.3 first, second, and third hops). In some cases, there is no appropriate entry in the routing table or the associated node is not reachable. Then the message is forwarded to a node whose prefix is the same as the local node, but numerically closer.

2) *Leaf set:* The leaf set contains a fixed number of nodes whose `NodeIds` are numerically closest to each node, which assists in rebuilding routing tables and reconstructing application's state when any operator fails (see Sec. 5.2.3, next, for more details).

5.2.3 Fragmented Parallel State Recovery

The parallel recovery mechanism of FP4S leverages the key idea from erasure codes. Erasure codes operate by converting a data object into a larger set of code blocks such that any sufficiently large subset of the generated code blocks can be used to reconstruct the original data object. For example, (32, 16)-Reed-Solomon (RS) code [RS60] divides a data object into 16 blocks and transforms these blocks into 32 coded blocks, guaranteeing that any 16 out of the 32 coded blocks are sufficient to reconstruct the original data object. Erasure codes have been widely used in massive storage systems (e.g., OceanStore [KBC+00]), Bar codes (e.g., QR Code [KLM+10]), data transmission tech-

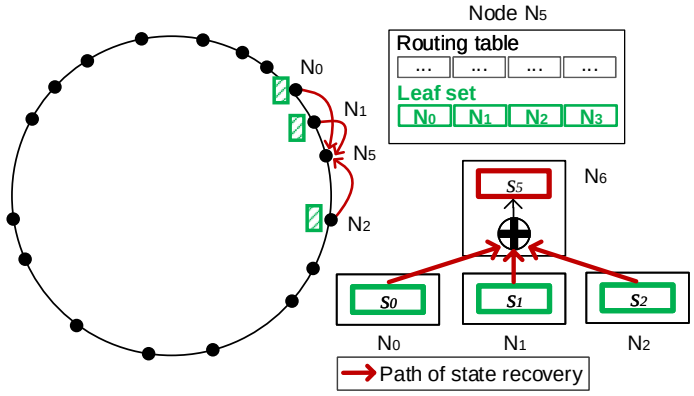


Figure 5.4: The fragment-based parallel state recovery process.

nologies (e.g., DSL [GDJ07]) and space transmission technologies (e.g., Galileo Probe). Figure 5.4 shows the steps of the erasure-code-based parallel recovery algorithm.

Built upon Sec. 5.2.2's DHT-based ring overlay, each node maintains a routing table and a leaf set. Periodically, the state in each node's memory is encoded into n identically-sized fragments, which include m raw data fragments and k parity fragments, where $k >= 1, n = m + k$. Then these n fragments of the state are replicated to n nodes in the original node's leaf set in parallel. The error correction mechanism of the erasure codes guarantees that any m out of the n fragments are sufficient to correctly recompute, even though when some fragments are not available in the leaf set (denoted as e), to reconstruct the original state. Thus, as long as $n - e >= m$, the original state is safe to be accurately recomputed from the node's leaf set nodes.

- **Step 1: encode state.** For each node, FP4S converts its current version of state in a sliding window into n fragments (configurable parameter) according to RSCodes algorithm [RS60]. These n fragments include m raw data fragments and k parity fragments. The amount of m and k are configurable.
- **Step 2: save state.** Each node sends these n fragments to any n of its leaf set nodes. We ensure that the size of the leaf set is larger than n. We assign the `NodeIds`

to reflect the physical proximity in order to ensure that the leaf set nodes are also geographical closest nodes that have abundant bandwidth.

- **Step 3: retrieve state.** Once any failure happens, the retrieve routine is triggered. A request to obtain the lost state's fragments will be sent out. To recompute the lost state, FP4S only requires m amount of n total fragments. These fragments are stored at the leaf set nodes that are quite easy to access.
- **Step 4: recompute state.** Finally, the state recompute routine is triggered, which reconstructs the lost state using erasure codes [RS60]. After that, the recovered state will be used as input for the downstream operators and we can resume the normal stream processing.

The benefits are the following: (1) it allows for tolerating a maximum of $(n-m)$ simultaneous failures; (2) the recovery process is fast. For multiple failures, different nodes from non-overlapping leaf set nodes can work in parallel to recompute the lost state, which is faster than DStream's line-structured recovery that executes strictly in line with the original lineage graph; and (3) we achieve data locality because the leaf set contains nodes that are geographically close to the original nodes (e.g., in the same rack or in the same site) that have abundant upload bandwidth.

5.2.4 FP4S API

FP4S is platform-agnostic and can be easily integrated with stream processing platforms such as Storm [sto], Spark Streaming [spa], Flink [aa], Timely Dataflow [QHS+13], Heron [KBF+15], etc. In our design, using FP4S is essentially a configuration option. Depending on the usage scenario (e.g., stateful or stateless, latency requirement, reliability requirement), users can choose to configure whether and when they want FP4S support. Table 5.1 shows the FP4S API.

Table 5.1: FP4S API

List <Fragment>`Encode`(int rawDataNumber, int parityNumber, State inputState) The function is invoked to encode a state into many fragments. The fragment number is decided by the inputs of `rawDataNumber` and `parityNumber`. The output is a list of encoded fragments with the length of `rawDataNumber` + `parityNumber`.
Boolean[] `Save`(List<>fragment, DHTNetwork dhtNetwork, int numberOfThreads) The function is invoked to save state into the DHT's overlay. It generates multiple threads to concurrently save the fragments. The inputs are the fragments, the DHT overlay information and the number of threads. The output is a Boolean array that indicates the status of each fragment.
List<Fragment>`Retrieve`(String stateName, DHTNetwork dhtNetwork, int numberOfThreads) The function is invoked when a state recovery request is issued.
String Recompute(List<>fragments) The function is invoked to recover the state. It loops through all the retrieved fragments. If the number of fragments is equal or larger than the number of raw fragments, the function will perform further computation to recompute the retrieved fragments into the original state.

5.3 Adaptivity Analysis

5.3.1 Adaptive Parameter Tuning

We provide a theoretical analysis model that can dynamically adjust the size and number of FP4S fragments to achieve the adaptability of our system. FP4S collects the instrument data during each episode, uses this data to train the model that will be exported next, and then configures system parameters for the next episode. The input to the model includes hardware properties (e.g., network and CPU speed), application characteristics and also user preferences. Using this information, our models can accurately moderate the number of data fragments m and parity fragments k to match the requirements for the subsequent episodes.

FP4S can adjust the value of k so that it can accommodate multi-fragments failures during the recovery process. Such extra cushion of reliability is particularly desirable when the application nodes are known to be more failure prone and unreliable. In con-

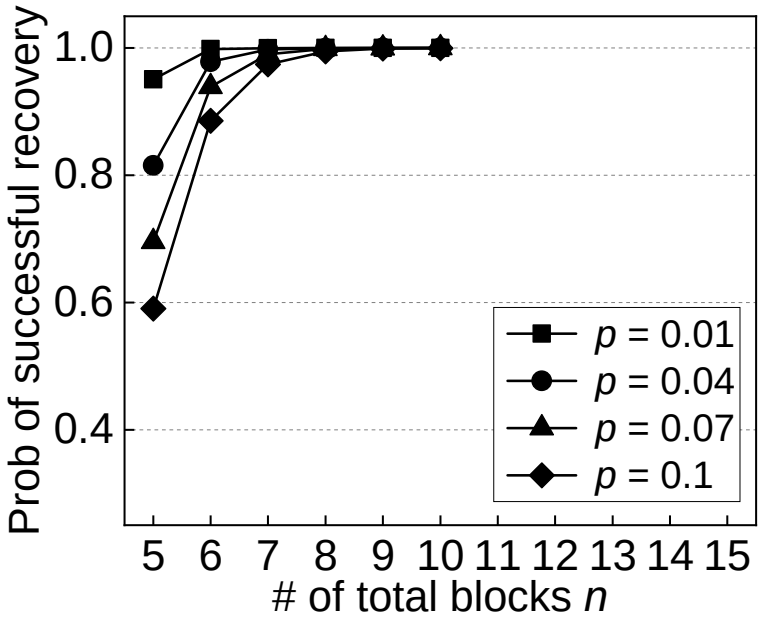


Figure 5.5: Probability of successful recovery.

trary, some other applications (e.g., real-time network monitoring) may opt for faster recovery time over 100% reliability. FP4S can adjust m and thereby the default size of each fragment to reduce the recovery latency.

5.3.2 Analysis

In this subsection, we analyze the performance of FP4S with adaptive number of fragments and compare it with the checkpoint-based recovery (e.g., Apache Storm) in the cluster. We compare the different methods of failure recovery based on three aspects: (1) the hardware properties (e.g., network bandwidth, disk speed), (2) the application characteristics (e.g., size of the state, size of fragment), and (3) the DAG length characteristics.

Assume the volume of state that is saved in each operator is s bytes. Considering in a DAG, where the operator A sends its output to the operator B. Let s be the volume of state in bytes. That means that A retains s bytes of records that it has passed down to B since the last checkpoint. For simplicy, we only consider buffer state and ignore processing state

for the time being. When (and if) operator B fails, operator C takes over and receives s bytes from A, which must come from reading HDFS or some sort of network file system. Assuming HDFS bandwidth to be h-bytes/sec, a checkpoint-based recovery scheme such as used in Apache Storm will take:

$$R_c = \frac{s}{h}. \tag{5.1}$$

When implementing FP4S recovery instead of checkpointing in the DAG, the buffer state of node A is periodically backed up in its leaf set nodes. Note that s bytes of buffer state is first split into m block (i.e., each being s/m bytes), that are then erasure-coded into n blocks stored in n leaf-set nodes, where $n > m$. Therefore, s bytes of buffer state requires sn/m bytes of storage in FP4S, leading to an overhead factor of $(n-m)/m$.

Note that although only s bytes of state is needed, C still issues for all n coded blocks in anticipation of any potential failures among the sending nodes. However, after m blocks are received correctly, C can ignore the remaining amount. Assuming network bandwidth of η-bytes/sec, this retrieval takes s/η seconds. After that, C can recompute s bytes of state from these coded blocks, say, at a rate c-bytes/second, which takes s/c seconds. Therefore, the recovery time of FP4S, denoted by R_f, is:

$$R_f = \frac{s}{\eta} + \frac{s}{c}. \tag{5.2}$$

We next consider the reliability aspect of the derived models, which is an important metric to consider because of the random node failure that can lead to some non-determinism. Assume that p is the probability for a node failure at any time. We also assume that node failure is a Poisson process, which means that previous failures do not affect the current failure.

When B fails in the DAG and another operator C needs at least m out of n leafset nodes of A to recover fully. That means, C will recover in a single hop of data transfer, which

we assume in our model, if $n-m$ or less nodes from A's leafset fail. Let the random variable X denote the event when this happens, i.e., C can recover using A's leafset nodes in a single hop, which requires at least m leafset nodes of A are alive at that moment. Note that if fewer leafset nodes are available, the DHT overlay will reorganize itself and provide functioning leafset nodes for A;s leafset. However, that would take more time and our model in (5.2) does not cover that. Therefore, our model reliability is given by:

$$P(X=1) = \sum_{i=0}^{n-m} \binom{n}{i} p^i (1-p)^{n-i}. \tag{5.3}$$

While there is no close-form solution to the above expression, we perform numerical evaluations with varying m, then varying n in range $[m, 2m]$, and also varying the the node failure probability p. Results are shown in Fig. 5.5.

Note that another way to interpret our model reliability equation in (5.3) is that it also works as a measure of FP4S's efficiency in that the failure recovery takes the minimum time when the right parameters m and k are chosen. Of course, FP4S will continue to work without such parameter tuning, although potentially sub-optimal.

Combining (5.1) and (5.2), we get:

$$R_c > R_f \Rightarrow \frac{s}{h} > \frac{s}{\eta} + \frac{s}{c} \Rightarrow h < \frac{c\eta}{c+\eta}. \tag{5.4}$$

We show the effect of (5.4) in Fig. 5.6, where we compare the maximum allowed HDFS bandwidth h against FP4S's recompute rate c. The goal is to see up to what speed of HDFS it is still viable to use FP4S for a given network bandwidth η. It is clear that for the most realistic values of c, η and h, FP4S is the preferred choice in terms of performance.

5.3.3 Instrumentation requirements

Here we describe the instrumentation requirements FP4S imposes and discuss the issues we encountered when integrating it with the Apache Storm processing engine.

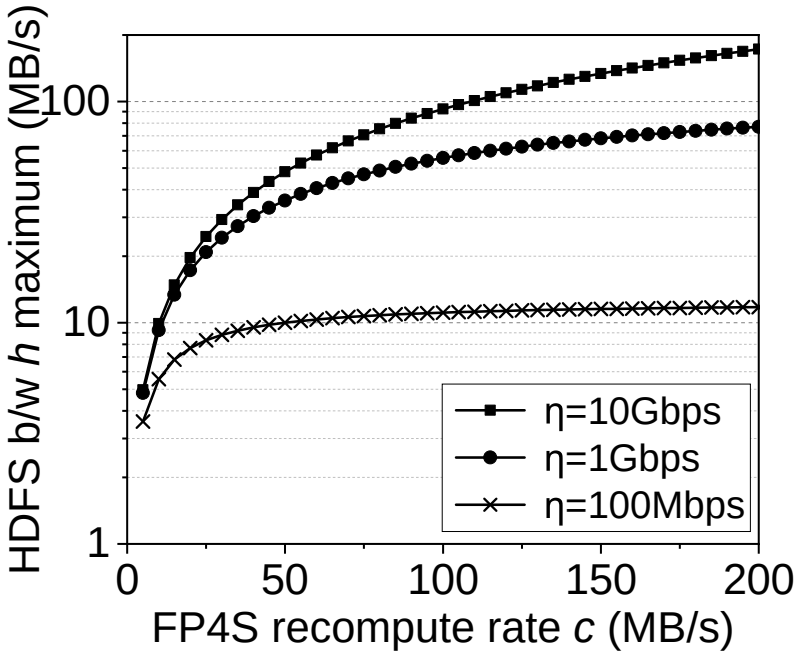


Figure 5.6: Maximum limit for HDFS rate vs Recompute rate c in FP4S.

In Apache Storm [sto], stream processing applications are deployed and executed as *topologies*. The *topologies* contain the business logics. These logics are formed as a DAG (directed acyclic graph) and implemented by *spouts* and *bolts*. *Spouts* are the data sources of the stream, which accept input data from raw data sources like Twitter Streaming API [as], Apache Kafka queue [ae], etc. *Bolts* are the logical processing units. *Spouts* pass data to bolts and *bolts* process and produce a new output stream. `IRichBolt` is the common interface for implementing *bolts*.

FP4S interacts with the `IRichBolt` interface in Storm [sto]. If FP4S is enabled, FP4S periodically saves the states into the DHT-based ring overlay for all stateful operators (*bolts*). For record-at-a-time systems like Storm, saving every operator's state may incur a lot of overhead. Instead, we aggregate the states for all the operators except for sources (*spouts*) and *sinks*. The aggregated state size is configurable in order to satisfy different real-world stream applications' requirements. After the size reaches a certain threshold, the `Encode` function encodes the states into fragments and the `Save` function puts these

fragments into the DHT-based overlay. If any node fails, the leaf set nodes call the routines to `Retrieve` and `Recompute` states on failover nodes. Any qualified available subset of fragments will be sufficient to recover the lost states by the `Recompute` function.

5.4 Evaluation

We integrate FP4S with Apache Storm and evaluate it using large scale real-world experiments, demonstrating its scalability, efficiency, and fast failure recovery. Experimental evaluations answer the following questions:

- How does the FP4S-enabled Storm system scale with the size of state, the number of concurrently running applications and the number of simultaneously failed operators?
- How does the efficiency of the fragment-based parallel state recovery algorithm change with different parameters, e.g., the number of the raw fragments (m) and the number of the coded fragments (n), and how does FP4S balance the workload?
- What are the performance and functionality benefits of FP4S compared to the state-of-the-art solutions?
- What is the overhead and the instrumentation used by FP4S?

5.4.1 Setup

We run all FP4S experiments on up to 4 machines, each with 16 Intel Xeon Gold 6130@2.10GHz cores and 256GB of RAM, running GNU/Linux 3.10.0. On top of these machines, we boot up 50 virtual machines to host 650 stream operators in total, each with 4 cores and 8GB of memory, running Linux Ubuntu 4.4.0. We use Apache Storm 2.0.0 [ah] configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36).

Table 5.2: Real-world application's dataset.

Application	Dataset	Size
Trending Topics	Twitter Streaming API [as]	>1TB
Bargain Index	Google Finance [al]	>1TB
Word Count	Project Gutenberg [ap]	8GB
	Wikimedia Dumps [at]	9GB
Traffic Monitoring	Dublin Bus Traces [aj]	4GB

We use Pastry 2.1 [ao] configured with leafset size of 24, max open sockets of 5000 and transport buffer size of 6MB.

To demonstrate generality across diverse computations and streaming operators. We deploy Yahoo streaming benchmarks [CDE+16] and real-world stream applications to FP4S (see Table 5.2).

These stream applications contain various representative streaming operators: stateless streaming transformations (e.g., `map`, `filter`), stateful operators (e.g., `incremental join`), and various window operators (e.g., sliding window, tumbling window and session window). We compare FP4S with a state-of-the-art failure recovery solution: the checkpointing recovery approach commonly used in TimeStream [QHS+13], Storm [sto], and Trident [ai]. We were not able to compare with Drizzle [VPO+17] because its source code is not publicly available. We choose the checkpointing recovery approach as the baseline approach because the replication recovery already costs twice the hardware and the DStream-based lineage recovery approach is not generalized because it sacrifices programming model transparency by forcing programmers to declare and maintain state using Spark's RDDs [ZCD+12].

The base value of raw fragments (m) and the coded fragments (n) are derived from production systems such as Pond [REG+03] and Sia [ar], which set $m = 16$, $n = 16$ and $m = 10$, $n = 20$ respectively. To fully evaluate the FP4S performance, we vary the values of m, n and the input state size.

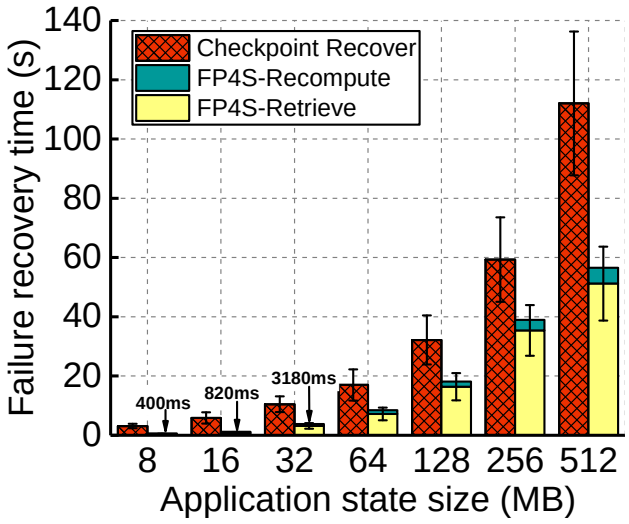


Figure 5.7: State recovery time for different input state sizes.

5.4.2 FP4S vs Checkpointing Recovery

We compare the failure recovery time of FP4S with the checkpointing recovery by varying the size of the state and the number of concurrently running stream applications.

The FP4S fragmented parallel recovery process consists of two steps: saving the state to leaf set nodes in the DHT-based overlay, and recomputing the state if any failure happens. Similarly, the checkpointing recovery process also consists of two steps: checkpointing the state to the HBase [ad] or HDFS [ac], and retrieving the state from HBase or HDFS if failure happens. Note that, for both approaches, the first step can run asynchronously with the second step so the first step may not impact the failure recovery time if they are executed in a pipeline.

Failure recovery time. Figure 5.7 shows the failure recovery time comparison of FP4S vs Storm's checkpointing recovery. In this experiment, we focus on a single stream application that has only one operator failure and we vary the state size. As Figure 5.7 shows, FP4S achieves 40.3% to 87.1% less failure recovery time compared to Storm's checkpointing recovery. The improvement gap increases as the size of the state increases. The rationale behind the result lies in that FP4S fragmented parallel recovery involves

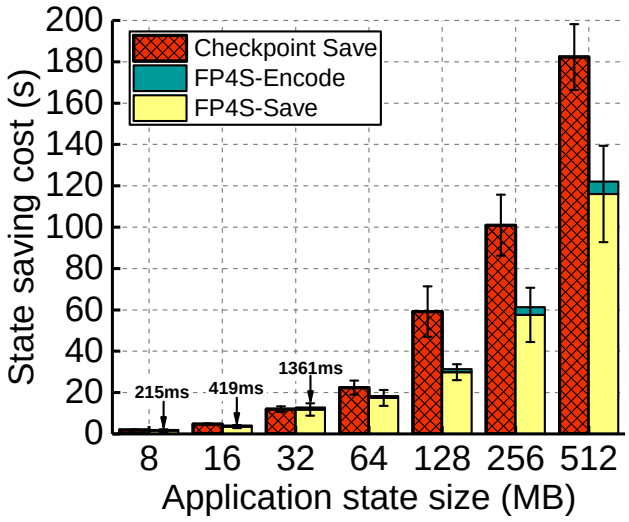


Figure 5.8: State saving time for different input state sizes.

many nodes to help re-compute the state in parallel which significantly reduces the failure recovery time. Instead, the checkpointing recovery only relies on a single node to retrieve the state which is constrained by the HBase I/O rate and the network bandwidth.

State saving cost. Figure 5.8 shows the state saving cost comparison of FP4S vs Storm's checkpointing recovery. The FP4S state saving cost includes the time cost for dividing the state into fragments, encoding each state, and then writing the encoded fragments into leaf set nodes. We write them into the leaf set nodes serially to enable a fair comparison with the checkpointing recovery. As Figure 5.8 shows, the state saving cost of FP4S is less than the state saving cost of the checkpointing recovery especially for large state because it runs in parallel with the operators execution.

Scale with the number of applications. Figure 5.9 shows the total failure recovery time for FP4S and Storm's checkpointing recovery when there are a large number of concurrently running stream applications on the platform. We set the failure rate of stream operators to be 1% according to Zorro [PLGC15]. As Figure 5.9 shows, compared to Storm's checkpointing recovery that has linearly increasing state recovery time, FP4S can handle many simultaneous failures with relatively stable state recovery time. This

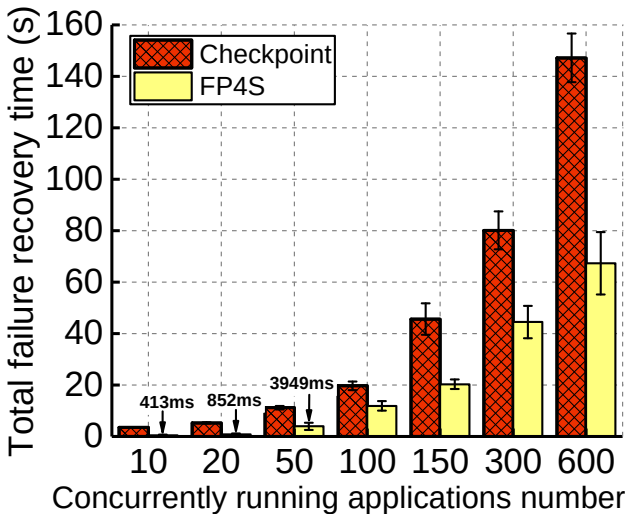


Figure 5.9: Total failure recovery time by varing # concurrently running stream applications.

is because FP4S leverages the DHT consistent ring overlay to distribute the total failure recovery load across all participating nodes and thus simultaneously failing operators' states can be recovered in parallel, which significantly improves the scalability.

5.4.3 Fragmented Parallel Recovery Algorithm

Several important factors affect the recovery performance, including the number of the raw fragments m in a state, the number of the parity fragments k in a state, the number of unavailable blocks e in a state and the amount of leaf nodes.

Number of the raw fragments (m). We evaluate the impact of the number of the raw fragments m in a state on the recovery performance by varying m from 11 to 20, where k is set to be 10. Figure 5.10 plots the performance of state recomputing time when recovering from single failure by varying m. We can observe that the state recomputing time increases as the number of raw fragments m increases. The reason lies in that the recovery time of FP4S is mainly determined by $mB/(m+k-1)$, where B is the amount of data that any providing peer uploads. $mB/(m+k-1)$ increases with the increases of

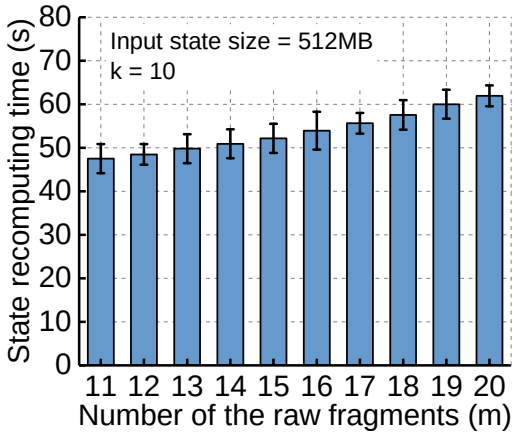


Figure 5.10: Adjust raw fragment (m) parameter.

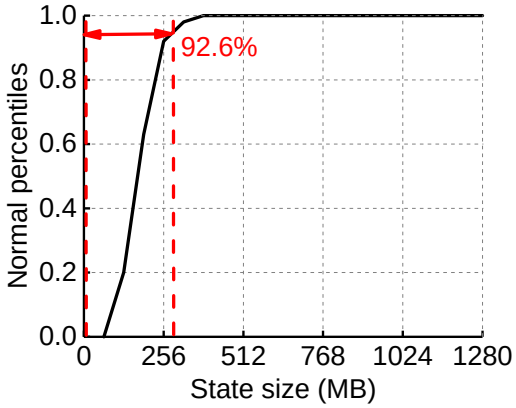


Figure 5.11: Normal probability plot of the state size across all DHT nodes.

m when the values of k and B are given. Thus, the performance of FP4S is more sensitive to m when k is smaller.

Number of the parity fragments (k). We evaluate the impact of the number of the parity fragments k in a state on recovery performance by varying k from 11 to 20, where m is set to be 10. Figure 5.12 plots the performance of state recomputing time when recovering from single failure by varying k. We can observe that FP4S achieves better recovery performance when k is increasing from 11 to 20. The reason is that the recovery time of FP4S is mainly determined by by $mB/(m+k-1)$, where B is the amount of data

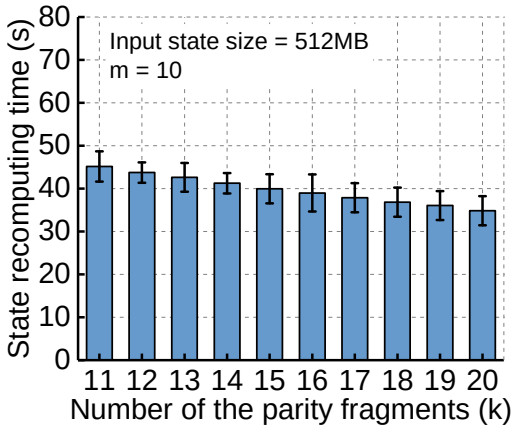


Figure 5.12: Adjust parity fragment (k) parameter.

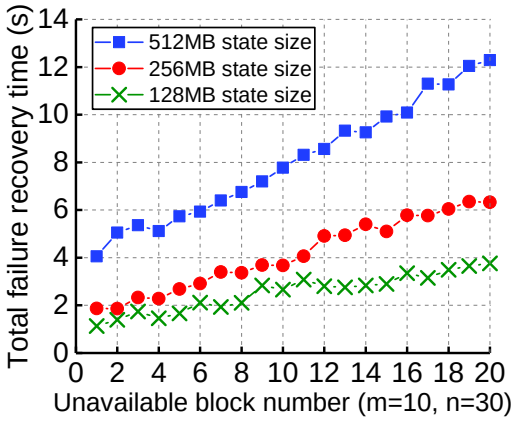


Figure 5.13: Adjust unavailable block number (e).

that any providing peer uploads. $mB/(m+k-1)$ decreases with the increases of k when the values of m and B are given.

Number of unavailable blocks (e). We evaluate the impact of the number of unavailable blocks e on recovery performance by varying e from 1 to 20. We can see from Figure 5.13 that the total failure recovery time of FP4S increases linearly with the increase of e. This is due to the current star-like structure of the FP4S prototype, in which the performance bottleneck of FP4S is mainly the upload speeds of providing peers. Thus, the recovery performance of FP4S is inversely proportional to the amount of data a providing peer uploads. In the future, we plan to enable fragments to be transmitted and combined

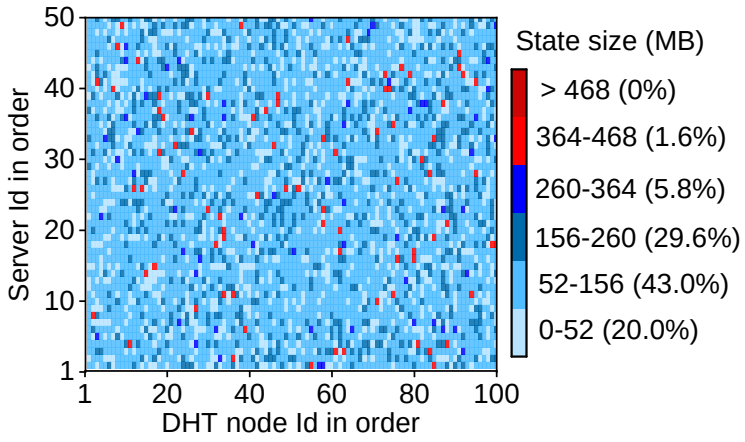


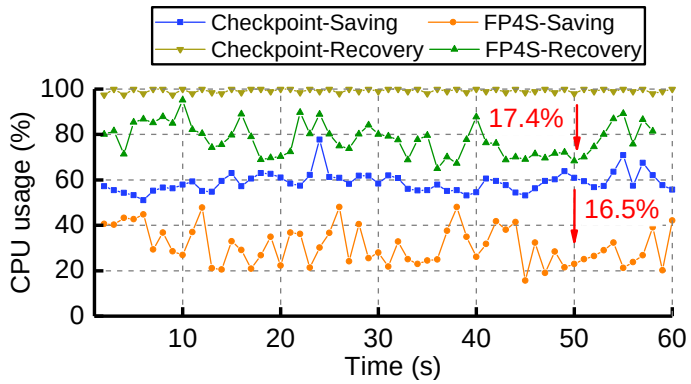
Figure 5.14: Heatmap of the state size across all DHT nodes.

through a spanning tree covering the replacing node and all providing nodes to mitigate this performance bottleneck.

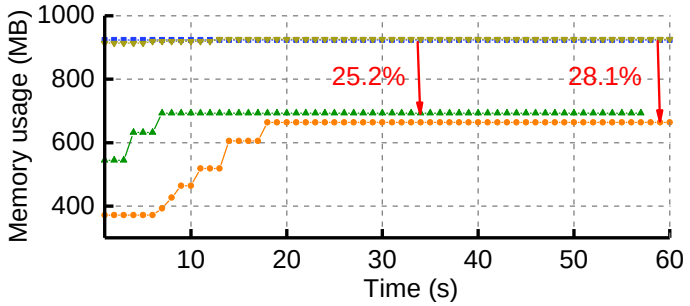
5.4.4 Load Balance

FP4S has an attractive load balance feature because it assigns each operator a non-overlapping set of leaf set nodes, which distributes the total load of state saving and recovery all over the overlay. We evaluate the load balance of FP4S by running 500 stream applications on the platform of 50 virtual servers that have 5000 DHT nodes. Each application has 512 MB state and we have 780GB states in total that need to be saved in the above 5000 nodes.

Figure 5.14 plots the heatmap of the state size on each DHT node. Figure 5.11 shows the normal probability plot for the size of stored state per node. We can observe that over 90% (20%+43%+29.6%) of nodes store state with less than 260 MB (see the blue zone in Figure 5.14), while less than 7.5% nodes store state over 260 MB, demonstrating its attractive load balance and scalability features.



(a) The CPU overhead.



(b) The memory overhead.

Figure 5.15: The overhead analysis of the FP4S-enabled Storm at runtime.

5.4.5 Overhead Analysis

We evaluate FP4S runtime overhead, particularly those pertaining to its saving and recovery execution, and compare them with the checkpointing recovery approach. The FP4S saving and recovery require additional CPU to compute the fragments and additional memory for maintaining intermediate results. Figure 5.15 presents these costs, explained next.

CPU overhead. Figure 5.15a shows the per-node CPU runtime overhead comparison of FP4S vs checkpointing recovery. The CPU runtime overhead of FP4S is less than the checkpointing recovery. This is because fragment calculations account for only a small fraction ($<$10%) of the entire save and restore execution.

Memory overhead. Figure 5.15b shows the per-node memory run-time overhead comparison of FP4S vs checkpointing recovery. The memory overhead of FP4S is less than the checkpointing recovery. This is because the checkpointing recovery involves a centralized daemon process such as Zookeeper for coordination. Instead, FP4S nodes are independent, which does not require the centralized daemon process to maintain these relationships.

5.5 Summary

In this paper we have described and evaluated FP4S, a novel fragment-based parallel state recovery mechanism that can handle many simultaneous failures for stateful stream applications. Unlike existing failure recovery approaches based on replication or checkpointing which are either slow, resource-expensive or fail to handle many simultaneous failures, FP4S leverages DHTs and erasure codes to divide each operator's in-memory state into many fragments and periodically save them in each node's leaf set nodes in a DHT ring, ensuring that different sets of available fragments can reconstruct failed state in parallel. By doing that, this failure recovery mechanism is scalable to the size of the lost state, which significantly reduces the failure recovery time and can tolerate many simultaneous operator failures.

FP4S is framework-agnostic and broadly applicable to a range of streaming systems. We have implemented FP4S atop the state-of-the-art stream processing engine Apache Storm, and demonstrated its scalability, efficiency, and fast failure recovery features that incur negligible instrumentation overheads. Note that dividing data blocks, encoding raw fragments, uploading encoded fragments, and reconstructing states are non-overlapping operations, so many interesting questions for future work arise: such as how to pipeline them to speed up the recovery process? How to adjust the ratio of m and n to tradeoff

the reduced upload data and the increased computation complexity? How to provide a theoretic model to estimate network I/O cost and computation complexity?

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

How should we provide and optimize a system that support next-generation distributed data processing in cloud? This dissertation presents a possible solution perspective: a distributed and easy-to-use system, which defined processing abstract, instantly collaborates with the processing data information, can provide useful knowledge that people need and promising performances in functionality.

In this dissertation, three research works are introduced from different perspectives to optimize distributed application in cloud environment. More specifically, they mainly focus on the system-level design in supporting scalable, online, and efficient system.

Specifically, DocMan, a toolset that adopts a black-box approach to discover container ensembles and collect information about intra-ensemble container interactions. It uses a combination of techniques such as distance identification and hierarchical clustering.

For storage optimization, we analyze the possibility to improve the HDFS data blocks replication mechanism. The caching strategy MemCached is leveraged in the design to effectively replicate the popular blocks in memory. The experimental results show that the proposed design is able to improve the HDFS based applications' performance from different perspectives and has the potential to overcome the critical issues of traditional Big Data storage systems.

FP4S is framework-agnostic and broadly applicable to a range of streaming systems. We have implemented FP4S atop the state-of-the-art stream processing engine Apache Storm, and demonstrated its scalability, efficiency, and fast failure recovery features that incur negligible instrumentation overheads.

In conclusion, the first work, a system named DocMan, is introduced to support application dependencies detection in cloud. The second work proposed a solution to optimize distributed applications' storage. The third work is a system named FP4S, is introduced to achieve applications state failure recovery by using the erasure code method. They correspond to specific areas but interrelated functions of next-generation distributed data processing applications in cloud environment.

6.2 Lessons Learned

Non-intrusive approach. To fulfill the same purpose, a non-intrusive approach should be considered in advance of intrusive approach. The system, DocMan, is used to detect dependencies. We implement the non-intrusive approach via capturing the CPU, memory and I/O logs. The rationale behind our approach is that we observed that real-world containerized multi-tier applications exhibited strong correlation among their resource usage statistics. It makes the approach having a wide usage instead of just working for a specific application.

Handle real problem scenarios. When we consider solving a problem, we need to know if the problem is the same or similar to the real scenarios. For example, when we do the project for failure recovery, if we only consider to recover a state at a time, it will be quite different from the real scenario. Then the following optimization will not make much sense. So we solve the problem based on the problem of how to scale recovery with the size of the state, the number of simultaneous failures and the number of concurrently running stream applications on a shared platform, which makes the problem the same as a real problem that is happening in the industrial world.

Imposing low hardware cost. When we archive a target, we cannot ignore the hardware cost, otherwise it will make the solution not realistic. For example, state-of-the-art

data processing systems offer failure recovery mainly through three approaches: replication recovery, checkpointing recovery and DStream-based lineage recovery, which are either slow, resource-expensive or fail to handle many simultaneous failures. Replication recovery adds significant hardware cost because multiple copies must concurrently run on distinct nodes for failover. Checkpointing recovery is known to be prohibitively expensive, and users in many domains disable it as a result. DStream-based lineage recovery is slow when the lineage graph is long and falls short in handling multiple simultaneous failures, so we propose the solution FP4S in a lightweight manner.

6.3 Broader Impact

In this dissertation, the three proposed systems are focus on different optimization perspectives. However, they are inter-connected, which can be implemented into distributed applications platform together or partially. They are framework-agnostic and broadly applicable to a range of distributed systems. The source code and presentations of the proposed systems are public to the community, which will make they have a broader impact.

6.4 Future Work

Many open questions and challenges are needed further consideration and research efforts in supporting scalable, efficient, and distributed data applications, as shown in the following:

For dependencies detection, DocMan's methods are fully implemented, but additional work is required for using it to continuously detect and manage containers at cloud-scale. For example, we need to filter out background traffic noises (i.e., heartbeat packets), since

such traffic might otherwise be interpreted as intra-ensemble communications. Further, it would be interesting to integrate DocMan into management solutions like Kubernetes and Docker Swarm. We also plan to leverage DocMan's insights to directly guide the container placement, thus improving containerized application's performance and amortizing the expenses related to their debugging and maintenance.

For storage optimization, there might be different levels of content access speed instead of just memory and hard disk. For example, the hard disk access speed may vary from 1.7MBps to 520MBps for HDD and SSD. For different types of memory also have this kind of variation. To accurately optimize these different types of storage, we need to define more complicated algorithms to cater to these scenarios.

For FP4S, note that dividing data blocks, encoding raw fragments, uploading encoded fragments, and reconstructing states are non-overlapping operations, so many interesting questions for future work arise: such as how to pipeline them to speed up the recovery process? How to adjust the ratio of m and n to trade off the reduced upload data and the increased computation complexity? How to provide a theoretic model to estimate network I/O cost and computation complexity?

We hope that the continuous research experience of system support in distributed applications can help us solve these problems and challenges. We also wish more research efforts can be conducted to design innovative next-generation cloud distributed systems.

BIBLIOGRAPHY

[0] Cloud computing trends: 2018 state of the cloud survey. https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2018-state-cloud-survey.

[aa] Apache flink. http://flink.apache.org/.

[ab] Apache flume. http://flume.apache.org/.

[ac] Apache hadoop hdfs. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/.

[ad] Apache hbase. http://hbase.apache.org/.

[ae] Apache kafka. http://kafka.apache.org/.

[af] Apache samza. http://samza.apache.org/.

[ag] Apache spark. http://spark.apache.org/.

[ah] Apache storm 2.0.0. https://storm.apache.org/2019/05/30/storm200-released.html.

[ai] Apache trident. http://storm.apache.org/releases/current/Trident-tutorial.html.

[aj] Dublin bus gps sample data from dublin city council. https://data.gov.ie/dataset/.

[ak] Faroo. https://en.wikipedia.org/wiki/FAROO.

[al] Google finance data api. http://finance.google.com/finance/feeds/.

[am] Leveldb. https://github.com/google/leveldb/.

[an] Mongodb. http://www.mongodb.com/.

[ao] Pastry. https://www.freepastry.org/FreePastry/.

[ap] Project gutenberg. http://www.gutenberg.com/.

[aq] Rocksdb. http://rocksdb.org/.

[ar] Sia: a decentralized storage platform secured by blockchain technology. http://sia.tech/.

[as] Twitter streaming apis. https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data.

[at] Wikimedia dumps. https://dumps.wikimedia.org/.

[A+10] Jonathan Appavoo et al. Providing a cloud network infrastructure on a supercomputer. In *19th ACM International Symposium on High Performance Distributed Computing*, pages 385–394. ACM, 2010.

[A+11] Ganesh Ananthanarayanan et al. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proc. 6th conf. on Computer systems*, pages 287–300. ACM, 2011.

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[ABB+16] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[ACÇ+03] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[AFRR+10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[AIR17] Alexandru Agache, Mihai Ionescu, and Costin Raiciu. Cloudtalk: Enabling distributed application optimisations in public clouds. EuroSys '17, pages 605–619, New York, NY, USA, 2017. ACM.

[ALC11] Cristina L Abad, Yi Lu, and Roy H Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *CLUSTER*, pages 159–168. Ieee, 2011.

[Ama] Amazon ec2 container service. `https://aws.amazon.com/ecs/`.

[AO15] Hnin Htet Htet Aung and Nyein Nyein Oo. Edas: Efficient data access scheme of data replication for hadoop distributed file system (hdfs). pages 177–183, 2015.

[B+08] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53, 2008.

[B+15] Dinh-Mao Bui et al. Replication management framework for hdfs based on prediction technique. In *Advanced Cloud and Big Data*, pages 58–63. IEEE, 2015.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2005.

[BDIM04] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[big] Bigdatabench. http://prof.ict.ac.cn/.

[C+12] Zhendong Cheng et al. Erms: An elastic replication management system for hdfs. In *CLUSTER WORKSHOPS*, pages 32–40. IEEE, 2012.

[CBM+17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th SOSP*, pages 153–167. ACM, 2017.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Cidr*, volume 2, page 4, 2003.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[CEF+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[cen] Centralized cache management in hdfs. https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html.

[CKF+02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[CKS13] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. *SIGCOMM Comput. Commun. Rev.*, 43(4):231–242, August 2013.

[clu] Cluster. https://en.wikipedia.org/wiki/Cluster_analysis.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[CS12] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[CZM+11] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[CZS14] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DL12] S. Ding and Y. Li. Lru2-mru collaborative cache replacement algorithm on multi-core system. In *CSAE*, volume 2, pages 395–398, May 2012.

[DL15] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Advances in neural information processing systems*, pages 3079–3087, 2015.

[EPPB11] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. SOSP '11, pages 311–326, New York, NY, USA, 2011. ACM.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[FPR+10] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. SIGCOMM '10, pages 339–350, New York, NY, USA, 2010. ACM.

[GDJ07] Philip Golden, Hervé Dedieu, and Krista S Jacobsen. *Implementation and applications of DSL technology*. CRC press, 2007.

[GHJ+09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[goo] Google cloud platform container. `https://cloud.google.com/container-engine`.

[GXD+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th Symposium on Operating Systems Design and Implementation*, pages 599–613, 2014.

[had] Hadoop. `http://hadoop.apache.org`.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[HLL16] Zhiming Hu, Baochun Li, and Jun Luo. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[HLM13] Fabien Hermenier, Julia Lawall, and Gilles Muller. Btrplace: A flexible consolidation manager for highly available applications. *IEEE Trans. Dependable Secur. Comput.*, 10(5):273–286, September 2013.

[HSG+12] Liting Hu, Karsten Schwan, Ajay Gulati, Junjie Zhang, and Chengwei Wang. Net-cohort: Detecting and managing vm ensembles in virtualized data centers. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 3–12, New York, NY, USA, 2012. ACM.

[HSK+12] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[I+14] NS Islam et al. In-memory i/o and replication for hdfs with memcached: Early experiences. in 2014 ieee intl. In *IEEE BigData*, 2014.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[ipe] Iperf. `https://iperf.fr`.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 190–201. ACM, 2000.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[KC13] A Kala Karun and K Chitharanjan. A review on hadoop—hdfs infrastructure extensions. In *ICT*, pages 132–137. IEEE, 2013.

[KLM+10] Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Lindsay Munroe, Sebastian Schrittwieser, Mayank Sinha, and Edgar Weippl. Qr code security. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, pages 430–435. ACM, 2010.

[kub] Kubernetes. `https://kubernetes.io`.

[LLP+12] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and An-Hai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[LMC15] Mingyong Li, Yan Ma, and Meilian Chen. The dynamic replication mechanism of hdfs hot file based on cloud storage. *International Journal of Security and Its Applications*, 9(8):439–448, 2015.

[LSD+16] Chaochun Liu, Huan Sun, Nan Du, Shulong Tan, Hongliang Fei, Wei Fan, Tao Yang, Hao Wu, Yaliang Li, and Chenwei Zhang. Augmented lstm

framework to construct medical self-diagnosis android. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 251–260. IEEE, 2016.

[LW14] Xueping Liu and Ji Wang. The study on capacity enhancement of distributed systems cloud services. In *LEMCS*. Atlantis Press, 2014.

[LZC14] Sai-Qin Long, Yue-Long Zhao, and Wei Chen. Morm: A multi-objective optimized replication management strategy for cloud storage cluster. *Journal of Systems Architecture*, 60(2):234–244, 2014.

[map] Mapreduce. `https://en.wikipedia.org/wiki/MapReduce`.

[MBMDS10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[mica] Micro-service. `http://microservices.io/`.

[micb] Microsoft azure container service. `https://azure.microsoft.com/en-us/services/container-service/`.

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[MPZ10] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. INFOCOM'10, pages 1154–1162, Piscataway, NJ, USA, 2010. IEEE Press.

[MYAFM10] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. NSDI'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[N+08] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[N+13] Rajesh Nishtala et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[NPP+17] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[num] Numpy. `http://www.numpy.org`.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. 2010.

[PLGC15] Mayank Pundir, Luke M Leslie, Indranil Gupta, and Roy H Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 195–208. ACM, 2015.

[PMC] Pearson correlation coefficient. `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`.

[PSLJ11] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM.

[QHS+13] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[r] R project. `https://www.r-project.org/about.html`.

[Ran20] Steve Ranger. What is cloud computing? `https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/`, 2020.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[red] Redis. `https://redis.io`.

[REG+03] Sean C Rhea, Patrick R Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, volume 3, pages 1–14, 2003.

[RS60] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[rub] Rubis. `http://rubis.ow2.org`.

[S+10] Konstantin Shvachko et al. The hadoop distributed file system. In *MSST*, pages 1–10. IEEE, 2010.

[SCW+12] Dawei Sun, Guiran Chang, Dongqi Wang, Dong Chen, and Xingwei Wang. Modeling and Managing Energy Efficiency Data Center by a Live Migration Mechanism in Mobile Cloud Computing Environments. *Sensor Letters*, 10(8):1855–1861, 2012.

[SGH+16] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 6. ACM, 2016.

[SHB04] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[spa] Spark. `https://spark.apache.org`.

[SRC10] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *ISPASS*, pages 122–133. IEEE, 2010.

[sto] Storm. `http://storm.apache.org`.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[TAB13] Radu Tudoran, Gabriel Antoniu, and Luc Bouge. Sage: geo-distributed streaming data analysis in clouds. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 2278–2281. IEEE, 2013.

[TTZ+09] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. USENIX'09, pages 19–19, Berkeley, CA, USA, 2009. USENIX Association.

[VPO+17] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[VRMB11] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[W+10] Qingsong Wei et al. Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster. In *CLUSTER*, pages 188–196. IEEE, 2010.

[War63] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory

cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

VITA

PINCHAO LIU

	Born, Tianjin, China
2004-2008	B.A., Computer Science and Technology Tianjin University of Commerce. Tianjin, China
2008-2011	M.S., Computer Science Tianjin University of Science and Technology Tianjin, China
2011-2016	Software Engineer Standard Chartered Bank Tianjin, China
2016-2021	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS

1. Pinchao Liu, Liting Hu, Hailu Xu, Zhiyuan Shi, Jason Liu, Qingyang Wang, Jai Dayal, and Yuzhe Tang, "A Toolset for Detecting Containerized Application's Dependencies in CaaS Clouds", *2018 IEEE International Conference on Cloud Computing (IEEE CLOUD), June 2018*.
2. Pinchao Liu, Adnan Maruf, Farzana Beente Yusuf, Labiba Jahan, Hailu Xu, Boyuan Guan, Liting Hu, and Sitharama S. Iyengar, Towards Adaptive Replication for Hot/Cold Blocks in HDFS using MemCached", *In Proceedings of 2019 International Conference on Data Intelligence and Security (ICDIS 2019), June 2019.*
3. Pinchao Liu, Hailu Xu, Dilma Da Silva, QingyangWang, Sarker Tanzir Ahmed, and Liting Hu. "FP4S: Fragment-based Parallel State Recovery for Stateful Stream Applications", *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2020)*.
4. Hailu Xu, Pinchao Liu, Susana Cruz-Diaz, Dilma Da Silva, and Liting Hu, SR3: Customizable Recovery for Stateful Stream Processing Systems, *In Proceedings of the 21st International Middleware Conference, December, 2020 (Middleware' 20).*
5. Boyuan Guan, Liting Hu, Pinchao Liu, Hailu Xu, Jennifer Fu, QingyangWang, dpSmart: A Flexible Group Based Recommendation Framework for Digital Repository Systems", *In Proceedings of the 2019 International Congress on Big Data (Big Data Congress), July 2019.*

6. Hailu Xu, Boyuan Guan, Pinchao Liu, William Escudero, and Liting Hu. "Harnessing the Nature of Spam in Scalable Online Social Spam Detection", *2018 IEEE Big Data workshop on Big Social Media Data Management and Analysis, in conjunction with IEEE Big Data, 2018.*

7. Hailu Xu, Liting Hu, Pinchao Liu, Yao Xiao,WentaoWang, Jai Dayal, Qingyang Wang and Yuzhe Tang, "Oases: An Online Scalable Spam Detection System for Social Networks", *2018 IEEE International Conference on Cloud Computing (IEEE CLOUD), June 2018.*

8. Hailu Xu, Liting Hu, Pinchao Liu, and Boyuan Guan, Exploiting the Spam Correlations in Scalable Online Social Spam Detection", *In Proceedings of the 2019 International Conference on Cloud Computing (CLOUD), June 2019.*