# A CONVERGENCE OF NOSQL STORAGE SYSTEMS FROM CLOUDS TO SUPERCOMPUTERS

BY

TONGLIN LI

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
October 2014

TABLE OF CONTENTS

Page

ABSTRACT . . . . . vii

CHAPTER

1. INTRODUCTION . . . . . 1

2. ZHT: A LIGHT-WEIGHT RELIABLE PERSISTENT DYNAMIC SCALABLE ZERO-HOP DISTRIBUTED HASH TABLE . . . 7

2.1 Background . . . . . 8

2.2 ZHT Design and Implementation . . . . . 12

2.2.1 Basic operations . . . . . 14

2.2.2 Terminologies . . . . . 16

2.2.3 Membership management . . . . . 18

2.2.4 Server architecture . . . . . 20

2.2.5 Hash Functions . . . . . 20

2.2.6 Fault Tolerance . . . . . 21

2.2.7 Socket level thread safe . . . . . 22

2.2.8 Consistency . . . . . 23

2.2.9 Consistency Persistence . . . . . 23

2.2.10 Implementation . . . . . 24

2.3 Performance Evaluation via Synthetic benchmarks . . . . . 25

2.3.1 Testbeds, Metrics, and Workloads . . . . . 25

2.3.2 Hash Functions . . . . . 27

2.3.3 Individual data store synthetic benchmark . . . . . 28

2.3.4 System Latencies . . . . . 28

2.3.5 Synthetic benchmarks On Supercomputers and clusters 28

Synthetic benchmarks On Cloud . . . . . 30

2.3.6 System Throughput . . . . . 34

Synthetic benchmarks On Supercomputers and clusters . . 34

Synthetic benchmarks On Cloud . . . . . 35

Throughput35 Running Cost36

2.3.7 Scalability and efficiency . . . . . 37

2.3.8 Aggregated performance . . . . . 39

2.4 ZHT as a Building Block for Distributed Systems . . . . . 40

2.4.1 FusionFS: a Distributed File System with Distributed Metadata Management . . . . . 40

2.4.2 IStore: an Erasure Coding Enabled Distributed Storage System . . . . . 41

2.4.3 MATRIX: a Distributed Many-Task Computing Scheduling Framework . . . . . 42

2.4.4 Slurm++: a Distributed HPC Job Launch . . . . . 43

2.4.5 ZDMQ: a Distributed Message Queue . . . . . 44

2.5 Related work . . . . . 45

2.6 Summary . . . . . 46

3. WAGGLEDB: A CLOUD-BASED INTERACTIVE DATA INFRASTRUCTURE FOR SENSOR NETWORK APPLICATIONS 48

3.1 Introduction . . . . . 49

3.2 Design and Implementation . . . . . 50

3.2.1 Challenges and solutions . . . . . 50

3.2.2 Architecture . . . . . 51

3.2.3 Implementation . . . . . 52

3.2.4 Performance Evaluation . . . . . 53

3.3 Summary . . . . . 54

4. SCALABLE STATE MANAGEMENT FOR SCIENTIFIC APPLICATIONS ON CLOUD . . . . . 55

4.1 Introduction . . . . . 56

4.2 Background . . . . . 58

4.2.1 Use cases . . . . . 58

4.2.2 Challenges . . . . . 59

4.2.3 FRIEDA framework . . . . . 59

4.3 System Design and Implementation . . . . . 60

4.3.1 State Description . . . . . 61

4.3.2 Static state capture . . . . . 62

4.3.3 Dynamic state capture . . . . . 62

4.3.4 State Storage . . . . . 63

4.3.5 Storage architectures . . . . . 64

4.3.6 Event reordering . . . . . 66

4.4 Evaluation . . . . . 67

4.4.1 Testbeds . . . . . 68

4.4.2 Scientific Workloads . . . . . 68

4.4.3 Experiment Setup . . . . . 68

4.4.4 Metrics . . . . . 69

4.4.5 Synthetic benchmark . . . . . 70

4.4.6 Scientific applications . . . . . 71

4.5 Related work . . . . . 73

4.5.1 Provenance . . . . . 73

4.5.2 Monitoring . . . . . 73

4.5.3 Key-value stores . . . . . 74

4.5.4 Unsynchronized Time Clocks and Event Ordering . . 75

4.6 Summary . . . . . 76

5. FUTURE WORK . . . . . 77

5.1 Collective request handling and buffering in Key-Value Storage: ZHT+ . . . . . 77

5.2 Integrate ZHT with Swift parallel scripting language . . . . 78

6. RESEARCH TIMELINE AND PUBLICATION SUBMISSION PLAN . . . . . 79

6.1 Timeline Milestones . . . . . 79

6.2 Publication submission plan . . . . . 79

BIBLIOGRAPHY . . . . . 80

ABSTRACT

This work presents a convergence of distributed NoSQL storage systems in clouds and supercomputers. It specifically presents ZHT, a zero-hop distributed key-value store system, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed systems, such as parallel and distributed file systems, distributed job management systems, and parallel programming systems. The goals of ZHT are delivering high availability, good fault tolerance, high throughput, and low latencies, at extreme scales (millions of nodes). ZHT has some important properties, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication, persistent, scalable, and supporting unconventional operations such as append, compare and swap, callback in addition to the traditional insert/lookup/remove. We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 64-nodes, an Amazon EC2 virtual cluster up to 96-nodes, to an IBM Blue Gene/P supercomputer with 8K-nodes. Using synthetic benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput on Blue Gene/P. On Amazon EC2 cloud, on 96-node scale it offers 0.8ms latency and 1.2M ops/s throughput.

We compared ZHT against other key/value stores (e.g. Casandra, Memcached, DynamoDB) and found it offers superior performance for the features and portability it supports. This work also presents several real systems that have adopted ZHT as well as other NoSQL systems, namely FusionFS (distributed metadata management and data provenance capture/query), IStore (data chunk metadata management), MATRIX (distributed scheduling), Slurm++ (distributed HPC job launch), ZDMQ (distributed message queue management), FREIDA-State (state management for scientific applications on cloud), and WaggleDB (a Cloud-based interactive data infras-

tructure for sensor network applications); all of these real systems have been simplified due to NoSQL storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. It's important to highlight that some of these systems are rooted in HPC systems from supercomputers, while others are rooted in clouds and ad-hoc distributed systems; through our work, we have shown how versatile NoSQL storage systems can be in such a variety of environments. This work will also outline planed future work, in terms of hierarchical architectures for ZHT, and the integration of ZHT into the Swift parallel programming system.

# CHAPTER 1

# INTRODUCTION

Today's science is generating datasets that are increasing exponentially in both complexity and volume, making their analysis, archival, and sharing one of the grand challenges of the 21st century. As supercomputers gain more parallelism at exponential rates, the storage infrastructure performance is increasing at a significantly lower rate. This implies that the data management [109, 108, 94, 107, 98, 130] and data flow between the storage and compute resources is becoming the new bottleneck for large-scale applications. The support for data intensive computing is critical to advancing modern science as storage systems have experienced a gap between capacity and bandwidth that increased more than 10-fold over the last decade. There is an emerging need for advanced techniques to manipulate, visualize and interpret large datasets. Many domains (e.g. astronomy, bioinformatics, and financial analysis) share these data management challenges, strengthening the potential impact from generic solutions.

"A supercomputer is a device for turning compute-bound problems into I/O bound problems"[14]. The quote from Ken Batcher reveals the essence of modern high performance computing and implies an ever-growing shift in bottlenecks from compute to I/O. For exascale computers, the challenges are even more radical, as the only viable approaches in next decade to achieve exascale computing all involve extremely high parallelism and concurrency. Up to 2014, some of the biggest systems already have more than 3 million general-purpose cores. Many experts predict that exascale computing will be a reality by the end of the decade; an exascale system is expected to have millions of nodes, billions of threads of execution, hundreds of petabytes of memory, and exabyte of persistent storage.

In the current decades-old architecture of HPC systems, storage is completely

segregated from the compute resources and are connected via a network interconnect (e.g. parallel file systems running on network attached storage, such as GPFS[83], PVFS[18] and Lustre[84]). This approach is not able to scale several orders of magnitude in terms of concurrency and throughput, and will thus prevent the move from petascale to exascale. If we do not solve the storage problem with new storage architectures, it could be a "show-stopper" in building exascale systems. The need for building efficient and scalable distributed storage for high-end computing (HEC) systems that will scale three to four orders of magnitude is on the horizon.

One of the major bottlenecks in current state-of-the-art storage systems is metadata management. Metadata operations on parallel file systems can be inefficient at large scale. Experiments on the Blue Gene/P system at 16K-core scales show the various costs (wall-clock time measured at remote processor) for file/directory create on GPFS. Ideal performance would have been constant, but we see the cost of these basic metadata operations (e.g. create file) growing from tens of milliseconds on a single node (four-cores), to tens of seconds at 16K-core scales; at full machine scale of 160K-cores, we expect a file create to take over two minutes for the many directory case, and over 10 minutes for the single directory case. Previous work shows these times to be even worse, putting the full system scale metadata operations in the hour range, but the test bed as well as GPFS might have been improved over the last several years. Whether the time per metadata operation is minutes or hours on a large-scale HEC system, the conclusion is that the distributed metadata management in GPFS does not have enough degree of distribution, and not enough emphasis was placed on avoiding lock contention. GPFS's metadata performance degrades rapidly under concurrent operations, reaching saturation at only 4 to 32 core scales (on a 160K-core machine).

Other distributed file systems (e.g. Google's GFS and Yahoo's HDFS for

Hadoop) that have centralized metadata management make the problems observed with GPFS even worse from the scalability perspective. Future storage systems for high-end computing should support distributed metadata management, leveraging distributed data-structure tailored for this environment. The distributed data-structures share some characteristics with structured distributed hash tables, having resilience in face of failures with high availability; however, they should support close to constant time inserts/lookups/removes delivering the low latencies typically found in centralized metadata management (under light load). Metadata should be reliable and highly available, for which replication (a widely used mechanism) could be used.

HPC storage is not the only area that suffers the storage bottleneck. Similar with the HPC scenarios, cloud based distributed systems also have to face storage bottleneck. Furthermore, due to the dynamic nature of cloud applications, a suitable storage system needs to satisfy more requirements.

As an initial attempt to meet these needs, we propose and build ZHT(zero-hop distributed hash table [48, 46, 47, 15, 43], an instance of NoSQL database[33], which has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies, and scientific computing data-access patterns). ZHT aims to be a building block for future distributed systems, with the goal of delivering excellent availability, fault tolerance, high throughput, scalability, persistence, and low latencies. ZHT has several important features making it a better candidate than other distributed hash tables and key-value stores, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication and by handling failures gracefully and efficiently propagating events throughout the system, a customizable consistent hashing function, supporting persistence for better recoverability in case of faults, scalable, and supporting unconventional operations such as append (providing lock-free con-

current key/value modifications) in addition to insert/lookup/remove. To provide ZHT a persistent back end, we also created a fast persistent key-value store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers as well as clouds. We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 512-cores, to an IBM Blue Gene/P supercomputer with 160K-cores. Using micro-benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput. We compared ZHT against two other systems, Cassandra [40] and Memcached [72] and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes. We also compared it to DynamoDB[4] in the Amazon AWS Cloud, and found that ZHT offers significantly better performance and economic cost than DynamoDB.

This work also presents several real systems that have adopted ZHT as well as other NoSQL systems, namely FusionFS [127][121](distributed metadata management and data provenance capture/query), IStore[122] (data chunk metadata management), MATRIX (distributed scheduling), Slurm++(distributed HPC job launch), ZDMQ (distributed message queue management), FREIDA-State [45](state management for scientific applications on cloud), and WaggleDB[44](a Cloud-based interactive data infrastructure for sensor network applications); all of these real systems have been simplified due to NoSQL storage systems, and have been shown to outperform other leading systems by orders of magnitude in some cases. It's important to highlight that some of these systems are rooted in HPC systems from supercomputers, while others are rooted in clouds and ad-hoc distributed systems; through our work, we have shown how versatile NoSQL storage systems can be in such a variety of environments.

The contributions of this work are as follows:

- Design and implementation of ZHT and optimized for high-end computing
- Verified scalability on 32K-cores scale
- Prove that Distributed NoSQL key/value storage systems that are light-weight, dynamic, resilient, portable, supporting both low latency and high throughput, are a reality
- Both real systems and simulations were used to evaluate NoSQL at extreme scales, up to thousands of real nodes, and millions of simulated nodes
- NoSQL systems are a fundamental building block for more complex distributed systems
- Evidence is shown through the variety of systems that have been implemented over ZHT, such as FusionFS, IStore, MATRIX, Slurm++, ZDMQ, WaggleDB, and FRIEDA-State

These contributions have led to 6 peer reviewed publications, and one publication that is under review.

- Tonglin Li, Ioan Raicu, Lavanya Ramakrishnan, Scalable State Management for Scientific Applications in the Cloud, IEEE International Congress on Big Data (BigData) 2014
- Tonglin Li, Kate Keahey, Rajesh Sankaran, Pete Beckman, Ioan Raicu, A Cloud-based Interactive Data Infrastructure for Sensor Networks, ACM/IEEE Supercomputing/SC 2014
- Tonglin Li, Xiaobing Zhou, Ioan Raicu, etc. ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table, 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013.

- Tonglin Li, Raman Verma, Xi Duan, Hui Jin, Ioan Raicu. Exploring Distributed Hash Tables in High-End Computing, ACM SIGMETRICS Performance Evaluation Review (PER), 2011

- Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems, IEEE International Conference on Big Data 2014

- Ke Wang, Xiaobing Zhou, Tonglin Li, Dongfang Zhao, Michael Lang, Ioan Raicu. Optimizing Load Balancing and Data-Locality with Data-aware Scheduling, IEEE International Conference on Big Data 2014

- Tonglin Li, Xiaobing Zhou, Ioan Raicu, etc., A Convergence of Distributed Key-Value Storage in Cloud Computing and Supercomputing, IEEE Transaction of Service Computing 2014, under review

These contributions have also lead to 13 additional [81, 100, 102, 124, 89, 101, 76, 126, 99, 74, 123, 125, 78, 103] peer-reviewed publications which have used the work from this proposal as a building block towards more complex distributed systems.

The rest of this proposal is organized as follows: section 2 describes ZHT, a Design and implementation of ZHT, A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table, and proved that Distributed NoSQL storage systems can be light-weight, dynamic, resilient, portable, supporting both low latency and high throughput. Section 3 and 4 describe two real cloud-based application systems that used NoSQL databases to boost the performance and simplify the design. Section 5 and 6 describe the future work and research timeline.

# CHAPTER 2

# ZHT: A LIGHT-WEIGHT RELIABLE PERSISTENT DYNAMIC SCALABLE ZERO-HOP DISTRIBUTED HASH TABLE

This work presents a convergence of distributed key-value storage systems in clouds and supercomputers. It specifically presents ZHT, a zero-hop distributed hash table, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed systems, such as parallel and distributed file systems, distributed job management systems, and parallel programming systems. The goals of ZHT are delivering high availability, good fault tolerance, high throughput, and low latencies, at extreme scales (millions of nodes). ZHT has some important properties, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication, persistent, scalable, and supporting unconventional operations such as append, compare and swap, call-back in addition to the traditional insert/lookup/remove. We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 64-nodes, an Amazon EC2 virtual cluster up to 96-nodes, to an IBM Blue Gene/P supercomputer with 8K-nodes. Using synthetic benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput on Blue Gene/P. On Amazon EC2 cloud, on 96-node scale it offers 0.8ms latency and 1.2M ops/s throughput. We compared ZHT against other key/value stores (e.g. Cassandra, Memcached, DynamoDB) and found it offers superior performance for the features and portability it supports. This work also presents several real systems that have adopted ZHT, namely FusionFS (distributed metadata management and data provenance capture/query), IStore (data chunk metadata management), MATRIX (distributed scheduling), Slurm++ (distributed HPC job launch) and ZDMQ (distributed message queue management); all of these real systems have been simplified due to ZHT, and have been shown to outperform other leading systems by orders of magnitude in some cases.

## 2.1 Background

Exascale computers (e.g. capable of $10^{18}$ ops/sec), with a processing capability similar to that of the human brain, will enable the unraveling of significant scientific mysteries and present new challenges and opportunities. Major scientific opportunities arise in many fields (such as weather modeling, understanding global warming, national security [114, 118, 116, 117, 34, 132, 131, 115, 35, 26, 113], micro electro mechanical systems [119, 88, 87, 29], and computational chemistry [86]) and may rely on revolutionary advances that will enable exascale computing.

"A supercomputer is a device for turning compute-bound problems into I/O bound problems" [14]. The quote from Ken Batcher reveals the essence of modern high performance computing and implies an ever-growing shift in bottlenecks from compute to I/O. For exascale computers, the challenges are even more radical, as the only viable approaches in next decade to achieve exascale computing all involve extremely high parallelism and concurrency. Up to 2014, some of the biggest systems already have more than 3 million general-purpose cores. Many experts predict [82] that exascale computing will be a reality by the end of the decade; an exascale system is expected to have millions of nodes, billions of threads of execution, hundreds of petabytes of memory, and Exabyte of persistent storage.

In the current decades-old architecture of HPC systems, storage is completely segregated from the compute resources and are connected via a network interconnect (e.g. parallel file systems running on network attached storage, such as GPFS [83], PVFS [18] and Lustre [84]). This approach is not able to scale several orders of magnitude in terms of concurrency and throughput, and will thus prevent the move from petascale to exascale. If we do not solve the storage problem with new storage architectures, it could be a show-stopper in building exascale systems. The need

for building efficient and scalable distributed storage for high-end computing (HEC) systems that will scale three to four orders of magnitude is on the horizon.

One of the major bottlenecks in current state-of-the-art storage systems is metadata management. Metadata operations on parallel file systems can be inefficient at large scale. Experiments on the Blue Gene/P system at 16K-core scales show the various costs (wall-clock time measured at remote processor) for file/directory create on GPFS.

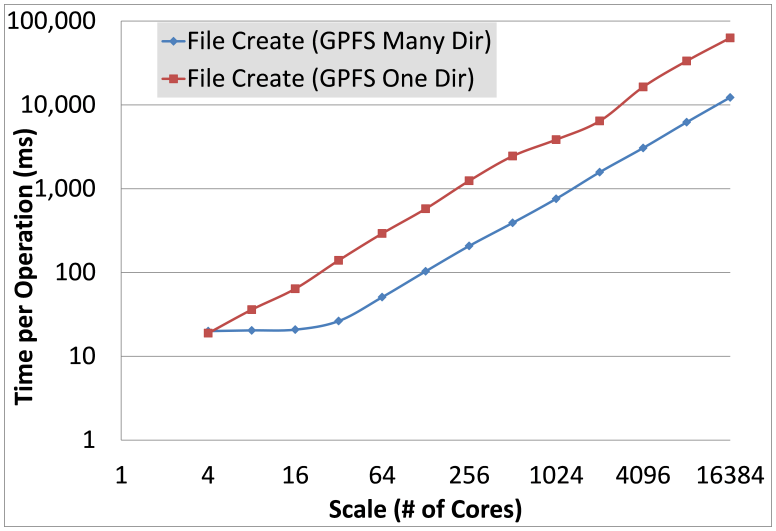



Figure 2.1: Time per operation (touch) on GPFS on various numbers of processors on a IBM Blue Gene/P

Ideal performance would have been constant, but we can see the cost of these basic metadata operations (e.g. create file) growing from tens of milliseconds on a single node (four-cores), to tens of seconds at 16K-core scales; at full machine scale of 160K-cores, we expect a file create to take over two minutes for the many directory case, and over 10 minutes for the single directory case. Previous work[77] shows these times to be even worse, putting the full system scale metadata operations in the hour range, but the test bed as well as GPFS might have been improved over the last several years. Whether the time per metadata operation is minutes or hours on a

large-scale HEC system, the conclusion is that the distributed metadata management in GPFS does not have enough degree of distribution, and not enough emphasis was placed on avoiding lock contention. GPFSs metadata performance degrades rapidly under concurrent operations, reaching saturation at only 4 to 32 core scales (on a 160K-core machine).

Other distributed file systems (e.g. Google's GFS and Yahoo's HDFS[111]) that have centralized metadata management make the problems observed with GPFS even worse from the scalability perspective. Future storage systems for high-end computing should support distributed metadata management, leveraging distributed data-structure tailored for this environment. The distributed data-structures share some characteristics with structured distributed hash tables, having resilience in face of failures with high availability; however, they should support close to constant time inserts/lookups/removes delivering the low latencies typically found in centralized metadata management (under light load). Metadata should be reliable and highly available, for which replication (a widely used mechanism) could be used.

This work presents a significant extension to our prior work on developing a zero-hop distributed hash table (ZHT)[48, 46], which has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent churn, low latencies, and scientific computing data-access patterns). ZHT aims to be a building block for future distributed systems, with the goal of delivering excellent availability, fault tolerance, high throughput, scalability, persistence, and low latencies. ZHT has several important features making it a better candidate than other distributed hash tables and key-value stores, such as being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication and by handling failures gracefully and efficiently propagating events throughout the system, a customizable consistent hashing function, supporting persistence for better

recoverability in case of faults, scalable, and supporting unconventional operations such as append (providing lock-free concurrent key/value modifications) in addition to insert/lookup/remove. To provide ZHT a persistent back end, we also created a fast persistent key-value store that could be easily integrated and operated in lightweight Linux OS typically found on todays supercomputers.

We have evaluated ZHT's performance under a variety of systems, ranging from a Linux cluster with 512-cores, to an IBM Blue Gene/P supercomputer with 160K-cores. Using micro-benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput. We compared ZHT against two other systems, Cassandra and Memcached and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes. We also compared it to DynamoDB in the Amazon AWS Cloud, and found that ZHT offers significantly lower latency and higher throughput than DynamoDB while costing less money.

This work also covers at a high-level five real systems (FusionFS, IStore, MATRIX, Slurm++, and ZDMQ) that have integrated with ZHT, and evaluates them at modest scales. 1) ZHT was used in the FusionFS distributed file system to deliver distributed meta-data management and data provenance capture/query. On a 512-nodes deployment at Los Alamos National Lab, FusionFS reached 509kops/s metadata performance. 2) ZHT was used in the IStore, an erasure coding enabled distributed object storage system, to manage chunk locations delivering more than 500 chunks/sec at 32-nodes scales. 3) ZHT was also used as a building block to MATRIX, a distributed task scheduling system, delivering 13k jobs/sec throughputs at 4K-core scales. 4) Slurm++, a distributed job launch system that avoids the centralized gateway nodes in Slurm. 5) ZDMQ is a distributed message queue which uses ZHT to store messages reliably, and load balance the resource requirements.

The contributions of this chapter are as follows:

- Design and implementation of ZHT, a lightweight, high performance, fault tolerant, persistent, dynamic, and highly scalable distributed hash table, optimized for high-end computing.
- Support for unconventional operations, such as append, cswap and callback, making key/value stores more flexible for a variety of different applications.
- Micro-benchmarks up to 32K-core scales, achieving latencies of 1.1ms and throughput of 18M ops/sec on a supercomputer and 0.8ms and 1.2M ops/sec on a cloud.
- Performance evaluation comparison between ZHT and many other key-value stores such as Memcached, Cassandra and DynamoDB on different platforms from supercomputers (IBM BlueGene/P), a Linux cluster, and public cloud (Amazon AWS).
- Simulated ZHT on 1 million-node scale for the potential use in extreme scale systems.
- Integration and evaluation with five real systems (FusionFS, IStore, MATRIX, Slurm++, and ZDMQ), managing distributed storage, metadata, task/job scheduling, and message queues.

## 2.2 ZHT Design and Implementation

Most HEC environments are batch oriented, which implies that a system that is configured at run time, generally has information about the compute and storage resources that will be available. This means that the amount of resources (e.g. number of nodes) would not increase or decrease dynamically, and the only reason to decrease

the allocation is either to handle failed nodes, or to terminate the allocation. Because nodes in HEC are generally reliable and have predicable uptime (nodes start on allocation, and shut down on de-allocation), it implies that node "churn" in HEC is virtually non-existent. We believe that dynamic membership is important for some environments, especially for cloud computing systems, and hence have made efforts to support it without affecting basic operations time complexity. This principle guided our design of the proposed dynamic membership support in ZHT.

The primary goal of ZHT is to get all the benefits of DHTs, namely excellent availability and fault tolerance, while achieves the benefits of minimal latencies normally associated with idle centralized indexes. The data-structure is kept as simple as possible for ease of analysis and efficient implementation.

The API of ZHT is kept simple and follows similar interfaces for hash tables. The six operations ZHT supports are 1. insert(key, value); 2. lookup(key); 3. remove(key), and 4. append(key, value), 5. cswap(key, seenValue, newValue), 6. callback(key, expectedValue). Keys are typically a variable length ASCII text string. Values can be complex objects, with varying size, number of elements, and types of elements.

In static membership, every node at bootstrap time knows how to contact every other node in ZHT. In a dynamic environment, nodes may join (for system performance enhancement) and leave (node failure or scheduled maintenance) any time, although in HEC systems this churn occurs much less frequently than in traditional DHTs.

ID Space and Membership Table are treated as a ring-shaped key name space. The node ids in ZHT can be randomly distributed throughout the network. The random distribution of the ID space has worked well up to 32K-cores.

The hash function maps an arbitrarily long string to an index value, which can then be used to efficiently retrieve the communication address (e.g. host name, IP address, MPI-rank) from a membership table (a local in-memory vector). Depending on the level of information that is stored (e.g. IP - 4 bytes, name - ¡100 bytes, socket - depends on buffer size), storing the entire membership table should consume only a small (less than 1

On 1K-nodes scale, one ZHT instance has a memory footprint of only 10MB (from an available 2GB memory), achieving our desired sub 1% memory footprint. The memory footprint consists of ZHT server binary in memory, entries in hash table, membership table and ZHT server side socket connection buffers. Among them, only membership table and socket buffers will increase with the scale of nodes. Entries in hash table will be flushed to disk finally. But membership is very small, it takes 32 bytes per entry (for each node), 1million nodes only need 32MB memory. By tuning the number of Key-Value pairs that are allowed to stay in memory, users can achieve the balance between performance and memory consumption.

### 2.2.1 Basic operations

Similar to other key-value stores, ZHT offers conventional operations, namely insert, lookup and remove. These three operations are implemented efficiently so the latency is low. Based on the requirements that we have faced during developing large-scale distributed system, we abstract three extra operations. These operations can significantly simplify the system design for many scenarios.

**append operation:** This allows user assigning multiple value to a same key. This is not a feature that many hash maps do, and is especially rare in persistent ones as well. The benefit of the append is that it allows fast concurrent modification (appending) of a value in the map. We found the append operation critical in supporting lock-free

concurrent modification in ZHT (eliminating the need for a distributed system lock); using append, we were able to implement a highly efficient metadata management for a distributed file system, where certain metadata (e.g. directory lists) could be concurrently modified across many clients. Consider a typical use case in distributed and parallel file systems: creating 10K files from 10K processes in one directory; the concurrent metadata modification occur via distributed locks.

**cswap (compare and swap) operation**: In some applications when clients read a value and set it to another value according to the read value, there comes the resource contention problem. For example, resource contention happens in a distributed resource management systems when different servers try to allocate the same resources. A naive way to solve the problem is to add a global lock for each queried key in the key-value stores, which is apparently not scalable. A better approach is to implement an atomic operation in the key-value stores that can tell the controllers whether the resource allocation succeeds or not. Learned from the traditional compare and swap atomic instruction, we implemented a specific compare and swap (abbreviated to cswap) atomic algorithm that could address resource contention problem.

**callback operation:** callback operation is used to notify a client upon a specified value changes. Sometimes an application (such as a state machine) needs to wait on specific state change in the key-value store before moving on. A simple way to do the blocking is let the client poll the data server periodically (e.g. every 1 sec) until the state changes, which brings too much unnecessary communication. To solve this problem, we move the value checking from the client side to the server side and introduce a new operation called **state_change_callback** (abbreviated to callback): The data server creates a special thread for the state change callback requests, and the main thread keeps processing requests. Within a given period of time (time_out), if the server finds the value being changed to expected value, it returns success to the client; otherwise returns to the client, which will do the callback operation again.

### 2.2.2 Terminologies

In this section, we briefly introduce the terms used in the this work.

**Physical node:** A physical node is an independent physical machine. Each physical node may have several ZHT instances that are differentiated with an IP address and port.

**Instance:** A ZHT instance is a ZHT server process that handles the requests from clients. Each instance takes care of some partitions. By adjusting the number of instance, ZHT can fit in heterogeneous systems with various storage capacities and computing power. A ZHT instance can be identified by a combination of IP address and port, and each ZHT instance maintains many partitions. We only need to store addresses for ZHT instances, no need to do so for partitions. Therefore number of partitions can be much larger than the number of addresses.

**Partition:** A partition is a contiguous range of the key address space; a file on disk is associated with each partition for persistence.

**Manager:** A Manager is a service process running on each physical node and takes charge of starting and shuting down ZHT instances. The manager is also responsible for managing membership table, starting/stopping instances, and partition migration. As traditional consistent hashing does, initially we assign each of the k physical nodes a manager and one or more ZHT instances, each with a universal unique id (UUID) in the ring-shaped space. The entire name space N (a 64-bit integer) is evenly distributed into n partitions where n is a fixed big number indicating the maximal number of nodes that can be used in the system. It is worth noting that while n (the number of partitions, also the maximal number of physical nodes) cannot be changed without potentially rehashing all the key/value pairs stored in ZHT, i (the number of ZHT instances) as well as k (the number of physical nodes) is

changeable with changes only to the membership table. Each physical node has one manager, holds n/k partitions, with each partition storing N/n key-value pairs and i/k ZHT instances serving requests. Each partition (which can be persisted to disk) can be moved across different physical nodes when nodes join, leave, or fail.

For example, in an initial system of 1000 ZHT instances (potentially running on 1000 nodes), where each instance contains 1000 partitions, the overall system could scale up to 1 million instances on 1 million physical nodes. Experiments validate this approach showing that there is little impact (0.73ms vs. 0.77ms per request when scaling from 1 partition to 1000 partitions respectively) on the performance as we increase the number of partitions per instance. This design allows us to avoid a potentially expensive rehash of many key/value pairs when the need arises to migrate partitions.

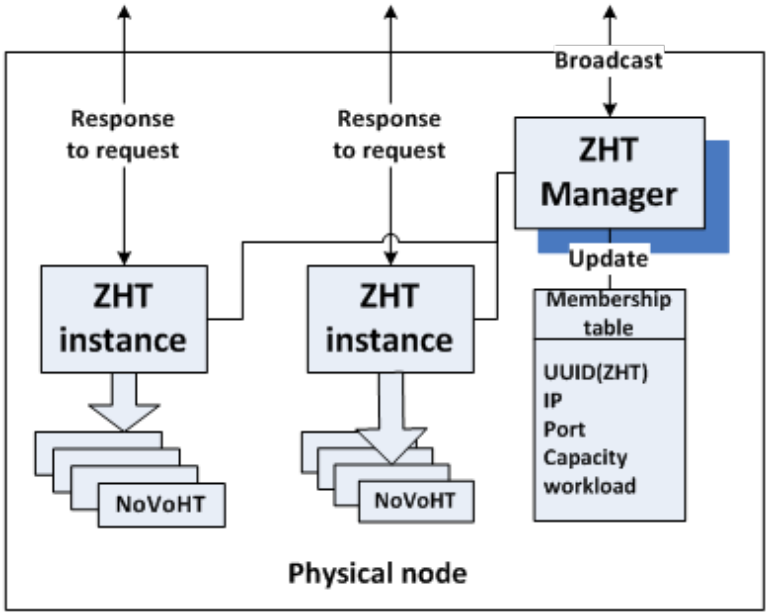



Figure 2.2: ZHT server single node architecture

### 2.2.3 Membership management

ZHT supports both static and dynamic node membership. In the static case, the bootstrapping phase gets neighbor list from the batch job scheduler or a given file. Once the membership is established, no new nodes would be allowed to join the system. Nodes could leave the system due to failures; we assume failed nodes do not recover. For the dynamic membership, nodes are allowed to dynamically join and leave the system. Most DHTs support dynamic membership, but typically deliver this through logarithmic routing. DHTs use consistent hashing which sacrifices performance in order to achieve scalability under potentially dynamic conditions. We address this issue with a zero-hop consistent hashing mechanism. With this novel design, we offer the desired flexibility of dynamic membership while maintaining high performance with constant time routing.

**Data migration and membership update:** The design goal is to ensure that only minimal impact on performance and scalability would be posed by dynamic membership. With dynamic membership, there comes the need to potentially migrate data from one physical node to another. In order to achieve this, ZHT organizes its data in partitions, and migrates entire partitions when needed. This avoids having to rehash key-value pairs that make up the migrated data, as most DHTs do. Moving an entire partition is significantly more efficient than rehashing many key/value pairs. When migration is in progress, the partition state cannot be modified. All requests are queued, until the migration is completed. In the meanwhile because the partition state is locked, corresponding replicas also wont change. This keeps the entire system state consistent. If failure occurs during migration, simply dont apply the changes (in terms of discarding the queued requests and reporting error to clients) to corresponding partitions and replicas, this will eventually force the system roll back to a consistent earlier state.

**Node Joins:** On a node join operation, it checks out a copy of membership table from the ZHT Manager on a random physical node. In this table, the new node can find the physical nodes with the most partitions, then join the ring as this heavily loaded nodes neighbor and move some of the partitions from the busy node to itself. Migrating a partition is as easy as moving a file, without having to rehash the key/value pairs stored in the partition.

**Node departures:** On planned node departures (e.g. an administrator wants to take down part of the system for maintenance), the administrator would get a current membership table from a random physical node, modify it accordingly, then broadcast the incremental table to other managers to update their local tables. The managers, which will be departing, first migrates their partitions to neighboring nodes, and then continue to depart. For an unplanned departure (e.g. due to a node failure), it will be detected first by a client which sends a request and times out waiting for a response, or due to another ZHT instance initiating a server-to-server operation (e.g. migration, replication, etc.). Upon a certain number of failures, it will mark the entire physical node unavailable on its local membership table and inform a random manager about this failure. The client then sends the request to the first replica of the failed node. At the same time, the manager updates its local membership table and broadcasts the change, and initiates a rebuilding of the replicas, specifically increasing replicas on all partitions stored on the failed physical node in order to maintain the specified level of replication.

**Client Side State:** In case that client and server are not on the same nodes, its necessary to keep client side membership table updated. Since the node joining and leaving will change the number of partitions covered by a ZHT instance, clients might send requests to wrong nodes if the local membership table is not updated. To address this issue, we adopt lazy updating. Only when the requests are sent

mistakenly, the ZHT instance will send back a copy of latest membership table to the clients.

### 2.2.4 Server architecture

We explored various architectures for ZHT server. Since typical Key-Value store operations are very short but frequent, we designed ZHT to be able to respond fast with little resource consumption. In early prototypes, we explored a multi-threading design, in which each request had a separate thread, but the overheads of starting, managing, and stopping threads was too high in comparison to work each thread was performing. We eventually converged on a notably more streamlined architecture, an event-driven model server architecture based on epoll. The current epoll-based ZHT outperforms the multithread version by 3X. Well discuss the performance difference in more detail in the evaluation section.

### 2.2.5 Hash Functions

Each hashing function has a set of properties and designed goals, such as: 1) minimize the number of collisions, 2) distribute signatures uniformly, 3) have an avalanche effect ensuring output varies widely from small input change, and 4) detect permutations on data order. Hash functions such as the Bob Jenkins hash function, FNV hash functions, the SHA hash family, or the MD hash family all exhibit the above properties. We have explored the use of Bob Jenkins and FNV hash functions, due to their relatively simple implementation, consistency across different data types (especially strings), and the promise of efficient performance.

### 2.2.6 Fault Tolerance

ZHT gracefully handles failures, by lazily tagging nodes that do not respond to requests repeatedly as failed (using exponential back off). ZHT uses replication to ensure data will persist in face of failures. Newly created data will be pro-actively replicated asynchronously to nodes in close proximity (according to the UUID) of the original hashed location. By communicating only with neighbors in close proximity, this approach will ensure that replicas consume the least amount of shared network resources when we succeed in implementing the network-aware topology (see future work section). Despite the lack of network-aware topology in the current ZHT, the asynchronous nature of the replication adds relatively little overhead with increasing numbers of replicas at modest scales up to 4K-cores.

ZHT is completely distributed, and the failure of a single node does not affect ZHT as a whole. The (key, value) pairs that were stored on the failed node were replicated on other nodes. Upon failures, the replicas will answer the queries for data that were on the failed node.

In the event that ZHT is shut down (e.g. maintenance of hardware, system reboot, etc.), the entire state of ZHT could be loaded from local persistent storage (e.g. the SSDs on each node); note that every change to the in-memory DHT is in fact persisted to disk (assuming there is one), allowing the entire state of the DHT to be reconstructed if needed. Given the size of memory and SSDs of today, as well as I/O performance improvements in the future, it is expected that a multi-gigabyte amount of state could be retrieved in just seconds.

Once ZHT is bootstrapped, the verification of its nearest neighbors should not be related to the size of the system. In the event that a fresh new ZHT instance is to be bootstrapped, the process is quite efficient in its current static membership form,

as there is no global communication required between nodes (see 2.3). Nevertheless, we expect the time to bootstrap ZHT to be insignificant in relation to the cost to the batch schedulers overheads on a HEC, which could potentially include node provisioning, OS booting, starting of network services, and perhaps the mounting of some parallel file system. At 1K-node scales, the time to start the batch scheduled job is about 150 seconds, after which the ZHT bootstrap takes another 8 seconds at 1K-node scale and 10 seconds to bootstrap at 8K-node scale. 2.3 shows the bootstrap time increase with the scale.

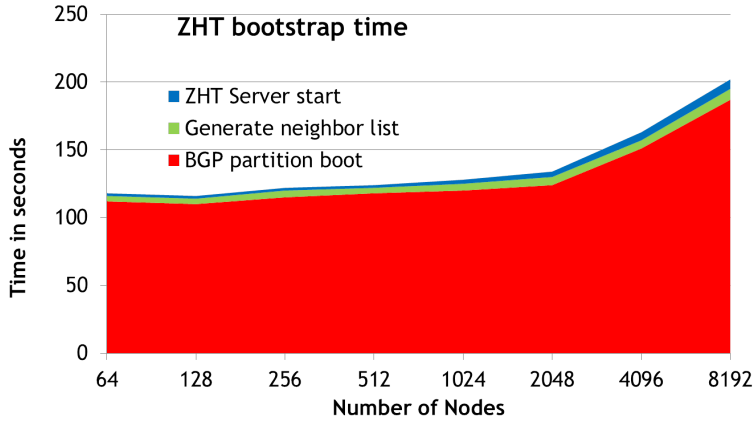



Figure 2.3: ZHT Bootstrap time on Blue Gene/P from 64 to 8K nodes

### 2.2.7 Socket level thread safe

Network multithreaded problems are mostly socket related, In other words, if the socket is thread safe, many message sending or receiving out-of-order related issues are smoothed away. The previous version of ZHT client is thread-safe in operation level, namely, all ZHT API e.g. insert, delete, etc. It relies on a shared mutex to avoid any contention problem. This was not as efficient as it could be. We have made ZHT client as thread safe not only in operation level but also in socket level. In ZHT client, sockets are stored in a LRU cache. The key is combination of IP and port of

ZHT server that client talks to. When the client needs to communicate with a server it will first try to find the key as mentioned. If not, socket is initialized and bound to IP and port of the server, and then put into the cache for reuse. In the same time, a mutex is initialized for that socket for protection. We removed the single mutex shared by ZHT API(s) and only rely on socket level thread-safe to realize overall ZHT client thread-safe.

### 2.2.8 Consistency

ZHT uses replication to enhance reliability. Current version only allows clients interact with a single replica (e.g. primary replica). This decision is based on the fact that if we allowed multiple replicas to be concurrently modified, a more complicated consistency mechanism such as Paxos protocol has to be maintained, at the cost of loss of performance advantages. In the event that the primary replica becomes temporarily inaccessible, a secondary replica will interact directly with clients (which would cause modifications to happen concurrently on both the primary and secondary replicas). The ZHT primary replica and secondary replica are strongly consistent; other replicas are asynchronously updated after the secondary replica is complete, causing ZHT to follow an eventual consistency model. Using this approach, ZHT achieves high throughput and availability while maintains reasonable consistency level.

### 2.2.9 Consistency Persistence

ZHT is a distributed in-memory data-structure. In order to withstand failures and restarts, it also supports persistence. We designed and implemented a Non-Volatile Hash Table (NoVoHT) for ZHT, which uses a log-based persistence mechanism with periodic checkpointing. We evaluated several existing systems, such as KyotoCabinet

HashDB and BerkeleyDB, but performance and missing features prompted us to implement our own solution.

NoVoHT is a custom-built lightweight hash table at the core, with added features built on top. The design of the map structure is an array of linked lists. This structure makes collision handling efficient. Also, it helps lookup time, by eliminating the worst case of iterating the entire array in the case of it being full. Finally, it allows the application to overfill the map, with more keys than buckets. While this would impact time of insert and remove, it keeps the space used for the array lower. It also allows the key/value store to allow lock-free read operations.

NoVoHT employs a log-based approach to achieve persistence. When a key/value pair is inserted, it writes the key-value pair to the file specified, and records where it was written with the key-value pair in the map. By recording the location in the file, removal is efficient. When an element is removed it removes the pair from the map, and marks the spot in the file. By marking the file, if the application crashes, that pair will not be inserted into the map when the file state is recovered. NoVoHT allows a customized threshold, which determines how many removes to do before the file is rewritten with the pairs in the map (effectively eliminating the pairs that were marked for removal from the file). NoVoHT also supports periodic garbage collection to reclaim free space at timed intervals.

### 2.2.10 Implementation

ZHT has been under development for 3.5 years with 7 years of man-hours. It is implemented in C/C++, and has very few dependencies. It consists of 14900 lines of code, and is an open source project accessible at GitHub. The dependencies of ZHT are NoVoHT (discussed in the next section) and Google Protocol Buffers.

## 2.3 Performance Evaluation via Synthetic benchmarks

In this section, we describe the performance of ZHT, including hashing functions, persistence, throughput, latencies, and replication. Firstly we introduce the test beds and micro benchmark configuration. Secondly a comprehensive performance evaluation will be presented. We compare ZHT with Memcached and Cassandra, two popular systems offering similar functionality or features to ZHT, and with DynamoDB on EC2[1] cloud.

### 2.3.1 Testbeds, Metrics, and Workloads

We used several machines to evaluate ZHTs performance.

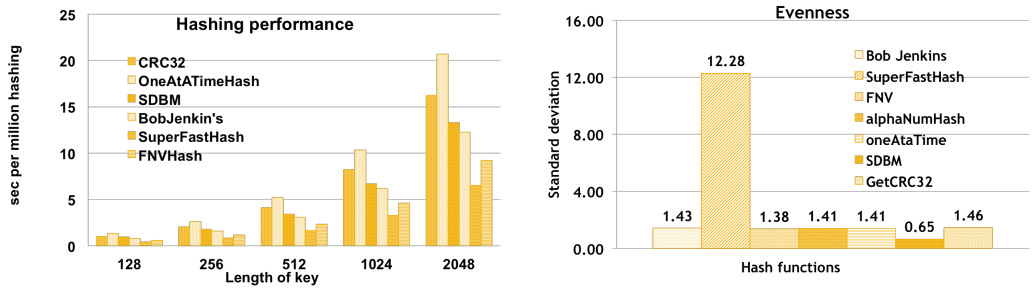
- Intrepid: an IBM Blue Gene/P[8] supercomputer at Argonne Leadership Computing Facility[2]. We used 8K nodes (32K cores), where each node has a 4-core PowerPC 450 processor and 2GB of RAM. This testbed was used to compare ZHT to Memcached. Note that this system does not have persistent local node disks, and RAM-based disks were used for persistence.
- Kodiak: A Parallel Reconfigurable Observational Environment (PROBE)[10] at Los Alamos National Laboratory, 1024 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory.
- HEC-Cluster: a 64-node (512-core) cluster at IIT: each node has a dual processor quad-core, 8GB RAM, used to compare ZHT with Cassandra.
- DataSys: an 8-core x64 server at IIT: dual Intel Xeon quad-core w/ HT processors, 48 GB RAM, used to compare NoVoHT, BerkeleyDB and KyotoCabinet.

- Fusion: a 48-core x64 server at IIT: quad AMD Opteron 12-core processors, 256GB RAM, used to compare NoVoHT, BerkeleyDB and KyotoCabinet.
- Amazon EC2 Cloud: up to 96 cc.8xlarge VMs.

On each node, one or more ZHT client-server pairs are deployed, namely ZHT instances. Each instance is configured with one partition known as NoVoHT. Test workload is a set of key-value pairs where the key is 15 bytes and value is 132 bytes. Clients sequentially send all of the key-value pairs through a ZHT Client API for insert, then lookup, and then remove. The additional operations such as append are evaluated separately due to their different nature of the operation. Since the keys are randomly generated, the communication pattern is All-to-All, with same number of servers and clients.

The metrics measured and reported are:

- Latency: The time taken for a request to be submitted from a client and a response to be received by the client, measured in milliseconds. Since the latencies of various operations (insert/lookup/remove) are fairly close, we use average of the three operations to simplify results presentation. Note that the latency includes the round trip network communication and storage access time. Since Blue Gene/P doesnt have persistent storage for each work node, ram-disks are used in the experiment, while regular spinning hard drives are used in experiments on cluster.
- Throughput: The number of operations (insert/lookup/remove) the system can handle over some period of time, measured in Kilo Ops/s.
- Ideal throughput: Measured throughput between two nodes times the number of nodes.




(a) Time per hash for a variety of hashing functions

(b) Key distribution evenness across different hash functions

Figure 2.4: Hash function comparisons

- Efficiency: Ratio between measured throughput and ideal throughput.

### 2.3.2 Hash Functions

Similar with what we observed, the time spent on hashing keys to nodes is not major of the total cost, but with the time passed, the accumulation could be observed. We investigate some of the usual hash functions for figuring out the tradeoff between performance and evenness.

As shown in figure 2.4a that some hash functions are faster than others, but a more important concern rather than performance is the evenness (figure 2.4b). Since the worst hash function we investigated has performance of 0.02ms/hash, and thus negligible compared to other overhead. Meanwhile the evenness is essential to the entire performance. An ideal hash function should be able to spread keys evenly do as to provide a natural load balancing mechanism.

### 2.3.3 Individual data store synthetic benchmark

We compared NoVoHT with persistence to KyotoCabinet[9] with identical workloads for 1M, 10M, and 100M inserts, gets, and removes, operating on fixed length key value pairs (see figure 2.5). When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we see significantly better scalability properties on NoVoHT. Although BerkeleyDB has some advantages such as memory usage (not shown in the figure), it does this at the cost of slower performance. When comparing NoVoHT persistence to non-persistence, we noticed that most of the overhead of the operations is held in the disk I/O portion.

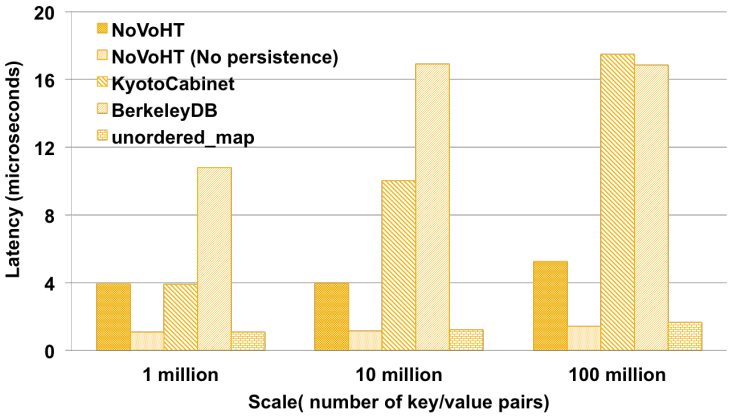



Figure 2.5: Average latency of NoVoHT, KyotoCabinet and BerkeleyDB

### 2.3.4 System Latencies

### 2.3.5 Synthetic benchmarks On Supercomputers and clusters

We evaluated the latency metric on both the Blue Gene/P and HEC-Cluster testbeds. We evaluated several communication variations, such as UDP, TCP with connection

caching, and compare them with Memcached and Cassandra.

At 8K-node scale, ZHT shows great scalability. As shown in figure 2.6, on one node, the latency of both TCP with connection caching and UDP is extremely low (0.5ms). When scaling up, ZHT shows low latency, up to 1.1ms at 8K-node scales. We see that TCP with connection caching can deliver essentially the same performance as UDP, for all the scales measured. Memcached also scaled well, with latencies ranging from 1.1ms to 1.4ms from 1 node to 8K nodes (note that this represents a 25% to 139% slower latency, depending on the scale). Note the IBM Blue Gene/P network for communication is a 3D Torus network, which does multi-hop routing to send messages among compute nodes. That means the number of hops will increase when communicate across racks. This explains the performance slow down on large scale, since one rack of Blue Gene/P has 1024 nodes, any larger scale than 1024 will involve more than one rack. We found the network to scale very well up to 32K-cores, but there is not much we can do about the multi-hop overheads across racks.

The CDF plot (figure 2.7) shows very similar trends for different scales that imply excellent scalability. On 64 node-scale 90% requests finish in 853us and 99% requests finish in 1259us. When scaling up to 1024 node-scale, the latencies are only slightly increased, 90% finishes in 1053us and 99% finishes in 3105us (2.1).

Table 2.1: ZHT Latencies ON Blue Gene/P in Microsecond

| Scales | 75% | 90% | 95% | 99% | Average | Throughput |
|---|---|---|---|---|---|---|
| 64 | 713 | 853 | 961 | 1259 | 676 | 90632 |
| 256 | 755 | 933 | 1097 | 1848 | 748 | 356137 |
| 1024 | 820 | 1053 | 1289 | 3105 | 1007 | 1316942 |

Because of Cassandras implementation in Java, and the lack of support for Java on the Blue Gene/P, we evaluated Cassandra, Memcached, and ZHT on a traditional Linux cluster, the HEC-Cluster. Not surprisingly, as shown in 2.8, ZHT has notably

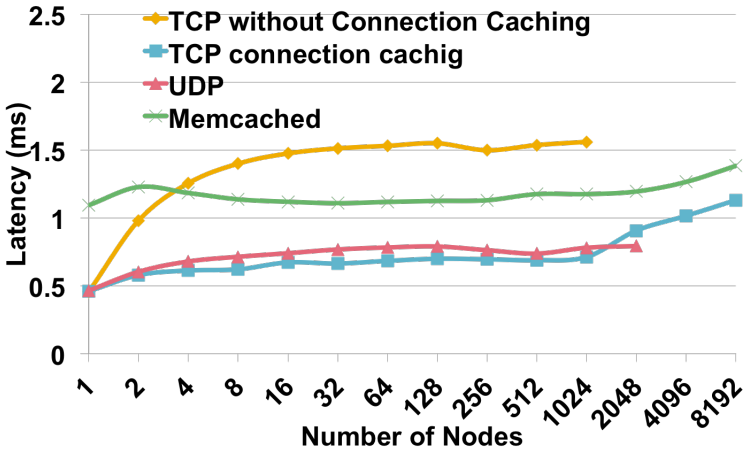



Figure 2.6: Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 8K nodes on the Blue Gene/P)

lower latency than Cassandra. ZHT also shows superior scalability over Cassandra. This is mainly because Cassandra has to take care of a logarithmic-routing-time dynamic member list and ZHT use constant routing. Surprisingly, Memcached only shows slightly better performance than ZHT up to 64-node scales. We attributed the slight loss in performance to the fact that ZHT must write to disk, while Memcacheds data stayed completely in-memory.

### Synthetic benchmarks On Cloud

We conduct micro benchmark on Amazon EC2 cloud as well to compare against Amazon DynamoDB. The EC2 instance type we used are m1.medium and cc2.8xlarge, the details are shown in table 2.2.

Different from the result that we got on supercomputers, the results on EC2 cloud reveal interesting inconsistency on various scales. Ideally the request latencies on large scale should fall into a narrow window like they do on smaller scale. On smaller instances such as m1.medium, ZHT latency CDF plots are quite different

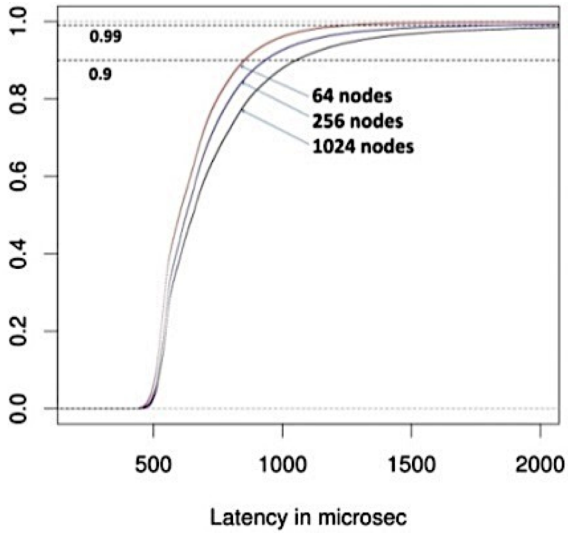



Figure 2.7: CDF of Benchmark on Blue Gene/P CDF

(Fig. 10). Although 90% requests are still finished within similar time (600 to 800us), 99% requests latency doubled when scales increase by 4 times. In other words, on EC2, latencies have much longer tails in larger scales than in smaller scales.

If the experiments are conducted on smaller instances, the long tail will be even longer (see figure 2.9b). On larger instances such as cc2.8xlarge, ZHT latency CDF plots are closer than they are on smaller instances. Figure 2.9 shows that the latency trends of 4, 16 and 64 nodes scales are quite similar. This difference confirms a fact that smaller EC2 instances (such as m1.medium) share more hardware resources (include CPU and network bandwidth) than larger instances (like cc2.8xlarge). Therefore small instances have more interference than large ones, so the application performance will also be influenced. Because of the less shared resource, big EC2 instance types may act more like regular cluster or supercomputer, since little interference exists. Note that on each cc2.8xlarge instance we start 8 ZHT servers and clients to better utilize the resource.

We also conducted micro benchmarks for Amazon DynamoDB as a comparison. Since DynamoDB default maximum throughput is 10K/s, all benchmarks are

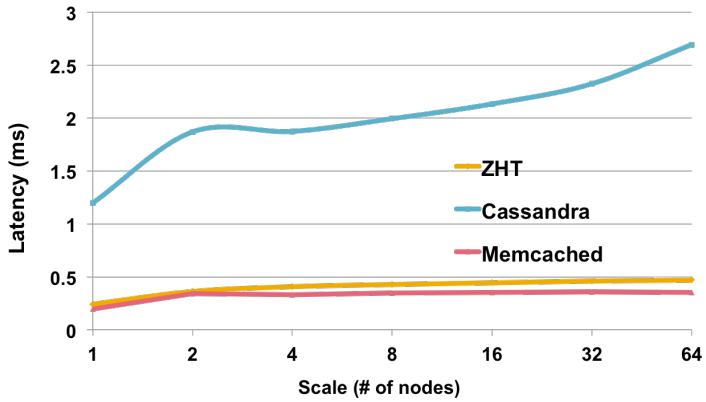



Figure 2.8: Performance evaluation of ZHT, Memcached and Cassandra plotting latency vs. scale (1 to 64 nodes on an AMD Cluster)

under that provision. There is no information released about how many nodes are used to offer a specific throughput. Since we have observe that the latency of DynamoDB doesnt change much with scales, and the value is around 10ms, we have to use many clients to saturate the capacity. We deployed clients for DynamoDB micro benchmarks on cluster computing instance, namely cc2.8xlarge. 8 clients were started on each instance.

Table 2.2: Profile of EC2 Instances Used in Experiments

| Instance type | m1.medium | cc2.8xlarge |
|---|---|---|
| CPU | 2 EC2 Compute Unit | 88 EC2 Compute Units |
| Memory | 3.75GB | 60.5 |
| Storage | 160GB | 3370GB |
| I/O Performance | Moderate | High (10 Gb/s Ethernet) |
| Cost | $0.112/hour | $2.4/hour |

As expected, DynamoDB has much longer latency on all scales. On 4-node (32 clients) scale it is 22 times slower than ZHT. In the CDF comparison DynamoDB shows that its 90% latencies fall into a 20x wider time window than ZHT. When we ran 8 clients on 64 nodes, DynamoDB started to give errors that complain about

Table 2.3: ZHT Latencies on cc2.8xlarge EC2 Instance in Microsecond

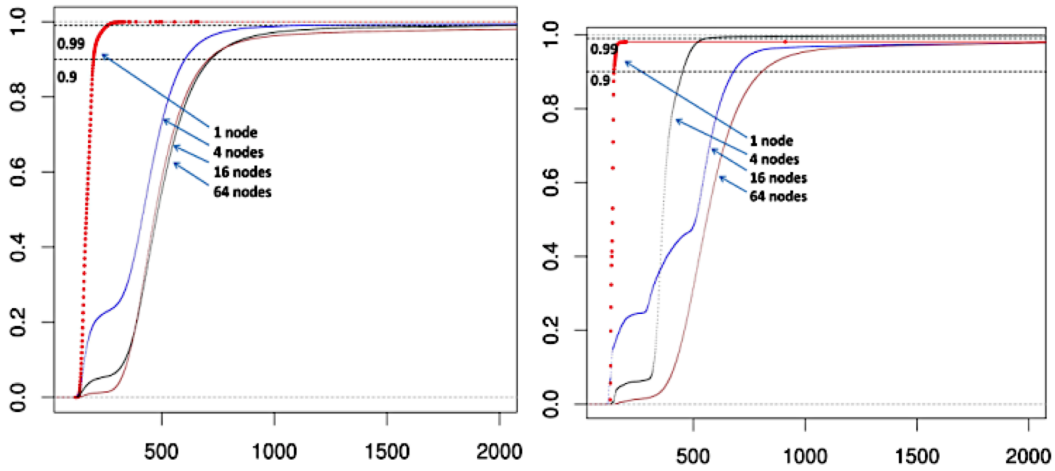
| Scales | 75% | 90% | 95% | 99% | Average | Throughput |
|---|---|---|---|---|---|---|
| 8 | 186 | 199 | 214 | 260 | 172 | 46421 |
| 32 | 509 | 603 | 681 | 1114 | 426 | 75080 |
| 128 | 588 | 717 | 844 | 2071 | 542 | 236065 |
| 512 | 574 | 708 | 865 | 3568 | 608 | 841040 |

Table 2.4: ZHT Latencies on m1.medium EC2 Instance in Microsecond

| Scales | 75% | 90% | 95% | 99% | Average | Throughput |
|---|---|---|---|---|---|---|
| 1 | 142 | 146 | 154 | 4887 | 229 | 4892.4 |
| 4 | 591 | 680 | 767 | 12500 | 760 | 4978.5 |
| 16 | 369 | 452 | 482 | 556 | 388.3 | 11351 |
| 64 | 665 | 807 | 970 | 3880 | 711.5 | 91201 |

over used throughput so we cant continue to push experiments on larger scales. The slowest 5% requests latency increased by 3 times.

It is worth noting that DynamoDB latencies dont vary much with the system scales. It seems to show an excellent scalability and a better aggregated-throughput. However considering that Amazon only guarantees the limited maximum throughput, instead of latency, users wont get faster response when they only use low throughput. In other words, DynamoDB with more clients doesnt work as fast as it with fewer clients; instead, with fewer clients it works as slow as with many clients. This characteristic prevents the users from reaching the provisioned capacity by lowering down the latency when they only have fewer clients. When we tried with scales larger than128 clients for DynamoDB, more than half request failed, because the throughput was beyond the provisioned one.




(a) cc2.8xlarge instance (b) m1.medium instance

Figure 2.9: CDF graph: ZHT latencies on EC2 cloud, in microseconds

Table 2.5: DynamoDB Latencies With Clients ON EC2 cc2.8xlarge Instance, 8 Clients/Instance

| Scales | 75% | 90% | 95% | 99% | Average | Throughput |
|---|---|---|---|---|---|---|
| 8 | 11942 | 13794 | 20491 | 35358 | 12169 | 83.39 |
| 32 | 10081 | 11324 | 12448 | 34173 | 9515 | 3363.11 |
| 128 | 10735 | 12128 | 16091 | 37009 | 11104 | 11527 |
| 512 | FAILED | FAILED | FAILED | FAILED | FAILED | FAILED |

### 2.3.6 System Throughput

#### Synthetic benchmarks On Supercomputers and clusters

We conducted several experiments to measure the throughput. The throughput of ZHT (TCP with connection caching) as well as that of Memcached increases near-linearly with scale, reaching nearly 7.4M ops/sec at 8K-node scale in both cases.

On the HEC-Cluster, as expected, ZHT has higher throughput than Cassandra. We expect the performance gap between Cassandra and ZHT to grow as system

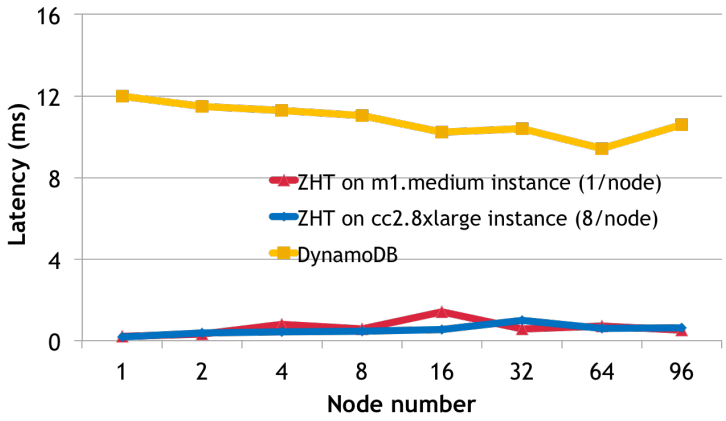



Figure 2.10: Latency comparison: ZHT v.s DynamoDB on EC2

scales grows. Figure 2.12 shows the nearly 7x throughput difference between ZHT and Cassandra. Memcached performed as expected better as well, with a similar 27% higher overall throughput.

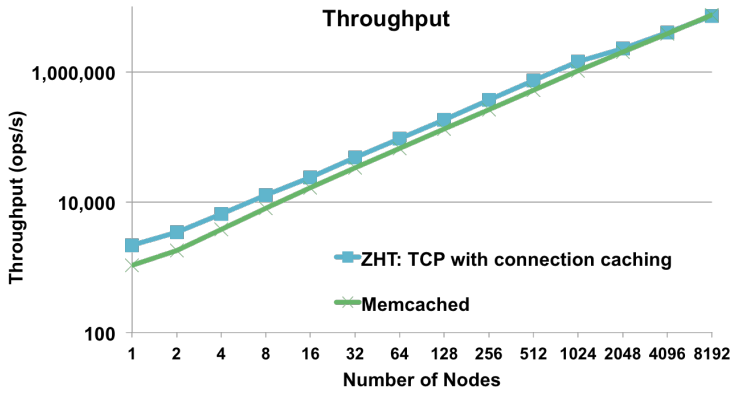



Figure 2.11: Performance evaluation of ZHT and Memcached plotting throughput vs. scale (1 to 8K nodes on the BLUE GENE/P)

### Synthetic benchmarks On Cloud

**Throughput** In figure 2.13 since the interference between m1.medium instances, ZHT shows mild fluctuation in throughput. On 2cc.8xlarge instances, the fluctuation almost disappears and the throughput is close to linear. Although DynamoDB seems

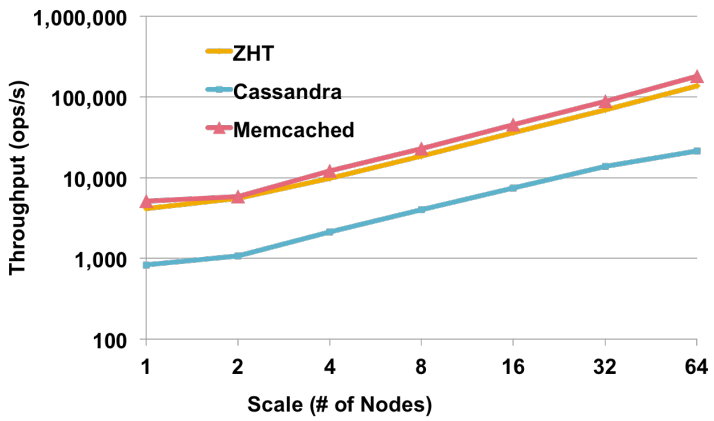



Figure 2.12: Performance evaluation of ZHT, Memcached and Casandra plotting throughput vs. scale (1 to 64 nodes on the HEC-Cluster)

to stay with a linear growth, the absolute throughput is quite low. Comparing with ZHT, DynamoDB was more than 20 times slower at all scales. For different EC2 instance types, we tried with various numbers of ZHT servers and clients on each instance so as to explore the aggregated throughput. In our experiments, on larger instance type such 2cc.8xlarge, running multiple ZHT server/client wont influence latency. Thus the aggregated throughput may have a linear growth as long as there is still CPU and network bandwidth resource. On 96 nodes scale with 2cc.8xlarge instance type, ZHT offers 1215.0 K ops/s while DynamoDB failed the test since it saturated the capacity. The measured maximum throughput of DynamoDB is 11.5K ops/s that is found at 64-node scale. For a fair comparison, both DynamoDB and ZHT have 8 clients per node.

Its worth noting that DynamoDB has a maximum throughput that is provisioned (namely capacity) by the users. When the throughput is beyond provisioned capacity, DyndmoDB will saturate and give errors, requests start to fail.

**Running Cost** When discussing cloud, the cost is always a big concern. We calculated hourly cost for both ZHT and DynamoDB on different scales. We

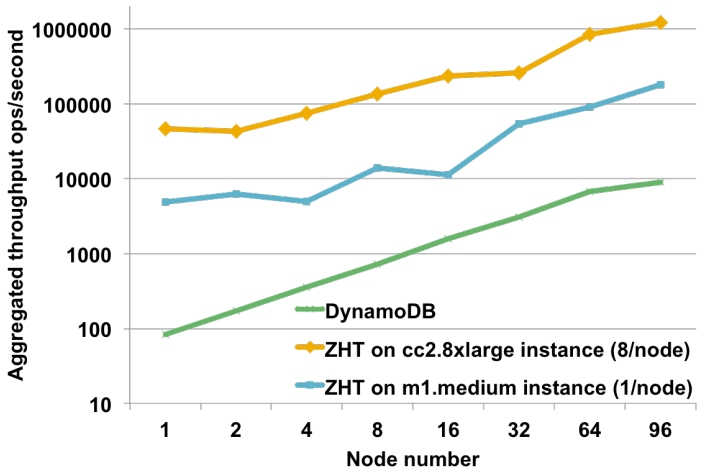



Figure 2.13: Aggregated throughput of ZHT and DynamoDB on EC2

calculate the ideal cost for DynamoDB, assuming the user always provisions the same throughput to fit their need, then according to Amazons pricing policy, for 1k ops/sec throughput, the cost is 0.65 cents per hour.

On 2-node scale DynamoDB cost 65 times more than ZHT; on largest scale that DynamoDB can support, it still cost 32 times more than ZHT for a same throughput (figure 2.14). Note the cost for DynamoDB doesnt include the EC2 instances for running clients, it will cost even more if include the client cost. These are huge cost savings applications could have by running their NoSQL distributed key/value stores on their own, at the expense of managing their own NoSQL setup.

### 2.3.7 Scalability and efficiency

Although the throughputs achieved by ZHT are impressive at many millions of ops/sec, it is important to investigate the efficiency of the system when compared to the performance at 2-node scale (the smallest test bed that involving the network) of the best performance system. In Figure , we show that ZHT and Memcached achieve different levels of efficiency up to 8K-node scales. Efficiency was computed by

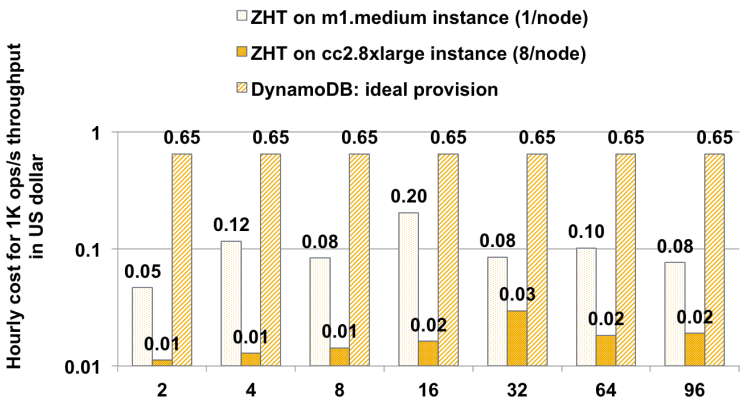



Figure 2.14: Running cost comparison

comparing ZHT and Memcached performance against the ideal latency/throughput (which was taken to be the better performer at 2-node scale ZHT). The reason why the performance over 1K-nodes degrades more sharply is because on Blue Gene/P system, 1K-nodes form a rack, and communication across the rack is more expensive (at least this is the case for TCP/UDP.

Although we were not able to run experiments at more than 8K-node scales due to time allocation on the Blue Gene/P system, we have simulated ZHT on a PeerSim-based simulator. It was interesting that the simulator results were able to closely match the results up to 8K-node scales (where we achieved 8M ops/sec), giving on average only 3% of difference. The simulations showed efficiency dropping to 8% at exascale levels (1M nodes). This sounds as if ZHT would not scale well to an exascale system, but a careful look at what 8% really means is worthwhile. 100% efficiency implies a latency of about 0.6ms per operation (ZHT latency at 2 node scales). 51% efficiency implies about 1.1ms latency (this is the performance of ZHT at 8K-node scales). 8% efficiency implies 7ms latency, at 1M node scales which is still extremely low. At 1M node scales and latencies of 7ms, we would achieve nearly 150M ops/sec throughputs.

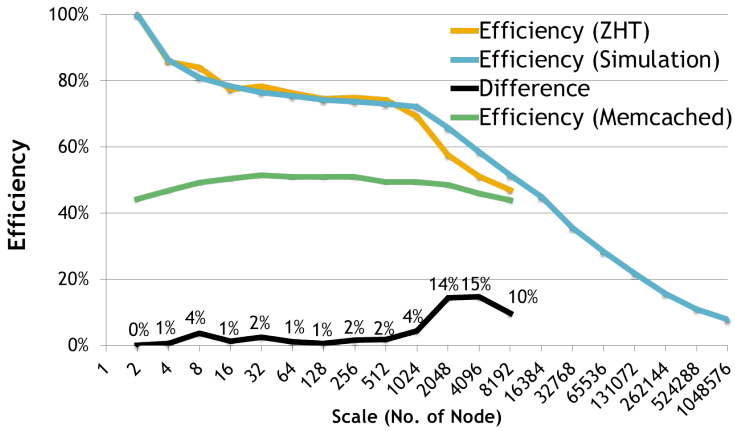



Figure 2.15: Performance evaluation of ZHT plotting measured efficiency and simulated efficiency vs. scale (1 to 8K nodes on the Blue Gene/P and 1 to 1M nodes on PeerSim)

### 2.3.8 Aggregated performance

Like most of the modern supercomputers that have multi-core processors, each Blue Gene/P node has 4 cores. Since we used an event driven architecture for the ZHT system, many components in ZHT are single threaded. Our hypothesis was that we might be able to achieve higher aggregate throughput by running multiple ZHT instances per node. We start 1 to 8 ZHT instances on each node and measure the latency and throughput. We found that assigning one instance to each core yields the best resource utilization and efficiency. As expected, in a setting with up to 4 instances per node, the aggregated throughput is excellent and the latency is still extremely low (2.08ms on 8K-nodes scale with 32K-instances). Comparing with the latency of 1.1ms on 8K-nodes with 8K-instances, the aggregated throughput is compelling (16.1M ops/sec as opposed to 7.3M ops/sec, a 2.2X increase).

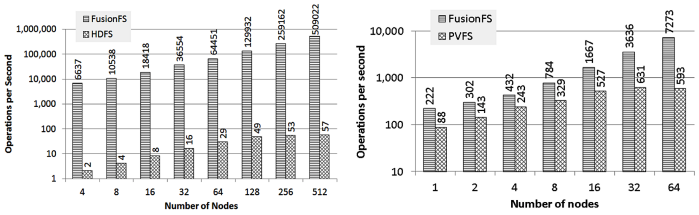
## 2.4 ZHT as a Building Block for Distributed Systems

This section presents some real systems that have adopted ZHT as a building block.

### 2.4.1 FusionFS: a Distributed File System with Distributed Metadata Management

We have an ongoing project to develop a new highly scalable distributed file system, called FusionFS. FusionFS is optimized for a subset of HPC and many-task computing (MTC) workloads. In FusionFS, every compute node serves all three roles: client, metadata server, and storage server. The metadata servers use ZHT, which allows the metadata information to be dispersed throughout the system, and allows metadata lookups to occur in constant time at extremely high concurrency. Directories are considered as special files containing only metadata about the files in the directory. FusionFS leverages the FUSE[112] kernel module to deliver a POSIX compatible interface as a user space filesystem.

We compare the metadata performance between FusionFS and HDFS on Kodiak. Both storage systems have FUSE/POSIX disabled. We have each node create (i.e. touch) a large number of empty files (with unique names), and we measure the number of files created per second. In essence, each touched file indicates a metadata operation. The aggregate metadata throughput of different scales is reported in figure 2.16a. The gap between FusionFS and HDFS is about more than 3 orders of magnitude. Note that, HDFS starts to flatten out from 128 nodes, while FusionFS keeps doubling the throughput all the way to 512 nodes, ending up with almost 4 orders of magnitude speedup (509022 vs. 57).




(a) Metadata performance comparison: performance on Kodiak

(b) Metadata performance comparison: performance on Intrepid

Figure 2.16: FusionFS: metadata performance comparison

We then compare the metadata performance between FusionFS and PVFS on Intrepid. The result is reported in figure 2.16b. FusionFS outperforms PVFS on a single node, which justifies that our metadata optimization for big directory (i.e. append vs. update) is quite efficient. FusionFS shows a linear scalability, where PVFS is saturated at 32 nodes.

### 2.4.2 IStore: an Erasure Coding Enabled Distributed Storage System

IStore is a simple yet high-performance Information Dispersed Storage System that makes use of erasure coding[68][27][42], and distributed metadata management with ZHT. Figure shows IStores metadata performance throughput on 8 to 32 nodes in the HEC-Cluster. The workload consisted of 1024 files of different sizes ranging from 10KB to 1GB. The workload performed read and write operations on these files through the IStore. At each scale of N nodes, the IDA algorithm was configured to chunk up files into N chunks, and storing this information in ZHT for later retrieval and the N chunks would be sent to or read from N different nodes.

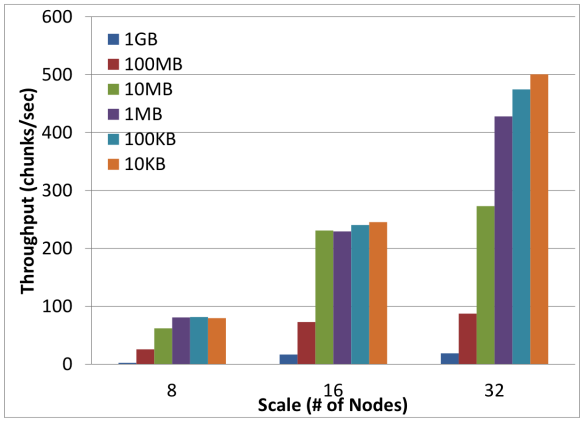



Figure 2.17: IStore metadata performance on HEC-Cluster

### 2.4.3 MATRIX: a Distributed Many-Task Computing Scheduling Framework

MATRIX is a distributed many-task computing execution framework, which utilizes the adaptive work stealing algorithm to achieve distributed load balancing, and uses ZHT to submit tasks and monitor the task execution progress by the clients. By using ZHT, the client could submit tasks to arbitrary node, or to all the nodes in a balanced distribution. The task status is distributed across all the compute nodes, and the client can look up the status information by relying on ZHT.

We performed several synthetic benchmark experiments to evaluate the performance of MATRIX, and how it compares to the state-of-the-art Falkon [77] lightweight task execution framework. Figure 2.18 shows the results from a study of how efficient we can utilize up to 2K-cores with varying size tasks using both MATRIX and the distributed version of Falkon (which used a nave hierarchical distribution of tasks). We see MATRIX outperform Falkon across the board with across all size tasks, achieving efficiencies starting at 92% up to 97%, while Falkon only achieved 18% to 82%.

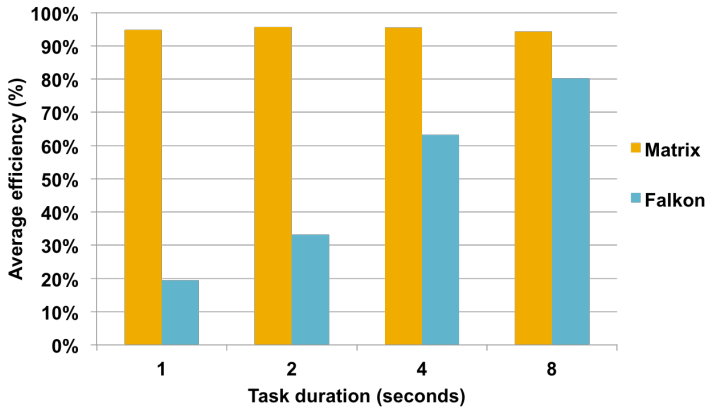



Figure 2.18: Comparison of MATRIX and Falkon average efficiency (between 256 and 2048 cores) of 100K sleep tasks of different granularity (1 to 8 seconds)

### 2.4.4 Slurm++: a Distributed HPC Job Launch

We have developed a distributed job launch prototype, SLURM++ based on SLURM [36], which serves as a core part for distributed job management system to do resource allocation and job launching. We used the ZHT as the data storage system to keep the job and resource metadata information in a globally accessible system.

We see that the average per-job ZHT message count shows decreasing trend (from 30.1 messages / job at 50 nodes to 24.7 messages at 500 nodes) with respect to the scale. This is likely because when adding more partitions, each job that needs to steal resource would have higher chance to get resource, as there are more options. This gives us intuition about how promising the resource stealing and compare and swap algorithms would solve the resource allocation and contention problems of distributed job management system towards exascale ensemble computing.

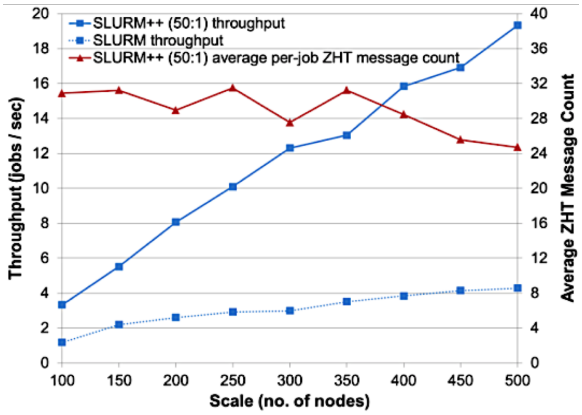


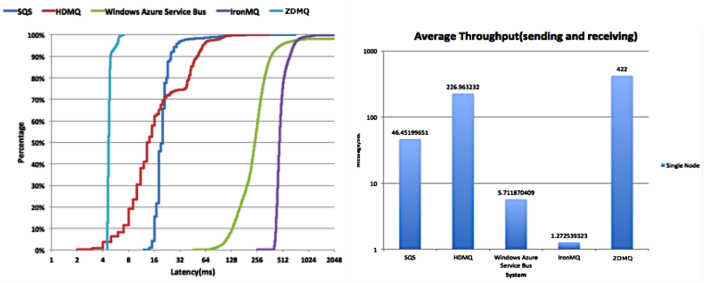

Figure 2.19: Throughput comparison for Slurm and Slurm++

### 2.4.5 ZDMQ: a Distributed Message Queue

We architected and implemented a new distributed message queuing system (ZDMQ), leveraging the highly scalable ZHT distributed key/value store, and supporting reliable exactly once delivery. ZDMQ was developed to address functionality and/or performance limitations of existing systems, such as SQS, HDMQ, Windows Azure Service bus, and IronMQ. ZDMQ consist of collection of ZHT server that can be used to store messages. ZDMQ provides replication of messages for high reliability.

The main goals of ZDMQ are to provide high throughput, low latency, single delivery high reliability and high scalability. Our inspirations were primarily SQS and HDMQ. We designed this system that stores messages in distributed key-value store that are structured in an multiple ZHT server where each ZHT server is a part of a storage where the queue messages will be stored in key-value manner.

In the preliminary testing we performed evaluation and compared ZDMQ to the commonly used commercial distributed queues measuring throughput and latency. We found ZDMQ to outperform distributed message queuing systems by up to 1.86-351x times in throughput and 3.6-177x times in latency.




(a) Latency comparison in CDF graph

(b) Latency and throughput

Figure 2.20: Comparison for ZDMQ vs. SQS, IronMQ, HDMQ, and Windows Azure Service Bus

## 2.5 Related work

In this section we introduce some existing works related to ZHT, including distributed hash tables and key-value stores. There have been many distributed hash table (DHT) algorithms and implementations proposed over the years. We discuss DHTs in this section due to their important role in building support for scalable metadata service across extreme scale systems. Some of the DHTs from the literature are Kademlia [67], CAN [79], Chord [92], Pastry[80], Tapestry[120], Memcached[72], Dynamo[25], RIAK[12], Cassandra[40]. Most of these DHTs scale logarithmically with system scales. Dynamo is a key-value storage system that some of Amazons core services use to provide an always-on experience. Dynamo calls itself as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly. Cassandra, an implementation inspired by Amazons Dynamo, strives to be an "always writable" system. Cassandras drawbacks include poor support on many supercomputers due to a lack of Java stack. Cassandra also uses logarithmic routing strategy, which makes it less scalable. Memcached is an in-memory implementation of a key/value store. It was designed as a cache to ac-

celerate distributed application execution, no persistence, replication, and dynamic membership. There are strict limitations on the size of the keys and values (250B and 1MB respectively).

Some recent key-value store projects generally focus on providing new features. For example HyperDex[28] is a distributed key-value store that provides a search primitive that enables queries on secondary attributes. Some adopt new storage backend technology to boost the performance, such as SkimpyStash[24]. SkimpyStash uses a hash table directory in RAM to index key-value pairs stored in a log-structured manner on flash. Due to the relatively simple yet complete basic functionality of key-value stores, there are also many research projects use them as demonstrative prototypes to verify their designs. For example Pileus [96] is a replicated key-value store that allows applications to declare their consistency and latency priorities via consistency-based service level agreements (SLAs).

## 2.6 Summary

ZHT is optimized for high-end computing systems and is designed and implemented to serve as a foundation to the development of fault-tolerant, high-performance, and scalable storage systems. We have used mature technologies such as TCP, UDP, and an epoll-based event-driven model, which makes it easier to deploy. It offers persistency with NoVoHT, a persistent high performance hash table. ZHT can survive various failures while keeping overheads minimal. Its also flexible, supporting dynamic nodes join and departure. We have shown ZHTs performance and scalability are excellent up to 8K-node and 32K instances. On the 32K-core scale we achieved more than 18M operations/sec of throughput and 1.1ms of latency at 8K-node scale. The experiments were conducted on various machines, from a single node server, to a

64-node cluster, an IBM BlueGene/P supercomputer, to the Amazon cloud. On all these platforms ZHT exhibits great potential to be an excellent distributed key-value store, as well as a critical building block of large scale distributed systems, such as job schedulers and file systems. In future work, we expect to extend the performance evaluation to significantly larger scales, as well as involve more applications.

We believe that ZHT could transform the architecture of future storage systems in HEC, and open the door to a broader class of applications that would have not normally been tractable. Furthermore, the concepts, data-structures, algorithms, and implementations that underpin these ideas in resource management at the largest scales, can be applied to emerging paradigms, such as Cloud Computing, Many-Task Computing, and High-Performance Computing.

# CHAPTER 3

# WAGGLEDB: A CLOUD-BASED INTERACTIVE DATA INFRASTRUCTURE FOR SENSOR NETWORK APPLICATIONS

In this section, we combined multiple distributed data structures (NoSQL database and distributed message queue) together to offer a cloud-based interactive data infrastructure for sensor network applications.

As small, specialized sensor devices, capable of both reporting on environmental factors and interacting with the environment, become more ubiquitous, reliable, and cheap, increasingly more domain sciences are creating "instruments at large" dynamic, often self-organizing, groups of sensors whose outputs are capable of being aggregated and correlated to support experiments organized around specific questions. This calls for an infrastructure able to collect, store, query, and process data set from sensor networks. The design and development of such infrastructure faces several challenges. The first group of challenges reflects the need to interact with and administer the sensors remotely. The sensors may be deployed in inaccessible places and have only intermittent network connectivity due to power conservation and other factors. This calls for communication protocols that can withstand unreliable networks as well as an administrative interface to sensor controller. Further, the system has to be scalable, i.e., capable of ultimately dealing with potentially large numbers of data producing sensors. It also needs to be able to organize many different data types efficiently. And finally, it also needs to scale in the number of queries and processing requests.

In this work we present a set of protocols and a cloud-based data store called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges above with a scalable multi-tier architecture, which is designed in such

way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

## 3.1 Introduction

The last several years have seen a raise in the use of sensors, actuators and their networks for sensing, monitoring and interacting with the environment [22]. There is a proliferation of small, cheap and robust sensors for measuring various physical, chemical and biological characteristics of the environment that open up novel and reliable methods for monitoring qualities ranging from the geophysical variables, soil conditions, air and water quality monitoring to growth and decay of vegetation[31][17][30][64][71]. Structured deployments, such as the global network of flux towers[5], are being augmented by innovative use of personal mobile devices [23, 52, 95, 54, 53, 61, 55, 59, 38, 58, 57, 62, 56, 60], and GIS(Geographic Information System) [51, 50, 63, 104, 49] use of data from social networks, and even citizen science. As small, specialized sensor devices, capable of both reporting on environmental factors and interacting with the environment, become more ubiquitous, reliable, and cheap, increasingly more domain sciences are creating instruments at large dynamic, often self-organizing, groups of sensors whose outputs are capable of being aggregated and correlated to support experiments organized around specific questions. In other words, rather than construct a single instrument comprised of millions of sensors, a virtual instrument might comprise dynamic, potentially ad hoc groups of sensors capable of operating independently but also capable of being combined to answer targeted questions. Projects organized around this approach represent important areas ranging from ocean sciences, ecology, and urban construction to hydrology.

## 3.2 Design and Implementation

### 3.2.1 Challenges and solutions

The online analysis needs of such instruments at large create the need for data store management system with the following properties:

**Write scalability**: The data store will need to be able to sustain many concurrent writes generated by thousands of sensor controller nodes that continuously send data to the database with the same quality of service, i.e., reliability (not losing any writes), time to acknowledge, etc. For achieving these goals, we propose to use a multi-layer architecture. A high performance load balancer is used as the first layer to accept and forward all write requests from sensor controller nodes evenly to a distributed message queue. In our system the message queue works as a write buffer and handles requests asynchronously. A separate distributed data agent service keeps pulling messages from the queue, preprocess it and then write to the data store. On the client side (sensor controller nodes), we adopt a collective sending method, which accumulates the sensor data and send a batch of message to the cloud periodically. The client sending frequency is customizable so to meet the different needs of data freshness.

**Support for various data types**: Since sensor data can be of various types and sizes; there is no fixed scheme for the data formats from all the different types of sensor. Therefore we need a flexible data schema so to enable a unified API to collect and store the data, and to organize data in a scalable way for further use (query/analytics). To address this issue, we design a flexible and self-describing message data structure that easily fits into a large category of scalable distributed databases, called column-oriented databases (or BigTable-like data stores[19]). Pop-

ular solutions include BigTable, Cassandra[40], Hive[97], HBase[7] etc. This design enables us to elevate the rich features, performance advantage and scalability from column-oriented databases, as well as to define a unified data access API.

**Transactional interaction between admin users and sensors**: Administrators need to push commands or queries to sensor controllers for development or maintenance purposes. However sensor network connection is not reliable and the connection can be lost any time, which makes conventional remote login mechanism such as SSH fail to work. We use a database table to track the sessions between admins and sensor controllers. A session contains all communication events and their orders. Admin submit a series of commands and the nodes that will run these commands to the system database, sensor controllers check out the commands from the same database whenever they are online. The controllers responses are also push to the database for admins.

**Dynamic scalable services**: The request rate can change in a wide range. The system needs to be able adjust its capability to satisfy multiple requests for processing with qualities of service. We distribute the functionality to independent tiers in the architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

### 3.2.2 Architecture

We design a loosely coupled multi-layer architecture to boost the scalability while maintaining a good performance. As shown in figure 3.1, the system is composed of a sensor controller node and a data server that is both written to by the sensors and read from by the clients. On the server side, there are 5 layers of components, namely load balancer, message queue, data agent, database, and query execution

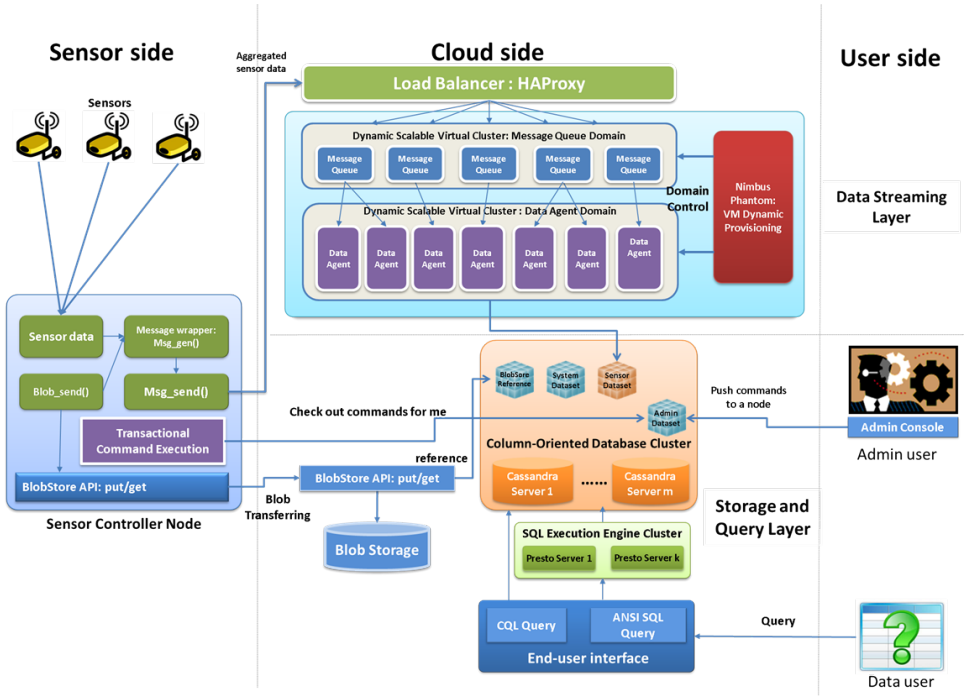



Figure 3.1: System architecture. Sensor controller nodes send messages and blobs to the cloud storage through APIs. Load balancer forwards client requests to data streaming layer. Nimbus Phantom[37] controls dynamic scaling of queue servers and data agent servers.

engine. Each layer can be deployed on a dedicated or shared virtual cluster. If any layer becomes bottleneck, it can be scaled easily by simply adding more resource.

### 3.2.3 Implementation

We have implemented a functioning prototype of WaggleDB and deploy it on a public cloud platform, FutureGrid. We use RabbitMQ as the message queue, Cassandra as the database, HAProxy as the load balancer. All tiers are deployed on multiple VMs. The prototype now has following features: accepting and storing data from different type of sensors, a blob store for potential big files from sensors such as full-spectrum cameras, transactional interaction between admin and sensor controllers, sensor and

controller registration, CQL and SQL query on the database, dynamic scalability.

### 3.2.4 Performance Evaluation

We have conducted a preliminary performance evaluation of WaggleDB system on FutureGrid, which is an OpenStack based public cloud. Since we dont have many sensor controller nodes at this point, we run the clients on virtual machines and send random data in a tight loop to a WaggleDB queue server as a simulation.

In this experiment, we use the number of clients and message size as parameters. We measured request latency from client side, message queue processing bandwidth and requests throughput on the queue server side. Figure 3.2 shows that the latency slowly increases with the concurrency level but much better than being linear. Worth to note that the message size only has very few influence on latency. This indicates that the major parts of overhead consists connection creation and closing, but not network transferring. This is also proved by our bandwidth measurement, as shown in figure 3.3. Clients have proportionally high bandwidth when using bigger messages. With 32 clients sending messages of 10,000 bytes, the system reaches 50MB/s bandwidth, while it only has 59KB/s with 10 bytes messages. The request throughput benefits from adding from concurrent clients as well.

When adding concurrent clients up to 128, the latency increases rapidly for one server, while the systems that have more servers perform better. Single server is saturated at 32 clients scale, 2, 4 and server systems saturated at 64 and 128 clients respectively. 8 servers system performs still good. This implies excellent server scalability.

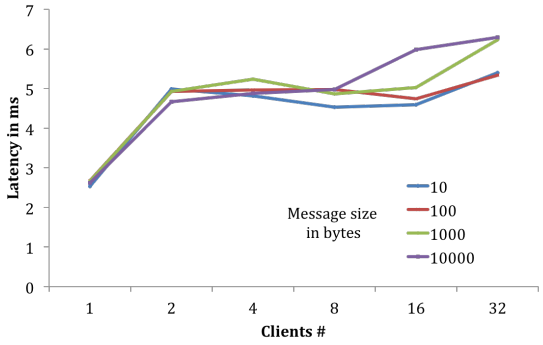



Figure 3.2: When the client scale increased by 32 times, the average latency only increased by 2.2 times. The differences between the latencies of 10 to 10k bytes message sizes were small.

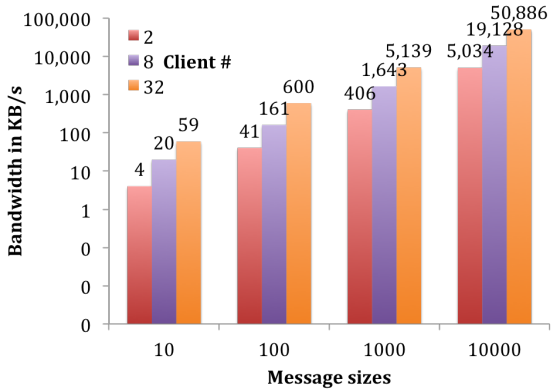



Figure 3.3: he bandwidth gain from bigger message size is close to that from adding more clients.

## 3.3 Summary

The emerging sensor network usage calls for an infrastructure able to collect, store, query, and process data set from sensor networks. We present the challenges and gives tentative solution through WaggleDB, a cloud-based interactive data infrastructure for sensor networks. WaggleDB is elastic on both scales and data presentation, and shows excellent potential to scale each tier in the architecture.

# CHAPTER 4

# SCALABLE STATE MANAGEMENT FOR SCIENTIFIC APPLICATIONS ON CLOUD

In the cloud era people are building increasingly bigger distributed application systems for different purposes. In recent days complex distributed applications can be developed and deployed more easily due to the presence of a wide variety of components and frameworks[106, 105, 93, 110], such as distributed data storage systems, distributed message queues, publish-subscribe systems and so forth. Data storage is one of the most important. This category covers a wide range of systems, such as file systems, SQL databases, NoSQL databases, blob stores, object stores and so forth. They play critical roles in stateful distributed system design and development, as most of such systems need globally accessible storage, such as a database or a file system. Especially NoSQL databases, with their help, the users atop can write their applications or upper layer easily while enjoy the advantages in performance, capacity and scalability.

However, using NoSQL instinctive does not always bring the best performance. In this section we describe a type of applications that not work the best when using NoSQL database in conventional manner.

The data generated by scientific simulations and experimental facilities is beginning to revolutionize the infrastructure support needed by these applications. The on-demand aspect and flexibility of cloud computing environments makes it an attractive platform for data- intensive scientific applications. However, cloud computing poses unique challenges for these applications. For example, cloud computing environments are heterogeneous, dynamic and non-persistent which can make reproducibility a challenge. The volume, velocity, variety, veracity and value of data combined with the characteristics of cloud environment make it important to track the state of ex-

ecution data and applications entire lifetime information to understand and ensure reproducibility. This chapter proposes and implements a state management system (FRIEDA-State) for high-throughput and data-intensive scientific applications running in cloud environments. Our design addresses the challenges of state management in cloud environments and offers various configurations. Our implementation is built on top of FRIEDA (Flexible Robust Intelligent Elastic Data Management), a data management and execution framework for cloud environments. Our experiment results on two cloud test beds (FutureGrid and Amazon) show that the proposed solution has a minimal overhead (1.2ms/operation at a scale of 64 virtual machines) and is suitable for state management in cloud environments.

## 4.1 Introduction

Data analysis is central to next-generation scientific discoveries. Cloud is as an emerging platform and increasingly attractive to scientists due to its flexibility and convenience. But cloud environments are typically transient. Virtual machine instances are terminated after applications complete execution. Users cannot leave data and/or revisit the resource setup to diagnose discrepancies. In the cloud environment, users have the responsibility to capture everything before the virtual machines are shutdown.

Big data scientific applications need to track every step of the scientific process, data access and environment for lineage, reconstruction, validity and reproducibility purposes. It is important to know the environment in which the applications run (e.g., floating point operations could give different results on different machines). Users might also wish to "rerun" some (e.g., only what failed) or all of the tasks.

Provenance tools have tracked workflow and data lineage at various levels

(e.g., operating system [70], file systems[69], databases[16], and workflow tools ([128] [77] [129][75]. Many monitoring tools ([13] [65] [85]) have been developed to monitor real-time system changes. These systems provide methods to collect, aggregate, and query monitoring data. However, this data is often insufficient for reproduction since they do not capture human knowledge. Furthermore, state management in cloud environments needs to tackle additional challenges due to its characteristics. First, the transient nature of the environments makes it important to capture metadata and state at various levels. Second, the performance and reliability characteristics of virtual machines is important to consider in the design of the collection system. Finally, different clock drifting rates on physical machines make it hard to have a unified time view for the end-user to rebuild meaningful semantics.

In this chapter, we propose FRIEDA-State, a state management system for cloud environments. We use the term state to represent the metadata from both execution framework and applications. FRIEDA-State addresses the transient nature, performance concerns and clock drifting issue[73] in its design. FRIEDA-State is currently implemented atop of FRIEDA [32], a data management and execution framework for cloud environments, which supports a high-throughput and data-intensive scientific applications,. We present a key-value based collection system to manage state in dynamic transient environments. We design and implement a vector clock[66] based event-ordering mechanism to address the clock drifting issue.

FRIEDA-State collects static and dynamic state data. Static state data is the information that doesnt change when the system is running (e.g., CPU/Memory info, environment variables and software stack information). Dynamic state data, on the other hand, changes during application running, such as the information on details of the input file that is processed, the time taken for a machine to finish execution or failure of jobs.

Specifically, the contributions of our work are:

- Design and implementation of FRIEDA-State, a state management system for scientific applications running in cloud environments, with lightweight capturing, efficient storage and vector clock-based event ordering
- Evaluation on multiple platforms (FutureGrid and Amazon EC2) at scales of up to 64 VMs; results show good efficiency with minimal overhead (1.2ms/operation at 64-node scales)

## 4.2 Background

In this section, we describe the background required to understand the design considerations in FRIEDA-State.

### 4.2.1 Use cases

Scientific applications need to track their scientific process for a number of reasons including a) real-time monitoring b) tracking data lineage c) validation of results d) reconstruction or repeating some or all of the experiments and, e) reproducibility of research results. Users might want to track their configuration and environment settings and repeat some or all of the experiment or validate a certain result (e.g., floating point). The state information might also be used for post-execution analysis. For example, the users might like to query job statistics and understand why some jobs took longer than others. Users might want to rerun the same experiment and/or run the same experiment with slightly different parameters.

### 4.2.2 Challenges

Next, we discuss the challenges on state collection, storage, and event synchronization. State collection and storage: State information is generated on each of VMs and multiple VMs are part of an application execution. High capture latency may degrade the application performance. Thus, scalable collection of data is important in the design of FRIEDA-State. Information aggregation and appropriate storage mechanisms are also important and different solutions might have different trade-offs. Centralized storage system (e.g., databases) could result in concurrent read/write bottlenecks and be the source of single-point failures. Distributed solutions often suffer from high operation latency and often require extra dedicated hardware. Event Synchronization: Cloud environments are dynamic. Virtual machines may not run on the same physical machines. This implies that the physical time clocks may not run at exactly the same speed because of the slight difference between crystal oscillators on different machines thus result in drifting [73]. With different drifting rates, at the end of a long run, the base-time gap between each virtual machine could be big. The drifting issue is serious in large-scale distributed systems and is even more serious due to transient nature of clouds. Synchronized bootstrap time clocks may not be guaranteed in distributed cloud environments. It is important that the events/states captured on the machines is unified for the end-user to build meaningful semantics.

### 4.2.3 FRIEDA framework

Our state management system is built on top of FRIEDA[32]. FRIEDA is a Flexible Robust Intelligent Elastic Data Management framework. FRIEDA manages the lifecycle of data that includes storage planning and provisioning, data placement and application execution of scientific applications in cloud environments.

FRIEDA enables users to plug-in flexible data management strategies for different application patterns by separating data control from execution. FRIEDA supports a Master-Worker execution model. There are three major components in FRIEDA architecture, namely controller, master and workers. The controller takes charge of environment setup and configurations for data management and application execution. The master is responsible for managing application execution and data distribution. The workers accept data and computation jobs from the master and execute them locally. After all workers finish their jobs, the framework will collect output data from all nodes. State management system collects information from the resource provisioning and execution phases.

## 4.3 System Design and Implementation

Figure 4.1 shows the system architecture of the state management system (FRIEDA-State). FRIEDA-State has a collection component in each of the main FRIEDA actors the controller, master and worker. FRIEDA-State works in two phases: capturing and storage. It allows multiple storage solutions to be plugged into the framework to meet different usage needs. The runtime state capture component collects two types of state: static and dynamic. The static states are collected mainly from a configuration YAML file, which is used in FRIEDA to configure the virtual machines (e.g. it defines the roles and the setups of the master and workers). The YAML file is populated in the state management system from the controller node once the experiment starts. The remaining static information (e.g. system information) is collected from the worker virtual machines directly. Dynamic states are captured from the FRIEDA framework through built-in functions. Once captured, states are encapsulated in key-value pairs and pushed to one of three storage solutions that

is selected by the user. FRIEDA-State currently supports raw files, Cassandra or DynamoDB (on Amazon Web Services).

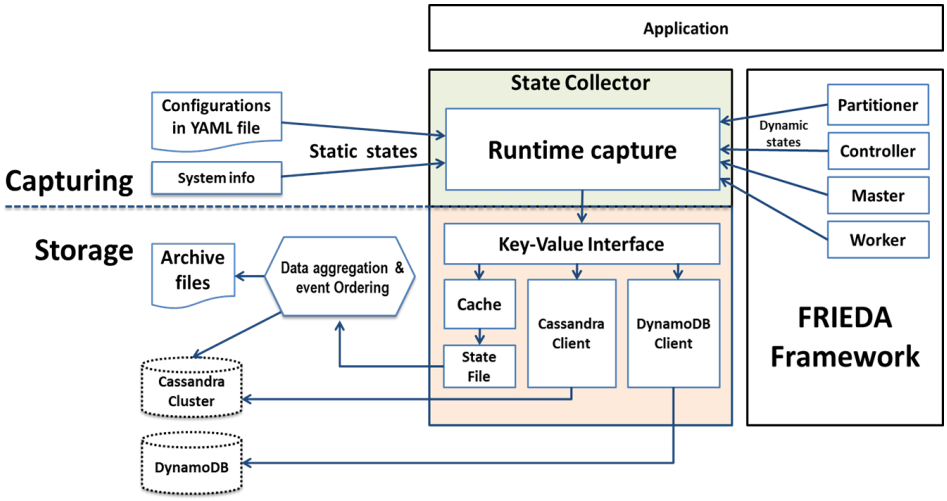



Figure 4.1: FRIEDA-State system architecture. Capturing is the first layer. State collector captures static states from two source, configuration YAML file and system information. Dynamic states are captured from FRIEDA-State functions, which are called in FRIEDA framework. Captured states are encapsulated into the form of key-value pairs and pushed to one of three storage solutions as selected by the user.

### 4.3.1 State Description

Each state in our system has the following fields. The state name is used as the key and the rest are used as values.

**State name**: This is used to represent the type of event.

**State information**: This field captures the state content.

**Role**: This field captures the source of the event or the role of the host (i.e., master or worker),

**Hostname/IP**: This captures the identity of the host where the state was collected.

**Logical timestamp**: We set a field for logical time for ordering the events captured

from distributed nodes. The logical timestamps is used as a part of vector clock.

**Local timestamp**: The local timestamp is also captured that indicates the time when the event is captured on a local host and be used to order events within a virtual machine.

### 4.3.2 Static state capture

Static state represents the data that will not change during application execution. In FRIEDA, most of the static states are covered in a configuration YAML file. The YAML file includes platform name, image ID, instance type, authentication information, application details. The YAML file allows users to setup environment, software installation and the application running details. The YAML file is loaded into memory and stored as structured data items and then dumped into a data store or state file as a record. Other static states, such as hardware info (CPUINFO/MEMINFO), software stack information and so on are captured when the virtual machines are launched.

### 4.3.3 Dynamic state capture

Dynamic state information represents the data that changes during the run-time of applications. For example, the identification of input files processed by a worker and the duration of the execution are captured by FRIEDA-State. We capture two types of information a) communication events and b) application execution details.

All communication events, such as connection made (and to which machine), data received (and from where), etc., are captured. These events not only describe the communication itself, but also carry vector clock information for later event reordering (section III.F). Application execution and data flow details include the commands

executed on each worker, I/O operations and application execution time etc. This can be used to track the application execution and to analyze run-time problems.

### 4.3.4 State Storage

The essence of state management is to capture data in distributed environment and store for future queries. For designing such a scalable system with low latency, the major concern is storage architecture. The state operation latency must be very low to prevent degradation of the application performance. Scalability is also important since the storage system could be a bottleneck when serving many clients for writing and/or query. FRIEDA-State currently supports three storage options: files, Cassandra data store and DynamoDB (on Amazon). This allows users and applications to select the right storage while accounting for the tradeoffs for their needs.

**Files:** This mode uses files for capturing state. Captured states are first written to files, which are later aggregated from all machines at the end of execution. Files as a storage mechanism provides some advantages over key-value stores and databases. First, simple memory-file operation is significantly faster than single node key-value stores, due to the fact that memory-file operation executes sequential writes while key-value stores execute writes randomly (hash table or B-Tree). Second, file-based mechanisms do not require any additional services and hence does not add any overheads on the nodes. Third, merging files are simple and fast since the files are already naturally ordered due to sequential writes on each node. Therefore sorting the state files has a linear time complexity. Files are not good to query on, but they are easy to manipulate and archive.

We capture states from all FRIEDA components on VMs and store them in an in-memory cache before being flushed to disks. The cache size is customizable to

address the tradeoffs between robustness and performance. If application fails, the states can still be found on those VMs since they are flushed to disk. For even better durability, it can write to a distributed file system or a block store that can survive beyond the life of the virtual machine, based on the configuration. Finally all states files are copied to a target machine for merging. If application fails, states are still saved within the FRIEDA-State framework. But if VMs or FRIEDA fails during execution, users might lose unsaved states.

**Cassandra**: We include Cassandra as one of the storage solutions[40], as it provides rich features for managing semi-structured data. It is easy to plug-in other NoSQL databases in FRIEDA-State. Users control the number of Cassandra instances according to their performance and capacity needs. Cassandra could share the virtual machines with the application or run on a separate cluster. Key-value stores are known to work well when deployed on dedicated machines. Practically, users can use a private cluster to host the Cassandra cluster. They can also setup a dedicated virtual cluster on cloud to serve the requests. In this case, users will need to periodically move state data to a more permanent storage.

**DynamoDB**: The third storage solution is based on DynamoDB, a NoSQL database available on Amazon Web Services. With this type of cloud databases services, users dont have to deploy a software stack to run and configure those data stores, but have to pay extra money for the service.

### 4.3.5 Storage architectures

FRIEDA-State supports multiple storage architectures: centralized, distributed and local storage. This sub-section will describe each of these architectures in detail.

**Centralized Storage Solution**: In our current implementation, centralized

solution is based on a single node key-value store or database. When all nodes in the system generate states, collecting and storing can result in a storage bottleneck. Depending on the application type, the rate of state generation can be different. If the data generated is minimal and at a low rate, centralized storage solution will work perfectly in practice. An important advantage of centralized solution is that all events can be naturally ordered as they arrive at the storage server and assigned a timestamp based on servers local time. The solution naturally provides persistence of the state data beyond the lifetime of the virtual machines.

**Distributed storage solution**: Similar to centralized solution, FRIEDA-State support distributed storage solution through NoSQL databases (e.g., Cassandra). High write concurrency is a big challenge for all types of storage systems. Distributed storage solutions, such as distributed databases, key-value stores can serve large amounts of write requests and spread them to many nodes to achieve scalable performance and load balancing[18][19]. In this type of solution, a group of dedicated data storage servers will be started prior to application execution. States generated on any node in the system will be written remotely to the data store. The latency of this operation depends on the data store solution and could be up to a few milliseconds [18]. To deploy data stores on all the nodes that will generate states will not help much on performance, because running the data stores consumes extra CPU and memory resources, and messages still need to be sent between all VMs over the network.

**Local storage solution**: We implemented this solution based on traditional files. Local storage eliminates network latency (the major part of operation latency).. Considering the data collected is likely to be queried after an application run, offline storage solutions are reasonable. Writing to local disk/memory is extremely fast compared to remote access and no extra resources are consumed by data store programs

on VMs. The hard part of using local storage is aggregating data from many VMs and to merge into a form that offers a single query interface. File-based solution is not perfect though. If users want to query the states during the system running, extracting the desired records is complicated and slow.

### 4.3.6 Event reordering

Distributed event ordering is an important topic in large-scale distributed system design. The goal is to keep a global logical order of events based on timestamps. In FRIEDA-State, we use modified vector clock[66] to maintain time order. FRIEDA-State uses 1-to-n communication pattern since communication only happens between master and workers. We use the master node clock as major clock and all workers logically synchronize to it using vector clock. By comparing attached master clock value in communication messages, we can tell which event happened earlier. If the master clock reads are same, then these events occurred in the same machine. It is trivial to order events within one machine by sorting the local timestamps. When using file-based storage solutions, events are sequentially written to files and thus are naturally ordered. Each state record has two fields for vector clock: one for local clock, another for master clock. Events on master node have same values for both clocks.

In the beginning, all logical clocks are set to be 0. Once a new state is captured and stored, the corresponding clock value is increased. The local clocks increase naturally along with the events happening, and the master clock can only be updated when a message is received that contains a new master clock value. Workers states collection can be divided into independent event groups by master clock value. In each group, events are naturally ordered and do not interleave with those in other groups. The possible causal relation between different groups, if exist, is determined

by master clock. Thus, the problem of reordering and merging different events is reduced to sorting the event groups. Sorting groups is simple and has the same time complexity as the merging phase in merge-sort, namely O(n), where n is number of groups. For example, in 4.2, sorting by master clock M value, all the events are divided into five groups. Each group presents an atomic sequence in a machine. Since the inner events of a group are naturally ordered, the reordering is efficient.

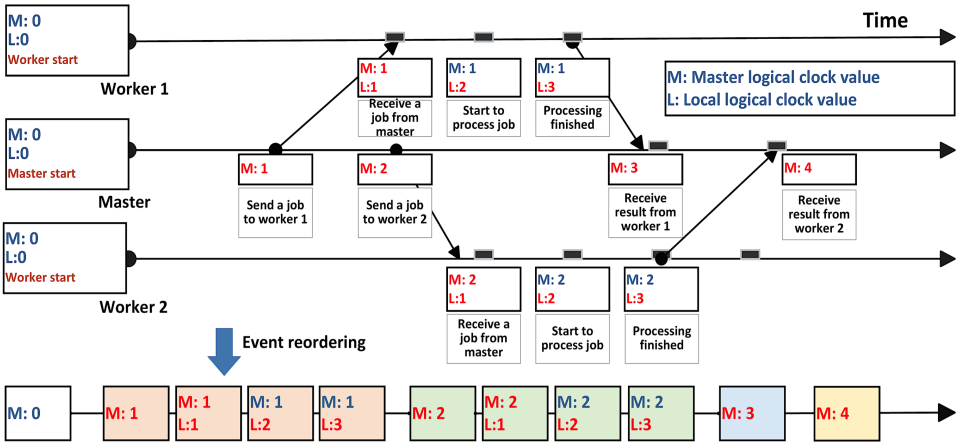



Figure 4.2: Event reordering example. Three machines have their local clock and maintain a vector clock. Each event will increase the local clock value; each received message will update others clock value in their local vector. The masters clock is used to maintain the time order when reordering the events. Sorting by master clock M value, all the events are divided into 5 groups. Just sort the groups will order all the event.

## 4.4 Evaluation

In this section, we describe the performance of the state management system, with different storage solutions.

### 4.4.1 Testbeds

- FutureGrid Sierra: a research purpose public cloud, experiments used up to 16 virtual machines.
- Amazon EC2 cloud: up to 128 c1.medium virtual machines, each has 2 virtual CPU, 5 elastic compute units and 1.7GB memory.
- DataSys: an 8-core x64 server at IIT: dual Intel Xeon quad-core w/ HT processors, 48 GB RAM. This machine is used for experiment to study the merging overhead.

### 4.4.2 Scientific Workloads

We use two applications that are representatives of scientific workloads using cloud environments: Image Comparison and Event Processing. Image Comparison compares an image with other images in the set to find similarities. These applications are representative of typical data processing scientific workflows.

### 4.4.3 Experiment Setup

We use the same workload for three storage solutions, respectively based on files, Cassandra and DynamoDB. For synthetic benchmarks, on each state client, we send 10K requests in a tight loop to simulate an extremely operation-intensive scenario. Each request consists of 20 bytes key and 80 bytes value. Both key and value are randomly generated.

**File-based solution**: For file-based solution, the key-value pairs are saved to a local file on each client. Next, all these files are copied to a shared NFS directory,

located on a dedicated VM where the files are merged. This is a simple solution for demonstrating state aggregation. Apparently its vulnerability to single point failures and the bottleneck can be addressed by well-known techniques such as mirroring or parallel file systems. We measure the time of writing to files, moving files to NFS server and merging events. We amortize the cost of file moving to state storage to get the average equivalent latency per state.

**Cassandra-based solution**: We use 1 to 8 Cassandra servers on dedicated VMs, and send requests from 1 to 128 state clients on VMs. **DynamoDB-based solution**: DynamoDB is a service provided by Amazon. The data servers dont need to be deployed on the VMs. The VMs only need to communicate to the remote Amazon data stores and this has a minimal performance impact on local VMs. We provision the maximum available throughput for DynamoDB, which is 10K ops/sec. Up to 128 VMs on Amazon EC2 are used as state clients to send requests.

### 4.4.4 Metrics

The metrics measured and reported are average latency and throughput.

**Average Latency**: We consider the average latency as per request to write a state to data stores, measured in microseconds. Note that the latency includes the round trip communication and storage access time. Measuring latency for Cassandra and DynamoDB is straightforward, but file-based solution needs more care. We use the formula below to calculate the average equivalent latency $t_{ave}$ for file-based solution, where $t_w$ is the average file write latency, $T_{moving}$ and $T_{merging}$ are the total time spent on moving and merging respectively, n is the total number of operations:

$t_{ave} = t_w + (T_{moving} + T_{merging})/n$

**Throughput**: The number of operations the system can handle over some period of time, measured in Operations per seconds (Ops per second).

### 4.4.5 Synthetic benchmark

**Capture overhead**: We conduct micro benchmarks on scales of up to 128 VMs. Note that the latency of file-based state management includes amortized cost for file moving, reordering and merging. Cassandra data stores crashed frequently and cannot serve requests at a scale of 8 servers with 128 clients. Similarly, DynamoDBs maximal throughput is reached at this scale and started to give errors, thus we only show results at a scale of 64 clients. Figure 4.3 shows that file-based state solution has significant advantage over other two capture methods. When clients number increases, moving files to a single server causes contention. But this cost gets amortized across all requests. A single node Cassandra is saturated with 8 clients. On larger scales, multiple servers show some benefits but it is still limited, compared to the file based approach. DynamoDB shows very stable performance when facing different client request pressure before it is saturated, but its at least three times slower than Cassandra at most scales.

Similar to the latencies, file-based solution delivers significantly higher throughput than other two. At 64 nodes, file-based solution achieves 52K ops/s, which is five times faster than a dedicated eight nodes Cassandra cluster and 18 times faster than DynamoDB based solution.

**Events reordering and file merging overhead:** On an 8-core Xeon server, we generate up to 512 state files. Each state file contains 10000 state records. Using a simple single thread merging program, 4 files cost 16ms, and 512 files cost 8209 ms. This can be further improved with more sophisticated merging algorithms in the future.

**File-based state management**: We measured the time for capturing state and writing to file, moving and merging files respectively. We set an in-memory cache

to boost the disk write performance. As shown in Figure 4.6 a full-size cache setting brings around 10% performance increase.

Since the state capturing on a local machine doesnt involve any contention, the latency is actually constant, around 500us. Simultaneously moving a large number of files can cause contention, either on network or disks. The time spent on moving files keeps increasing even when the time is amortized. Better methods to aggregate data will be needed when running at larger scales. Merging overhead increases as well, but still negligible compare to other two overheads.

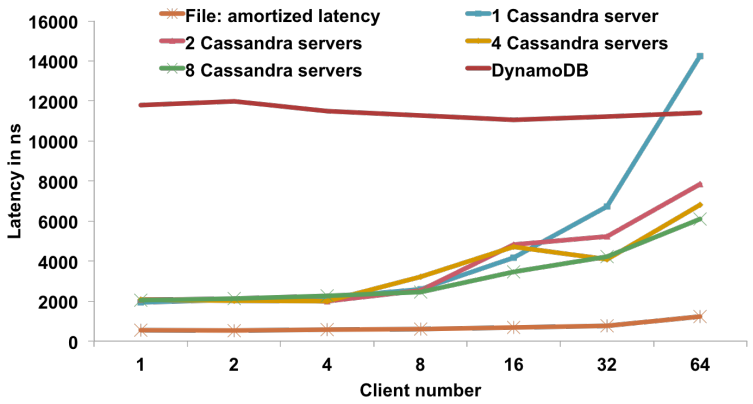



Figure 4.3: File-based solution has lower average latency. Cassandra performance decreases with the scale. DynomoDB latency doesnt change much with scale, but failed 128 clients test.

### 4.4.6 Scientific applications

With integrated state management system in FRIEDA, we run two scientific applications (Image Comparison and Event Processing) to evaluate the overall performance impact of state collecting on real applications. Both applications are evaluated on FutureGrid [6] system.

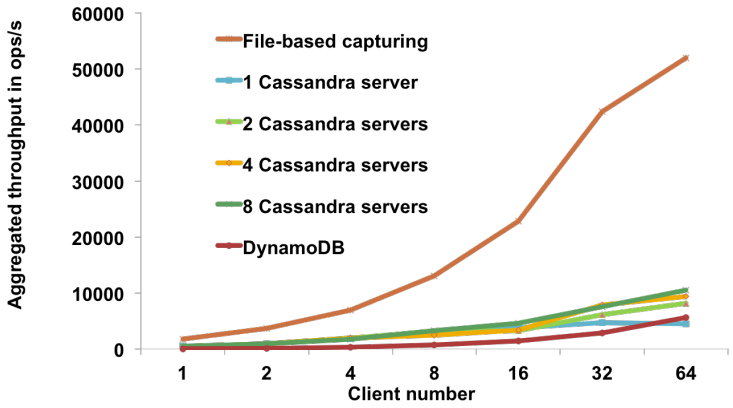



Figure 4.4: System throughput comparison. File-based solution obtains the maximum aggregated performance increases with scale.

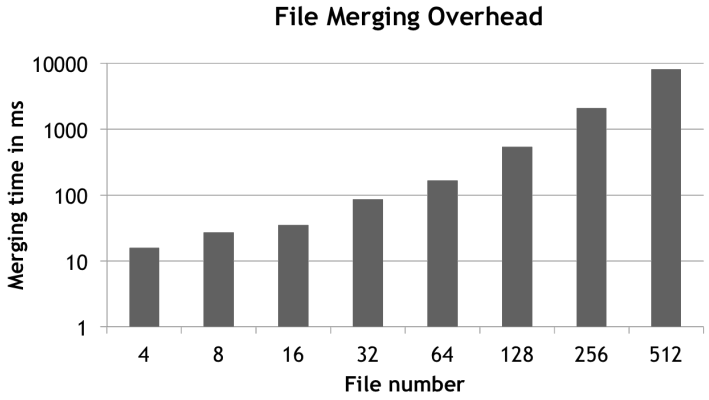



Figure 4.5: File merging time becomes longer with number of files.

Although in synthetic benchmarks we observed huge difference of performance among different storage solutions, in application tests, we see no significant difference (state-introduced overhead is less than 5%). This is mainly because the micro benchmark tests execute operations in a tight loop while real applications have sparser and random patterns, so the total time spent on state management is very low compared to the application running time.

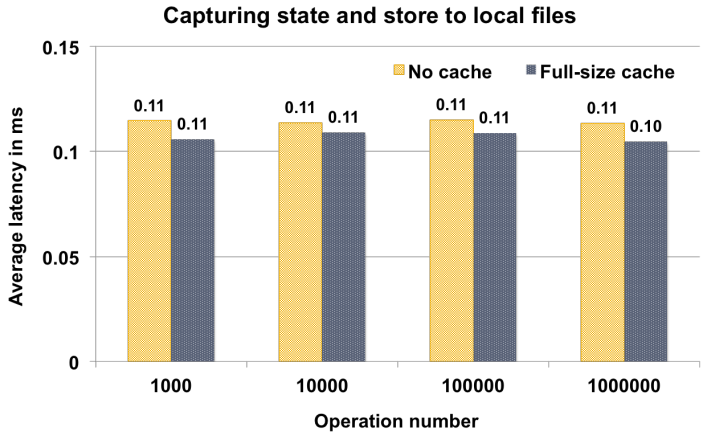



Figure 4.6: File write operation scales well. Full-size cache brings around 10% performance gains

## 4.5 Related work

### 4.5.1 Provenance

Traditional data provenance represents the change history of data objects. Previous works on data provenance [90] have addressed different aspects, from operating system [70] to file systems[69] to from databases [16]. In our previous work [48], we have shown that distributed key-value stores can boost performance. Karma [91] provenance framework gives a set of tools for collecting provenance from workflow and process. Milieu [20] focuses on provenance collection for scientific experiments in HPC systems.

### 4.5.2 Monitoring

Monitoring gives users a perspective that combines resource utilization, cost efficiency and performance. Previous work has focused on runtime model and attempt to reach the balance between runtime overhead and monitoring capability[13]. Earlier works

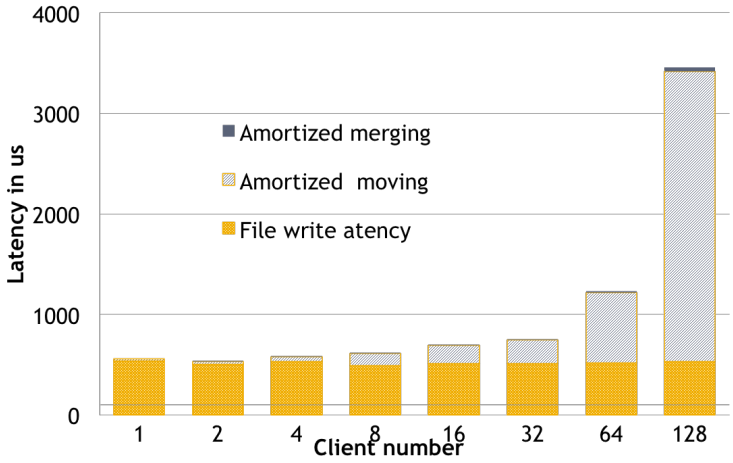



Figure 4.7: File-based storage write latency is constant while merging time slightly increases. The amortized moving time increases exponentially.

include Ganglia [65], a distributed monitoring system for clusters and grid systems. FRIEDA-State is event-driven i.e., it does not proactively go to fetch information, and hence is more efficient.

### 4.5.3 Key-value stores

Key-value stores (or distributed hash tables) are widely used as building blocks in many production systems, such as Amazon shopping cart with Dynamo[25], Facebook with Memcached [72]. Active key-value store projects include Cassandra[40], ZHT[48] [46], Riak[12] and CouchDB[3]. This approach has many advantages, such as simplified API, encapsulated communication methods, the promise that to inherit desired features from key-value stores such as load balance, fault tolerance and scalability.

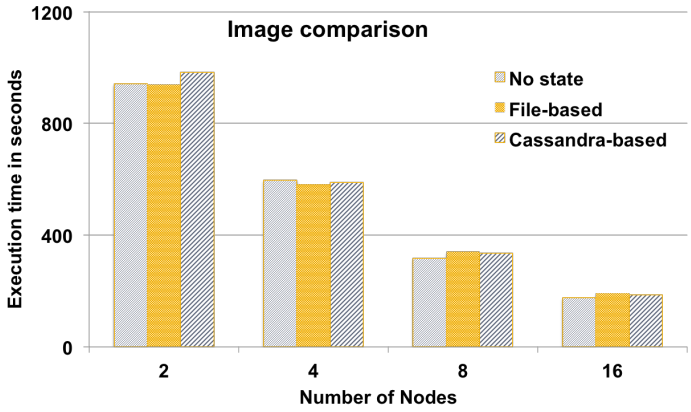



Figure 4.8: Application: image comparison

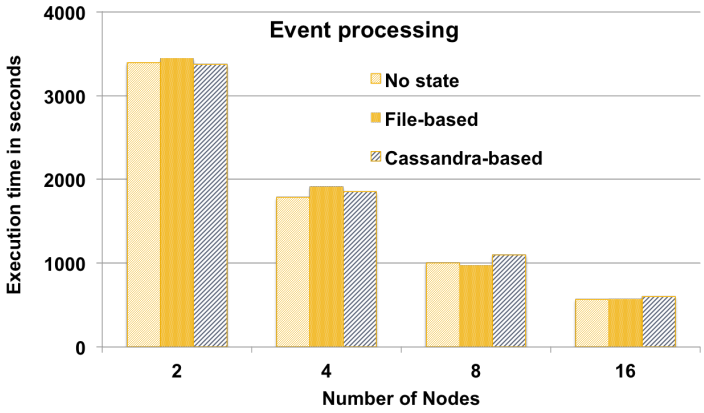



Figure 4.9: Application: event processing

### 4.5.4 Unsynchronized Time Clocks and Event Ordering

In large scale distributed systems, unsynchronized clocks and drifting issue are inevitable. Based on different time baselines, its hard to build meaningful semantics from distributed events or logs without synchronization or logical clock mechanisms.

Synchronization to a standard time source (atomic clock or GPS clock) is simpler. Typical cases are Precision Time Protocol [11] and NTP[39]. In recent projects, Google Spanner [21] adopts similar way to offer a synchronized clock to global scale databases and offers 5ms accuracy in global scales. Many works have

been done for distributed event ordering. Beside Lamports timestamp [41], Vector Clock[66] is another popular approach in todays systems.

## 4.6 Summary

Scientific applications are increasingly using cloud environments and need a way to track the applications entire lifetime information both for monitoring and ensuring reproducibility. We propose and implement a state management system (FRIEDA-State) for a broad type of scientific applications running in cloud environments. FRIEDA-State has an innovative design that allows various storage mechanisms to be plugged-in while providing different trade-offs in durability, performance and usability. In this paper, we discussed our implementations based on files, Cassandra and DynamoDB respectively and evaluated them on two cloud platforms. The evaluation showed that FRIEDAState has very low overhead even when running at a scale of 64 virtual machines. File-based storage solution offers significantly better performance than key-value stores (e.g. Cassandra) on moderate scales. Furthermore, in some conditions, file-based storage is better than cloud databases services (e.g. DynamoDB) as well, in terms of latency and aggregated throughput. The major part of overhead of file-based storage solution is file moving, when using a centralized data server. Further scalability can be achieved with better merging algorithms for file-based systems or deploying larger number of NoSQL data nodes. We expect that as we increase scale into 100s and 1000s of VMs, that the centralized data server will become a bottleneck, and distributed key-value stores would begin to offer better performance.

# CHAPTER 5

# FUTURE WORK

Based on ZHT, we have many ideas for future work. There are also many possible use cases where ZHT could make a significant contribution in performance or scalability. Among them, we plan to extend our current work into the following projects.

## 5.1 Collective request handling and buffering in Key-Value Storage: ZHT+

Various applications can tolerate very different latency ranges. For example, a shopping cart application can satisfy customers with 50 ms latency, instant messaging users are totally fine with 500ms while a metadata service for databases or file systems requires as low as possible latency, ideally no longer than 5ms. Giving all application same efforts and optimizing on the same aspect (single request latency) is not necessarily appropriate, because it may harm the total provided bandwidth of the system, thus lower down the resource utilization. This is especially true when storage system or database are considered as services and need to server many different applications and users. We propose to introduce a collective request handling and buffering mechanism to key-value storage so that each request from different applications can be treated differently. As a result, requests latency requirements can be satisfied while the total system bandwidth can be increased greatly. Next, we propose to further optimize the system for multiple metrics and multiple applications. Beside latency limit, we will also enable more options to users, such as consistency level, replica number, such that users can reconfigure the key-value store on-the-fly.

## 5.2 Integrate ZHT with Swift parallel scripting language

Swift is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows. It supports applications that execute many tasks coupled by disk-resident datasets - as is common, for example, when analyzing large quantities of data or performing parameter studies or ensemble simulations. Were closely working with Argonne National Lab on MTC (Many Task Computing) Swift so as to boost its performance and scalability through ZHT.

CHAPTER 6

RESEARCH TIMELINE AND PUBLICATION SUBMISSION PLAN

## 6.1 Timeline Milestones

- 10/2014 Comprehensive Exam
- 01/2015 Hierarchical ZHT
- 02/2015 ZHT+ Multi-metric Optimization
- 04/2015 ZHT Swift integration
- 07/2015 Dissertation first draft
- 10/2015 Dissertation Defense
- 12/2015 Graduation

## 6.2 Publication submission plan

- Extended WaggleDB work: IEEE/ACM CCGrid 15 (10/27, 2014)
- ZHT+ initial work with request collective handling and buffering: HPDC15 (1/12, 2015)
- ZHT MPI implementation: ICDCS/HDPC workshop (Jan/Feb, 2015)
- ZHT+ with multi-metric optimization SC15 (Apr, 2014)
- ZHT+ extended with QoS/SLA: IEEE transactions
- ZHT integrated with Swift: SC15 (Apr, 2014)

BIBLIOGRAPHY

[1] Amazon ec2 cloud. http://aws.amazon.com/ec2/. Accessed: 2014-11-30.

[2] Argonne leadership computing facility. https://www.alcf.anl.gov/. Accessed: 2014-11-30.

[3] Couchdb. http://couchdb.apache.org/,2014. Accessed: 2014-05-30.

[4] DynamoDB. http://aws.amazon.com/dynamodb/. Accessed: 2014-12-15.

[5] FLUXNET: Integrating worldwide co2, water and energy flux measurements. http://fluxnet.ornl.gov/. Accessed: 2012-1-30.

[6] Futuregrid. https://portal.futuregrid.org/. Accessed: 2014-11-30.

[7] Hbase. http://wiki.apache.org/hadoop/Hbase. Accessed: 2014-11-30.

[8] Ibm BlueGene supercomputers. http://en.wikipedia.org/wiki/Blue_Gene. Accessed: 2012-1-30.

[9] Kyotocabinet. http://fallabs.com/kyotocabinet/. Accessed: 2012-09-30.

[10] Parallel reconfigurable observational environment. http://www.nmc-probe.org/. Accessed: 2014-11-30.

[11] Precision time protocol. http://www.ieee1588.com/. Accessed: 2014-05-30.

[12] Riak. http://docs.basho.com/riak/latest/. Accessed: 2014-1-30.

[13] P. T. Barbon, M. Pistore, and M. Trainotti. run-time monitoring of instances and classes of web service compositions, in Web Services. ICWS, 2006.

[14] K. Batcher. Supercomputer io quote. http://en.wikipedia.org/wiki/Ken_Batcher. Accessed: 2014-12-30.

[15] K. Brandstatter, T. Li, X. Zhou, and I. Raicu. NoVoHT: a lightweight dynamic persistent NoSQL key/value store. GCASR' 13, 2013.

[16] P. Buneman and W.-C. Tan. Provenance in databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2007.

[17] R. Cardell-Olivier, M. Kranz, and K. Smettem. A reactive soil moisture sensor network: Design and field evaluation. *International Journal of Distributed Sensor Netowrks*, 1.(2.), 2005.

[18] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *In Proceedings of Annual Linux Showcase and Conference*, pages 317–327, Atlanta, Georgia, 2000.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[20] Y.-W. Cheah, R. Canon, B. . R. Plale, and L. Milieu: Lightweight and congurable big data provenance for science, big data (BigData congress). BigData Congress, 2013.

[21] J. C. Corbett, J. Dean, and M. Epstein. Spanner: Google's globally-distributed database. OSDI, 2012.

[22] P. Corke, T. Wark, R. Jurdak, W. Hu, P. Valencia, and D. Moore. Environmental wireless sensor networks. *Proceedings of the IEEE*, 98(11):1903–1917, Nov 2010.

[23] A. D'Alessandro and G. D'Anna. *Suitability of low-cost three-axis MEMS accelerometers in strong-motion seismology: tests on the LIS331DLH (iPhone) accelerometer*. Bulletin of the Seismological Society of America, 2013.

[24] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, Athens, Greece, 2011.

[25] D. H. DeCandia, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. dynamo: Amazons highly available key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP, pages 205–220, New York, NY, USA, 2007.

[26] S. A. DeLoach, X. Ou, R. Zhuang, and S. Zhang. Model-driven, moving-target defense for enterprise network security. In *Models@ run. time*, pages 137–161. Springer International Publishing, 2014.

[27] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE/ACM Trans. Netw.*, 14(SI):2809–2816, June 2006.

[28] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the 2012 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM '12, pages 25–36, Helsinki, Finland, 2012.

[29] S. Fan, Z. Xiang, L. Qian, and S. Changyu. Numerical simulation of micro injection moulding based on mesh free method. *Sciencepaper Online*, 2010.

[30] M. Gashinova, M. Cherniakov, N. Zakaria, and V. Sizov. Empirical model of vegetation clutter in forward scatter radar micro-sensors. Radar Conference, 2010.

[31] M. Gerboles and D. Buzica. *Evaluation of Micro-Sensors to Monitor Ozone in Ambient AIr*. Joint Research Center for Environment and Sustainability, 2009.

[32] D. Ghoshal and L. Ramakrishnan. Frieda: Flexible robust intelligent elastic data management in cloud environments. SCC, 2012.

[33] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. M. Capretz. data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing, Advances, Systems and Applications, Springer*, 2:22, 2013.

[34] J. Homer, S. Zhang, X. Ou, D. Schmidt, Y. Du, S. R. Rajagopalan, and A. Singhal. Aggregating vulnerability metrics in enterprise networks using attack graphs. *Journal of Computer Security*, 21(4):561–597, 2013.

[35] H. Huang, S. Zhang, X. Ou, A. Prakash, and K. Sakallah. Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2011.

[36] M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60, Seattle, WA, USA, 2003.

[37] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau. Infrastructure outsourcing in multi-cloud environment. FederatedClouds '12, pages 33–38. ACM, 2012.

[38] M. S. L. Khan, S. U. Réhman, L. Zhihan, and H. Li. Head orientation modeling: Geometric head pose estimation using monocular camera. In *The 1st IEEE/IIAE International Conference on Intelligent Systems and Image Processing 2013 (ICISIP2013)*, 2013.

[39] D. L. *Computer Network Time Synchronization: The Network Time Protocol*. Taylor & Francis, 2010.

[40] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[41] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun*, 21(7):558–565, 1978.

[42] M. Li, J. Shu, and W. Zheng. GRID Codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage*, 4(4):15:1–15:22, Feb. 2009.

[43] T. Li, A. P. de Tejada, K. Brandstatter, Z. Zhang, and I. Raicu. ZHT: a zero-hop DHT for high-end computing environment. GCASR' 12.

[44] T. Li, K. Keahey, R. Sankaran, P. Beckman, and I. Raicu. A cloud-based interactive data infrastructure for sensor networks. IEEE/ACM Supercomputing/SC'14.

[45] T. Li, I. Raicu, and L. Ramakrishnan. Scalable state management for scientific applications in the cloud. BigData Congress '14.

[46] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu. Exploring distributed hash tables in highend computing. *SIGMETRICS Performance Evaluation Review*, 2011.

[47] T. Li, X. Zhou, K. Brandstatter, and I. Raicu. Distributed key-value store on HPC and cloud systems. GCASR' 13, 2013.

[48] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. IPDPS '13.

[49] X. Li, Z. Lv, J. Hu, L. Yin, B. Zhang, and S. Feng. Virtual reality gis based traffic analysis and visualization system. *Advances in Engineering Software*, 2015.

[50] X. Li, Z. Lv, J. Hu, B. Zhang, L. Shi, and S. Feng. Xearth: A 3d gis platform for managing massive city information. In *Computational Intelligence and Virtual Environments for Measurement Systems and Applications (CIVEMSA), 2015 IEEE International Conference on*, pages 1–6. IEEE, 2015.

[51] X. Li, Z. Lv, W. Wang, C. Wu, and J. Hu. Virtual reality gis and cloud service based traffic analysis platform. In *The 23rd International Conference on Geoinformatics (Geoinformatics2015)*. IEEE, 2015.

[52] Z. Lu, M. S. Lal Khan, and S. Ur Réhman. Hand and foot gesture interaction for handheld devices. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 621–624. ACM, 2013.

[53] Z. Lu, S. U. Réhman, and G. Chen. Webvrgis: Webgis based interactive online 3d virtual community. In *Virtual Reality and Visualization (ICVRV), 2013 International Conference on*, pages 94–99. IEEE, 2013.

[54] Z. Lu and S. Ur Réhman. Touch-less interaction smartphone on go! In *SIGGRAPH Asia 2013 Posters*, page 28. ACM, 2013.

[55] Z. Lv, G. Chen, C. Zhong, Y. Han, and Y. Y. Qi. A framework for multi-dimensional webgis based interactive online virtual community. *Advanced Science Letters*, 7(1):215–219, 2012.

[56] Z. Lv, C. Esteve, J. Chirivella, and P. Gagliardo. Clinical feedback and technology selection of game based dysphonic rehabilitation tool. In *9th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth2015)*. IEEE, 2015.

[57] Z. Lv, L. Feng, S. Feng, and H. Li. Extending touch-less interaction on vision based wearable device. In *IEEE Virtual Reality Conference 2015, 23 - 27 March, Arles, France*. IEEE, 2015.

[58] Z. Lv, L. Feng, H. Li, and S. Feng. Hand-free motion interaction on google glass. In *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*. ACM, 2014.

[59] Z. Lv, S. Feng, M. S. L. Khan, S. Ur Réhman, and H. Li. Foot motion sensing: augmented game interface based on foot interaction for smartphone. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, pages 293–296. ACM, 2014.

[60] Z. Lv, A. Halawani, S. Feng, and H. Li. Touch-less interactive augmented reality game on vision based wearable device. *Personal and Ubiquitous Computing*, 2015.

[61] Z. Lv, A. Halawani, M. S. Lal Khan, S. U. Réhman, and H. Li. Finger in air: touch-less interaction on smartphone. In *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia*, page 16. ACM, 2013.

[62] Z. Lv and H. Li. Imagining in-air interaction for hemiplegia sufferer. In *International Conference on Virtual Rehabilitation (ICVR2015)*. IEEE, 2015.

[63] Z.-H. Lv, R.-N. Ma, J.-B. Fang, Y. Han, and G. Chen. Index structure for multi-scale representation of multi-dimensional spatial data in webgis [j]. *Application Research of Computers*, 9:053, 2010.

[64] J. Manobianco. *Global Environmental Micro Sensors (GEMS): A Revolutionary Observing System for the 21st Century*. NASA Institute for Advnced Concepts: Atlanta, GA, 2005.

[65] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30, July 2004.

[66] F. Mattern. Virtual time and global states of distributed systems. Workshop on Parallel and Distributed Algorithms, 1988.

[67] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2001.

[68] A. J. McAuley. Reliable broadband communication using a burst erasure correcting code. In *Proceedings of the 1990 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM '90, pages 297–306, Philadelphia, Pennsylvania, USA, 1990.

[69] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. USENIX Annual Technical Conference, 2006.

[70] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[71] P. Namour, M. Lepot, and N. Jaffrezic-Renault. Recent trends in monitoring of european water framework directive priority substances using micro-sensors: A 2007 to 2009 review. *Sensors*, page 79477978, 2010.

[72] R. Nishtala, H. Fugal, S. Grimm, and M. Kwiatkowski. Scaling memcache at facebook. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI, pages 385–398, Lombard, IL, 2013.

[73] R. Ostrovsky and B. Patt-Shamir. Optimal and efficient clock synchronization under drifting clocks. PODC, 1999.

[74] D. Patel, F. Khasib, I. Sadooghi, and I. Raicu. Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues. SCRAMBL'14, 2014.

[75] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain. *Towards Data Intensive Many-Task Computing*. Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2009.

[76] I. Raicu, I. T. Foster, and P. Beckman. Making a case for distributed file systems at exascale. LSAP '11, 2011.

[77] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: A fast and light-weight task execution framework. SC '07, pages 1–12, Reno, Nevada, 2007.

[78] A. Rajendran and I. Raicu. Matrix: Many-task computing execution fabric for extreme scales. 2013.

[79] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM International Conference on on Data Communication*, SIGCOMM, pages 161–172, New York, NY, USA, 2001.

[80] Rowstron and P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. Middleware. London, UK, UK, 2001.

[81] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu. Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing. CCGrid'14, 2014.

[82] V. Sarkar et al. *ExaScale Software Study: Software Challenges in Extreme Scale Systems*. DARPA IPTO, 2009.

[83] R. H. Schmuck. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'02, pages 16–16, Monterey, CA, 2002.

[84] P. Schwan. Lustre: Building a file system for 1000-node clusters. Proc. of the 2003 Linux Symposium, pages 174–186, Berkeley, CA, USA, 2003.

[85] J. Shao, H. Wei, Q. Wang, and H. Mei. A runtime model based monitoring approach for cloud. CLOUD, 2010.

[86] F. Shi, A. M. Coffey, K. W. Waddell, E. Y. Chekmenev, and B. M. Goodson. Nanoscale catalysts for nmr signal enhancement by reversible exchange. *The Journal of Physical Chemistry C*, 119(13):7525–7533, 2015.

[87] F. Shi, X. Zhang, Q. Li, and C. Shen. Mould wall friction effects on micro injection moulding based on simulation of mis. *IOP Conference Series: Materials Science and Engineering*, 2010.

[88] F. Shi, X. Zhang, Q. Li, and C. Shen. Notice of retraction particle tracking in micro-injection molding simulated by mis. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*. IEEE, 2010.

[89] C. Shou, D. Zhao, T. Malik, and I. Raicu. Towards a provenance-aware a distributed file system. TaPP13, 2013.

[90] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2005.

[91] Y. L. Simmhan, B. Plale, D. Gannon, and S. Marru. Performance evaluation of the karma provenance framework for scientific workflows. IPAW, 2006.

[92] R. M. Stoica, D. Karger, M. F. Kaashoek, and H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. ACM SIGCOMM. San Diego, CA, USA, 2001.

[93] Y. Su, Y. Wang, and G. Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 61–72. ACM, 2015.

[94] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu. SDQuery DSI: Integrating Data Management Support with a Wide Area Data Transfer Protocol. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2013.

[95] A. Tek, B. Laurent, M. Piuzzi, Z. Lu, M. Chavent, M. Baaden, O. Delalande, C. Martin, L. Piccinali, B. Katz, et al. Advances in human-protein interaction-interactive and immersive molecular simulations. *Biochemistry, Genetics and Molecular Biology"Protein-Protein Interactions-Computational and Experimental Tools'*, pages 27–65, 2012.

[96] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13.

[97] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. pages 1626–1629, 2009.

[98] J. J.-Y. Wang, Y. Wang, S. Zhao, and X. Gao. Maximum mutual information regularized classification. *Engineering Applications of Artificial Intelligence*, 37:1–8, 2015.

[99] K. Wang, K. Brandstatter, and I. Raicu. Simmatrix: Simulator for manytask computing execution fabric at exascales. HPC, 2013.

[100] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu. Using simulation to explore distributed key-value stores for extreme-scale system services. SC '13, 2013.

[101] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, and I. Raicu. Paving the road to exascale with many-task computing. SC'12 Poster, 2012.

[102] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu. Next generation job management systems for extreme-scale ensemble computing. HPDC '14, 2014.

[103] K. Wang, X. Zhou, T. Li, M. Lang, and I. Raicu. Optimizing load balancing and data-locality with data-aware scheduling. IEEE BigData'14.

[104] W. Wang, Z. Lv, X. Li, W. Xu, B. Zhang, and X. Zhang. Virtual reality based gis analysis platform. In *22nd International Conference on Neural Information Processing (ICONIP2015)*. Springer, 2015.

[105] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. Smart: A mapreduce-like framework for in-situ scientific analytics. Technical report, OSU-CISRC-4/15-TR05, Ohio State University, 2015.

[106] Y. Wang, W. Jiang, and G. Agrawal. Scimate: A novel mapreduce-like framework for multiple scientific data formats. CCGRID '12, 2012.

[107] Y. Wang, A. Nandi, and G. Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, page 9. ACM, 2014.

[108] Y. Wang, Y. Su, and G. Agrawal. Supporting a Light-Weight Data Management Layer Over HDF5. In *CCGrid'13*. IEEE.

[109] Y. Wang, Y. Su, and G. Agrawal. A novel approach for approximate aggregations over arrays. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 4. ACM, 2015.

[110] Y. Wang, Y. Su, G. Agrawal, and T. Liu. SciSD: Novel Subgroup Discovery Over Scientific Datasets Using Bitmap Indices. Technical report, OSU-CISRC-3/15-TR03, Ohio State University, 2015.

[111] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[112] W.Vogels. Filesystem in userspace. `http://fuse.sourceforge.net/`. Accessed: 2011-09-17.

[113] S. Zhang. Deep-diving into an easily-overlooked threat: Inter-vm attacks. *Whitepaper, provided by Kansas State University, TechRepublic/US2012, available online: http://www. techrepublic. com/resourcelibrary/whitepapers/deep-diving-into-an-easilyoverlooked-threat-inter-vm-attacks*, 2013.

[114] S. Zhang. *QUANTITATIVE RISK ASSESSMENT UNDER MULTI-CONTEXT ENVIRONMENTS*. PhD thesis, Kansas State University, 2014.

[115] S. Zhang, D. Caragea, and X. Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

[116] S. Zhang, X. Ou, and J. Homer. Effective network vulnerability assessment through model abstraction. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 17–34. Springer, 2011.

[117] S. Zhang, X. Ou, A. Singhal, and J. Homer. An empirical study of a vulnerability metric aggregation method. In *The 2011 International Conference on Security and Management (SAM'11), special track on Mission Assurance and Critical Infrastructure Protection (STMACIP'11)*, 2011.

[118] S. Zhang, X. Zhang, and X. Ou. After we knew it: empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across iaas cloud. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 317–328. ACM, 2014.

[119] X. ZHANG, Q. LI, F. SHI, and C. SHEN. Micro injection molding simulation based on sph method. *CIESC Journal*, 2012.

[120] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communication*, 22(1):41–53, 2004.

[121] C. S. Zhao, Z. Zhang, I. Sadooghi, X. Zhou, T. Li, and I. Raicu. FusionFS: a distributed file system for large scale data-intensive computing. GCASR, 2013.

[122] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu. Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms. IEEE Cluster'13, 2013.

[123] D. Zhao, K. Qiao, and I. Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. 2014.

[124] D. Zhao, C. Shou, T. Malik, and I. Raicu. Distributed data provenance for large-scale data-intensive computing. CLUSTER, 2013.

[125] D. Zhao, J. Yin, K. Qiao, and I. Raicu. Virtual chunks: On supporting random accesses to scientific data in compressible storage systems. IEEE BigData'14, 2014.

[126] D. Zhao, D. Zhang, K. Wang, and I. Raicu. Exploring reliability of exascale systems through simulations. HPC '13, pages 1:1–1:9, 2013.

[127] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Big Data, 2014 IEEE International Conference on*.

[128] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. IEEE Workshop on Scientific Workflows'07, 2007.

[129] Y. Zhao, I. Raicu, S. Lu, and X. Fei. Opportunities and challenges in running scientific workflows on the cloud. IEEE CyberC, 2011.

[130] G. Zhu, Y. Wang, and G. Agrawal. SciCSM: Novel Contrast Set Mining over Scientific Datasets Using Bitmap Indices. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 38. ACM, 2015.

[131] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal. Investigating the application of moving target defenses to network security. In *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pages 162–169. IEEE, 2013.

[132] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National Symposium on Moving Target Research*, 2012.