CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Convergence of Key-Value Storage Systems from Clouds to Supercomputers

Tonglin Li[1], Xiaobing Zhou[3], Ke Wang[1], Dongfang Zhao[1],Iman Sadooghi[1], Zhao Zhang[4], Ioan Raicu [1,2]

[1] *Computer Science Department, Illinois Institute of Technology, Chicago, IL, USA)*
[2] *MCS Division, Argonne National Laboratory, Lemont, IL, USA*
[3] *Hortonworks, Palo Alto, CA, USA*
[4] *AMP Lab, University of California, Berkeley, CA, USA*

SUMMARY

This paper presents a convergence of distributed Key-Value 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. 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. We compared ZHT against other key/value stores and found it offers superior performance for the features and portability it supports. This paper also presents several real systems that have adopted ZHT, namely FusionFS (a distributed file system), IStore (a storage system with erasure coding), MATRIX (distributed scheduling), Slurm++ (distributed HPC job launch), Fabriq (distributed message queue management); all of these real systems have been simplified due to Key-Value 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 Key-Value storage systems can be in such a variety of environments.

Received ...

KEY WORDS: NoSQL Database, Distributed Key-value store, supercomputer, cloud computing

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 [1]. 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 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 [2], 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" [3]. 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 the next decade to achieve exascale computing all involve extremely high parallelism and concurrency [4]. Up to 2015, 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(e.g. parallel file systems, such as GPFS [5], PVFS [6] and Lustre [7]) is completely separated from compute resources. The connection between them is a high speed network. 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. The unscalable storage architecture could be a "show-stopper" in building exascale systems [4]. Although there are works such as burst-buffer [8, 9] to alleviate the parallel file system bottleneck, in the long run the need for building efficient and scalable distributed storage for high performance computing (HPC) 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 most of parallel and distributed file systems can be inefficient at large scales. Our previous work(fig.1) on a Blue Gene/P supercomputer with 16K-cores shows the various costs for file/directory creating(metadata operation of file systems) on GPFS. GPFS's metadata performance degrades rapidly under concurrent operations, reaching saturation at only 4 to 32 core scales (on a 160K-core machine). Ideal performance would have been constant at different scales, but we see the cost of these basic metadata operations (e.g. create file) growing exponentially, 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 one file creation 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 hours range, although GPFS might have been improved over the last several years. On a large scale HPC system, whether the time per metadata operation is minutes or hours, the conclusion is that the metadata management in GPFS does not have enough degree of distribution, and not enough emphasis was placed on avoiding lock contention.

Other parallel or distributed file systems (e.g. Google's GFS and Yahoo's HDFS) 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 operations and deliver the low latencies typically found in centralized metadata management (under light load).

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, such as being able to handle dynamic nodes join and leave on the fly and the flexibility to run on different cloud instance types simultaneously.

As an initial attempt to meet these needs, we propose and build ZHT(zero-hop distributed hash table [10–13]), an instance of NoSQL database [14]. ZHT has been tuned for the specific requirements of high-end computing (e.g. trustworthy/reliable hardware, fast networks, non-existent "churn", low latencies). ZHT aims to be a building block for future distributed

Prepared using cpeauth.cls
DOI: 10.1002/cpe

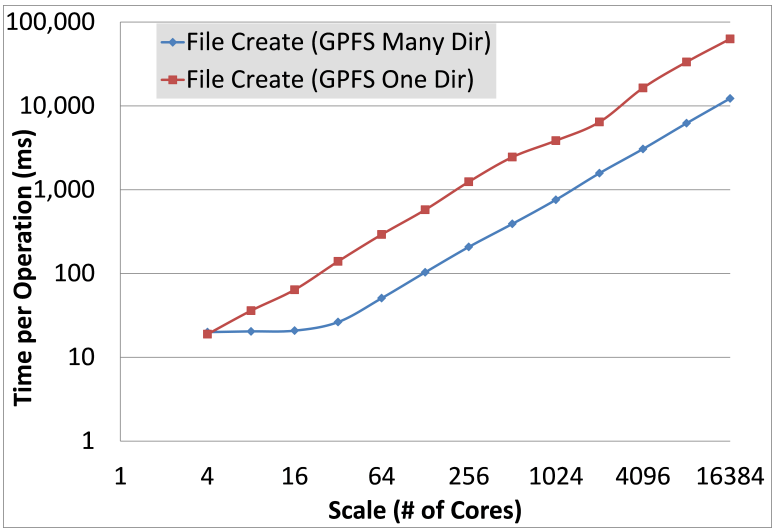


Figure 1. Time per operation (touch) on GPFS on various numbers of processors on a IBM Blue Gene/P

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. Highlighted features include being light-weight, dynamically allowing nodes join and leave, fault tolerant through replication, handling failures gracefully, efficiently propagating events throughout the system, a customized consistent hashing mechanism. Unlike conventional key-value store, ZHT implemented new operations such as append,compare_swap/ and state_change_ callback in addition to insert/lookup/remove. To provide ZHT a persistent back end, we also created a fast persistent single node data 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 [15] and Memcached [16] 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 [17] in the Amazon AWS Cloud, and found that ZHT offers significantly better performance and economic cost than DynamoDB.

This paper presents a significant extension to our prior work on developing a zero-hop distributed hash table (ZHT) [10]. It also covers five real systems (FusionFS, IStore, MATRIX, Slurm++, and Fabriq) at a high-level. They have been integrated with ZHT, and evaluated at modest scales. 1) ZHT is 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 is 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 is also used as a building block to MATRIX, a distributed task scheduling system, delivering 13k jobs/sec throughput at 4K-core scales. 4) Slurm++, a distributed job launch system that avoids the centralized gateway nodes in Slurm. 5) ZHT is used in Fabriq, a distributed message queue, to store messages reliably, and load balance the resource requirements. 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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

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 the original conference paper [10] that this journal paper has extended are:

- Design and implementation of ZHT, a light-weight, high performance, fault tolerant, persistent, dynamic, and highly scalable distributed hash table, optimized for supercomputers and clusters.
- Support for unconventional operations, such as append, in order to reduce lock contention.
- Extensive system micro benchmarks conducted on up to 8K real nodes. Simulations used to evaluate ZHT and some competitors at up to millions of simulated nodes.
- Integration and evaluation with three real systems (FusionFS, IStore, and MATRIX), managing distributed storage metadata and distributed job scheduling information.

The contributions of this particular extended paper are as follows: This paper has extended the original conference paper [10] through the following significant contributions:

- 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.
- Extended primitive operations with new operations(compare_swap and state_change_callback), which can significantly simplify the design of upper level applications.
- Extended evaluations of ZHT to the Amazon AWS cloud for both performance and economics.
- Showcased two new real system that have been built with ZHT, namely Slurm++ (HPC job launch system) and Fabriq(distributed message queue system)

2. ZHT DESIGN AND IMPLEMENTATION

Most high performance computing environments are batch oriented, in which an allocation is configured in the beginning at run time. Such an allocation generally has information about the available hardware and software resources, and the amount of resources (e.g. number of nodes) generally would not change until the allocation is terminated. The only possible reason to decrease the allocation is hardware(nodes, hard drive or network) or low level software system (such as monitoring and scheduling system)failure. Because nodes in HPC systems are generally reliable and have predicable uptime (from the start of an allocation, to shut down on de-allocation), it implies that node "churn" in HPC occurs much less frequently than in traditional DHTs. In ZHT's static membership (for HPC), every node at bootstrap time knows how to contact each other.

However in some dynamic environments, the system properties are different. In dynamic environments such as clouds or data centers, nodes may join (for system performance boosting) and leave (node failure or scheduled maintenance) at any time. We believe that dynamic membership would be important for such environments, especially for cloud computing systems, and hence have made efforts to support it without affecting primitive operations' time complexity. This principle guided our design of the proposed dynamic membership management in ZHT.

The node ID space and membership table are treated as a ring-shaped 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. A hash function maps arbitrarily long strings to index values, which can then be used to efficiently retrieve the communication address (e.g. host name, IP address, port, MPI-rank) from a local in-memory membership table . Depending on the volume of information that is stored, storing the entire membership table consumes only a small (less than 1%) portion of available memory on each node.

DOI: 10.1002/cpe

With a 1K-nodes scale allocation on Intrepid, an IBM BlueGene/P supercomputer, one ZHT instance's memory footprint is less than 8MB. The memory footprint consists of ZHT server binary, 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 periodically. The membership table is very small, each entry(presenting a node) only takes 32 bytes, 1 million nodes only need 32 MB space. By tuning the number of key/value pairs that are cached in memory, users can reach the balance between performance and memory consumption.

2.1. Primitive operations

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

insert operation: Insert a string type key-value pair into data store; return execution status.

lookup operation: Lookup a given key string and return a value string, if the key is ever modified by `append` operation, it will return a list of value.

remove operation: Remove a key-value pair and return execution status.

append operation: This allows user assigning multiple value to a same key (algorithm 1). This is not a feature that many hash maps do, and is especially rare in persistent ones as well. 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 service 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 occurs usually via distributed locks, which is known to be inefficient. Append primitive looks like multi-version concurrency control (MVCC) appeared in some data stores like Voldemort, Riak and HBase, but it's implemented differently for different purpose. In MVCC, only one version of data is marked as current and active, while in append primitives all data fields are current and are treated equally.

cswap (compare and swap) operation: In some applications when a client reads a value and sets it to another value that depends on the read value, it will need to lock the keyvalue record. If another client wants to access the same key, there comes the lock contention problem. A naive way to implement is to add a global lock for each queried key in the key-value stores, which is apparently not scalable. A better approach is used in ZHT to implement an atomic operation that is executed on the key-value store server side, which finish the value update before return to the client (algorithm 2). This primitive is similar with "Check-and-Set" in Memcached and Couchbase Server [18].

callback operation: Callback operation is used to notify a client upon a specified value change (algorithm 3). 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 this is letting the client to pull 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 dedicated thread for all state change callback requests, and the main thread keeps processing other requests. Within a given period of time (time_out), if the server finds the value being changed to expected value, it returns success signal to the client; otherwise returns failure signal to the client.

2.2. Terminologies

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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Algorithm 1 Append

```
1: procedure APPEND_CLIENT(key, appended_value)
2:     pack ← pack(key, appended_value)
3:     send(serialize(pack))
4: procedure APPEND_SERVER_RESPONSE(key, appended_value)
5:     record_p ← lookup(key)                          ▷ return an entry pointer
6:     while cursor_p.value.next! = Null do
7:         cursor_p ← record_p.value.next
8:     cursor_p.next.value ← appended_value
```

Algorithm 2 Compare_and_swap

```
1: procedure COMPARE_SWAP_SERVER_RESPONSE(key, update())
2:     record_p ← lookup(key)
3:     record_p.lock()
4:     seenValue ← record_p.value
5:     newValue ← update(seenValue)
6:     record_p.value ← newValue
7:     record_p.unlock()
```

Algorithm 3 Callback

```
1: procedure CALLBACK_SERVER_RESPONSE(key, expectedValue)
2:     while record_p.value! = expectedValue do
3:         record_p ← lookup(key)
4:         record_p ← record_p.value.next
5:     notify_client()
```

Physical node: A physical node is an independent physical machine. Each physical node may run several ZHT instances that are differentiated with a combination of 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 per physical node, ZHT can fit in heterogeneous computing systems with various storage capacities and performance. A ZHT instance can be identified by a combination of IP address and port. Therefore the partitions can be many more than the instances and physical nodes.

Partition: A partition is a contiguous range of the key address space; a file on disk is associated with each partition for persistence. We developed a single-node persistent key-value store (NoVoHT) as ZHT's back-end, which also takes care of each partition.

Manager: A Manager is a service process running on each physical node and takes charge of starting and shutting down ZHT instances, managing membership table 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.

DOI: 10.1002/cpe

For example, in an initial system of 1000 ZHT instances (typically running on 1000 nodes), where each instance contains 1000 partitions, the overall system could scale up to 1 million instances with 1 million physical nodes. Experiments validate this approach by 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.

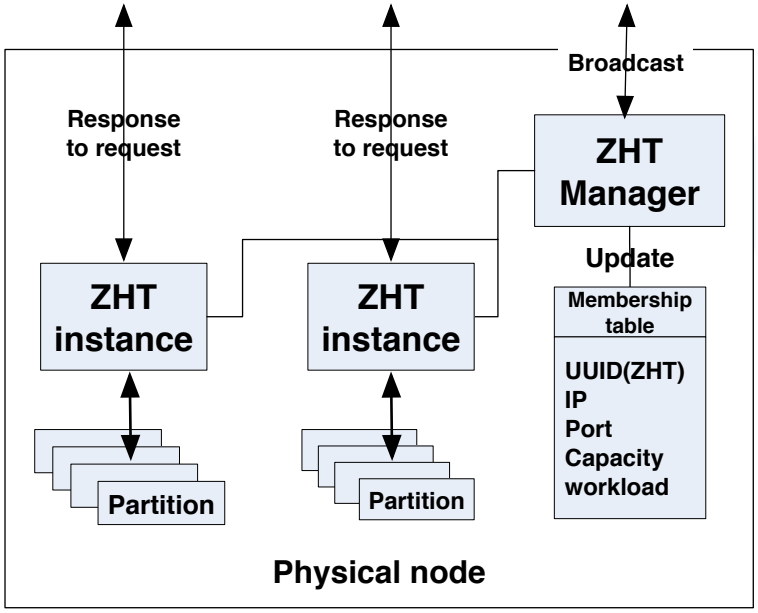


Figure 2. ZHT server single node architecture. Each physical can run multiple ZHT instances, which further manages multiple partitions. Each partition stores a contiguous key space.

2.3. Membership management

ZHT supports both static and dynamic node membership. In static membership, clients and servers fetch neighbor list from the batch job scheduler or a given file during the bootstrapping phase. Once the membership is established, no new nodes is allowed to join. Nodes could leave the system due to failures; we assume failed nodes do not recover. For the dynamic membership, nodes are allowed to join and leave the system dynamically. Many DHTs and key-value stores support dynamic membership, but typically deliver this through logarithmic routing. They use consistent hashing which sacrifices performance for scalability under dynamic environment. We address this issue with an improved consistent hashing mechanism that requires constant-number-hop(typically 1, at most 2) routing. With this novel design, we offer the desired flexibility of dynamic membership while maintaining very low latency through 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 adopting 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 a partition as a whole instead of many individual key-value pairs. This avoids rehashing all effected key-value pairs, as most DHTs that adopt consistent hashing. Moving an partition altogether is remarkably more efficient than rehashing individual key/value pairs. When migration is in progress, the partition state is locked. All incoming requests during this time are queued, until the migration is completed. In the meanwhile because the

Prepared using cpeauth.cls
DOI: 10.1002/cpe

partition state won't change, corresponding replicas also won't change. This keeps the entire system state consistent. If failure occurs during migration, simply don't apply the changes to corresponding partitions and replicas (discard the queued requests and report error to clients and administrators), this will eventually bring the system to roll back to a consistent earlier state.

Node Joins: Upon a node joins the network, it firstly checks out a copy of membership table from a ZHT Manager on a random physical node. Based on this table, the new node can find the physical nodes with the most partitions, then join the ring as one of the heavily loaded node's neighbor and migrates some of the partitions from the "busiest" node to itself. Migrating a partition is as easy as moving a file, without having to rehash the key/value pairs in the partition.

Node departures: On planned node departures (e.g. during system maintenance), the administrator fetches current membership table from a random physical node, modify it accordingly, then broadcast the incremental table to other ZHT managers to update their local tables. The departing managers firstly migrate their data partitions to neighboring nodes, and then proceed to depart. For an unplanned departure (e.g. due to a node failure), it will be firstly detected by the client that sends a request and doesn't get response within a given timeout, or due to another ZHT instance initiates a server-to-server operation and fails(e.g. migration, replication, etc.). Upon a certain number of try-and-fails on a certain server, the client marks the corresponding physical node down on its local membership table and informs a random ZHT manager about this failure. The client then sends the request to the first available replica of the failed node. At the same time, the manager updates its local membership table and broadcasts the change to the whole network, and initiates a rebuilding of the replicas, specifically increasing replicas for all partitions that are stored on the failed physical node in order to maintain the specified replication level.

Client Side State: In case that clients and servers are not on the same physical nodes, it's necessary to keep the client-side membership table updated. Since the node joining and departure changes the number of partitions covered by a ZHT server, clients might send requests to wrong nodes if it's local membership table is not updated. To address this issue, we lazily update clients' membership table. Only when the requests are sent mistakenly, the ZHT server will send back a copy of latest membership table to the clients.

2.4. Server architecture

We have explored various architectures for ZHT server. Since typical Key-Value store operations are very small but frequent, we optimize ZHT more for small requests. In early prototypes, we adopted a multi-threading design, in which a server throw a thread for each request, but the overheads of starting, switching, and tearing down threads was too high compare to the work to be done for a request. We eventually converged on a notably more streamlined architecture, an event-driven server architecture based on *epoll* [19]. The current epoll-based ZHT outperforms the multithread version by at least 3X. We'll discuss the performance difference in detail in the evaluation section.

2.5. Fault tolerance

ZHT handles failures gracefully by lazily tagging non-responding nodes failed. ZHT uses replication to ensure data persistence in face of failures. Newly inserted or modified key/value data will be replicated asynchronously to secondary replicas that have closer hashed location. By communicating only with near neighbors, this approach ensures that replicas only consume less network resources when we succeed in implementing the topology-aware and locality-aware protocols (similar approach can be found in [20,21]). Despite the lack of topology-aware in the current ZHT, the asynchronous replication only adds relatively small overhead when adding more replicas at modest scales (up to 4K-cores).

ZHT is fully distributed, and single node failures do not affect the functionality. The key/value pairs that were stored on the failed node can be found on replica nodes. Upon

DOI: 10.1002/cpe

failures, the replicas answers the requests for data that were originally stored on the failed node.

When ZHT is shut down due to hardware maintenance or system reboot purposes, the entire state of ZHT could be dumped to local persistent storage; note that every change to the in-memory data structure is dumped to disk periodically, ensuring the entire state of the data store can be recovered if needed. Considering the increasing size of memory and SSDs, as well as I/O performance improvements in the future, it is expected that a multi-gigabyte of state could be retrieved in just seconds.

Once ZHT is bootstrapped, the system verification time 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 very efficient with its current static membership table, as there is no global communication required between nodes (see fig.3). Nevertheless, we expect the time to bootstrap ZHT to be insignificant compare to the batch scheduler's overheads on a high-end computing system, which could potentially include node provisioning, OS booting, starting of network services, and perhaps the mounting of some parallel file systems. At 1K-node scale on IBM Blue/Gene machine, 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 it at 8K-node scale. Fig.3 shows the bootstrap time increase with the scale.

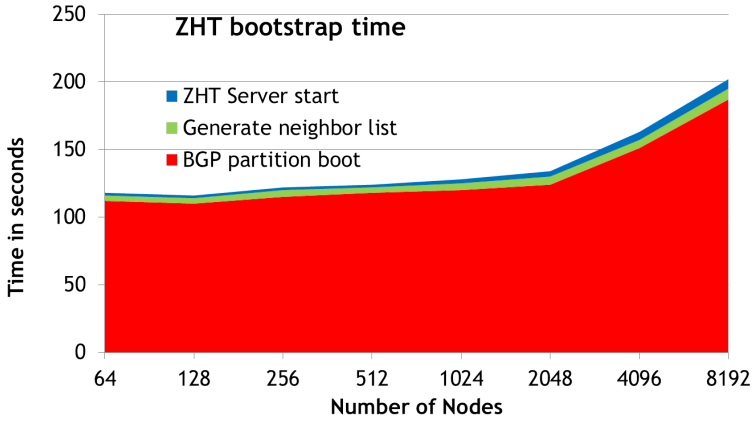


Figure 3. ZHT Bootstrap time on Blue Gene/P from 64 to 8K nodes

2.6. Socket level thread safety and connection caching

In a dynamic network environment, many multi-threaded problems are related to socket. In other words, if the sockets are thread safe, many message transferring out-of-order related issues are smoothed away. The previous version of ZHT client is thread-safe in operation level, such as 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 found, the 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.

2.7. Consistency

ZHT uses replication to handle server failures. Current version only allows clients interact with a single primary replica for write operations(insert/remove/append). This decision is based on

DOI: 10.1002/cpe

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, which may cause serious performance loss. If enabling all replicas to accept read request, the read performance in terms of throughput will gain some boost, however, to ensure consistent results for all read requests to different replicas will need version checking and updating, which may increase the request latency significantly. Upon a primary replica failure, a corresponding secondary replica will take place of primary and directly talk to clients. Strong consistency is maintained between ZHT primary and secondary replica, operation completion thus will be acknowledged to clients right after the second replica is updated. Other replicas are asynchronously updated after the secondary replica updating is complete, bringing ZHT to eventual consistency. By this hybrid consistency approach, ZHT attains high throughput, good availability and reasonable consistency level at the same time.

2.8. 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 [22]) 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. 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.

2.9. Implementation

ZHT 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 [23].

3. PERFORMANCE EVALUATION VIA SYNTHETIC BENCHMARKS

In this section, we evaluate ZHT's performance, in terms of including request latencies, system throughput, performance of hashing functions, persistence overhead, and replication cost. Firstly we describe the configuration of test beds and benchmark setup. Secondly we presented a comprehensive performance evaluation. Two popular NoSQL systems (Memcached and Cassandra) that offer similar functionality or features are compared against ZHT, along with a cloud database service, DynamoDB on EC2 [24] cloud.

3.1. Testbeds, Metrics, and Workloads

We used several platforms to evaluate ZHT's performance.

DOI: 10.1002/cpe

- Kodiak: A Parallel Reconfigurable Observational Environment (PROBE) [25] at Los Alamos National Laboratory, it has 1024 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory.
- Intrepid: an IBM Blue Gene/P [26] supercomputer at Argonne Leadership Computing Facility [27]. 8K physical nodes (32K cores) are used, each of which has a 4-core PowerPC 450 processor and 2GB of RAM. Intrepid was used to compare ZHT to Memcached. Note that this system does not have persistent local node disks so the RAM-based disks were used as persistence option.
- HEC-Cluster: a 64-node (512-core) cluster at IIT: each node has two quad-core processors, 8GB RAM, 200GB HDD, it's used to compare ZHT to Cassandra.
- DataSys: an 8-core x64 server at IIT: two Intel Xeon quad-core processors with HT, 48 GB RAM, used to compare KyotoCabinet, BerkeleyDB and NoVoHT.
- Fusion: a 48-core x64 server at IIT: four AMD Opteron 12-core processors, 256GB RAM, used to compare KyotoCabinet, BerkeleyDB and NoVoHT.
- 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: Latency presents the time taken from a request to be submitted from a client to a response to be received by the client, measured in milliseconds (ms). Since various operations (insert/lookup/remove) latencies are quite close, we use average of the three operations to simplify the results presentation. Note that the latency consists of round trip network communication, system processing, and storage access time. Since Blue Gene/P doesn't have persistent storage for each work node, ram-disks are used for the experiment, while regular spinning 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.
- Efficiency: Ratio between measured throughput and ideal throughput.

3.2. Hash Functions

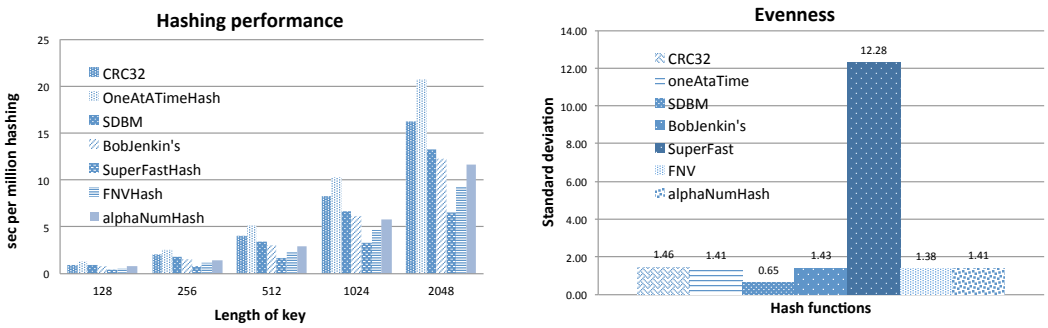
Similar with what we observed, the time spent on hashing keys to nodes is not major of the total cost, but with the time passed, the accumulation could be observed. We investigate some of the usual hash functions for figuring out the trade-off between performance and evenness.

As shown in fig.4a that some hash functions are faster than others, but a more important concern rather than performance is the evenness (fig.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.

3.3. Individual data store synthetic benchmark

We compared KyotoCabinet [28] to NoVoHT with persistence. We used identical workloads of 1M, 10M, and 100M operations (insert/get/remove), operating on fixed length key value pairs (see fig.5). When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we observed significantly better capacity scalability on NoVoHT. Although BerkeleyDB has some advantages such as memory usage, it does this at the cost of higher latency. When comparing

DOI: 10.1002/cpe



(a) Hash functions performance (b) Key distribution evenness

Figure 4. Hash function comparisons

NoVoHT persistence to non-persistence, we noticed that most of the overhead of the operations is on the hard disk I/O.

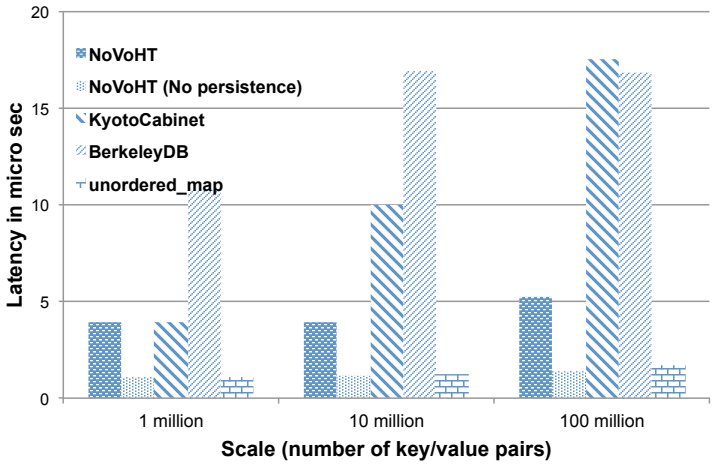


Figure 5. Average latency of NoVoHT, KyotoCabinet and BerkeleyDB

3.4. System Latencies

3.4.1. Synthetic benchmarks On Supercomputers and clusters We evaluated the latency on both the Blue Gene/P supercomputer and HEC-Cluster. We evaluated different implementations of ZHT with various communication protocols, such as UDP, TCP with connection caching, and compare them to Cassandra and Memcached.

Since ZHT latency is mostly dominated by cross node communication overhead, in-node operations mostly happened in memory, the latency difference between insert and lookup is minimal (fig.6). Because of this performance characteristic of ZHT, we use average latency of insert, lookup and remove operations to simplify presentation of the figures in rest of the paper.

ZHT shows great scalability at up to 8K-node scale. As shown in fig.7, on single node, the latency of both TCP with connection caching and UDP are extremely low (0.5ms). When scaling up, ZHT shows slowly increased latency, up to 1.1ms at 8K-node scales. We see that TCP with connection caching delivers practically same performance of that of UDP, at all the scales we measured. Memcached scaled well too, the latencies ranging from 1.1 ms to 1.4 ms from single node to 8K nodes (note that this represents a 25% to 139% increased latency,

Prepared using cpeauth.cls
DOI: 10.1002/cpe

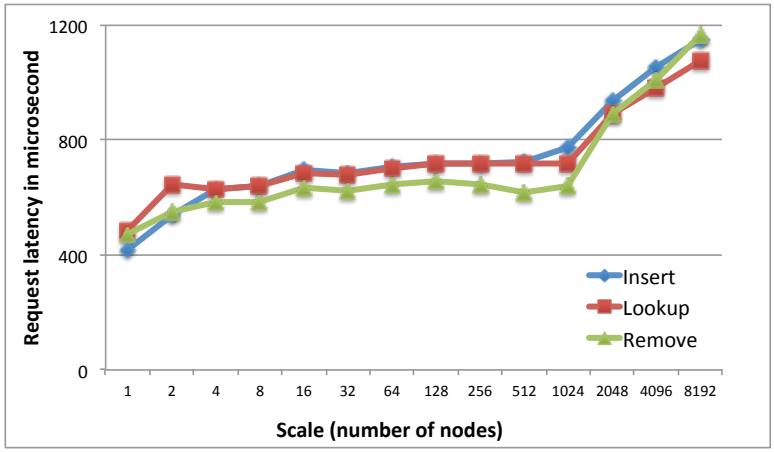


Figure 6. Basic operation latency comparison on Blue Gene/P. Note that insert/lookup operation latencies are extremely close, because majority of the latency are communication overhead, which are same for insert and lookup.

depending on the scale, implying some scalability issue). Note that IBM Blue Gene/P uses a 3D Torus network [29] for communication, which means the routing needs increasing number of hops at larger scales to send messages cross compute nodes. This also explains why the latency start to increase on large scale – one rack of Blue Gene/P has 1024 nodes, any scale larger than 1024 involves 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. If running on a Fat-tree network [30], we expect more constant latency (before the network is saturated) due to the constant routing hops.

The CDF plot (fig.8) 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 (table I).

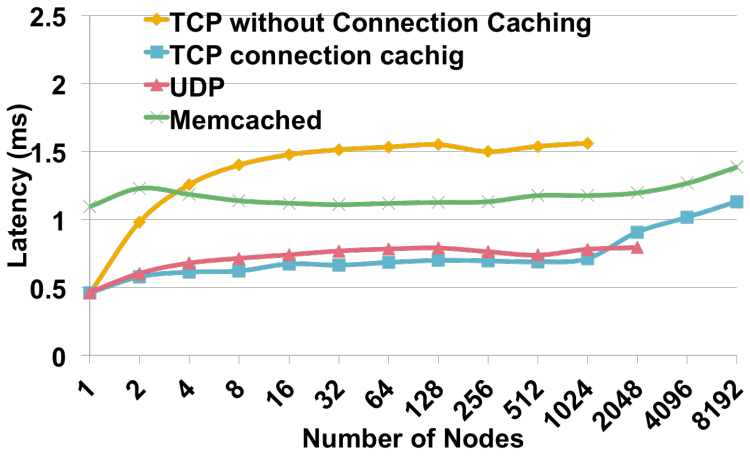


Figure 7. Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 8K nodes on the Blue Gene/P)

Because of Cassandra's implementation in Java, and the the Blue Gene/P machine lacks of support for Java , we evaluated Cassandra, Memcached, and ZHT on a conventional Linux cluster, the HEC-Cluster. Not surprisingly, as shown in 9, ZHT's latency is notably lower than that of Cassandra. ZHT also shows superior scalability over Cassandra. This is mainly

Prepared using cpeauth.cls
DOI: 10.1002/cpe

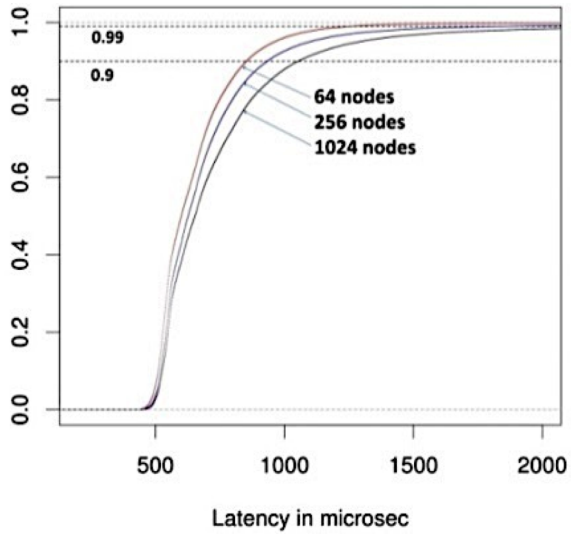


Figure 8. CDF of Benchmark on Blue Gene/P CDF

Table I. 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 Cassandra adopts a logarithmic routing algorithm and ZHT uses constant routing. Interestingly, Memcached only shows slightly better performance than ZHT at up to 64-node scales. We attributed ZHT's slight increment in latency to the fact that ZHT must write to disk, while Memcached's data stayed in memory completely.

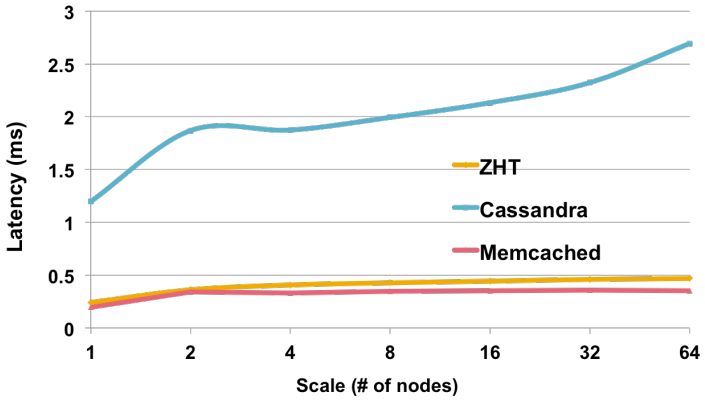


Figure 9. Performance evaluation of ZHT, Memcached and Cassandra plotting latency vs. scale (1 to 64 nodes on an AMD Cluster)

3.4.2. 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 II.

DOI: 10.1002/cpe

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 (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 fig.10b). On larger instances such as cc2.8xlarge, ZHT latency CDF plots are closer than they are on smaller instances. Fig.10 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 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 doesn't 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 II. 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

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

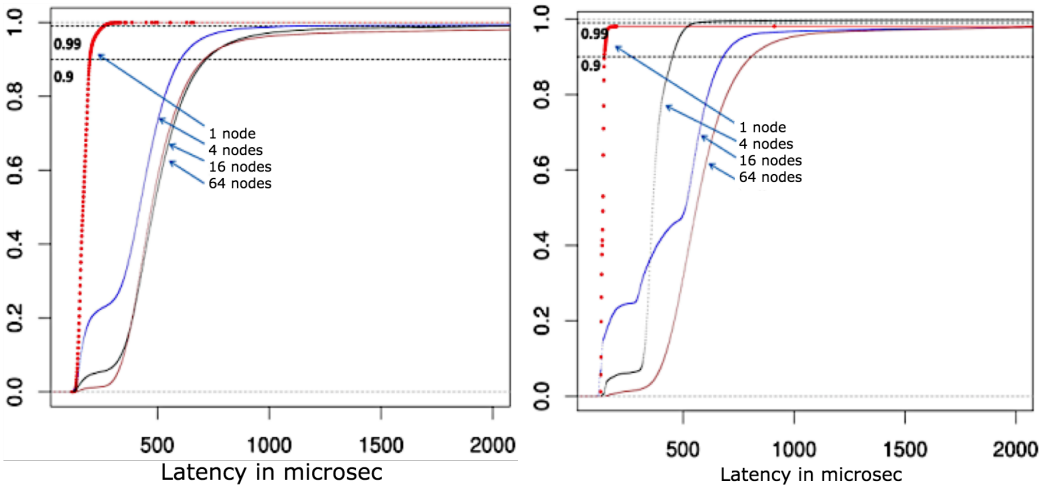
Scales	75%	90%	95%	99%	Average	Throughput
8	186	199	214	260	172	46421
32	509	603	681	1114	426	75080
128	588	717	844	2071	542	236065
512	574	708	865	3568	608	841040

Table IV. 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

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

Prepared using cpeauth.cls
DOI: 10.1002/cpe



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

Figure 10. CDF graph: ZHT latencies on EC2 cloud, in microseconds

Table V. 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

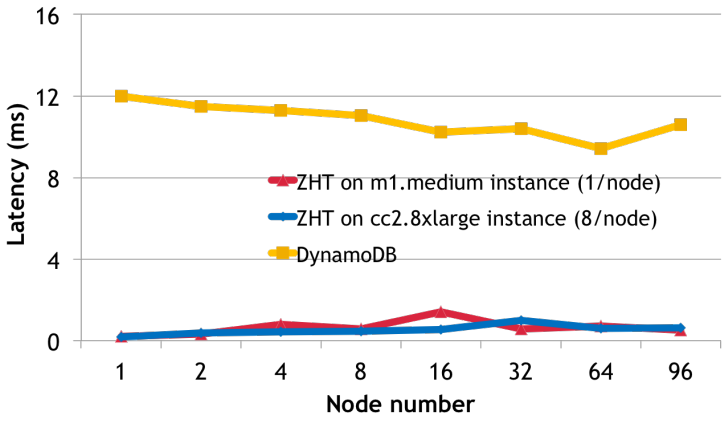


Figure 11. Latency comparison: ZHT v.s DynamoDB on EC2

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 over used throughput so we can't continue to push experiments on larger scales. The slowest 5% requests latency increased by 3 times.

It is worth noting that DynamoDB latencies don't 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 won't get faster response when they only use low throughput. In other words, DynamoDB with more clients doesn't 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

DOI: 10.1002/cpe

capacity by lowering down the latency when they only have fewer clients. When we tried with scales larger than 128 clients for DynamoDB, more than half request failed, because the throughput was beyond the provisioned one.

3.5. System Throughput

3.5.1. 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 scaling, reaching nearly 7.4M ops/sec at 8K-node scale in both cases.

On the HEC-Cluster, ZHT has higher throughput than Cassandra as expected. We expect the performance gap between Cassandra and ZHT to grow as system scales grows due to ZHT's faster routing algorithm. Fig.13 shows the nearly 7x throughput difference between Cassandra and ZHT. As expected, Memcached performed better as well, with 27% higher total throughput.

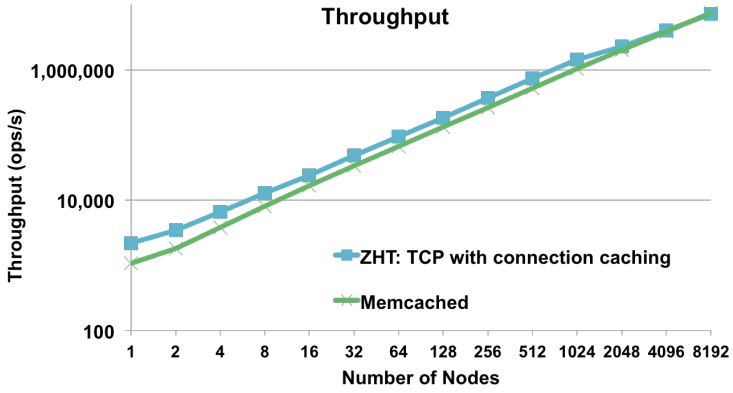


Figure 12. Performance evaluation of ZHT and Memcached plotting throughput vs. scale (1 to 8K nodes on the BLUE GENE/P)

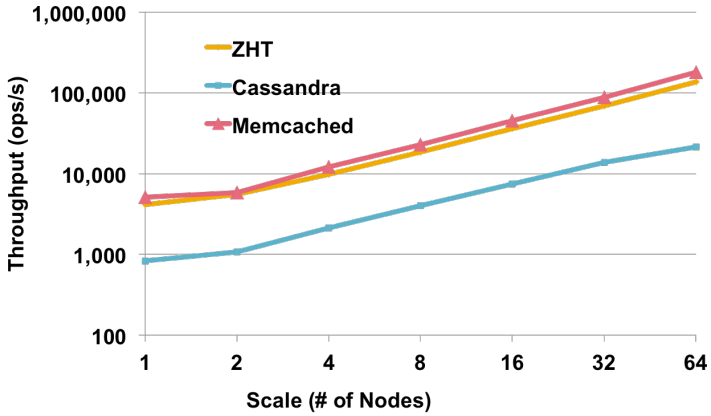


Figure 13. Performance evaluation of ZHT, Memcached and Casandra plotting throughput vs. scale (1 to 64 nodes on the HEC-Cluster)

3.5.2. Synthetic benchmarks On Cloud

DOI: 10.1002/cpe

Throughput In fig.14, due to 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 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 won't 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.

It's 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.

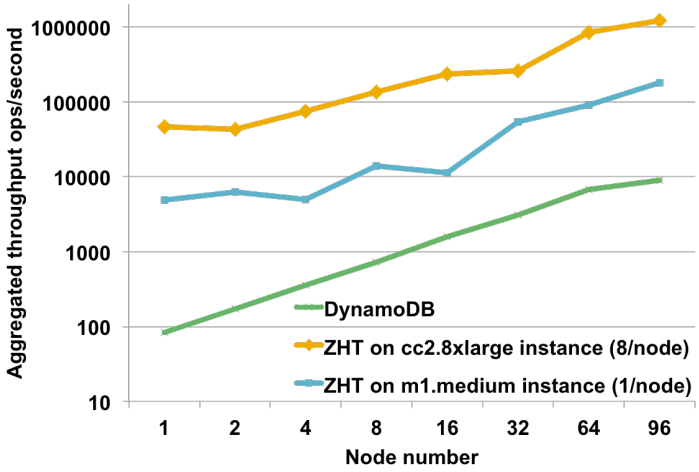


Figure 14. Aggregated throughput of ZHT and DynamoDB on EC2

Running Cost When discussing cloud, the cost is always a big concern [31, 32]. We calculated hourly cost for both ZHT and DynamoDB on different scales. We calculate the ideal cost for DynamoDB, assuming the user always provisions the same throughput to fit their need, then according to Amazon's 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 (fig.15). Note the cost for DynamoDB doesn't 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.

3.6. Scalability and efficiency

Although the throughput achieved by ZHT is impressive at many millions of ops/sec, it is important to investigate the efficiency of the system. Efficiency was computed by comparing ZHT and Memcached performance against the ideal latency/throughput (which was taken to be the better performer at 2-node scale, the smallest test that involving the network communication). In fig.16, we show that Memcached and ZHT achieve different levels of

Prepared using cpeauth.cls
DOI: 10.1002/cpe

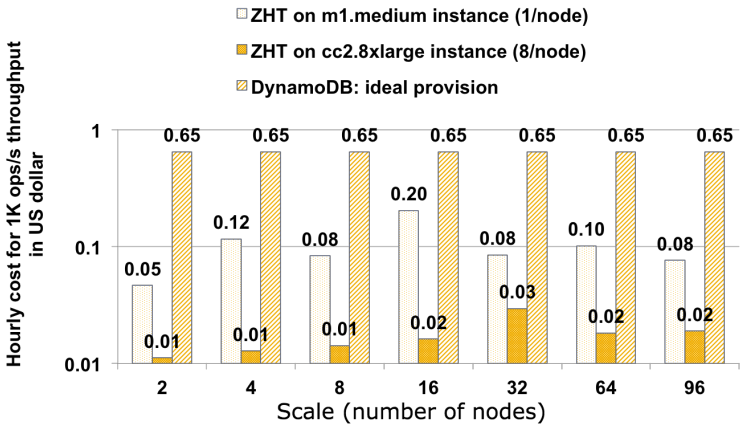


Figure 15. Running cost comparison

efficiency at up to 8K-node scales. 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 Intrepid, 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 simulation showed efficiency drop to 8% at exascale levels (1M nodes). This sounds like ZHT would not scale to an exascale system, but a closer look at what 8% really means is worthy. 100% efficiency implies a latency of about 0.6ms per operation (ZHT latency at 2 node scales). 51% efficiency means 1.1ms latency (this is the performance of ZHT at 8K-node scales). 8% efficiency means the latency is as low as 7ms, at 1M node scales which is still extremely low. At 1M node scales and with latencies of 7ms, ZHT would achieve 150M ops/sec throughput.

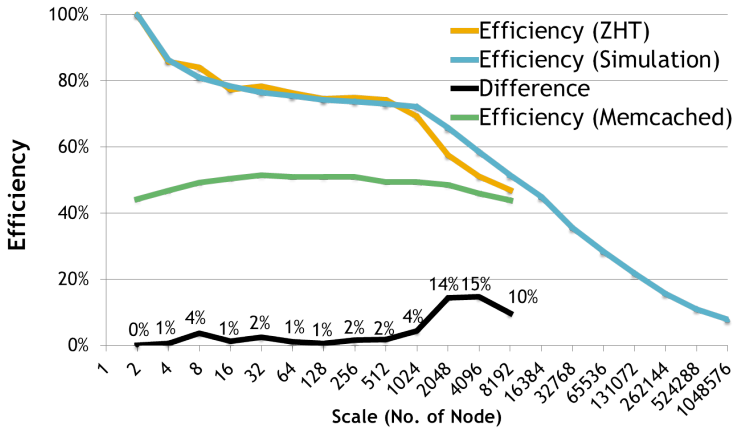
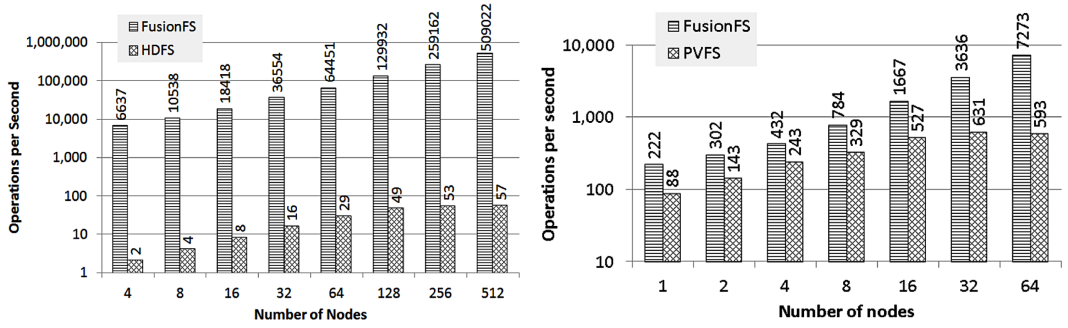


Figure 16. Performance evaluation of ZHT plotting measured efficiency and simulated efficiency vs. scale (1 to 8K nodes on the Blue Gene/P and 1 to 1M nodes on PeerSim)

3.7. Aggregated performance

Each Blue Gene/P compute node has 4 cores, to fully utilize the compute resource, we conduct experiments with various numbers of ZHT instances on each node and measure the request throughput and latency. We expect to achieve higher aggregate throughput by running multiple

DOI: 10.1002/cpe



(a) Throughput on Kodiak (b) Throughput on Intrepid

Figure 17. FusionFS: metadata performance comparison

ZHT instances per node. The experiment results implies that the best resource utilization and efficiency can be achieved by assigning one instance to each core. In a setting with up to 4 instances per node, the aggregated throughput is the compelling (16.1M ops/sec as opposed to 7.3M ops/sec for 1 instance per node, a 2.2X increase), and the latency is still extremely low (2.08ms on 8K-nodes scale with 32K-instances).

4. ZHT AS A BUILDING BLOCK FOR DISTRIBUTED SYSTEMS

This section presents some real systems that have adopted ZHT as a building block. It also lead to additional publications [33–39].

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 [40–42]. 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 [43] 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 fig.17a. 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).

We then compare the metadata performance between FusionFS and PVFS on Intrepid. The result is reported in fig.17b. 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.

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 [44–46], and distributed metadata management with ZHT. Fig.18 shows IStores' metadata performance throughput on 8 to 32 nodes in the HEC-Cluster. The workload

DOI: 10.1002/cpe

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.

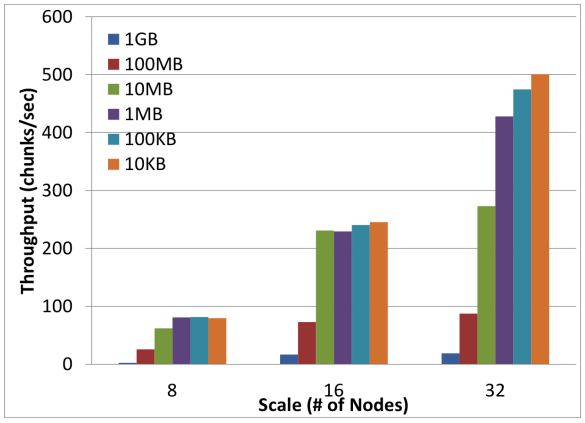


Figure 18. IStore metadata performance on HEC-Cluster

4.3. MATRIX: a Distributed Many-Task Computing Scheduling Framework

MATRIX [47, 48] 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 [49] lightweight task execution framework. Fig.19 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%.

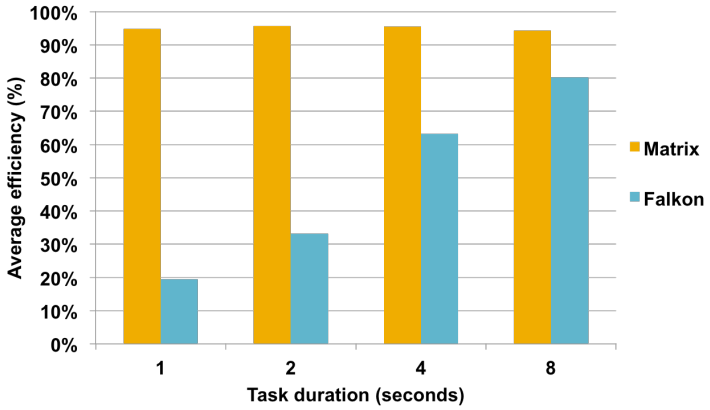


Figure 19. Comparison of MATRIX and Falkon average efficiency (between 256 and 2048 cores) of 100K sleep tasks of different granularity (1 to 8 seconds)

DOI: 10.1002/cpe

4.4. Slurm++: a Distributed HPC Job Launch

We have developed a distributed job launch prototype, SLURM++ based on SLURM [50], 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.

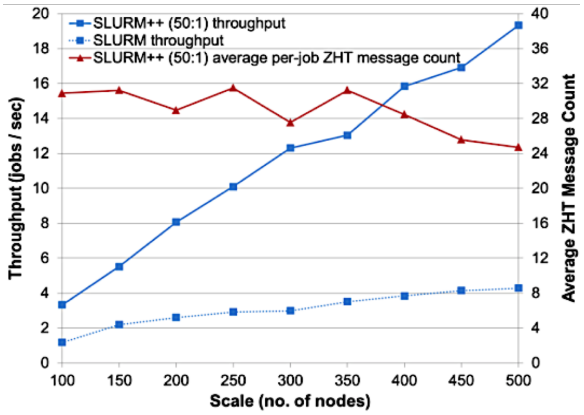


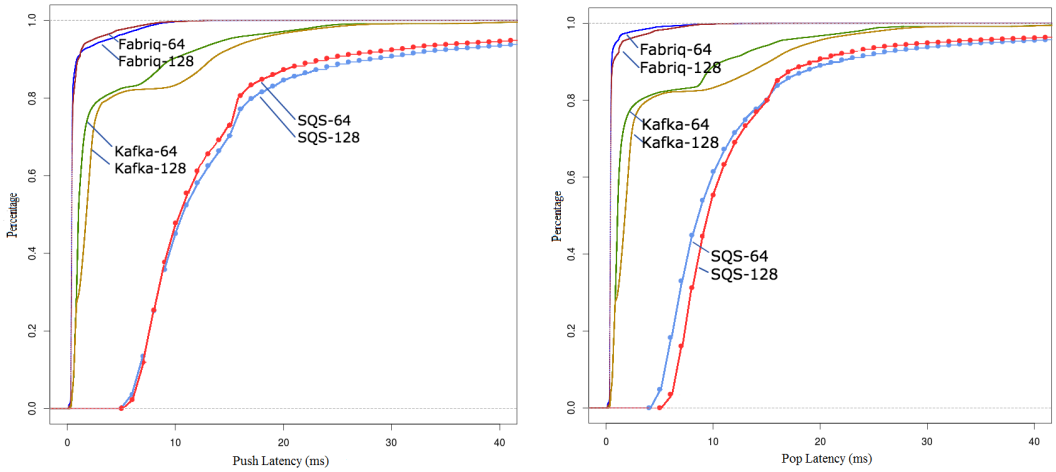
Figure 20. Throughput comparison for Slurm and Slurm++

4.5. Fabriq: a Distributed Message Queue

We propose Fast, Balanced and Reliable Distributed Message Queue (FaBRiQ), a persistent reliable message queue that aims to achieve high throughput and low latency while keeping the near perfect load balance even on large scales. Fabriq uses ZHT as its building block. Fabriq leverages ZHT components to support persistence, consistency and reliable messaging. Another unique feature of Fabriq is the guarantee of exactly once delivery. To our best knowledge, no other DMQ provides such guarantee. Most of the DMQs either provide no delivery guarantee or at least once delivery guarantee. The fact that Fabriq provides low latency makes it a good fit for HPC and MTC workloads that are sensitive to latency and require high performance. Furthermore, providing high throughput on larger scales and persistence makes Fabriq a good option for HTC applications.

At 128 nodes scale, Fabriqs throughput was as high as 1.8 Gigabytes/sec for 1 Megabytes messages, and more than 90,000 messages/sec for 50 bytes messages. At the same scale, Fabriqs latency was less than 1 millisecond. Our framework outperforms other state of the art systems including Kafka and SQS in throughput and latency. According to fig.21a, at the 50 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.42ms, 1.03ms, and 11ms. However, the problem with the Kafka is having a long tail on latency. At the 90 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.89ms, 10.4ms, and 10.8ms. We can notice that the range of latency on Fabriq significantly shorter than the Kafka. At the 99.9 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 11.98ms, 543ms, and 202ms. Similarly, fig.21b shows a long range on the pop operations for Kafka and SQS. The maximum pop operation time on the on Fabriq, Kafka, and SQS were respectively 25.5ms, 3221ms, and 512ms.

DOI: 10.1002/cpe



(a) CDF: push latency comparison

(b) CDF: pop latency comparison

Figure 21. Comparison for Fabriq vs. SQS, IronMQ, HDMQ, and Windows Azure Service Bus

5. RELATED WORK

In this section we introduce some existing works related to ZHT, including distributed hash tables and key-value stores, because they play critical roles in building scalable distributed systems. Numerous distributed hash tables and key-value stores have been proposed and implemented over the years. Some widely cited and discussed projects and solutions (including Chord [51], CAN [52], Pastry [53], Kademlia [54], Tapestry [55], RIAK [56] and Cassandra [15]) adopt logarithmic routing algorithms, resulted in increased latency with systems scale. Some other works such as Dynamo [57] and Memcached [16] , use constant routing algorithms to achieve a nearly constant latencies like ZHT. Amazon's Dynamo is a key-value storage system that Dynamo hosts some of Amazon's core services. It inspired many similar projects and started a NoSQL trend in both academia and industries. The major focus of Dynamo is to provide an "always-on" experience to it's upper level applications. Dynamo claim to be a zero-hop DHT, where each server has enough routing information locally to route requests to the appropriate target server directly. Memcached is a simple but efficient in-memory a key-value store. It was designed as a cache to speed up distributed data access such as web page caches. Due to it's specific purpose, Memcached doesn't support dynamic membership, replication and persistence. The length of the keys and values are strictly limited to 250 and 1M bytes respectively. Cassandra is firstly inspired by Amazon's Dynamo, strives to be an "always writable" system. In later implementations, it's more considered as a column-family store, like Google's BigTable [58], although it still support key-value interfaces. Cassandra is very popular in industries, but it gets little use in high performance computing areas many supercomputers don't have good support on Java stack. Another drawback of Cassandra is its logarithmic routing strategy, which makes the performance scalability a big issue.

Some recent key-value store projects generally focus on providing new features or exploring new approaches to boost performance. HyperDex [59] 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 [60]. 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 [61] is a replicated key-value store that allows applications to declare their consistency and latency priorities via consistency-based service level agreements (SLAs). MICA [62] is another scalable key value store that can handle millions of operations per

DOI: 10.1002/cpe

second using single general multi purpose core system. MICA achieved this by encompassing all aspects of request handling by enabling parallel access to data, network request handling, and data structure design. SPANStore [63] presents a key value store that exports a unified view of storage services in geographically distributed data centers. SPANStore combines three main principles, span multiple cloud providers to minimize cost, estimating application workload at the right granularity and finally minimizing use of compute resources. SPANStore in some scenarios was able to lower cost by 10X. Masstree [64] presents anther key value store designed for SMP machines. Masstree functions by keeping all data in memory in a form of concatenated B+ trees. Lookups use optimistic concurrency control, a read-copy-update like technique but no writing on shared data. With these techniques Masstree is able to execute more than six million simple queries per second. For enabling NoSQL databases to handle highly concurrent distributed transactions, Claret [65] is recently proposed. It utilizes abstract data type (ADT) semantics in databases to provide a safe and scalable programming model for NoSQL databases. LOCS [66] is system equipped with customized SSD design, which exposes its internal flash channels to applications, to work with LSM-tree based KV store, specifically LevelDB in LOCS. Main motivation of LOCS was to overcome inefficiencies to naively combining LSM-tree based KV stores with SSD. They were able to show 4X increase in storage throughput after applying the proposed optimization techniques. Small Index Large Table (SILT) [67] presents a memory efficient high performance key value store system based on flash storage that can scale to serve billions of key value items on a single node. SILT focuses on using algorithmic and systemic techniques to balance the use of memory, storage and computation.

6. CONCLUSION

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. It's also flexible, supporting dynamic nodes join and departure. We have shown ZHT's 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 AMD Opteron 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 HPC, 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.

ACKNOWLEDGEMENT

This work was supported in part by the National Science Foundation grant NSF-1054974. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE- AC02-06CH11357. This research used Amazon EC2 cloud research grant.

DOI: 10.1002/cpe

REFERENCES

1. Wang, Y., Su, Y., Agrawal, G.: Supporting a Light-Weight Data Management Layer Over HDF5. In: CCGrid'13, Delft, the Netherlands, pp. 335–342 (2013). IEEE
2. Wang, Y., Agrawal, G., Ozer, G., Huang, K.: Removing sequential bottlenecks in analysis of next-generation sequencing data. In: IPDPSW, 2014 IEEE International, Phoenex, Arizona (2014). IEEE
3. Batcher, K.: Supercomputer IO Quote. `http://en.wikipedia.org/wiki/Ken_Batcher`. Accessed: 2014-12-30
4. Raicu, I., Foster, I.T., Beckman, P.: Making a case for distributed file systems at exascale. In: Proceedings of the Third International Workshop on Large-scale System and Application Performance. LSAP '11, pp. 11–18, San Jose, California, USA (2011)
5. Schmuck, R.H.: GPFS: A shared-disk file system for large computing clusters. In: Proceedings of the USENIX Conference on File and Storage Technologies. FAST'02, pp. 16–16, Monterey, CA (2002)
6. Carns, P.H., III, W.B.L., Ross, R.B., Thakur, R.: PVFS: A parallel file system for linux clusters. In: In Proceedings of Annual Linux Showcase and Conference, Atlanta, Georgia, pp. 317–327 (2000)
7. Schwan, P.: Lustre: Building a file system for 1000-node clusters. Proc. of the 2003 Linux Symposium, pp. 174–186, Berkeley, CA, USA (2003)
8. Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium On. MSST'12, pp. 1–1, Pacific Grove, CA, USA (2012)
9. Liu, N., Fu, J., Carothers, C.D., Sahni, O., Jansen, K.E., Shephard, M.S.: Massively parallel I/O for partitioned solver systems. Parallel Processing Letters **20**(4), 377–395 (2010)
10. Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z., Raicu, I.: ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IPDPS '13, pp. 775–787, Boston, MA (2013)
11. Li, T., Verma, R., Duan, X., Jin, H., Raicu, I.: Exploring distributed hash tables in highend computing. SIGMETRICS Performance Evaluation Review **39**(3), 128–130 (2011)
12. Li, T., Tejada, A.P.D., Brandstatter, K., Zhang, Z., Raicu, I.: ZHT: a zero-hop DHT for high-end computing environment. In: Greater Chicago Area System Research Workshop. GCASR, pp. 20–21, Chicago, IL (2012)
13. Li, T., Zhou, X., Brandstatter, K., Raicu, I.: Distributed key-value store on hpc and cloud systems. In: Greater Chicago Area System Research Workshop. GCASR' 13, pp. 6–7, Chicago, IL (2013)
14. Grolinger, K., Higashino, W.A., Tiwari, A., Capretz, M.A.M.: data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing, Advances, Systems and Applications, Springer **2**, 22 (2013)
15. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
16. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M.: Scaling memcache at facebook. In: Proceedings of the USENIX Conference on Networked Systems Design and Implementation. NSDI, pp. 385–398, Lombard, IL (2013)
17. DynamoDB. `http://aws.amazon.com/dynamodb/`. Accessed: 2014-12-15
18. Grolinger, K., Higashino, W., Tiwari, A., Capretz, M.: Data management in cloud environments: Nosql and newsql data stores. Journal of Cloud Computing: Advances, Systems and Applications **2**(1) (2013)
19. epoll - Linux man page. `http://linux.die.net/man/4/epoll`. Accessed: 2015-1-30
20. Yang, X., Zhou, Z., Tang, W., Zheng, X., Wang, J., Lan, Z.: Balancing job performance with system performance via locality-aware scheduling on torus-connected systems. In: Cluster Computing (CLUSTER), 2014 IEEE International Conference On, Madrid, Spain, pp. 140–148 (2014). IEEE
21. Zhou, Z., Yang, X., Lan, Z., Rich, P., Tang, W., Morozov, V., Desai, N.: Improving batch scheduling on Blue Gene/Q by relaxing 5d torus network allocation constraints. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing. IPDPS'15, Hyderabad, India (2015)
22. Brandstatter, K., Li, T., Zhou, X., Raicu, I.: Novoht: a lightweight dynamic persistent NoSQL key/value store. In: Greater Chicago Area System Research Workshop. GCASR' 13, pp. 27–28, Chicago, IL (2013)
23. Google Protocol Buffers. `http://code.google.com/apis/protocolbuffers`. Accessed: 2014-09-30
24. Amazon EC2 Cloud. `http://aws.amazon.com/ec2/`. Accessed: 2014-11-30
25. Parallel Reconfigurable Observational Environment. `http://www.nmc-probe.org/`. Accessed: 2014-11-30
26. IBM BlueGene Supercomputers. `http://en.wikipedia.org/wiki/Blue_Gene`. Accessed: 2012-1-30
27. Argonne Leadership Computing Facility. `https://www.alcf.anl.gov/`. Accessed: 2014-11-30
28. Kyotocabinet. `http://fallabs.com/kyotocabinet/`. Accessed: 2012-09-30
29. Liu, N., Carothers, C.: Modeling billion-node torus networks using massively parallel discrete-event simulation. In: Proceedings of Workshop on Principles of Advanced and Distributed Simulation , Washington, DC, USA, pp. 1–8 (2011)
30. Liu, N., Haider, A., Sun, X., Jin, D.: Fattreesim: Modeling a large-scale fat-tree network for hpc systems and data centers using parallel and discrete event simulation. In: Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), 2011 ACM SIGSIM, London, UK, pp. 199–210 (2015). ACM, IEEE
31. Sadooghi, I., Hernandez Martin, J., Li, T., Brandstatter, K., Zhao, Y., Maheshwari, K., Pais Pitta de Lacerda Ruivo, T., Timm, S., Garzoglio, G., Raicu, I.: Understanding the performance and potential of cloud computing for scientific applications. IEEE Transactions on Cloud Computing **PP**(99), 1–1 (2015)
32. Sadooghi, I., Zhao, D., Li, T., Raicu, I.: Understanding the cost of cloud computing and storage. In:

DOI: 10.1002/cpe

Greater Chicago Area System Research Workshop. GCASR' 12, pp. 14–15, Chicago, IL (2012)
33. Li, T., Raicu, I., Ramakrishnan, L.: Scalable state management for scientific applications in the cloud. BigData Congress '14, pp. 204–211, Anchorage, Alaska (2014)
34. Li, T., Keahey, K., Sankaran, R., Beckman, P., Raicu, I.: A cloud-based interactive data infrastructure for sensor networks. IEEE/ACM Supercomputing/SC'14, pp. 14–15, New Orleans, LA, USA (2014)
35. Zhao, D., Yin, J., Qiao, K., Raicu, I.: Virtual chunks: On supporting random accesses to scientific data in compressible storage systems. In: Big Data, 2014 IEEE International Conference On. IEEE BigData'14, pp. 231–240, Washington DC, USA (2014)
36. Wang, K., Kulkarni, A., Lang, M., Arnold, D., Raicu, I.: Exploring the design tradeoffs for extreme-scale high-performance computing system software. In: IEEE Transaction on Parallel and Distributed Systems (TPDS), pp. 1–1 (2015)
37. Li, T., Keahey, K., Wang, K., Zhao, D., Raicu, I.: A dynamically scalable cloud data infrastructure for sensor networks. ACM ScienceCloud 15, pp. 25–28, Portland, Oregon, USA (2015)
38. Li, T., Ma, C., Li, J., Zhou, X., Wang, K., Zhao, D., Sadooghi, I., Raicu, I.: Graph/z: A key-value store based scalable graph processing system. In: IEEE International Conference on Cluster Computing. Cluster'15, p. , Chicago, IL, USA (2015)
39. Zhao, D., Liu, N., Kimpe, D., Ross, R., Sun, X.-H., Raicu, I.: Towards exploring data-intensive scientific applications at extreme scales through systems and simulations. In: IEEE Transaction on Parallel and Distributed Systems. TPDS, pp. 1–1 (2015)
40. Zhao, D., Zhang, Z., Zhou, X., Li, T., Wang, K., Kimpe, D., Carns, P., Ross, R., Raicu, I.: Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: Big Data, 2014 IEEE International Conference On. IEEE BigData'14, pp. 61–70, Washington DC, USA (2014)
41. Zhao, C.S., Zhang, Z., Sadooghi, I., Zhou, X., Li, T., Raicu, I.: Fusionfs: a distributed file system for large scale data-intensive computing. In: Greater Chicago Area System Research Workshop. GCASR, pp. 17–18, Chicago, IL (2013)
42. Zhao, D., Shou, C., Zhang, Z., Sadooghi, I., Zhou, X., Li, T., Raicu, I.: Fusionfs: a distributed file system for large scale data-intensive computing. In: Greater Chicago Area System Research Workshop. GCASR' 13, pp. 10–11, Chicago, IL (2013)
43. W.Vogels: Filesystem in Userspace. `http://fuse.sourceforge.net/`. Accessed: 2011-09-17
44. McAuley, A.J.: Reliable broadband communication using a burst erasure correcting code. In: Proceedings of the 1990 ACM SIGCOMM International Conference on on Data Communication. SIGCOMM '90, pp. 297–306, Philadelphia, Pennsylvania, USA (1990)
45. Dimakis, A.G., Prabhakaran, V., Ramchandran, K.: Decentralized erasure codes for distributed networked storage. IEEE/ACM Trans. Netw. **14**(SI), 2809–2816 (2006)
46. Li, M., Shu, J., Zheng, W.: GRID Codes: Strip-based erasure codes with high fault tolerance for storage systems. ACM Transactions on Storage **4**(4), 15–11522 (2009)
47. Wang, K., Kulkarni, A., Lang, M., Arnold, D., Raicu, I.: Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In: Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing. ACM HPDC '15, pp. 219–222, Portland, Oregon, USA (2015)
48. Wang, K., Zhou, X., Li, T., Lang, M., Raicu, I.: Optimizing load balancing and data-locality with data-aware scheduling. In: Big Data, IEEE International Conference On. IEEE BigData'14, pp. 119–128, Washington DC, USA (2014)
49. Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I., Wilde, M.: Falkon: A fast and light-weight task execution framework. SC '07, pp. 1–12, Reno, Nevada (2007)
50. Jette, M.A., Yoo, A.B., Grondona, M.: Slurm: Simple linux utility for resource management. In: In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP), Seattle, WA, USA, pp. 44–60 (2003)
51. Stoica, R.M., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. ACM SIGCOMM, pp. 149–160, San Diego, CA, USA (2001)
52. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proceedings of the 2001 ACM SIGCOMM International Conference on on Data Communication. SIGCOMM, pp. 161–172, New York, NY, USA (2001)
53. Rowstron, Druschel, P.: Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. Middleware, pp. 329–350, London, UK, UK (2001)
54. Maymounkov, P., Mazières, D.: 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, pp. 53–65, London, UK, UK (2001)
55. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communication **22**(1), 41–53 (2004)
56. Riak. `http://docs.basho.com/riak/latest/`. Accessed: 2014-1-30
57. DeCandia, D.H., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: dynamo: Amazons highly available key-value store. In: Proceedings of the ACM Symposium on Operating Systems Principles. SOSP, pp. 205–220, New York, NY, USA (2007)
58. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4–1426 (2008)
59. Escriva, R., Wong, B., Sirer, E.G.: Hyperdex: A distributed, searchable key-value store. In: Proceedings of the 2012 ACM SIGCOMM International Conference on on Data Communication. SIGCOMM '12, pp.

DOI: 10.1002/cpe

25–36, Helsinki, Finland (2012)
60. Debnath, B., Sengupta, S., Li, J.: 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, pp. 25–36, Athens, Greece (2011)
61. Terry, D.B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M.K., Abu-Libdeh, H.: Consistency-based service level agreements for cloud storage. In: Proceedings of the ACM Symposium on Operating Systems Principles. SOSP '13, pp. 309–324, Farminton, Pennsylvania (2013)
62. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: a holistic approach to fast in-memory key-value storage. In: Proceedings of the USENIX Conference on Networked Systems Design and Implementation. NSDI'14, pp. 429–444, Berkeley, CA, USA (2014)
63. Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., Madhyastha, H.V.: Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In: Proceedings of the ACM Symposium on Operating Systems Principles. SOSP '13, pp. 292–308. ACM, New York, NY, USA (2013)
64. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems. EuroSys '12, pp. 183–196. ACM, New York, NY, USA (2012)
65. Holt, B., Zhang, I., Ports, D., Oskin, M., Ceze, L.: Claret: Using data types for highly concurrent distributed transactions. In: Workshop on Principles and Practice of Consistency for Distributed Data. PaPoC '15, pp. 1–4, Bordeaux, France (2015)
66. Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., Cong, J.: An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In: The European Conference on Computer Systems. EuroSys '14, pp. 16–11614. ACM, Amsterdam, The Netherlands (2014)
67. Lim, H., Fan, B., Andersen, D.G., Kaminsky, M.: Silt: A memory-efficient, high-performance key-value store. In: Proceedings of the ACM Symposium on Operating Systems Principles. SOSP '11, pp. 1–13. ACM, New York, NY, USA (2011)

DOI: 10.1002/cpe