ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table

Tonglin Li[1], Xiaobing Zhou[1], Kevin Brandstatter[1], Dongfang Zhao[1],
Ke Wang[1], Anupam Rajendran[1], Zhao Zhang[2], Ioan Raicu[1,3]
tli13@hawk.iit.edu, xzhou40@hawk.iit.edu, kbrandst@iit.edu, dzhao8@hawk.iit.edu,
kwang22@hawk.iit.edu, arajend5@hawk.iit.edu, zhaozhang@uchicago.edu, iraicu@cs.iit.edu

[1]*Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA*
[2]*Department of Computer Science, University of Chicago, Chicago IL, USA*
[3]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA*

Abstract— **This paper 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 of 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 (providing lock-free concurrent key/value modifications) in addition to insert/lookup/remove. 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. This work provides three real systems that have integrated with ZHT, and evaluate them at modest scales. 1) ZHT was used in the FusionFS distributed file system to deliver distributed meta-data management at over 60K operations (e.g. file create) per second at 2K-core scales. 2) ZHT was used in the IStore, an information dispersal algorithm 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 job scheduling system, delivering 5000 jobs/sec throughputs at 2K-core scales. We compared ZHT against other distributed hash tables and key/value stores and found it offers superior performance for the features and portability it supports.**

Keywords- Distributed hash tables, key/value stores, high-end computing

I. INTRODUCTION

Exascale computers (e.g. capable of 10^{18} ops/sec) [1], 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, drug discovery, and economics) 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".
-- Ken Batcher

The quote [46] 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 2012, some of the biggest systems already have more than 700,000 general purpose cores. Many experts predict [1] that exascale computing will be a reality by 2019; an exascale system is expected to have millions of nodes, billions of threads of execution, hundreds of petabytes of memory, and exabytes 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 [2], PVFS [3], and Lustre [4]). 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 (see [2]).

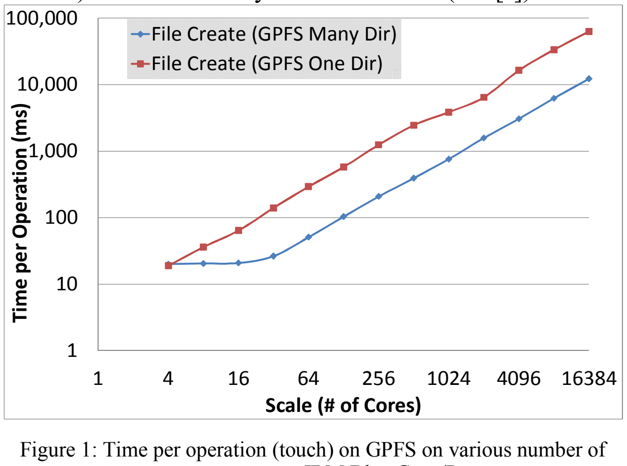

Figure 1: Time per operation (touch) on GPFS on various number of processors on a IBM Blue Gene/P

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 [5, 6] shows these times to be even worse, putting the full system scale metadata operations in the hour range, but the testbed 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 [7] and Yahoo's HDFS [8]) 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 [9], 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 zero-hop distributed hash table (ZHT), 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.

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 [38] and Memcached [20] and found it to offer superior performance for the features and portability it supports, at large scales up to 16K-nodes.

This work provides three real systems 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 at over 60K operations (e.g. file create) per second at 2K-core scales. 2) ZHT was used in the IStore [50, 65], an information dispersal algorithm 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 job scheduling system, delivering 5000 jobs/sec throughputs at 2K-core scales.

The contributions of this paper are as follows:

- Design and implementation of ZHT, a light-weight, high performance, fault tolerant, persistent, dynamic, and highly scalable distributed hash table, optimized for high-end computing.
- Support for unconventional operations, such as append allowing data to be incrementally added to an existing value, delivering lock-free concurrent modification on key/value pairs.
- Micro-benchmarks up to 32K-core scales, achieving latencies of 1.1ms and throughput of 18M ops/sec.
- Integration and evaluation with three real systems (FusionFS, IStore, and MATRIX), managing distributed storage metadata and distributed job scheduling information.

II. RELATED WORK

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 [15], CAN [16], Chord [17], Pastry [18], Tapestry [19], Memcached, Dynamo [21], Cycloid [22], Ketama [23], RIAK [24], Maidsafe-dht [25], Cassandra and C-MPI [26]. Most of these DHTs scale logarithmically with system scales, but some (e.g. Cycloid) go as far as reducing the number of operations to O(c) where c is a constant related to the maximum size of the network (instead of the actual size of the network), which in practice still results to c ~ log(N) [22].

There has been some uptake recently in using traditional DHTs in HEC, namely the C-MPI [26] project, in which the Kademlia DHT has been implemented and shown to run well on 1K nodes on a Blue Gene/P supercomputer. C-MPI is used to perform data management operations for the Swift project [27, 57], but it is rather simplistic (e.g. no support for data replication, data persistence, or fault tolerance via stateless protocols). C-MPI adopted the Message Passing Interface (MPI) for communication, making it a bridle at large scale and prone to system wide failures due to single node failures. Although MPI is attractive from a performance perspective on these HEC systems, it makes it hard to implement a fault tolerant system. Furthermore, C-MPI is based on new implementations of the Kademlia (with log(N) routing time) distributed hash table. Another recent project using DHTs on a HEC is DataSpaces [28], which deploys a DHT on a Cray XT5 to coordinate in-memory data management for simulation workflows. DataSpaces has similar drawbacks as C-MPI. In future work, we will consider supporting MPI, in addition to protocols such as TCP and UDP, as MPI 3.0 [29] promises to address many of the current MPI fault tolerance limitations.

Dynamo [21] is a key-value storage system that some of Amazon's 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. Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. A significant drawback of Dynamo is the fact that it is an internal Amazon project, which cannot be used outside of the Amazon infrastructure.

Cassandra, an implementation inspired by Amazon's Dynamo, strives to be an "always writable" system in that the system is designed to always accept writes even in light of node failures. It accomplishes this by deferring consistency until the time when data is read and resolving conflicts at that time, this means that Cassandra needs to offer different levels of consistency on reads. Cassandra's 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 accelerate distributed application execution. It is rather simplistic in which there is no data persistence, no data replication, and no dynamic membership. There are strict limitations on the size of the keys and values (250B and 1MB respectively). All these limit the use of Memcached for the purpose of making it a building block for large-scale distributed systems, but it offers a good baseline for comparison.

In section 4 we'll compare the performance of ZHT, Cassandra and Memcached. A brief overview of the differences between Cassandra, Memcached, C-MPI, Dynamo, and ZHT can be found in Table 1.

Table 1: Comparison between ZHT and other DHT implementations

Name	Impl.	Routing Time	Persistence	Dynamic membership	Append
Cassandra 38	Java	log(N)	Yes	Yes	No
Memcached [20]	C	2	No	No	No
C-MPI [26]	C/MPI	log(N)	No	No	No
Dynamo [21]	Java	0 to log(N)	Yes	Yes	No
ZHT [14]	C++	0 to 2	Yes	Yes	Yes

III. 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. By making dynamic membership optional, the complexity of the system can be reduced and a low average number of hops per operation can be achieved.

We do 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. 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. This in principle guided our design of the proposed dynamic membership support in ZHT.

It is also important to point out that nodes in a HEC system are generally trust-worthy, and that stringent requirements to encrypt communication and/or data would simply be adding overheads. HEC systems are generally locked down from the outside world, behind login nodes and firewalls, and although authentication and authorization are still needed, full communication encryption is wasteful for a large class of scientific computing applications that run on many HEC systems. Most storage systems used in HEC communicate between the client nodes and storage servers without any encryption.

A. Overview

The primary goal of ZHT is to get all the benefits of DHTs, namely excellent availability and fault tolerance, but concurrently achieve the benefits 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 application programming interface (API) of ZHT is kept simple and follows similar interfaces for hash tables. The four operations ZHT supports are 1. int *insert*(key, value); 2. value *lookup*(key); 3. int *remove*(key), and 4. int *append*(key, value). Keys are typically a variable length ASCII text string. Values can be complex objects, with varying size, number of elements, and types of elements. Integer return values return 0 for a successful operation, or a non-zero return code that includes information about the error that occurred.

In static membership, every node at bootstrap time has all information about 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.

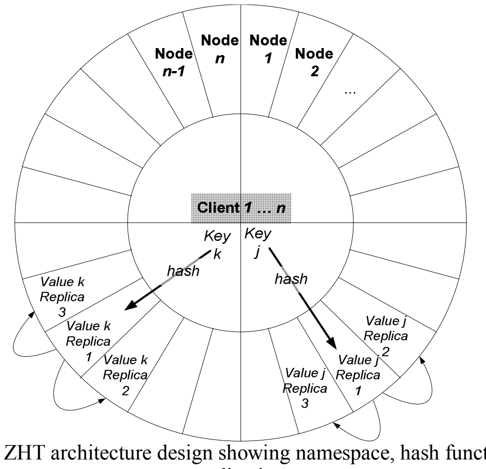

Figure 2: ZHT architecture design showing namespace, hash function, and replication

ID Space and Membership Table are shown in Figure 2 as a ring-shaped key name space. The node ids in ZHT can be randomly distributed throughout the network, or they can be closely correlated with the network distance between nodes. The correlation can generally be computed from information such as MPI rank or IP address. The random distribution of the

ID space has worked well up to 32K-cores, but we will explore a network aware topology in future work.

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%) portion of available memory of each node. 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 stay in memory, users can achieve the balance between performance and memory consumption.

B. *Terminologies:*

Physical node: A physical node is an independent physical machine. Each physical node may have several ZHT instances which are differentiated with IP address and port. By adjusting the number of instance, ZHT can fit in heterogeneous systems with various computing power.

Instance: A ZHT instance is a process which 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.

Manager: A Manager is a service 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.

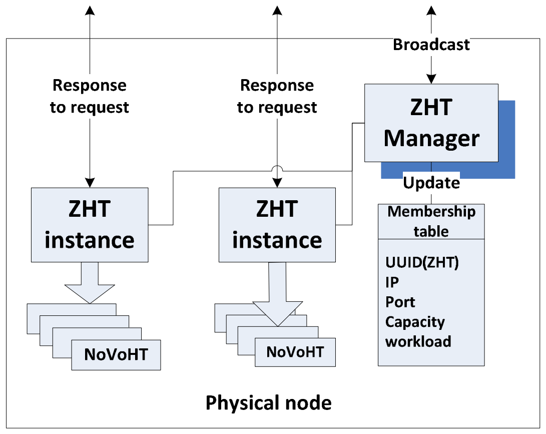

Figure 3: ZHT architecture per node

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) on the performance of partitions as we increase the number of partitions per instance (see Figure 4). This design allows us to avoid a potentially expensive rehash of many key/value pairs when the need arises to migrate partitions.

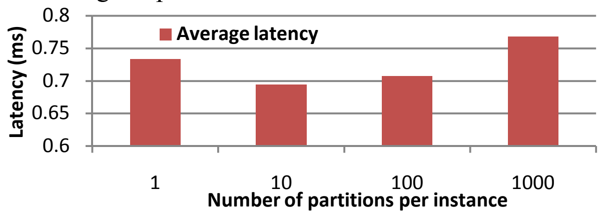

Figure 4: Concurent performance from 1 to 1K partition per ZHT instance

C. *Membership management*

ZHT supports both static and dynamic node membership. In the static case, the bootstrapping phase gets information from the batch job scheduler about the allocated node information (or perhaps the information could be extracted from the nodes at job start time). 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 extremely 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 maintain high performance with constant time routing.

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 node's neighbor and move some of the partitions from the "busy" node to itself. Migrating a *partition* is as easy as moving a file, all without having to rehash the key/value pairs stored in the partition. Some issues come up with ZHT requests to a migrating partition, as requests destined for a specific partition could continue to go to the original partition location until the partition migration is completed, and all membership tables have been updated. During this migration, every request received is acknowledged with a redirect message informing the recipient of the new partition location. This "lazy" membership updating mechanism on client side reduces the number of messages to 1 message per client for each partition. Once the migration is completed, the manager broadcasts out the incremental information of membership in an atomic manner, and all future requests will go to the new partition location.

Node departures: On planed 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 migrate 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. Replication is used to improve reliability, the client then sends the request to the first replica of the failed node. At the same time, the manager updates and broadcasts its local membership table, and initiates a rebuilding of the replicas, specifically increasing replication on all partitions stored on the failed physical node in order to maintain the specified level of replication.

Data Migration: An essential design decision was to ensure that minimal impact on performance and scalability would be posed by introducing dynamic membership. With dynamic membership, 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 with only membership table modifications. This avoids the need 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, and should be able to achieve near disk and network peak bandwidth performance. When migration is in progress, ZHT state cannot be modified for the migrated partitions. All requests are queued, until the migration is completed. In the meanwhile because the partition state is locked, corresponding replicas also won't change. This keeps the entire system state consistent. If failure occurs during migration, simply don't 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 the consistent state.

Client Side State: In case that client and server are not on the same nodes, it's necessary to keep client side membership table updated. Since the node joining and leaving will change the number of partitions covered by ZHT instance, clients might send request 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. Our typical deployment scenario has a 1:1 ratio between clients and servers, which implies that the client could share the membership table with a corresponding server on the same physical node, to reduce the number of membership tables that need to be synchronized on modification.

D. *Server arhitecture*

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 much more streamlined architecture, an event-driven model server architecture based on epoll. The current epoll-based ZHT outperforms the multithread version 3X. We'll discuss the performance difference in more detail in the evaluation section IV.

E. *Hashing Functions*

There are many good hashing functions in practice [31]. 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 [32, 33]. 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 [49].

F. *Lightweight 1-1 Communication*

We implemented ZHT with both TCP (with server returned result state) and UDP (acknowledge message based, which means every time a message is sent, the sender is waiting for an acknowledge message) protocols. In previous work [14], we showed that UDP offered some performance advantage at modest scales of nearly 6K cores. We anticipate that UDP's advantages will become more prevalent with even larger scales as connectionless communication protocols will be preferred to avoid having expensive connection establishments among many nodes. In ZHT, we implemented a LRU cache for TCP connections, which makes TCP works almost as fast as UDP does. We expect to extend the communication protocols in future iterations of ZHT, such as BMI [41], and perhaps even MPI if we are willing to sacrifice fault tolerance for potentially improved performance and accessibility to certain HEC systems that do not support the IP protocol.

G. *Complex Structures Support*

In order to support complex structures as values in ZHT, we adopted the Google protocol buffer [37] project, which serializes complex structures into a stream of bytes. The indicators for four basic operations (insert, lookup, remove, and append) are defined in the message prototype and compiled with Google Protocol Buffers. They are encapsulated with the key-value pair into a plain string and transferred through network. When a server receives a request, it just unpacks the message, read the indicator and execute the operation request.

H. *Fault Tolerance*

ZHT gracefully handle 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 in response to the failure, and queries asking for data that were on the failed node will be answered by the replicas.

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.

We have evaluated these mechanisms to work on modest scales and include the results in the evaluation section.

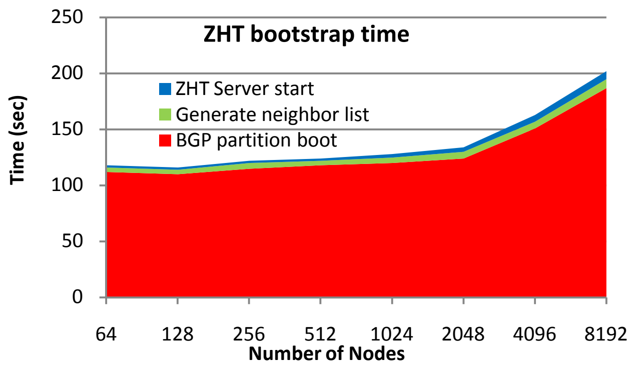

Figure 5: ZHT Bootstrap time on Blue Gene/P from 64 to 8K nodes

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 Figure 5).

Nevertheless, we expect the time to bootstrap ZHT to be insignificant in relation to the cost to the batch scheduler's 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 [5], after which the ZHT bootstrap takes another 8 seconds and it takes 10 seconds to bootstrap at 8K-node scale. Figure 5 shows the bootstrap time increase with the scale.

I. *Persistence*

ZHT is a distributed in-memory data-structure. In order to withstand failures and restarts, it also supports persistence. We evaluated several existing systems, such as KyotoCabinet HashDB [39] and BerkeleyDB, but low performance and missing features prompted us to implement our own solution. We designed and implemented a Non-Volatile Hash Table (NoVoHT) which uses a log-based persistence mechanism with periodic checkpointing. NoVoHT was designed to address several limitations of KyotoCabinet, specifically to enable specifying a size (to control memory footprint), re-size rate (how often to increase or decrease the size of the table), and garbage collection (how often to reclaim unused space on persistent storage).

Since all key-value pairs are kept in memory, it lends itself to low latency in lookups when compared to other persistent hash maps such as KyotoCabinet's HashDB[39], which are disk-based and any lookup must hit disk.

In addition to the standard get, put, remove functions that are inherent in a hash map, NoVoHT supports a fourth basic function, append. The append allows a string to be appended to a value that is currently in the hash map. 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 for fast concurrent modification of a value in the map, utilizing a local lock. 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. As shown in Figure 1, metadata operation on 16K processor scale could be as slow as 63 seconds per operation. By using append, all metadata servers can store entries under the same key (associated with the parent directory), all without distributed locking (simple local locks are still needed to prevent multiple threads from concurrently modifying the same memory location).

J. *Consistency*

ZHT uses replication to enhance reliability. Replicas have distinct ordering in terms of which ones are accessed by clients. This means that clients will generally be interacting with a single replica (e.g. primary replica), and consistency is straightforward to be maintained, at the cost of potential loss of performance advantages if we allowed multiple replicas to be concurrently modified. 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 a weak consistency model. Using this approach, ZHT achieves high throughput and availability while maintains reasonable consistency level.

K. *Implementation*

ZHT has been under development for 2 years with 4.5 years of man-hours. It is implemented in C/C++, and has very few dependencies. It consists of 6700 lines of code, and is an open source project accessible at [45]. The dependencies of ZHT are NoVoHT and Google Protocol Buffers [37]. NoVoHT itself has no dependencies other than a modern gcc compiler.

IV. PERFORMANCE EVALUATION VIA MICROBENCHMARKS

In this section, we describe the performance of ZHT, including hashing functions, persistence, throughput, latencies, and replication. Firstly we'll 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.

A. *Testbeds, Metrics, and Workloads*

We used several machines to evaluate ZHT's performance in this paper.

- **Intrepid**: an IBM Blue Gene/P supercomputer [40] at Argonne Leadership Computing Facility [44], Argonne National Lab: 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.
- **HEC-Cluster**: a 64-node (512-core) cluster at IIT: each node has a dual processor quad-core, 8GB RAM. This testbed was used to compare ZHT with Cassandra and Memcached.
- **DataSys**: an 8-core x64 server at IIT: dual Intel Xeon quad-core w/ HT processors, 48 GB RAM. This machine was used to compare NoVoHT, BerkeleyDB and KyotoCabinet.
- **Fusion**: a 48-core x64 server at IIT: quad AMD Opteron 12-core processors, 256GB RAM. This machine was used to compare NoVoHT, BerkeleyDB and KyotoCabinet.

The basic operations that ZHT supports include insert, append, lookup, and remove. On each node, one or more ZHT client-server pairs are deployed, namely ZHT instances. Each instance is configured with one or more partitions known as NoVoHT. Each client creates a long list of key-value pairs; here we set the length of the key to 15 bytes and length of value to 132 bytes. Clients sequentially send all of the key-value pairs through a ZHT Client API for insert, then lookup, and then remove. Append is evaluated separately due to its 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 doesn't have persistent storage for each work node, ramdisks 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 per second/s.
- **Ideal throughput:** Measured throughput between two nodes times the number of nodes.
- **Efficiency:** Ratio between measured throughput and ideal throughput.

B. *NoVoHT Persistencec*

We compared NoVoHT with persistence to KyotoCabinet with identical workloads for 1M, 10M, and 100M inserts, gets, and removes, operating on fixed length key value pairs. The results (see Figure 6) show NoVoHT scales nearly perfect in terms of time per operation; experiments not shown in this figure also show that memory overheads follow the same near perfect trends. It is interesting to note that persistency of writing key/value pairs to disk only adds about 3us of latency on top of the in-memory implementation.

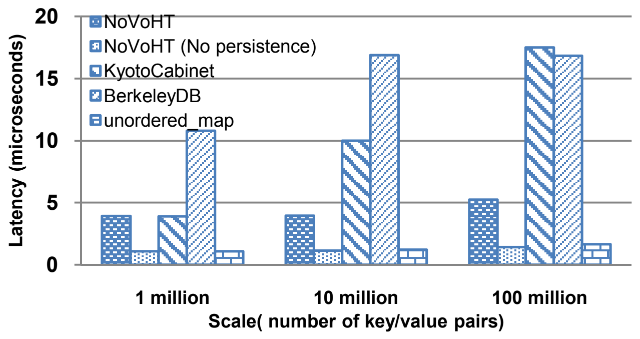

Figure 6: Performance evaluation of NoVoHT, KyotoCabinet and BerkeleyDB plotting latency vs. scale (1M to 100 million key/value pairs) on Fusion

When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we see much better scalability properties for NoVoHT. Although BerkeleyDB has some advantages such as memory usage (not shown in the figure), it does this at the cost of performance.

C. *Latencies*

We evaluated the latency metric on both the Blue Gene/P and HEC-Cluster testbeds. We evaluated several communication variations, such as UDP/IP, TCP/IP without connection caching, TCP/IP with connection caching, and compare them with Memcached and Cassandra.

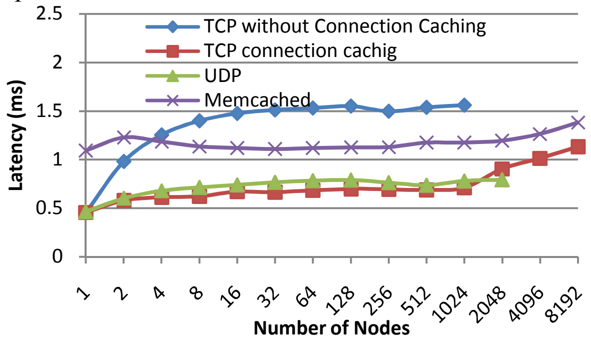

Figure 7: Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 8K nodes on the Blue Gene/P)

At 8K-node scale, ZHT shows great scalability. As shown in the , 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.

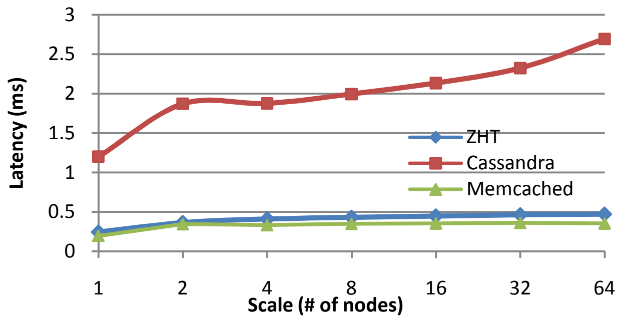

Figure 8: Performance evaluation of ZHT and Memcached plotting latency vs. scale (1 to 64 nodes on the HEC-Cluster)

Because of Cassandra's implementation in Java, and the lack of support for Java on the Blue Gene/P, we evaluated Cassandra, Memcached, and ZHT on the HEC-Cluster (a traditional Linux cluster). Not surprisingly, as shown in Figure 8, ZHT has much 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 Memcached's data stayed completely in-memory.

D. *Throughput*

We conducted several experiments to measure the throughput (see Figure 9). The throughputs 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.

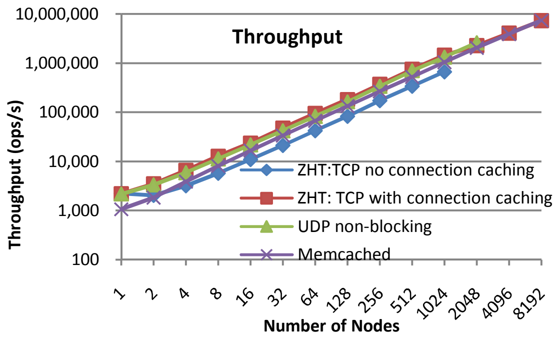

Figure 9: Performance evaluation of ZHT and Memcached plotting throughput vs. scale (1 to 8K nodes on the BLUE GENE/P)

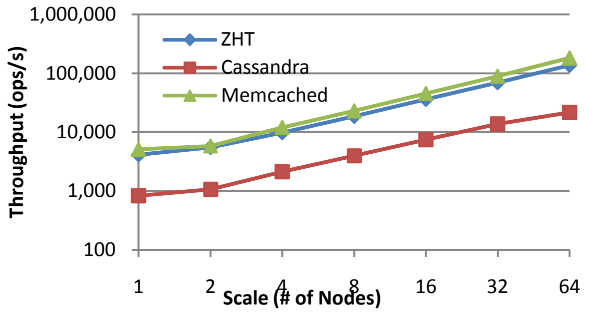

Figure 10: Performance evaluation of ZHT, Memcached and Casandra plotting throughput vs. scale (1 to 64 nodes on the HEC-Cluster)

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 scales grows. Figure 10 shows the nearly 7x throughput difference between ZHT and Cassandra. Memcached performed as expected better as well, with a similar 27% higher overall throughput.

E. *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 involving the network) of the best performance system. Efficiency is simply the measured throughput divided by the ideal throughput. In Figure 11, we show that ZHT and Memcached achieve different levels of efficiency (51%~100% for ZHT and 42%~53% for Memcached) up to 8K-node scales. Memcached's worse efficiency is attributed to having lower performance (higher latency) overall. 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 – 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/IP); we will investigate if MPI has the same performance characteristic.

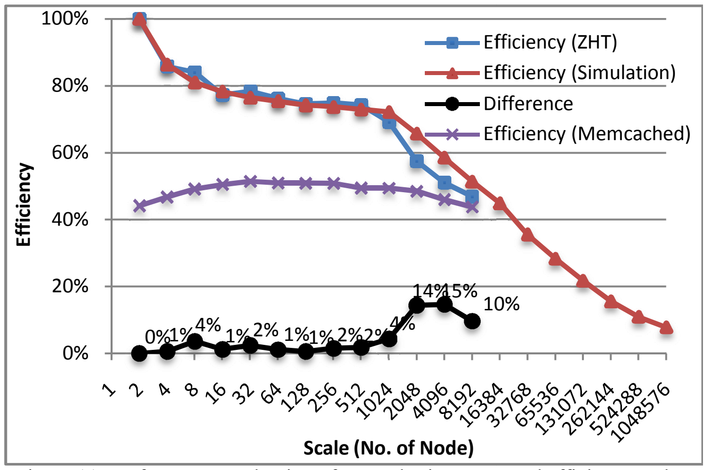

Figure 11: Performance evaluation of ZHT plotting measured efficiency and simulated efficiency vs. scale (1 to 8K nodes on the Blue Gene/P and 1 to 1M nodes on PeerSim)

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 decided to investigate the performance of ZHT in simulations, at larger scales. We have simulated ZHT on a PeerSim-based [54] simulator [51]. 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 (this is the performance of ZHT at 2 node scales). 51% efficiency implies about 1.1ms latency (this is the performance of ZHT at 8K-node scales). 8% efficiency implies about 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.

F. *Replication*

Because of the importance of fault tolerance, ZHT uses replication mechanism. It will certainly introduce some overhead. As shown in Figure 12, replication does increase the operation latency, but it is not a significant increase. One replica adds around 20% and 2 replicas add around 30% overhead compared with the latency of no replica. It is worth noting that the choice of replicating asynchronously likely helped keeping the overheads low. If replication would have been synchronous, the cost of each replica would have likely been 100% increment for 1 replica, and 200% for 2 replicas.

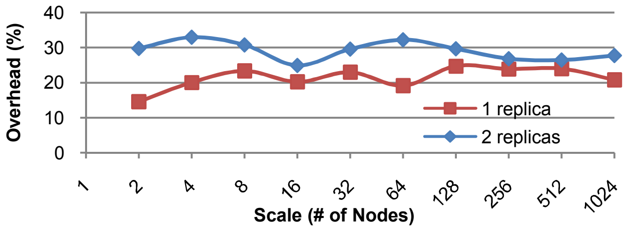

Figure 12: Performance evaluation of ZHT with different levels of replication plotting overheads vs. scale (2 to 1K nodes on the Blue Gene/P)

G. *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).

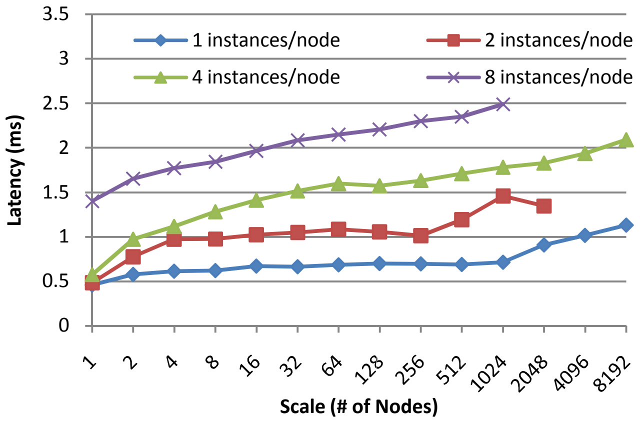

Figure 13: Performance evaluation of ZHT with different numbers of instances per node plotting latency vs. scale (1 to 8K nodes on the BLUE GENEBLUE GENE/P)

Not surprisingly, more instances will increase latency (shown in Figure 13), but the aggregated throughput implies that it's useful to continue to increase the concurrent instances to even more than one instance per core. Due to the fact that we were able to run ZHT with 32K instances on 32K-cores with good performance, leading us to believe that ZHT will scale even better to 32K physical nodes (with 32K instances). The full BLUE GENE/P machine is 40K-nodes, certainly within reach for the ZHT system, but lack of time allocation prevented us from running larger scale experiments. We plan to do full scale BLUE GENE/P experiments in future work.

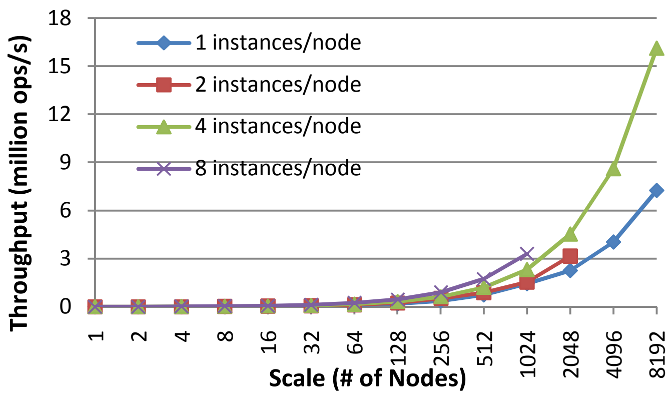

Figure 14: Performance evaluation of ZHT with different numbers of instances per node plotting throughput vs. scale (1 to 8K nodes on the BLUE GENE/P)

H. Overhead of Dynamic Membership

We performed experiments to evaluate the functionality and overhead of dynamical membership, and the cost of nodes joining dynamically to the system, on up to 32 nodes. We set up a benchmark that first starts 32 clients (spread over 32 nodes), and one ZHT server on a single node. While clients are active in performing operations to the ZHT server, we double the number of servers and measure the time to complete the resource increase operation.

Figure 15 shows the time spent on doubling number of servers. Up to 32 nodes, the trends seem relatively constant (with costs around 2 seconds) implying good scalability. We will conduct larger scale dynamic membership experiments in future work.

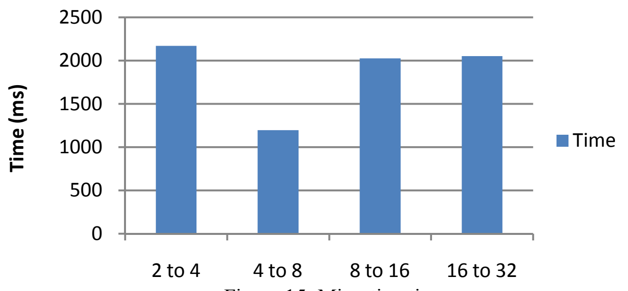

Figure 15: Migration time

V. A Building Block for Distributed Systems

This section presents some real systems that have adopted ZHT as a building block to build a large-scale distributed system.

A. *FusionFS: Distributed Metadata Management*

We have an ongoing project to develop a new highly scalable distributed file system, called FusionFS [13]. FusionFS is optimized for a subset of HPC and many-task computing (MTC) [12, 59, 62, 63] workloads, and it is designed for extreme scales [61]. These workloads are often extremely data-intensive [56, 58, 60], and optimizing data locality [55] becomes critical to achieving good scalability and performance. 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 kernel module to deliver a POSIX compatible interface as a user space filesystem.

In order to measure the metadata performance of FusionFS (which in turn is based on ZHT), we built a benchmark that creates 10K files per node, across *N* directories, where *N* was equal to the number of nodes, ranging from 64 to 512. In the case of FusionFS, it could use the simple insert/lookup API of ZHT, as every node/client could modify metadata information of different directories. We compared the performance of metadata management of FusionFS with that of GPFS which is commonly deployed in production large-scale HEC systems.

As shown in Figure 16 on Blue Gene/P, at 512-node scale (1 process per node), FusionFS has nearly two orders of magnitude higher performance over GPFS (8ms vs. 393ms for GPFS). The gap between GPFS and FusionFS metadata access cost will continue to grow as 8 nodes were enough to saturate the metadata servers of GPFS, but ZHT achieved excellent scalability up to 8K-nodes (as we discussed in Section IV). FusionFS shows excellent scalability (increasing 2X from 4.5ms to 8ms, from 1 node to 512 nodes) while GPFS latency grows 78X from 5ms to 393ms.

Figure 1: Time per operation (touch) on GPFS on various number of processors on a IBM Blue Gene/P shows the performance of concurrently creating many files in the same directory (on GPFS) performs even worse, with 2449ms at 512-node scales. FusionFS adopted the append operation in order to allow concurrent metadata modification without needing locks. In micro-benchmarks, the append operation is at least as fast as inserts, if not faster, even under concurrent appends to the same key/value pair. We expect the performance of FusionFS to be similar for the concurrent file creates in a single directory to those results of creating files across many directories.

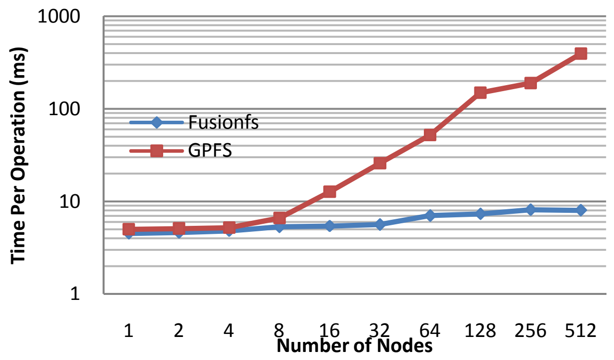

Figure 16: FusionFS vs GPFS (time per operation)

B. *IStore*

Large-scale storage systems require fault-tolerance mechanisms to handle failures, which are a norm rather than an exception. To deal with this, a new trend other than replication, includes the information dispersal algorithms [47, 48]. By implementing erasure coding, these algorithms encode the data into multiple blocks among which only a portion is necessary to recover the original data. IStore is a simple yet high-performance Information Dispersed Storage System that makes use of erasure coding and distributed metadata management with ZHT [50]. Figure 17 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. The IStore uses ZHT to manage metadata about file chunks. 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. The smaller the files, the more metadata intensive IStore became, requiring as many as 500 metadata operations per second at 32 node scales.

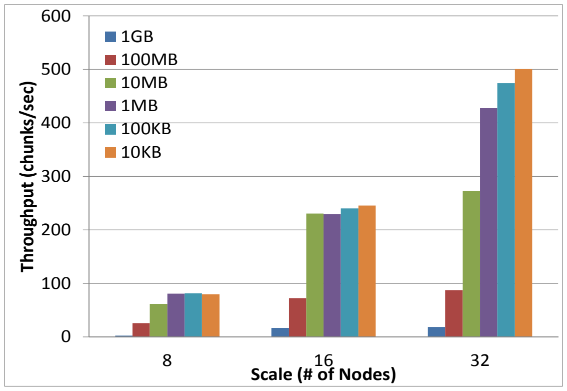

Figure 17: IStore metadata performance on HEC-Cluster

C. *MATRIX*

MATRIX is a distributed many-task computing [12] execution framework, which utilizes the adaptive work stealing algorithm to achieve distributed load balancing [51], and ZHT to submit tasks and monitor the task execution progress by the clients. We have a functioning prototype implemented in C, and have scaled this prototype on a BLUE GENE/P supercomputer up to 512 nodes (2K-cores) with good results. 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 [53] light-weight task execution framework (see Figure 18). The workload consisted of 100K tasks of various lengths, ranging from 0 seconds (NO-OP) to 1, 2, 4, and 8 seconds. It might be difficult to compare MATRIX with Falkon running on the SiCortex or the Linux Cluster, as MATRIX was run on the BLUE GENE/P. However, when comparing MATRIX with Falkon on the BLUE GENE/P for peak throughput, we see Falkon saturate at 1700 tasks/sec at 256-core scales (similarly as all the other test beds also saturate eventually). Falkon has a centralized architecture, and hence had limited scalability. MATRIX shows excellent growth in throughput, starting with 1100 tasks/sec at 256-core scales, up to almost 4900 tasks/sec at 2048-core scales. What is even more important is that there was no obvious sign of saturation, and the growth tracked well the increase in ZHT performance.

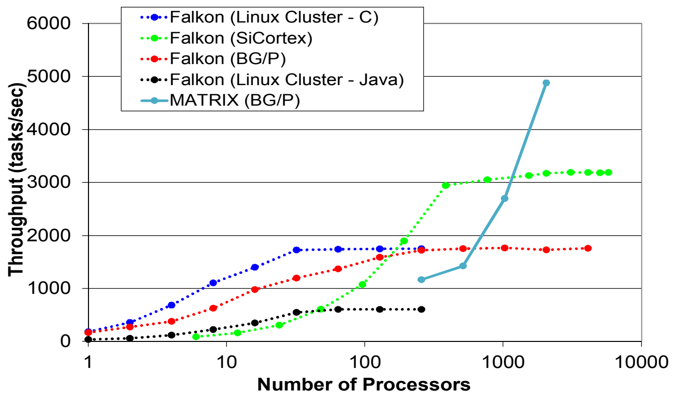

Figure 18: Comparison of MATRIX with Falkon [5, 53]

Figure 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 naïve hierarchical distribution of tasks) [5]. 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%.

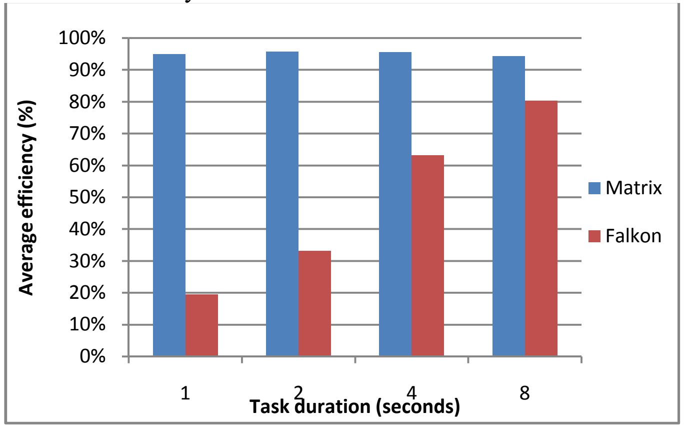

Figure 19: Comparison of MATRIX and Falkon average efficiency (between 256 and 2048 cores) of 100K sleep tasks of different granularity (1 to 8 seconds)

VI. FUTURE WORK

We have many ideas on how to improve ZHT. There are also many possible use cases where ZHT could make a significant contribution in performance or scalability.

Network-aware topology: Given the popularity of multi-dimensional Torus networks on HEC systems, we believe that making ZHT network topology aware is critical to making ZHT scalable by ensuring that communication is kept localized when performing 1-to-1 communication.

Broadcast primitive: We believe that a broadcast primitive (in addition to insert/lookup/remove/append) would be beneficial to transmit the key/value pairs efficiently to all nodes (potentially via a spanning tree).

Data Indexing: We will explore the possibility of using ZHT to index data (not just metadata) based on its content. For this indexing to be successful, some domain specific knowledge regarding the data to be indexed will be necessary.

MosaStore: MosaStore [30] is an experimental storage system under development at the University of British Columbia. MosaStore has a centralized manager to handle metadata, just like most other filesystems available. ZHT will be used to implement a distributed metadata manager for MosaStore.

Swift: Swift [27, 57] is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows on clusters, grids, supercomputers, and clouds [64]. It supports applications that execute many tasks coupled by disk-resident datasets. We will work with the Swift team to integrate ZHT into Swift in order to achieve scalable data management.

VII. 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 64-node cluster, and an IBM Blue Gene/P supercomputer. 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 much 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.

ACKNOWLEDGMENT

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.

REFERENCES

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

[2] F. Schmuck, R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," FAST 2002

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, R. Thakur. "PVFS: A parallel file system for linux clusters", Proceedings of the 4th Annual Linux Showcase and Conference, 2000

[4] P. Schwan. "Lustre: Building a file system for 1000-node clusters," Proc. of the 2003 Linux Symposium, 2003

[5] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Toward Loosely Coupled Programming on Petascale Systems," IEEE SC 2008

[6] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, I. Foster, M. Wilde. "Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming", IEEE MTAGS08, 2008

[7] S. Ghemawat, H. Gobioff, S.T. Leung. "The Google file system," 19th ACM SOSP, 2003

[8] A. Bialecki, M. Cafarella, D. Cutting, O. O'Malley. "Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware", 2005

[9] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. "Looking up data in P2P systems", Communications of the ACM, 46(2):43–48, 2003

[10] "Filesystem in Userspace", http://fuse.sourceforge.net/, 2011

[11] W. Vogels, "Eventually consistent," ACM Queue, 2008.

[12] I. Raicu, I. Foster, Y. Zhao. "Many-Task Computing for Grids and Supercomputers", IEEE MTAGS08, 2008

[13] D. Zhao, C. Shou, X. Zhou, T. Li, Z. Zhang, I. Raicu, "FusionFS: a distributed filesystem for extreme scaledata-intensive computing", Under MSST13 review, 2013.

[14] T. Li, R. Verma, X. Duan, H. Jin, I. Raicu, "ZHT: Zero-Hop Distributed Hash Table for High-End Computing", ACM Performance Evaluation Review (PER), 2012

[15] P. Maymounkov, D. Mazieres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", In Proceedings of IPTPS, 2002

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, "A scalable content-addressable network," in Proceedings of SIGCOMM, pp. 161–172, 2001

[17] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", ACM SIGCOMM, pp. 149-160, 2001

[18] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in Proceedings of Middleware, pp. 329–350, 2001

[19] 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, VOL. 22, NO. 1, 2004

[20] B. Fitzpatrick. "Distributed caching with Memcached." Linux Journal, 2004(124):5, 2004

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store." SIGOPS Operating Systems Review, 2007

[22] H. Shen, C. Xu, and G. Chen. Cycloid: A Scalable Constant-Degree P2P Overlay Network. Performance Evaluation, 63(3):195-216, 2006

[23] Ketama, http://www.audioscrobbler.net/development/ketama/, 2011

[24] Riak, https://wiki.basho.com/display/RIAK/Riak, 2011

[25] Maidsafe-DHT, http://code.google.com/p/maidsafe-dht/, 2011

[26] J.M. Wozniak, B. Jacobs, R. Latham, S. Lang, S.W. Son, and R. Ross. "C-MPI: A DHT implementation for grid and HPC environments", Preprint ANL/MCS-P1746-0410, 2010

[27] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, I. Foster. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", SciDAC09, 2009

[28] C. Docan, M. Parashar, S. Klasky. "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows", ACM HPDC 2010

[29] MPI Forum, "Mpi 3.0 standardization effort," http://meetings.mpi-forum.org/MPI_3.0_main_page.php, 2012

[30] S. Al-Kiswany, A. Gharaibeh, M. Ripeanu. "The Case for a Versatile Storage System", Workshop on Hot Topics in Storage and File Systems (HotStorage'09), 2009

[31] A. Petitet. "Block Cyclic Data Distribution". http://www.netlib.org/utk/papers/scalapack/node8.html, 2009

[32] B. Mulvey. "Hash Functions". http://bretm.home.comcast.net/~bretm/hash/, 2009

[33] S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk, "Cryptographic Hash Functions: A Survey". 1995

[34] J. Eisner. "State-of-the-art algorithms for minimum spanning trees: A tutorial discussion", University of Pennsylvania, 1997

[35] D. Karger, P.N. Klein, R.E. Tarjan. "A randomized linear-time algorithm to find minimum spanning trees", J. ACM, vol. 42, pp. 321–328, 1995

[36] M. Fredman, D. E.Willard. "Trans-dichotomous algorithms for minimum spanning trees and shortest paths". Proc. 31st IEEE Symp. Foundations of Comp. Sci., pp. 719–725, 1990

[37] Google Protocol Buffers: http://code.google.com/apis/protocolbuffers/, 2012

[38] Cassandra http://cassandra.apache.org/, 2012

[39] Kyotocabinet http://fallabs.com/kyotocabinet/, 2012

[40] Blue Gene supercomputer

http://en.wikipedia.org/wiki/Blue_Gene, 2012

[41] P. H. Carns, W. B. Ligon III, R. Ross, P. Wyckoff. "BMI: a network abstraction layer for parallel I/O". In Proceedings of IPDPS'05, CAC workshop, 2005

[42] MATRIX http://datasys.cs.iit.edu/projects/MATRIX/index.html, 2012

[43] S. Alam, R. Barrett, M. Bast. Early evaluation of IBM Blue Gene/P,

SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing.

[44] ALCF, Argonne Leadership Computing Facility, https://www.alcf.anl.gov

[45] ZHT source code. https://github.com/mierl/ZHT

[46] K. Batcher, http://en.wikipedia.org/wiki/Ken_Batcher, 2012

[47] A. J. McAuley, "Reliable broadband Communication using a burst erasure correcting code", in SIGCOMM '90 Proceedings of the ACM symposium on Communications architectures & protocols, 1990

[48] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols", in ACM SIGCOMM Computer Communication Review, Volume 27 Issue 2, Apr. 1997.

[49] K. Brandstatter, T. Li, X. Zhou, I. Raicu. "NoVoHT: a Lightweight Dynamic Persistent NoSQL Key/Value Store", Under MSST13 review, 2013

[50] C. Debains, P.M.A. Togores, I. Raicu. "Evaluating Information Dispersal Algorithms", 1st Greater Chicago Area System Research Workshop, 2012

[51] K. Wang, A. Rajendran, I. Raicu. "Extreme Scale Distributed Load-Balancing with Adaptive Work Stealing", under review at HPDC 2013

[52] K. Wang, A. Kulkarni, M. Lang, I. Raicu. "Exploring Design Tradeoffs for Exascale System Services through Simulation", under review at HPDC 2013

[53] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, M. Wilde. "Falkon: a Fast and Light-weight tasK executiON framework", IEEE/ACM SuperComputing/SC, 2007

[54] A. Montresor, M. Jelasity. "PeerSim: A scalable P2P simulator". In Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), 2009.

[55] A. Szalay, J. Bunn, J. Gray, I. Foster, I. Raicu. "The Importance of Data Locality in Distributed Computing Applications", NSF Workflow Workshop 2006

[56] I. Raicu, I. Foster, Y. Zhao, P. Little, C. Moretti, A. Chaudhary, D. Thain. "The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems", ACM International Symposium on High Performance Distributed Computing (HPDC09), 2009

[57] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008

[58] I. Raicu, I. Foster, A. Szalay, G. Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis", TeraGrid Conference 2006, June 2006

[59] I. Raicu, I. Foster, M. Wilde, Z. Zhang, A. Szalay, K. Iskra, P. Beckman, Y. Zhao, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, D. Thain. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[60] I. Raicu, I. Foster, A. Szalay. "Harnessing Grid Resources to Enable the Dynamic Analysis of Large Astronomy Datasets", IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC06), 2006

[61] I. Raicu, P. Beckman, I. Foster. "Making a Case for Distributed File Systems at Exascale", Invited Paper, ACM Workshop on Large-scale System and Application Performance (LSAP), 2011

[62] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C.M. Moretti, A. Chaudhary, D. Thain. "Towards Data Intensive Many-Task Computing", book chapter in Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2011

[63] I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Dept., University of Chicago, Doctorate Dissertation, March 2009

[64] Y. Zhao, I. Raicu, S. Lu, X. Fei. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE International Conference on Network-based Distributed Computing and Knowledge Discovery (CyberC) 2011

[65] C. Debains, P. Alvarez-Tabio, D. Zhao, Ioan Raicu. "IStore: Towards High Efficiency, Performance, and Reliability in Distributed Data Storage with Information Dispersal Algorithms", under review at IEEE MSST 2013