NoVoHT: a Lightweight Dynamic Persistent NoSQL Key/Value Store

Kevin Brandstatter[1], Tonglin Li[1], Xiaobing Zhou[1], Ioan Raicu[1,2]
kbrandst@iit.edu, tli13@iit.edu, xzhou40@hawk.iit.edu, iraicu@cs.iit.edu

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

Abstract—**With the increased scale of systems in use and the need to quickly store and retrieve information, key/value stores are becoming an important element in the design of large-scale storage systems. Key/value stores are well known for their simplistic interfaces, persistent nature, and excellent operational efficiency – they are also known as NoSQL databases. This paper presents the design and implementation of a non-volatile hash table (NoVoHT). NoVoHT was designed from the ground up to be lightweight, fast, and dependency-free. Our goal was to create a fast persistent key/value store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers. We also aimed to develop a system that performed as close as possible to an in-memory hash map, but with the added benefit of being persistent. We also extended the traditional key/value store interface (e.g. insert, lookup, remove) to include a novel operation (e.g. append) that has allowed NoVoHT to efficiently support lock-free concurrent write operations. NoVoHT is also dynamic, supporting live migration across node boundaries. We have run comparisons at significant scales against some of the more commonly used key value stores and have shown that NoVoHT can perform similarly or better than other systems such as Kyoto Cabinet, and BerkeleyDB. We observed up to 165K+ operations per second, up to 32X better performance than competing systems. We have evaluated NoVoHT with solid state disks (SSD) and have deployed NoVoHT as the persistent back-end of a distributed hash table (ZHT) on an IBM BlueGene/P supercomputer at up to 32K-cores.**

I. Introduction

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 [15], PVFS [16], and Lustre [17]). 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 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 (e.g. GPFS [15]) can be inefficient at large scale. Experiments on the BlueGene/P system at 16K-core scales showed 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. 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 4~32 core scales (on a 160K-core machine).

Other distributed file systems (e.g. Google's GFS [18] and Yahoo's HDFS [19]) have centralized metadata management, making the problems observed with GPFS even worse from the scalability perspective. We believe future storage systems for high-end computing should support distributed metadata management, in order to support close to constant time inserts/lookups/removes.

In previous work, we developed a zero-hop distributed hash table (ZHT) [9], which has been tuned for the specific requirements of high-end computing. ZHT aims to be a building block for future distributed file systems, with the goal of delivering excellent availability, fault tolerance, high throughput, scalability, persistence, and low latencies. ZHT has three important characteristics that are relevant to this paper, ***a customizable consistent hashing function, it supports persistence for better recoverability in case of faults, and supports unconventional operations such as append*** (providing lock-free concurrent key/value modifications) in addition to insert/lookup/remove.

We achieved these features in ZHT through the design and implementation of NoVoHT, the single node persistent NoSQL Key/Value store. NoVoHT was designed from the ground up to be lightweight, fast, and dependency-free. Our goal was to create a fast persistent key/value store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers. We also aimed to develop a system that performed similar to an in-memory hash map, but with the benefit of being persistent. Furthermore, we extended the traditional key/value store interface (e.g. insert, lookup, remove) to include a novel operation (e.g. append) that has allowed NoVoHT to efficiently support lock-free concurrent write operations. Finally, we have made NoVoHT dynamic supporting live migration across node boundaries, allowing the ZHT system to significantly simplify the needed logic for dynamic membership.

We have run comparisons at significant scales against some of the more commonly used key value stores and have shown that NoVoHT can perform similarly or better than other systems such as Kyoto Cabinet and BerkeleyDB. We observed up to a 32X performance improvement compared to Kyoto Cabinet at large scale, and on average a 4X performance improvement over both Kyoto Cabinet and BerkeleyDB on a variety of scales. We have evaluated NoVoHT with solid state disks (SSD) and have deployed NoVoHT as the persistent back-end of a distributed hash table (ZHT) on an IBM

BlueGene/P supercomputer at up to 32K-cores. We compared NoVoHT with other key/value stores and found it offers superior performance, features, and portability.

The contributions of this paper revolve around the design and implementation of NoVoHT, a Lightweight Dynamic Persistent NoSQL Key/Value Store. It's novel features are:

- ***Lock-Free:*** *Support for unconventional operation such as append, allowing the efficiently support of lock-free concurrent modification operations*
- ***Dynamic:*** *Support live migration across physical nodes*
- ***Lightweight:*** *Micro-benchmarks delivering over 165K+ operations/second on single-node deployment ➔ up to 32X faster than existing systems*
- ***Persistent:*** *Combines memory mapped and disk mapped approaches to deliver both fast data access and high data resilience at the same time*
- ***Real-System:*** *Adopted by ZHT and deployed on IBM BlueGene/P supercomputer at 32K-core scales*

II. NoVoHT Design and Implementation

NoVoHT is a persistent in memory hash map designed to be fast and lightweight for running on large systems. It was developed to support the ZHT distributed hash table in a variety of features. The application programming interface (API) of NoVoHT is kept simple and follows similar interfaces for hash tables. The four operations NoVoHT supports are 1. bool *insert*(key, value); 2. value *lookup*(key); 3. bool *remove*(key), and 4. bool *append*(key, value).

NoVoHT is built from scratch and features few dependencies outside of the standard libraries. This makes it easily portable and usable on large supercomputers which often have restricted Linux environments not supporting a large number of libraries and complex dependencies. It features an efficient garbage collection mechanism to keep the disk space utilized to a minimum. It is a basic building block towards ZHT project, as well as the FusionFS filesystem [9].

A. Design Goals

Our goals for the design of NoVoHT were to make a very lightweight key value store. For avoiding reliance on outside dependencies and to maximize support on larger systems with limited storage and libraries, we started with the basic building blocks from scratch.

Secondly, we aimed to increase speed by holding the entire table in memory so there wouldn't be noticeable performance degradation with large data sets. To do this, we relied on the traditional style of a hash map and worked to make the persistence layer as simple as possible. We aimed for simplicity to reduce the overhead of the persistence, thus keeping operation latency low.

Finally, we set out to solve the issue of garbage collection to reclaim unused disk space. As a persistent storage system, we expect the life of the data to be quite long, and varied in amount over time. This variance would bloat normal NoSQL systems as their database needs to expand to encapsulate the data. However, should this data be reduced significantly, the disk space is never reclaimed and continues to take up space on the system. The way these systems usually handle shrinking the storage, is to create a new instance and reinsert the data. For many systems this is a poor option. NoVoHT fixes this by implementing a system to transparently rebuild the persistent storage. This way, disk utilization can remain constrained to the size of the data. We have made this tunable.

B. Core Hash Table

NoVoHT is simply a custom built light-weight hash table at the core, with added features built on top. The general design of the map structure is an array of linked lists. This structure makes collision handling more efficient. Rather than searching for an open bucket, it simply adds it to the list structure. Also, this helps lookup time, because it eliminates the worst case of iterating the entire array in the case of it being very full. Finally, this gives the application the ability 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.

C. Resizability

In order for NoVoHT to scale to meet its dynamic demands, the ability to expand its storage size is necessary. Without resizing, over time inserts and removes would be progressively slower as the number of collisions would continue to increase. To resolve this, the constructor offers parameterization of a value that corresponds to a percentage of the size of the storage array. When the number of elements in the map is equal to the size of the map times the percentage, the map calls the re-size routine. The re-size routine creates a new array of twice the size of the old one, then places the address of each pair into the corresponding bucket of the new array. Then it sets the new array as the maps array and frees the old one. Since the base array simply holds the sentinel head of each buckets linked list, this operation is approximately as fast as rehashing each key, and adding the address of the pair to the appropriate list. Of course, if the number of elements to be inserted is known, the user could turn off resizing to ensure this overhead isn't incurred.

D. Persistence

NoVoHT's major feature is its ability to save its state to a POSIX file system, and allow it to recover its state in case of an application crash. NoVoHT employs a log-based approach to achieve persistence. When a key/value pair is inserted, it writes the key-value pair to the file specified, and records where it was written with the key-value pair in the map. By recording the location in the file, removal is extremely 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.

E. Garbage Collection

Through the constant adding and removal of key-value pairs, the file grows and shrinks. In order to handle file shrinking, NoVoHT allows the parameterization of a 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.

Garbage collection occurs by rewriting the entire state of NoVoHT. We chose this method because it allowed us to simplify the file format to optimize inserts as one does not have to keep track of available space or search for a fit -- insertions are then as simple as a write append. As a tradeoff, the garbage collection requires a rewrite to clean out the deleted records.

Normally the garbage collection would require a global lock of the map to prevent changes to the map during the rewrite. We designed a way to handle the majority of the file rewrite without locking access to the table. First, we spawn a background helper thread to manage the actual writing of the file so that it doesn't stall the application waiting for completion. This is especially important for large maps where the rewrite time can be potentially many seconds to minutes. During this time, the hash table writes its activity to a merge file by recording the pairs added, and the keys it deleted. Then, when the write is complete, the map locks insert and remove operations and performs the merge operation by implementing the changes from the merge file ensuring consistency with memory. The get operation needs not be locked because it does not rely on access to the file since it gets the data from memory. We chose this merge approach because we are under the assumption that under our use case in a distributed file system, inserts and removes will be less frequent while gets are common and need to be always accessible. Also, we could have tried to implement a queuing system for operations during the merge phase, however we decided that a more transactional operation model was appropriate and would like to only return when the record is persisted.

F. *Append Function*

Since NoVoHT was designed with the initial intent for file system metadata storage, we found it necessary to add a nonstandard fourth operation, append. This may seem to be a trivial addition; however it was complicated by the persistence layer. In memory it is as simple as appending the new value to the old value. For the persistence layer to do this it would require us to remove the old record and write the entire key and combined value at the end. This yielded degraded performance as subsequent appends would take progressively longer.

We achieved a constant time operation by fragmenting the appended segments on disk. Thus, when a value is appended, the appended value and the key are entered as a new record. This location is also recorded by NoVoHT for removal purposes. Also, these fragments will be merged on the next garbage collection when the entire pair will be rewritten as a single record.

G. *Implementation*

NoVoHT has been under development for 1.5 years, and it is an open source project accessible at [20]. It is implemented in C++, and has very few dependencies. Currently, NoVoHT only relies on a standard C++ compiler such as the one provided by the GNU Compiler Collection, the C++ standard libraries, and posix thread support. These very basic requirements assure that NoVoHT is very light and can be easily deployed in almost any Linux friendly environment. NoVoHT also is tested to work well with strings generated by the Google Protocol Buffers [3] (on the BG/P) to support the storage and retrieval of complex objects, but the Google Protocol Buffer is not necessary if key/value pairs are constrained to string types.

III. PERFORMANCE EVALUATION

All NoVoHT experiments were performed on Fusion.

- **Fusion**: a 48-core x64 server: quad AMD Opteron 12-core processors, 256GB RAM. This machine was used to compare NoVoHT, BerkeleyDB and KyotoCabinet.

A. *Hashing Functions*

In this section we discuss the hash functions that can be used to map keys to nodes. We investigate some of the usual hash functions for figuring out the performance and load balancing abilities. As shown in Figure 1 that some hash functions are faster than others, but a more important concern rather than performance is the evenness (see Figure 2).

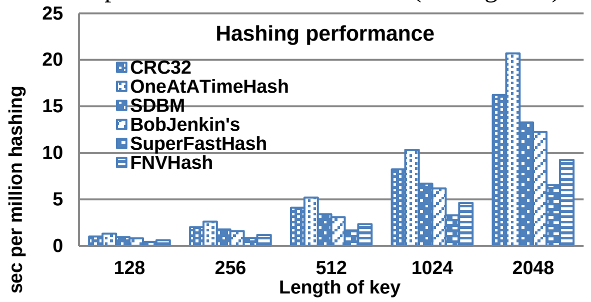

Figure 1: Time per hash for a variety of hashing functions

An ideal hash function should be able to spread keys evenly. Additionally it should not increment the time too rapidly as the length of key increased. We need the hash function to be even to provide a natural load balancing mechanism. But obviously sometimes we can't obtain both performance and evenness, as can be seen by the SuperFast hash unevenness (see Figure 2). We adopted the FNVHash hash function in NoVoHT. As observed in the charts below, FNVHash offers a good performance while providing evenness as well. We measure 1 million keys distribution over 1000 buckets, then find that the standard deviation is only 138 keys (or 1. 38%). This is on par with most of the other hash functions' standard deviation (see Figure 2).

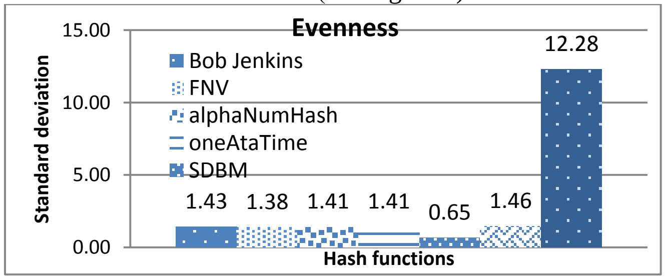

Figure 2: Load balancing across different hashing functions

B. NoVoHT Persistence

We compared NoVoHT with persistence to KyotoCabinet and BerkeleyDB with identical workloads up to 100 million inserts, gets, and removes, with fixed length key value pairs. The results (see Figure 3) show NoVoHT scales near 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.

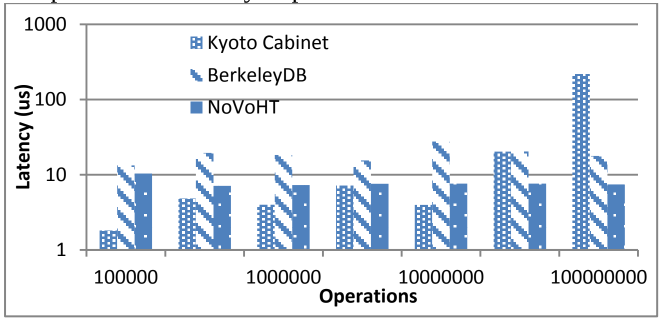

Figure 3: Comparison between NoVoHT, KyotoCabinet, and BerkeleyDB

When comparing NoVoHT with KyotoCabinet or BerkeleyDB, we see much better scalability properties, as they both show a significant increase in the cost per operation as the size increases from 1M to 100M key/value pairs. Also, it's worth pointing out that although BerkeleyDB has some advantages such as memory usage, it clearly does this at a big performance cost. When comparing NoVoHT persistence to non-persistence (see Figure 4, we noticed that most of the overhead of the operations is held in the disk I/O portion.

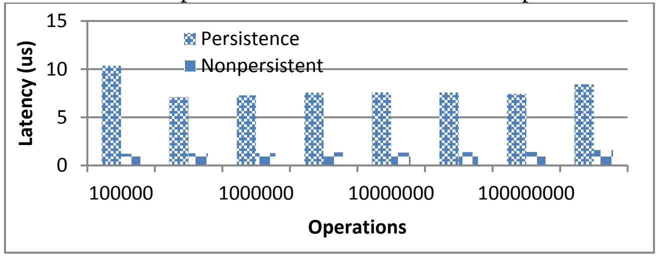

Figure 4: NoVoHT persistence vs. non persistent

Figure **5** shows the performance of NoVoHT with persistence disabled, and compared to unordered_map. It's interesting that NoVoHT was able to outperform the C++ native unordered_map by over 20%.

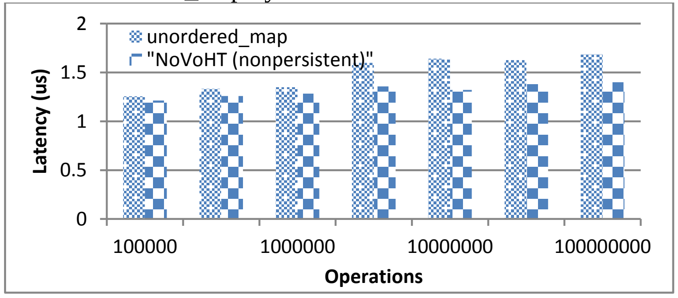

Figure 5: NoVoHT non persistant vs. unordered_map

C. NoVoHT Append

Since most hash tables do not implement and append function, we could not compare our implementation to others. However, we did run several tests to verify that the append function would scale properly and was a constant time operation as intended. To do this we first implemented a naïve implementation that rewrote the new record and removed the old. As expected, this did not scale very well but it provided us with a baseline measurement to judge our optimized version. The results as shown in Figure 6 show that our optimized append operation scales very well and has latency comparable to the other hash table functions.

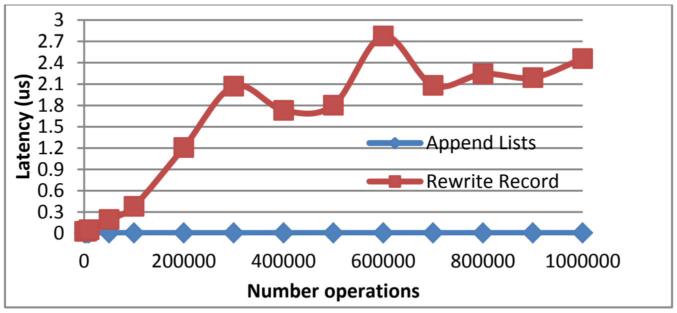

Figure 6 Append method latency comparisons

Comparing the range of operations on each hash table we can confirm that NoVoHT is able to exceed or match its competitors. Thus we have achieved the goal of providing a lightweight replacement without a performance degradation. Also, we can see that this performance is more scalable than the previous use of Kyoto Cabinet. For future tests we would like to try more varied workloads to avoid discrepancies due to specific optimization advancements.

IV. APPLICATIONS

This section presents a real systems that has adopted NoVoHT as a building block to build a large-scale distributed systems. ZHT [9] is a zero-hop distributed hash table, which uses NoVoHT as backend. 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. Using micro-benchmarks, we scaled ZHT up to 32K-cores with latencies of only 1.1ms and 18M operations/sec throughput (see Figure 9).

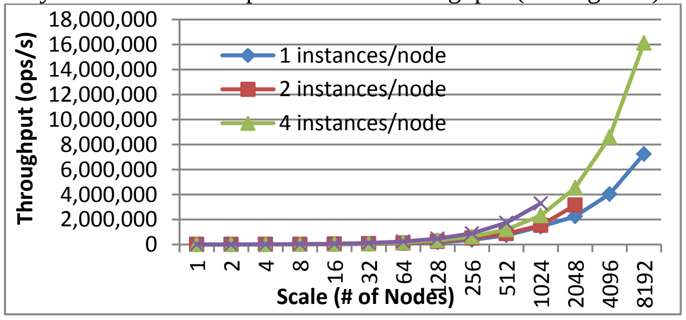

Figure 7: Performance evaluation of ZHT with different numbers of instances per node plotting throughput vs. scale (1 to 8K-nodes on BG/P)

V. Related work

NoSQL Key/Value stores have been around for some time, some dating back to the early 1990s (e.g. BerkeleyDB [13]). Instead of listing dozens of key/value stores, we focus this section on several systems that are most similar, namely BerkelyDB [13], KyotoCabinet [4], and LevelDB [14].

KyotoCabinet is a file based hash database that is optimized for speed. KyotoCabinet is a project by FAL Labs that provides various persistent database APIs and libraries [4]. We particularly explored their HashDB, which implements a hash database backed with file persistence. HashDB works by allocating a file as a database and managing it as it would an in memory hash map with use of mmap to map portions of the file directly to memory[10]. This allows them to have capped memory usage that is independent of the database size. However, since all key/value pairs are mapped directly to a file, garbage collection (the reclaiming of unused space in files) is not trivial; our experience with HashDB was that garbage collection was non-functional. Furthermore, since HashDB is file-based, it has to map the file to memory in order to perform retrieval functions. This is a significantly slower system call. At small scales, it is not very noticeable as it doesn't always have to remap the file, but as the database increases in size, the odds of successive queries being in the same region becomes very small, and the file must be constantly remapped. The constant remapping causes KyotoCabinet to scale significantly worse than NoVoHT.

BerkeleyDB is also a file based database that can either be managed as a hash database or a binary tree database. It is often very popular for storing file system metadata and as such was a good second model of performance we aimed to surpass. BerkeleyDB [13] is a persistent NoSQL database maintained by Oracle. It was designed to be very robust and able to recover from most failures using features such as write ahead logging to ensure data consistency [12]. It has four main methods of access, B-tree, Hash, recno, and queue, though we are mainly concerned with the B-tree and hash methods. While B-tree provides faster access due to locality of reference, for large data sets the index does not fit in memory and thus data access is not optimal [12]. BerkeleyDB, like KyotoCabinet, uses a memory buffer to store pages of the file in memory. This renders it vulnerable to the same scaling issues. The strong point of BerkeleyDB is its recovery functions and emphasis on fault tolerance for data integrity. However, this makes it slower and heavier than other option.

LevelDB [14] is a lightweight key value storage system written at Google. It features sorted data and data compression.

None of these other systems support the wide range of unique features of NoVoHT, such as support for unconventional operation (e.g. append) allowing the efficiently support of lock-free concurrent modification operations, support for live migration across physical nodes, support for both memory mapped and disk mapped approaches to deliver both fast data access and high data resilience at the same time, up to 32X better performance than existing systems.

VI. Conclusions

. This paper presented the design and implementation of a non-volatile hash table (NoVoHT). NoVoHT was designed from the ground up to be lightweight, fast, and dependency-free. Our goal was to create a fast persistent key/value store that could be easily integrated and operated in lightweight Linux OS typically found on today's supercomputers. We also aimed to develop a system that performed as close as possible to an in-memory hash map, but with the added benefit of being persistent. We also extended the traditional key/value store interface (e.g. insert, lookup, remove) to include a novel operation (e.g. append) that has allowed NoVoHT to efficiently support lock-free concurrent write operations. NoVoHT is also dynamic, supporting live migration across node boundaries. We have run comparisons at significant scales against some of the more commonly used key value stores and have shown that NoVoHT can perform similarly or better than other systems such as Kyoto Cabinet, and BerkeleyDB. We observed up to 165K+ operations per second, up to 32X better performance than competing systems. We have evaluated NoVoHT with solid state disks (SSD) and have deployed NoVoHT as the persistent back-end of a distributed hash table (ZHT) on an IBM BlueGene/P supercomputer at up to 32K-cores.

References

[1] FusionFS: Fusion distributed File System, http://datasys.cs.iit.edu/projects/FusionFS/index.html, 2013

[2] ZHT: Zero-Hop Distributed Hash Table for High-End Computing, Tonglin Li, Raman Verma, Xi Duan, Hui Jin, Ioan Raicu., ACM Performance Evaluation Review (PER), 2012

[3] Google Protocol Buffers: http://code.google.com/apis/protocolbuffers/, 2013

[4] Kyotocabinet http://fallabs.com/kyotocabinet/, 2013

[5] BlueGene supercomputer http://en.wikipedia.org/wiki/Blue_Gene, 2013

[6] MATRIX http://datasys.cs.iit.edu/projects/MATRIX/index.html, 2013

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

[8] ZHT source code. https://github.com/mierl/ZHT, 2013

[9] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table", IEEE IPDPS, 2013

[10] KyotoCabinet Specifications http://fallabs.com/kyotocabinet/spex.html, 2013

[11] KyotoCabinet HashDB http://fallabs.com/kyotocabinet/api/classkyotocabinet_1_1HashDB.html, 2013

[12] Margo Seltzer, Keith Bostic. "Berkeley DB", http://www.aosabook.org/en/bdb.html, 2013

[13] BerkeleyDB, http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html, 2013

[14] LevelDB, https://code.google.com/p/leveldb/, 2013

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

[16] 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

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

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

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

[20] NoVoHT GitHub Source, https://github.com/kev40293/NoVoHT/, 2013