HydraDB: A Resilient RDMA-driven Key-Value Middleware for In-Memory Cluster Computing

Yandong Wang† Li Zhang† Jian Tan Min Li
Yuqing Gao* Xavier Guerin* Xiaoqiao Meng* Shicong Meng*

IBM Thomas J. Watson Research Center, New York, USA

ABSTRACT

In this paper, we describe our experiences and lessons learned from building a general-purpose in-memory key-value middleware, called HydraDB. HydraDB synthesizes a collection of state-of-the-art techniques, including continuous fault-tolerance, Remote Direct Memory Access (RDMA), as well as awareness for multicore systems, *etc,* to deliver a high-throughput, low-latency access service in a reliable manner for cluster computing applications.

The uniqueness of HydraDB mainly lies in its design commitment to fully exploit the RDMA protocol to comprehensively optimize various aspects of a general-purpose key-value store, including latency-critical operations, read enhancement, and data replications for high-availability service, *etc*. At the same time, HydraDB strives to efficiently utilize multicore systems to prevent data manipulation on the servers from curbing the potential of RDMA.

Many teams in our organization have adopted HydraDB to improve the execution of their cluster computing frameworks, including Hadoop, Spark, Sensemaking analytics, and Call Record Processing. In addition, our performance evaluation with a variety of YCSB workloads also shows that HydraDB can substantially outperform several existing in-memory key-value stores by an order of magnitude. Our detailed performance evaluation further corroborates our design choices.

1. INTRODUCTION

Aggregating spare DRAM of a cluster of machines and serving them as a storage system through key-value abstraction has been proven as an effective approach to provision high-performance data access service [24, 28]. Unfortunately, remote data access over traditional commodity networking technology can incur latency that exceeds 100μs, three orders of magnitude higher than local memory access (100ns), therefore substantially compromising the full potential of in-memory key-value stores.

Recognizing the issue above, system researchers have been seeking high-performance networking technologies, *e.g.,* InfiniBand and Converged Ethernet, to improve the access to remote data. However, many previous studies [23, 16, 17] have unveiled that simply running existing TCP/IP-based in-memory key-value stores on top of fast networks is unable to efficiently exploit the performance available in the infrastructures. As a result, leveraging RDMA (Remote Direct Memory Access) protocol commonly offered by fast interconnects has obtained great attention over the past few years due to its high-bandwidth (56Gbps), low-latency (1$\sim$3μs), and one-sided zero CPU consumption capability. For example, Pilaf [23] has adopted RDMA Read to optimize the GET operations for Memcached [4]. HERD [18] has systematically analyzed the *pros and cons* of different RDMA primitives and proposed that combining RDMA Write with Send over Unreliable Datagram (UD) can be an appealing approach to build performance-critical operations for key-value systems. However, they all lack a holistic design to comprehensively imbue RDMA into different aspects of a general-purpose key-value store that can enhance the execution of cluster computing frameworks.

In this paper, we present HydraDB, a new in-memory key-value middleware that aims to exploit RDMA protocol to comprehensively optimize a general-purpose key-value system from multiple dimensions. HydraDB is multicore-friendly, resilient, and versatile, allowing users to configure it either as a cache or a reliable storage system to serve different purposes of their cluster computing applications.

To optimize performance, HydraDB has taken full advantages of both reliable RDMA Write and Read to efficiently strengthen widely used key-value operations and data replication. As testaments to such optimization, common latency-critical operations, *e.g.,* GET and PUT, obtain substantial latency reduction and the entire system achieves significantly higher throughput compared to existing in-memory key-value stores on the market. Both improvements sustain at an extent of one order of magnitude. Meanwhile, the design of HydraDB is friendly to multicore systems, taking account of the contention among CPU cores, NUMA-awareness and cache capacity to avoid curbing the performance of RDMA due to imprudent internal data manipulations on the server side.

With increasing maturity, many teams in our organization have adopted HydraDB to enhance the execution of their analytic applications. Examples include Apache Hadoop, Spark, Call Record Processing and IBM G2 Sensemaking analytics. Integrated performance results show that HydraDB is effective in accelerating those applications through removing the conventional I/O bottlenecks. In addition, we have also conducted a detailed evaluation of HydraDB over an InfiniBand cluster by using a variety of YCSB workloads. Performance results demonstrate that HydraDB can provide high-throughput, low-latency service to commonly observed workloads. Our efficient usage of RDMA Write for data replication is capable

†Contact: {yandong, zhangli}@us.ibm.com
*All authors contributed to this work while employed at IBM.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807614

of reducing the overhead incurred by offering high-availability.

The remainder of this paper is organized as follows; we begin with a description on how HydraDB has been leveraged by different cluster computing applications in Section 2. Section 3 discusses related research. Then we detail the principles that guide our design of HydraDB in Section 4 and its high-availability design in Section 5. Section 6 presents the performance evaluation. Finally, we conclude the paper in Section 7.

2. HYDRADB APPLICATIONS

HydraDB has demonstrated its efficacy and efficiency empirically in accelerating many cluster computing frameworks, including Apache Hadoop, Spark, and several IBM analytics systems. In this section, we briefly describe how HydraDB has been used by 4 representative applications to enhance their execution and highlight the performance advantages.

2.1 MapReduce Acceleration

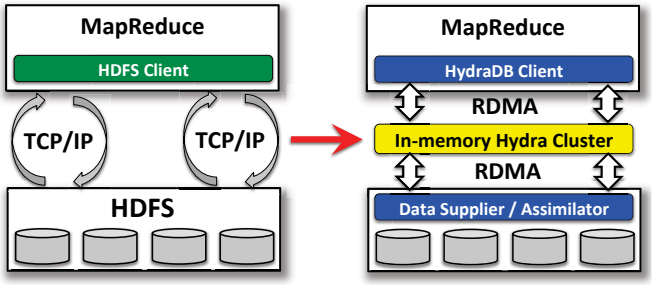


Figure 1: HydraDB has been used as a cache layer on top of HDFS to help mitigate the I/O bottleneck and exploit the benefits provided by RDMA-capable networks.

MapReduce programming model has been broadly adopted to analyze large-scale datasets. Apache Hadoop [1] and Spark [2] are two notable implementations that deliver scalable and reliable job execution. In addition, Spark also offers in-memory data processing for iterative algorithms and interactive queries. However, the I/O communication with the underlying disk-oriented storage systems, *e.g.,* HDFS or HBase, is still a major bottleneck that prevents fast analytics from fully performing in memory speed as revealed by many previous studies [32, 30, 33]. Also, existing frameworks within the MapReduce processing stacks completely rely on the TCP/IP protocol to transfer data through the stacks as shown in Figure 1. As a result, when deployed on clusters equipped with high-performance networks, *e.g.,* 10 Gigbit Ethernet or InfiniBand, such a mechanism is unable to efficiently exploit the performance benefits available in the infrastructures.

In light of the above issues, a system based on HydraDB has been designed to bridge the I/O gap and facilitate data movement over fast networks for the MapReduce analytics stacks. Figure 1 illustrates the key idea of such a system. Mainly acting as a cache layer on top of HDFS, the system takes responsibility to prefetch input from HDFS into a HydraDB cluster, service the I/O requests from upper-layer applications, conduct eviction and necessary data format conversion. Meanwhile, by leveraging HydraDB, frameworks across the stacks can transparently relish the benefits provided by the RDMA protocol without further engineering efforts.

Figure 2 demonstrates the speedup HydraDB can yield for a number of Hadoop and Spark applications. Also included is the speedup difference when HydraDB uses RDMA and TCP/IP respectively. To achieve a fair comparison, the speedup is obtained by comparing to the case in which in-memory HDFS is used. Overall, for Hadoop applications that contain intensive I/O communications with HDFS, *e.g.,* TestDFSIO or Data Loading, significant speedup of as high as 17.9× has been observed. Although Spark has thoroughly optimized the MapReduce framework, HydraDB is still beneficial to many of its applications by accelerating input data retrieval, delivering improvement ranging from 4%∼41%. Moreover, those applications effectively obtain the advantages of RDMA, which outperforms TCP/IP in all the cases, providing substantially higher speedup.

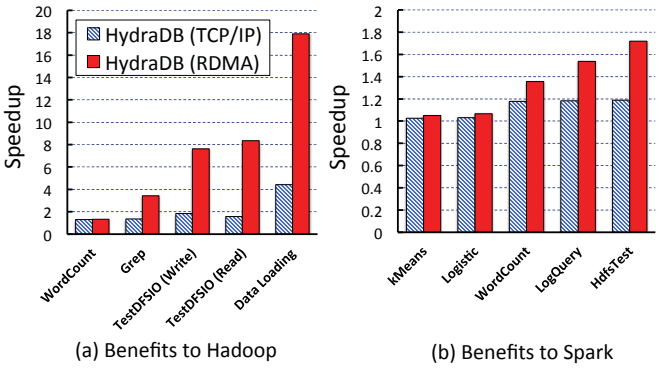


Figure 2: Comparing to in-memory HDFS, HydraDB can deliver efficient performance improvement to a number of Hadoop and Spark applications.

2.2 Scaling G2 Sensemaking Analytics

IBM *G2* Sensemaking [3] is a cutting-edge assertion-making analytics system based on continuous real-time events. It aims to address real-world challenges, including cyber security, voter registration, and fraud detection *etc*. To accomplish such goals, possessing the ability to immediately react to large volumes of observations is of critical importance to G2. However, existing relational data stores, *e.g.,* DB2, are gradually becoming the I/O bottleneck under the pressure of growing amount of data, impeding G2 from using more analytics engines to capture timely observations. Therefore, the G2 product has leveraged HydraDB as a complementary data store to absorb real-time events. Within the implementation, database tables are reorganized as key-value structures with the help of protobuf [5] to extract attributes residing in different columns. Figure 3 demonstrates that compared to in-memory database in the same setting, HydraDB allows 4× more G2 engines to effectively operate concurrently, delivering up to an order of magnitude higher throughput.

2.3 Call Record Processing

In telecommunication and mobile systems, processing large volumes of *Call Data Record* (CDR) imposes stringent and demanding throughput (≥ millions of access per seconds) and latency (≤ hundreds of microseconds) requirements. However, traditional practice using shared memory is unable to effectively scale to cope with rapidly increasing quantities of user data. Thus, to better satisfy the quality of service, HydraDB has been leveraged to enhance the data access, *e.g.,* user ID lookups or CDR updates, from a large number of stream Processing Elements (PE). Periodically, subscriber data and calling information of millions of users are extracted from the reference data source and loaded into HydraDB. HydraDB then reliably retains updated values and delivers low-latency lookups to the PEs.

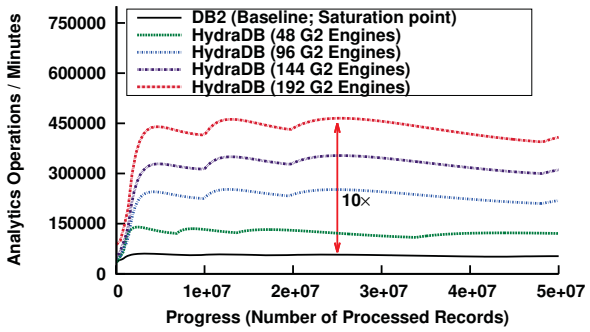


Figure 3: HydraDB provides G2 with abundant throughput and allows more G2 engines to efficiently capture more timely observations to refine decision-making.

3. RELATED WORK

General-Purpose In-Memory Key-Value Systems: The past few years have witnessed a burgeoning development of many distributed in-memory key-value systems. Among numerous implementations, Memcached [4] has been widely studied and deployed by leading organizations [24, 7] at scale to cache database query results and small objects. Its success is greatly attributed to its high-performance, simplicity, and scalability. However, Memcached cannot efficiently support RDMA protocol and ensure high availability. Redis [6] is another popular RAM-based key-value store, offering optional object persistence, master-slave replication with rich data structure support. Its single-threaded execution model allows Redis to deliver vertical scalability on multicore systems. However, sophisticated data partitioning or load rebalancing must be employed, otherwise skewed workloads can rapidly degrade the throughput of the entire system. RAMCloud [26] has drawn great attention from academia since its introduction. It demonstrates the capability of InfiniBand to reduce the read latency of small objects. RAMCloud has also proposed fast crash recovery [25] to minimize the unavailability period of data when failure occurs. However, none of above systems leverages RDMA protocol natively for data transfer and data replication.

RDMA-Optimized Key-Value Systems: Using RDMA to enhance the key-value systems has attracted many research interests in recent years. Jose, *et al* [16] have attempted to optimize the GET operation of Memcached by using RDMA. However, their solution requires a control message to notify the completion, thus needing an additional round trip. Mitchell, *et al* [23] have introduced Pilaf to optimize Memcached by leveraging RDMA Read to improve the performance of GET operations while continuing to rely on message passing for other performance-critical operations. At the same time, they propose self-verifying data structures to detect inconsistent read. Nevertheless, Pilaf lacks a holistic approach to leverage RDMA for different aspects of key-value stores, including message passing and fault-tolerance. HERD [18], introduced by Kalia, *et al*, strives to use best-effort RDMA Write with Send running over Unreliable Datagram (UD) to deliver high throughput with low access latency. However, HERD does not guarantee reliable data transfer, which is strongly required by enterprise applications.

Optimizations for Multicore Systems: Many works have also been conducted to optimize key-value stores on multicore systems. Based on Memcached, Fan, *et al*, have introduced MemC3 [11], which consists of a cache-friendly and concurrent Cuckoo hash table to improve the read throughput and a CLOCK-based eviction mechanism to reduce the memory consumption of indexing data structures. Masstree [21], introduced by Mao, *et al*, has leveraged a toolset of multicore optimization techniques to greatly improve an architecture-aware B^+-tree, so that the performance gap between cache and memory access can be efficiently hided. However, both works mainly target the design in which multiple threads share the same indexing data structure, thus differing from HydraDB, which resorts to single-threaded exclusive partition management. Nevertheless, their cache-friendly design can benefit HydraDB to minimize the DRAM access delay.

4. DESIGN PRINCIPLES OF HYDRADB

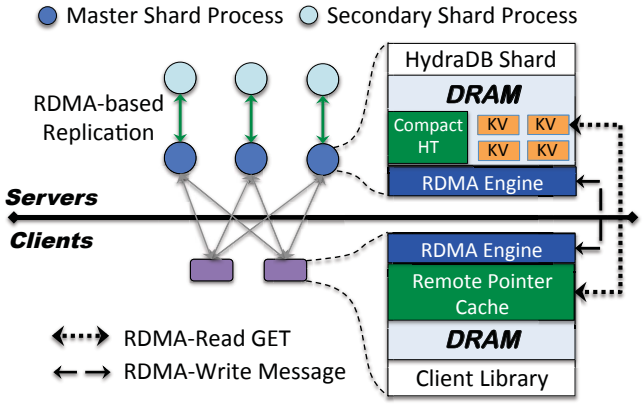


Figure 4: The architecture of HydraDB cluster, and major components within the HydraDB server and client sides.

HydraDB, as illustrated in Figure 4, is presented to users as a distributed hash table. All the data are maintained in the aggregated DRAM of all the servers and served to the clients across the RDMA-capable networks, *e.g.*, InfiniBand. To ensure high availability, each key-value pair is replicated into multiple servers. HydraDB provides data access services to applications through HydraDB client library (we call it client for short in the following). The client sides locate key-value pairs according to the consistent hashing algorithm [19].

Throughout the design, HydraDB leverages the RDMA protocol to accelerate message passing, GET operations, and minimize the overhead incurred by offering high availability. At the same time, HydraDB strives to exploit the computing power from multicore systems on the server sides via adopting single-threaded execution model, NUMA-awareness, along with cache-friendly compact HashTable. In the following sections, we detail the principles that guide our design from two main aspects: (1) How we adapt our design to multicore systems in Section 4.1, and (2) How we leverage RDMA protocol to enhance the performance-critical operations in Section 4.2.

4.1 Adapting to Multicore Systems

4.1.1 Parallel Execution Model

To efficiently utilize the available CPU cores and avoid expensive overhead incurred by conventional shared access control, *e.g.,* using mutexes, HydraDB has taken exclusive partition management model, in which a single CPU core exclusively manages a partition, *i.e.,* processing all the access requests for all the key-value pairs contained in the partition. The managing process, called *shard*, is pinned to a pre-determined core and retain the full computing power of that unit without sharing with others to eliminate the cost

of context switch. A key-value item is directly routed to a shard from a client side according to the 64-bit hashcode of its key using consistent hashing.

Although system designers have long been concerned with the incapability of partition-based design to handle highly skewed workloads, such issue can be effectively alleviated by HydraDB, which allows clients to cache remote pointers for previously accessed items (see section 4.2.2). Hence, there is a high likelihood for popularly read data to be retrieved via RDMA Read, which consumes zero CPU power of the shard that manages the datum. This substantially mitigates the penalty caused by skewed read distributions. In addition, previous studies [14] have also demonstrated that fine-grained data partitioning can further lessen the impact of imbalanced access patterns. Our results in sections 6 show that HydraDB is robust under higly skewed workloads.

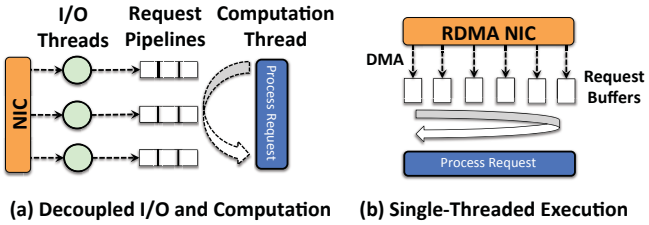


Figure 5: Decoupling I/O and computation is inefficient when RDMA is available, thus HydraDB resorts to the single-threaded execution model to use a single thread to conduct both request detection and handling.

Single-Threaded Execution: Due to the widening performance gap between I/O devices and computation units, many existing data processing and storage systems have decoupled I/O and computation, leveraging pipelined design to enhance the throughput as depicted in Figure 5(a). Such a design serves systems that rely on TCP/IP effectively as multiple threads can concurrently move data from kernel to the user space. However this decoupled execution model can result in underutilized CPU resources and unnecessary synchronization overhead when RDMA is available to directly transfer data from hardware to the user-level. In HydraDB, each shard process resorts to a single thread to conduct both request detection and handling, delegating the responsibility of intermediate data movement to the RDMA-capable NIC as shown in Figure 5(b). In our current implementation, the shard thread continuously polls the request buffers in a round-robin fashion. Upon detecting a new request, shard processes the request and sends the response out via RDMA Write before polling the next request buffer. Section 4.2.1 provides more details about our RDMA-based communication. By employing this single-threaded design, our performance investigation shows that HydraDB can deliver up to 94.8% higher throughput with above 50% lower latency comparing to the cases that use decoupled execution model. Moreover, such a single-threaded design provides the flexibility to control the partition size in a fine-grained manner, so that we can alleviate expensive overhead during failover, which is often the case for the monolithic process design.

4.1.2 NUMA-Awareness

During request processing, the location of data plays a critical role in achieving low latency access. Without prudence, frequent data movements across NUMA nodes can impose severe overhead resulted from many factors, such as interconnect bandwidth contention and higher remote access latency *etc*. Hence, to achieve NUMA-awareness, HydraDB has embedded NUMA domain information into all the shards, each of which strictly confines the memory allocation and future data access within the same domain as the shard thread. Though one may argue that interleaving the memory blocks over all available memory controllers can bring higher throughput, HydraDB can achieve so with lower latency by deploying multiple instances over different NUMA nodes so that aggregated memory bandwidth can be fully utilized.

4.1.3 Cache-friendly Hash Table

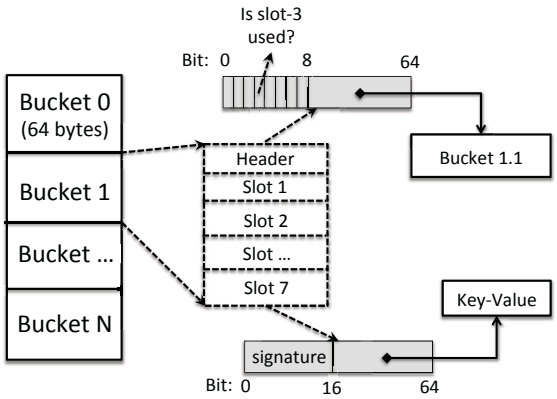


Figure 6: The layout of cache-friendly hash table aims to minimize the pointer dereferences and full-key comparison during lookup.

To efficiently handle read-intensive workloads consisting of variable key-value lengths, HydraDB server resorts to a hash table that stores the locations of key-values instead of actual data as the primary indexing data structure. However, naive implementations of hash table that depend on linked lists to resolve hash collisions is cache-unfriendly and can lead to excessive number of pointer dereferences under heavy workloads. To avoid such an issue, we have introduced a compact hash table that aims to reduce the number of pointer dereferences and unnecessary full key comparisons during indexing.

Figure 6 illustrates the details of our compact hash table. The main branch of the table is a continuous array of buckets, each of which contains 64 bytes that can fit in a single cache line on our system. Each bucket is structured to include a header (8-byte) with 7 slots (each one is 8-byte). Each slot is further partitioned into two areas. The first 16 bits are used to hold the signature of the key (implemented as a short hash) while the rest 48-bit links to the actual key-value data. Using this signature, a full key is only retrieved from the memory when the signatures match, thus we can efficiently cut down on the number of unnecessary full key comparisons. Meanwhile, to keep track of the usage of slots, the first 7 bits within the header byte are used as a filter. However, 7 slots may not be sufficient to resolve all the collisions, thus another 56-bit within the header are used to link a dynamically generated bucket. During execution, our hash table merges multiple buckets together after the remove operations.

By leveraging this compact hash table, HydraDB can resolve a large amount of hash collisions and lookups after a single cache-line read without traversing long linked lists. Also note that the above indexing only occurs when the servers receive message-based requests. The RDMA-Read accelerated GET operations completely bypass this server-side indexing procedure and directly retrieve items from the server-side memory. We are aware that there have been a large body of research on enhancing the performance of indexing

by using different data structures, such as Masstree [21], Cuckoo hashing [27], Hopscotch hashing [12] *etc*. We plan to evaluate their benefits to HydraDB in our future work.

4.2 Leveraging RDMA

HydraDB strives to take full advantages of RDMA protocol to enhance various aspects of a key-value store. In particular, we have introduced a message passing mechanism based on RDMA Write and leveraging RDMA Read to further optimize GET operations. Such design choice draws a lesson from a previous research [23] showing that detecting write-write races from many different remote clients is extremely difficult, if not impossible. In contrast, letting servers handle all the writes while using RDMA Read to accelerate content retrieval can efficiently utilize the fast networks and inhibit data inconsistency. The following sections further elaborate our RDMA design details.

4.2.1 RDMA Write driven Message Passing

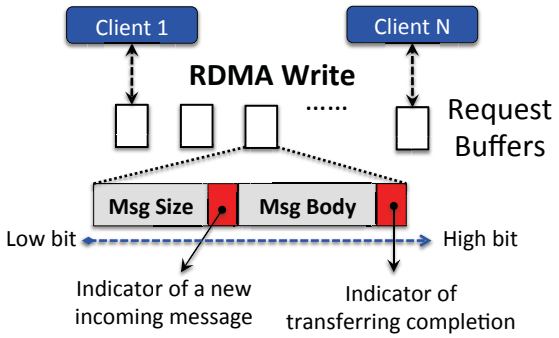


Figure 7: The memory layout and message format for conducting RDMA-Write driven message passing between clients and servers.

HydraDB relies on the servers to handle all the write requests, *e.g.,* INSERT or UPDATE. Those requests are issued to the servers in the form of messages. Meanwhile, since RDMA Write delivers the optimal latency and bandwidth, it is an ideal mechanism to pass messages. Unfortunately, one-sided RDMA Write lacks the capability to notify the server about the completion of a write operation, thus leading many system designers to either completely rely on Send/Receive methods or using RDMA Write with immediate to interrupt server-side CPU after data transfer. However, neither approach can perform comparably as RDMA Write. Hence, HydraDB has introduced a sustained-polling mechanism for both clients and servers to detect requests and responses issued by RDMA Writes.

As described in section 4.1.1, a single shard thread continuously polls a group of request buffers to detect incoming messages. Each request buffer exclusively belongs to a specific *Shard-Client* connection as shown in Figure 7. Similarly, a response buffer is also allocated on the client side and polled by a client CPU. To allow correct message detection, HydraDB employs an indicators-encapsulated message format as shown in Figure 7. Such a format leverages the fact that many RDMA adaptors provide in-order RDMA Write guarantees and was initially introduced in [20] for fast MPI message passing over RDMA networks. Following the increasing memory order, the first indicator (a word) helps detect the arrival of a new message. Meanwhile, the set of the 1st indicator also guarantees the consistency of the initial 4-byte, which tells the shard about the new message size. Upon perceiving the set of the 1st indicator, the shard then skips the next *Msg-Size* bytes to poll the last word of the message. After processing the request, the shard zeros out the request buffer and sends the response back to the client following the same format. We have compared the benefits of RDMA Write driven message passing with that of using Send/Receive in Section 6.2. The results demonstrate that such an approach can provide up to 162.6% higher throughput with significantly lower latency.

A major concern associated with the above approach is the high CPU usage. However, this issue can be greatly mitigated by leveraging high-resolution sleep functions. After polling the content for a certain number of cycles, shard puts CPU into the sleep mode (sleep 100*n*s in current case) if no new request is detected. By doing so, HydraDB imposes negligible CPU overhead under light workloads without sacrificing latency performance.

4.2.2 Optimizing GET via RDMA Read

Many production workloads [7, 8] consist of a large number of GET requests that access a small percentage of highly popular items. Under such scenarios, continuously sending requests for those key-values and relying on servers to process the requests can rapidly overload the nodes that accommodate those popular data, and dramatically increase the amount of redundant messages and processing. To tackle such issue, HydraDB exploits RDMA Read via allowing clients to directly retrieve data from server-side memory in a single round trip for previously-accessed key-values.

HydraDB achieves so by introducing a client-side remote pointer caching mechanism as shown in Figure 4. The first time a key-value is accessed, the request is issued to the corresponding server via RDMA Write as discussed in the previous section. Along with other returning information, a remote pointer that describes the location of the key-value pair on the server is also returned back to the client, who subsequently caches it locally. Later on, when the client needs to GET the value of the same key, it retrieves the remote pointer from the cache and directly fetches the key-value via RDMA Read. Comparing to the design that purely relies on message passing, remote pointer caching can further improve the throughput by over 29% for the read-intensive workloads used in our experiments. Such improvement increases when the workloads exhibit higher access locality.

4.2.3 Data Consistency Protection Mechanisms

RDMA Read can conflict with memory write operations conducted by the server, resulting in read-write race. Previous studies have attempted to address such issue by employing costly checksums [23] or cache-line versions [10] to allow clients to verify the consistency of fetched data. Differing from them, HydraDB takes a lightweight approach via combining the out-of-place update with lease-based deferred memory reclamation to guarantee data integrity.

For each key-value pair that allows RDMA Read, the server appends a guardian word at the end of the item to indicate whether the data has been deleted and prohibits in-place content modification. Upon receiving an update request, the shard firstly flips the guardian word of the original item in an atomic manner and creates a new area for the updated key-value. Since such guardian word is always fetched along with the key-value during a RDMA Read, clients can efficiently detect whether a retrieved item is outdated. In case outdated data is observed, the client then issues another GET message to the server to retrieve the latest version.

Meanwhile, how to efficiently reclaim the memory resource is challenging since the shard instances are not aware of the RDMA-Read GET operations, thus being unable to keep track of the number of remote pointers cached on the client sides. The breach of

data integrity can occur when servers imprudently free memory areas that are being referenced by many remote pointers. A tentative solution by forcing servers to send notification about every reclamation operation is at the cost of significant performance degradation.

In view of such challenge, HydraDB has introduced a *lease* mechanism to efficiently manage memory reclamation without impacting the performance. For each key-value pair, the server maintains a lease (64-bit timestamp), which stands as an agreement between servers and clients to guarantee the availability of the items within the server-side memory so that RDMA Read can be used to retrieve the data. Every server-aware GET operation extends the lease by a certain extent (varies from 1 second to 64 seconds) according to the approximate popularity of such key observed by the server. The lease term is also returned back to the client along with the remote pointer in order to notify the client about the time window within which RDMA Read can be used. In addition, clients periodically send lease renew messages to the servers to extend the leases of keys they deem as popular so that remote pointers of popular keys can remain valid within the local cache. On the server sides, upon receiving the update or remove requests, a server flips the guardian word and prevents the lease of outdated keys from being extended in the future. Later on, a background thread frees the memory area of the key-value when the corresponding lease has expired. Detailed design and evaluation of the lease mechanism introduced by HydraDB can be found in [31].

4.2.4 *Remote Pointer Sharing*

When many clients are collocated on the same machine, HydraDB allows the remote pointers used by a client to be shared with others. Besides accelerating the cache warm-up period, such design mainly aims to alleviate the cascading effect of remote pointer invalidation. Imaging a case in which 10 client processes are running on a single node with all showing strong preference for a single item. Without remote pointer sharing, when one client updates the item, each one of the rest 9 clients would experience an invalid RDMA Read, causing unnecessary data fetching and wasting the network resources. To facilitate the sharing, we have leveraged an IBM introduced Lock-Free hash table [22] to minimize the locking overhead when many clients try to access the same remote pointer cache. It is indeed a concern when security isolation is needed to separate the cache domains of multiple clients. In current implementation, HydraDB then simply disallows the sharing and uses exclusive cache for each client when secure access is enforced.

5. HIGH AVAILABILITY

Aiming to provision continuous data availability within a datacenter, HydraDB derives the High-Availability (HA) architecture from two core HA principles, including replicating data at multiple locations and establishing a state machine to provide consistent view of the status of all server instances. Meanwhile, we exploit RDMA Write to alleviate the overhead incurred by offering HA support. In the following, we firstly outline the HydraDB HA architecture, then we detail the design of RDMA-Write based Logging replication that allows us to reduce the cost for HA support.

5.1 HA Architecture

During runtime, HydraDB couples each primary shard with a number of secondary shards as shown in Figure 4 and synchronously replicates every key-value from the primary to the secondaries. The secondary shards are dedicated to the assigned primary without servicing other requests from any clients. Meanwhile, in order to satisfy the low-latency requirement, HydraDB employs star-formed primary-backup replication protocol instead of chain replication [29], letting the primary handle all the requests.

To monitor the aliveness of all the shard processes and guarantee a consistent view of process status, we have equipped HydraDB with a Zookeeper [13] cluster, which commonly comprises an ensemble of $3 \sim 5$ machines. Any status change, including process failures, node joining, *etc.,* is captured by the leader of an independent group of servers, dubbed **S**tatus **W**atcher and re**A**ct **T**eam (SWAT). Upon receiving a status change notification from the Zookeeper, the SWAT leader carries out necessary environment reconfigurations, *e.g.,* selecting a new primary shard from the secondaries, or notifying certain shards to migrate data to newly joined nodes, and synchronize the reconfiguration metadata with that stored in the znode. In the case of SWAT leader failure, a new leader from the SWAT group is elected and takes over future status change reactions.

5.2 RDMA Logging Replication

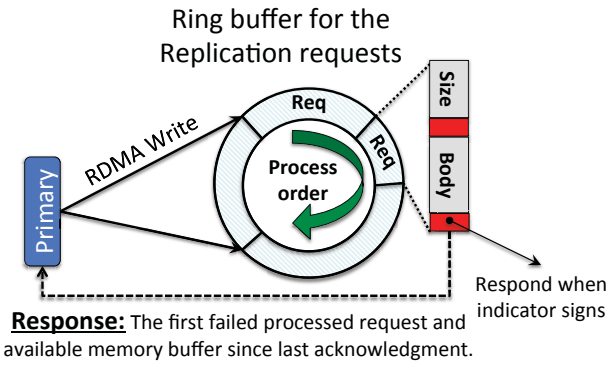


Figure 8: The HydraDB exposes a large chunk of memory of the secondary shard to the primary with a relaxed request/acknowledge model to efficiently use RDMA Write for request replication.

Given that the memory area of a secondary shard is exclusively used for backing up the content from a single primary without serving any request, it exhibits a *Single-Writer Zero-Reader* characteristic, making it unnecessary and inefficient to use conventional request/acknowledge for request replications.

Recognizing such issue, we have introduced an RDMA Logging Replication mechanism. Its key idea is to expose a large chunk of memory from the secondary shard to the primary, and allows the primary to directly replicate each write requests into the store using RDMA Write in a log-structured fashion. To further minimize the overhead during execution, HydraDB relaxes the requirement of immediate acknowledgment for every replication request.

To each replication request, the primary assigns a sequence number incremented by 1 after each assignment. Such sequence number will be returned back within the acknowledgment from the secondary to allow the primary to perceive failed replication requests. A replication request is written to the secondary using indicator-encapsulated format. Within the secondary shard, a dedicated thread polls replication requests from the exposed memory (the memory is organized as a ring buffer internally), then directly merges them into the main memory store. The acknowledgment number is incremented after successful processing. However, when the secondary fails to process a request, it stops incrementing the acknowledgment number and discards the following replication requests, then looping around the buffer until a request for acknowledgment from the primary is observed. Such a request is detected

via evaluating the value carried in the indicator. Upon detecting the sign of response, the secondary sends back the first failed requests and freed memory buffer since last acknowledgment. On the other side, when the primary observes unprocessed replication request, it rolls back to that request and re-sends all the replication requests following the failed one. To avoid severe performance degradation incurred by request resending (though it is rare), the primary asks for acknowledgment after several tens of requests.

6. PERFORMANCE ANALYSIS

HydraDB has been fully implemented using C++. It has exposed both C and Java interfaces through JNI. Though HydraDB also supports TCP/IP protocol, we do not present its performance in this paper. Recently, many different products have adopted HydraDB to enhance their applications as shown in Section 2. However, In order to comprehensively understand the performance of HydraDB, we have further conducted an extensive evaluation on an InfiniBand cluster that consists of 8 machines. Our major objective is to comprehend how well a HydraDB service running on a single multicore system can serve requests from many remote clients. In most of the following experiments, we dedicate a single machine as the HydraDB server. Unless otherwise stated, we allocate 4 shard instances on the server. We set up 50 clients on another 5 machines and use the rest 2 machines for Zookeeper and SWAT services. For the scale-out evaluation, we orchestrate 7 server machines to investigate aggregated performance HydraDB cluster can provide.

Server Machine Configuration: Each machine follows x86_64 architecture and is equipped with 4 2.60GHz Intel Xeon E5-4650L Octa-core processors located on 4 NUMA nodes, each of which connects to 64GB memory. Each node has a 40Gbps Mellanox ConnectX-3 InfiniBand adaptor connected to a single port of Mellanox IS5030 switch. We run RedHat 6.4 and OFED 1.5.4.1.

Evaluation Benchmark: We have used YCSB [9] to examine the performance of HydraDB. Considering that YCSB workload generation can be highly CPU-intensive and time-consuming, all the workloads are pre-generated. Throughout the experiments, all the clients first load the requests into the memory before starting the performance measurement. In the following sections, we mainly focus on 6 representative types of workloads that consist of different GET/UPDATE ratios, including *50% GET + 50% Update*, *90% GET + 10% Update* and *100% GET + 0% Update*. For each GET/UPDATE combination, we have measured the performance when the request distribution follows both Zipfian and Uniform distributions. Each workload consists of 60 million requests operating on 60 million records. Based on the previous product workloads analysis [24, 7], we have chosen 16-byte key with 32-byte value as the targeted item size for succinct results presentation. Note that, HydraDB can efficiently support much larger key-value items as demonstrated by the use case for the MapReduce enhancement in Section 2.1, in which each HDFS block is partitioned into several 4MB chunks and stored as key-value pairs within HydraDB. For all the tests, we report the average results of three runs.

6.1 Overall Performance

We start our evaluation by comparing HydraDB to several well-studied in-memory key-value stores, including Memcached (v1.4.21) [4], Redis (v2.8.17) [6] and RAMCloud [26]. To achieve fair comparison, we disable the data replication. Both Memcached and Redis run over InfiniBand through IPoIB [15] protocol, while RAMCloud uses its native InfiniBand support. For Memcached, we assign 8 threads and enable the memory lock-down. In the Redis tests, we run 8 Redis instances on our machine and leverage fine-grained sharding on the client sides. Similarly, a single RAMCloud server instance without replication is running with 8 threads allocated and logging silenced.

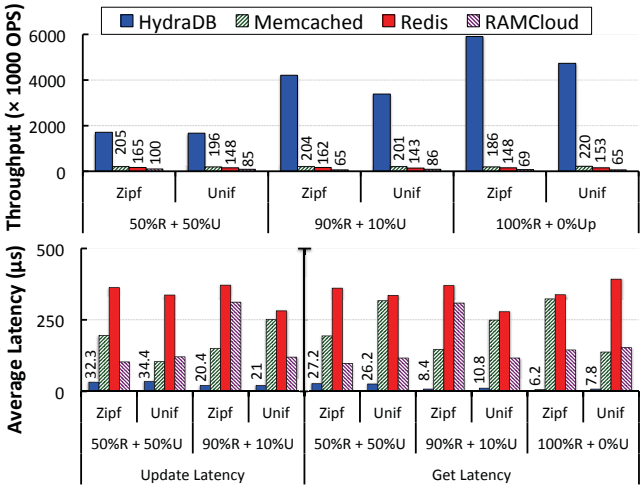


Figure 9: Performance comparison between HydraDB with 3 major key-value stores.

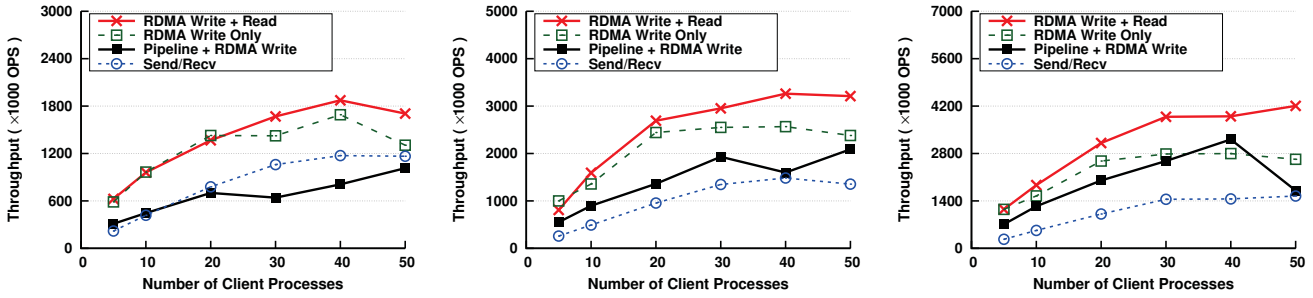
Figure 9 shows the peak throughput and corresponding average update/get latency comparison among 4 key-value stores. As we can see, HydraDB substantially outperforms 3 other key-value stores throughout all of the workloads due to its efficient RDMA support and adaption to the multicore system, providing an order of magnitude higher throughput with up to 50× lower latency.

Meanwhile, we have observed that the throughput of HydraDB increases effectively with higher GET/UPDATE ratios. This is mainly attributed to asymmetric read/write performance on the server side and RDMA Read optimization. Overall, when the GET percentage increases from 50% to 100%, HydraDB obtains 246.3%, 182.8% higher throughput for Zipfian and Uniform workloads respectively. Correspondingly, the average read latency reduces from 27.2μs to 6.2μs for Zipfian workloads and from 26.2μ to 7.8μ for Uniform workloads. Also revealed by the results of Zipfian workloads is that HydraDB displays better performance for skewed read-intensive workloads since a large amount of read requests can be conducted via RDMA Read, which efficiently decreases the number of messages over the network and lessens the amount of request processing on the server. Since many previous studies have unveiled that the production workloads tend to feature high GET/UPDATE ratios and exhibit skewed preference to a small amount of highly popular items, above results suggest that HydraDB is a promising solution for handling such types of workloads.

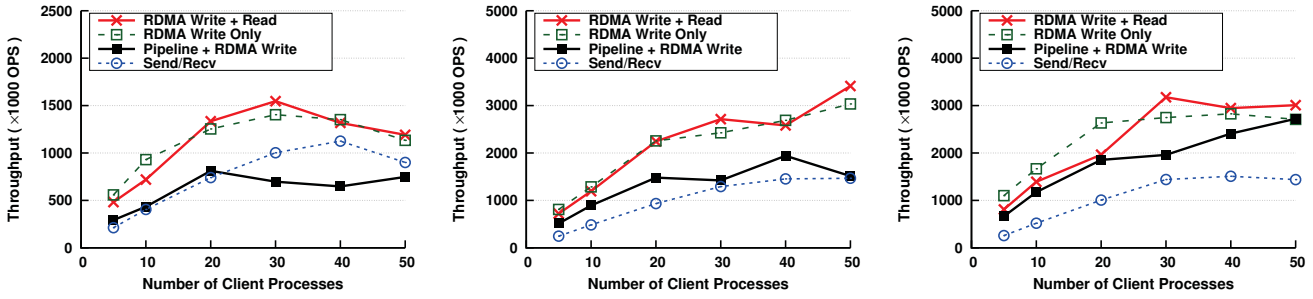
6.2 Impact of RDMA Design Choices

To investigate the impact of two major RDMA design choices, including *RDMA-Write driven Message Passing* and *Remote Pointer Caching for RDMA Read*, we have carried out an incremental evaluation through adding one feature at a time. Also, to establish a performance baseline, we have measured the performance of HydraDB that uses Send/Recv as the communication methods.

As shown in Figure 10, for all the workloads, using RDMA-Write (shown as "*RDMA Write Only*") to drive the message delivery substantially outperform Send/Recv based approach, yielding average improvements by up to 78.9% 142.8%, 155.2%, 74.7%, 136.1%, 162.6% for (a), (b), (c), (d), (e), (f) 6 workloads, respectively, thus adequately demonstrating the essentiality of leveraging RDMA Write for exchanging messages.



(a) 50%Get + 50%Update, Zipfian. (b) 90%Get + 10%Update, Zipfian. (c) 100%Get + 0%Update, Zipfian.



(d) 50%Get + 50%Update, Uniform. (e) 90%Get + 10%Update, Uniform. (f) 100%Get + 0%Update, Uniform.

Figure 10: Incremental evaluation of RDMA design choices using 6 YCSB workloads that consist of different GET/UPDATE percentages following Zipfian and Uniform distributions.

Then on top of RDMA-Write, we have further introduced remote pointer caching that allows clients to use RDMA Read to directly retrieve data from server-side memory (shown as "*RDMA Write + Read*"). As shown by the Zipfian workloads, using RDMA Read efficiently improves the average throughput for (a), (b), (c) workloads by 10.1%, 14.4%, 29.9%, respectively. However, as also indicated by the results that the benefit of caching remote pointer diminishes when more updates appear in the workloads. This is because higher ratio of updates not only reduces the chance for clients to leverage RDMA Read but also causes more invalid data retrieving (outdated item is observed) as well. Such phenomena is corroborated by the hit analysis shown in Figure 11. For the Zipfian workloads, successful remote pointer hits reduces by 75.5% when update ratio increases from 0% to 50%, but invalid hits increases by about 7 million times.

On the other hand, for the Uniform workloads, the chances of reusing cached remote pointers is substantially less than their counterparts in the Zipfian workloads as also demonstrated in Figure 11, thus less benefit is observed when remote pointer caching is employed for (d), (e), (f) three Uniform workloads.

6.2.1 *Single-Thread versus Pipelined-Structure*

Recall from section 4.1.1 that another design choice made surrounding the RDMA is to use single-threaded execution model since RDMA-capable NIC can spare CPU cores from conducting intensive I/O. Compared with the pipelined execution model, single threading can reduce the number of required cores and minimizes the synchronization overhead. To further examine the benefit against pipelined execution, we have modified HydraDB according to the pipelined structure depicted in Figure 5. During the experiments, we allocate 4 pipelined-shard instances on 4 NUMA

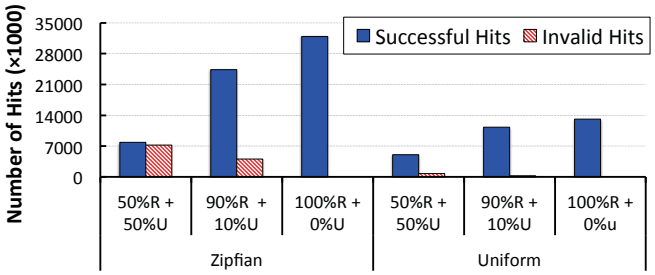
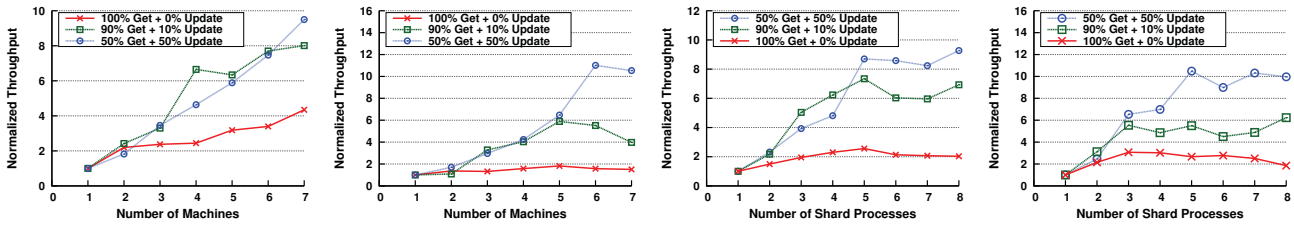


Figure 11: Detailed remote pointer hit analysis (50 clients cases). Remote pointer caching can efficiently benefit workloads that contain a large amount of read with skewed access pattern, while delivering less benefits for workloads with many update operations.

nodes. Meanwhile, to draw efficient parallelism, 2 shard threads are assigned to each instance, which contains another 2 threads to dispatch the requests from multiple clients. Essentially, such configuration uses 16 cores, 4× more than we have used for single-threaded tests. We have disabled remote pointer caching and only use RDMA Write for message passing.

As shown by "*Pipeline + RDMA Write*" in Figure 10, such pipelined design performs substantially worse than single-threaded HydraDB even though more resources are allocated. For all six workloads, single-threaded HydraDB (*RDMA Write Only*) can outperforms pipelined model by 94.8%, 53.1%, 27.4%, 86.9%, 60.6%, 34.4% for the workloads (a), (b), (c), (d), (e), (f) on average, respectively.

6.3 Scalability



(a) Scale-out Uniform (b) Scale-out Zipfian (c) Scale-up Uniform (d) Scale-up Zipfian

Figure 12: Scalability performance of HydraDB.

We investigate the scalability of HydraDB from two perspectives, including its capability to scale out and scale up, respectively. In both set of experiments, 60 clients spread on 6 machines are used to issue requests concurrently and we measure the aggregated throughput. Meanwhile, we plot the normalized throughput in Figure 12 to show the scalability.

During the scale-out performance measurement, we constrain the number of shard instance on each machine as 1 while increasing the number of machines in the cluster from 1 to 7. As shown in Figure 12 (a). For 3 uniform workloads, HydraDB has demonstrated efficient scalability since more instances effectively help rebalance evenly distributed requests. Both 50% Get+50% Update and 90% Get+10% Update workloads show linear scalability. While for the 100% Get workload, collocating the clients with the server instances attenuates the benefit of adding more NICs for RDMA Read operations, thus affected scalability is observed. On the other hand, for the Zipfian workloads, due to skewed distribution, increasing the number of machines cannot efficiently rebalance the requests, therefore, for each workload, a saturation point is observed, 6 for 50% Get + 50% Update, 5 for 90% Get+10% Update, and 100% Get respectively. In addition, collocating the clients severely limits the scalability for 100% Get workload.

To examine the scale-up on a multicore system, we use a single machine for the HydraDB server and increase the number of shard instances on the node from 1 to 8. Figure 12 (c) and (d) present the scale-out performance. As shown in the figures, for the uniform workloads, since both 50% Get + 50% Update and 90% Get + 10% workloads require intensive server-side CPU involvement, increasing the number of shards from 1 to 5 effectively scales the throughput linearly or even super-linearly. However, slow improvement rate is observed once the number of shards goes beyond 5. One major cause is that handling too many RDMA connections (Number of Shards $\times$ Number of Clients) starts triggering the scalability bottleneck within the network driver, offsetting the benefits provided by adding more shard instances. While, for the Zipfian workloads, both skewed request distribution and too many RDMA connections together inhibit the effectiveness of adding more shard once the number is beyond 3 for 90% Get + 10% update workload, and 5 for 50% Get + 50% Update workload. In addition, for all the 100% Get workloads, the processing capability of the network device is saturated by RDMA operations even with small number of shards (≤ 3).

An important implication from above experiment is that too many RDMA connections can prevent HydraDB from scaling out on a single machine. A potential solution to mitigate such issue is using sub-sharding mechanism to allow a single shard instance to use multiple cores for independent sub-shards while the main process maintains all the connections. Hence we may reduce the number of RDMA connections managed by the device driver when we add more cores to serve the requests.

6.4 Impact of RDMA Logging Replication

As discussed in Section 5.2, strictly following request-acknowledge path for data replication can idle the primary shard and neglect the optimization space exposed by the Single-writer Zero-Reader characteristic of HydraDB HA design. Therefore, we have introduced RDMA Logging replication with relaxed request/acknowledge model to accelerate the data replication. To evaluate the impact of our design, we have mustered different number of clients to concurrently issue INSERT requests into a single shard instance, which connects to a different number of secondary shards. Each client issues 3 millions of INSERTs and we measure the average latency of different replication protocols.

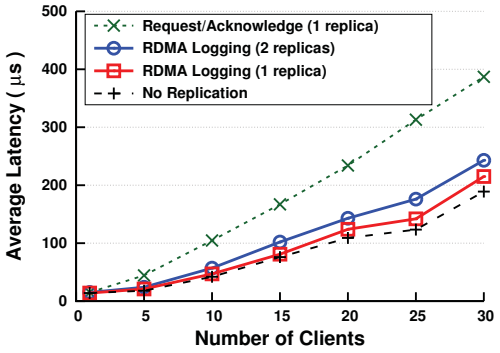


Figure 13: RDMA Logging replication can efficiently reduce the latency overhead incurred by key-value replication.

Figure 13 illustrates the performance comparison between strict request/acknowledge model with RDMA Logging replication. As shown in the figure, although HydraDB has optimized the message passing via using RDMA write, strictly requiring an acknowledgment for each replication request still imposes severe overhead during data replication. Compared to the case when there is no replication, strict request/acknowledge model consistently doubles the average latency. In contrast, via relaxing the acknowledgment requirement, RDMA Logging replication can accomplish most replication request in a single RDMA Write round-trip. In addition, the processing of the replication request on the secondary shard can be overlapped with that on the primary. As a result, substantially mitigated overhead is observed. As shown in the figure, using a single replica only increases the average latency by 12.3%, and the overhead only reaches 41.1% when each request is replicated twice.

7. CONCLUSION

Traditional networking technology is curbing the full potential of in-memory key-value systems. To tackle such challenge, this paper presents HydraDB, a new resilient in-memory key-value middle-

ware that has been built from ground up to fully take advantages of high-performance networking technologies, *e.g.*, InfiniBand and Converged Ethernet. In particular, HydraDB has strived to leverage Remote Direct Memory Access protocol to comprehensively optimize a key-value store from three main aspects, including message passing, read enhancement, and data replication. Meanwhile, to avoid restraining the performance of RDMA, HydraDB has been designed to be multicore-friendly so that data manipulations on the servers do not impede the access progressing. HydraDB has sufficiently demonstrated its effectiveness and efficiency not only under our performance evaluation with a variety of YCSB workloads, but also in accelerating many real-world cluster-computing frameworks, such as Hadoop, Spark and IBM G2 product. Although we have only described a few applications in this paper, HydraDB is showing strong benefits to many others. With more datacenters are adopting high-performance networks, we believe more cluster computing applications that require high-throughput and low-latency access will obtain performance benefits from HydraDB.

8. REFERENCES

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Apache Spark. http://spark.apache.org/.

[3] IBM InfoSphere Sensemaking. http://www-01.ibm.com/software/data/entity-analytics-solutions/.

[4] Memcached. http://memcached.org/.

[5] Protobuf. https://code.google.com/p/protobuf/.

[6] Redis. http://redis.io/.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[8] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[11] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[12] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[14] Lim Hyeontaek, Han Dongsu, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, nsdi'14, Berkeley, CA, USA, 2014. USENIX Association.

[15] J. Chu and V. Kashyap. Transmission of IP over InfiniBand(IPoIB). http://tools.ietf.org/html/rfc4391, 2006.

[16] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[19] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[20] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[21] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[22] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[23] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling

memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[25] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[26] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[27] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[28] Mike Svoboda and Diego Zamboni. Leveraging in-memory key value stores for large-scale operations. https://www.usenix.org/conference/lisa13/leveraging-memory-key-value-stores-large-scale-operations, November 2013.

[29] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[30] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and optimization of memory-resident mapreduce on HPC systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 799–808, 2014.

[31] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 23:1–23:13, New York, NY, USA, 2014. ACM.

[32] Yandong Wang, Xinyu Que, Weikuan Yu, Dror Goldenberg, and Dhiraj Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[33] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 569–578, May 2013.