2012 IEEE 26th International Parallel and Distributed Processing Symposium

High-Performance Design of HBase with RDMA over InfiniBand

Jian Huang[1], Xiangyong Ouyang[1], Jithin Jose[1], Md. Wasi-ur-Rahman[1], Hao Wang[1],
Miao Luo[1], Hari Subramoni[1], Chet Murthy[2], and Dhabaleswar K. Panda[1]

[1] *Department of Computer Science and Engineering,*
The Ohio State University
{huangjia, ouyangx, jose, rahmanmd, wangh, luom, subramon, panda}
@cse.ohio-state.edu

[2] *IBM T.J Watson Research Center*
Yorktown Heights, NY
{chet}
@watson.ibm.com

***Abstract*—HBase is an open source distributed Key/Value store based on the idea of BigTable. It is being used in many data-center applications (e.g. Facebook, Twitter, etc.) because of its portability and massive scalability. For this kind of system, low latency and high throughput is expected when supporting services for large scale concurrent accesses. However, the existing HBase implementation is built upon Java Sockets Interface that provides sub-optimal performance due to the overhead to provide cross-platform portability. The byte-stream oriented Java sockets semantics confine the possibility to leverage new generations of network technologies. This makes it hard to provide high performance services for data-intensive applications.**

The High Performance Computing (HPC) domain has exploited high performance and low latency networks such as InfiniBand for many years. These interconnects provide advanced network features, such as Remote Direct Memory Access (RDMA), to achieve high throughput and low latency along with low CPU utilization. RDMA follows memory-block semantics, which can be adopted efficiently to satisfy the object transmission primitives used in HBase.

In this paper, we present a novel design of HBase for RDMA capable networks via Java Native Interface (JNI). Our design extends the existing open-source HBase software and makes it RDMA capable. Our performance evaluation reveals that latency of HBase *Get* operations of 1KB message size can be reduced to 43.7 μs with the new design on QDR platform (32 Gbps). This is about a factor of *3.5* improvement over 10 Gigabit Ethernet (10 GigE) network with TCP Offload. Throughput evaluations using four HBase region servers and 64 clients indicate that the new design boosts up throughput by *3 X* times over 1 GigE and 10 GigE networks. To the best of our knowledge, this is first HBase design utilizing high performance RDMA capable interconnects.

***Keywords*-HBase, InfiniBand, RDMA, Cloud Computing and Clusters**

I. Introduction

Over the last several years there has been a substantial increase in the number of cloud data storage services being deployed to meet the requirements of data intensive web applications and services. NoSQL systems [1] have shown that they have enormous scalability to satisfy the immense data storage and look-up requirements of these applications. One good example is Google's BigTable [2], which is a semi-structured database intended to handle petabytes of data. It is a sparse, persistent multi-dimensional sorted distributed storage system that is built on top of Google File System [3]. Many projects at Google such as web indexing, Google Earth, and Google Finance use BigTable. Another example is HBase [4], the Hadoop [5] database. It is an open source distributed Key/Value store, aiming to provide BigTable like capabilities. It is written in Java to achieve platform-independence. HBase uses HDFS (Hadoop Distributed File System) [6], [7] as its distributed file system. Usually, HBase and HDFS are deployed in the same cluster to improve data locality. HBase is used by many Internet companies (e.g. Facebook and Yahoo!) because of its open source model and high scalability. At Facebook, a typical application for HBase is 'Facebook Messages' [8], which is the foundation of a 'Social Inbox' [9].

The existing open-source HBase implementation uses traditional Java (TCP) Sockets. This provides a great degree of portability, but at the price of performance. It is well known that the byte-stream socket model has inherent performance limits (both latency and throughput) [10], [11], due to issues such as multiple memory copies. The byte-stream model of Java Sockets requires a Java object be serialized into a block of bytes before being written to a socket (for additional rounds of data copy). On the receiver side de-serialization must be performed to convert an incoming data block into a Java object. All these aspects result in further performance loss.

High performance networks, such as InfiniBand [12], provide high data throughput and low transmission latency. Open source software APIs, such as OpenFabrics [13], have made these networks readily available for a wide range of applications. Over the past decade, scientific and parallel computing domains, with the Message Passing Interface (MPI) as the underlying basis for most applications, have made extensive usage of these advanced networks. Implementations of MPI, such as MVAPICH2 [14] achieves low one-way latencies in the range of 1-2μs. Even the best implementation of sockets on InfiniBand achieves 20-25μs one-way latency. InfiniBand is being embraced even in Datacenter domain. At ISCA 2010, Google published their work on high performance network design for data-center, which uses InfiniBand to achieve high performance and low power consumption [15]. Oracle's 'Exalogic Elastic Cloud' uses InfiniBand to integrate storage and compute resources [16].

With these known problems of TCP sockets, and the known advantages of InfiniBand, it seems natural to ask whether HBase could benefit from being able to use InfiniBand (and other RDMA-capable networks, in general). There have been some studies [17], [18] to exploit RDMA-capable networks to improve the performance of Memcached [19] - a popular distributed-memory object-caching system. In this paper, we want to study exactly this problem for HBase: the limitations of socket-based communication inside HBase, and approaches to deliver the advantages of a high performance network such as InfiniBand and its RDMA capability to HBase so as to accelerate HBase network communications. We address several challenges in this paper:

1) In the existing HBase design, can we characterize the performance of Java Sockets-based communication? Does the use of Java Sockets diminish performance and if so, by how much?
2) Can we improve HBase performance by using high-performance networks such as InfiniBand? In particular, can we achieve a hybrid design in which both the conventional socket interface and the new RDMA-capable communication channel co-exist in the same HBase framework?
3) With such a hybrid communication mechanism, will HBase show improvements in query latency and transaction throughput?

We perform detailed profiling to understand the performance penalty caused by Java Sockets in the HBase query processing cycle. Based on that finding, we extend the HBase data communication mechanism into a hybrid hierarchy that incorporates both conventional sockets and advanced RDMA-capable InfiniBand for efficient network I/O. We conduct extensive experimental studies evaluating the performance implications of this new RDMA-capable communication module in contrast to other conventional socket based networks including 1 GigE, 10 GigE, and IP-over-InfiniBand (mentioned in Section II-B1). Our RDMA-capable design achieves *Get* latency as low as 43.7 μs for 1 KB payload, which is *3.5 X* times faster than 10 GigE network with TCP offloading. Throughput evaluations using four HBase region servers and 64 clients indicate that the new design boosts up throughput by *3 X* times over 1 GigE and 10 GigE networks.

The major contributions of this paper are:

1) A detailed profiling to reveal the performance penalty caused by conventional sockets in end-to-end HBase data access operations.
2) A hybrid hierarchical design of HBase communication modules that incorporates both sockets and advanced RDMA features.
3) An extensive evaluation to study the performance implications of the new RDMA-capable module, in contrast to existing socket based solutions.

The rest of the paper is organized as follows. In Section II, we give a background about the key components involved in our design. In Section III and Section IV, we propose a hybrid communication hierarchy to incorporate InfiniBand and socket interface into existing HBase framework. In Section V, we present our experiments and evaluation. Related work is discussed in Section VI, and in Section VII we present conclusions and future work.

II. Background

A. HBase

HBase is an open-source database project based on Hadoop framework for hosting very large tables [4]. It is written in Java and provides BigTable [2] like capabilities. HBase consists of three major components as shown in Figure 1: HBaseMaster, HRegionServer and HBaseClient. HBaseMaster keeps track of active HRegionServers and takes care of assigning data regions to HRegionServer. It also performs administrative tasks such as resizing of regions, replication of data among different HRegionServers, etc. HBaseClients check with HBaseMaster to identify which server it should request for read/write operations. HRegionServers serve client requests by fetching or updating data stored in Hadoop Distributed File System (HDFS). HDFS [5] is a fault tolerant distributed file system. Files are divided into small blocks before they are stored in HDFS. The default size of each block is 64MB, and each block is replicated at multiple DataNodes. HBase calls DFSClient to load regions from HDFS to memory. In other words, HBase acts as a client of HDFS. Usually, HBase and HDFS are deployed in the same cluster to improve the data locality.

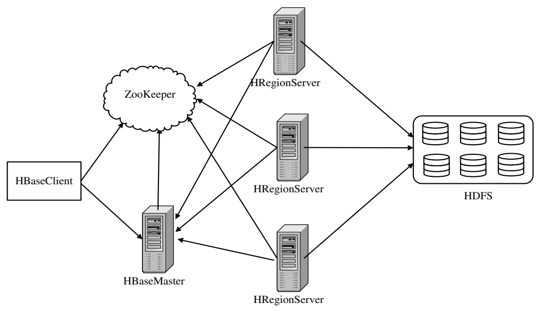


Figure 1. HBase Architecture

B. High Performance Networks

In this section, we present an overview of popular high performance networking technologies that can be utilized in data-center domain.

1) InfiniBand: InfiniBand [12] is an industry standard switched fabric that is designed for interconnecting nodes in High End Computing clusters. It is a high-speed, general purpose I/O interconnect that is widely used by scientific computing centers world-wide. The TOP500 [20] rankings released in June 2011 indicate that more than 41% of the computing systems use InfiniBand as their primary

interconnect. One of the main features of InfiniBand is Remote Direct Memory Access (RDMA). This feature allows software to remotely read or update memory contents of another remote process without any software involvement at the remote side. This feature is very powerful and can be used to implement high-performance communication protocols [21]. InfiniBand has started making inroads into the commercial domain with the recent convergence around RDMA over Converged Enhanced Ethernet (RoCE) [22]. InfiniBand offers various software layers through which it can be accessed. They are described below:
(a) **InfiniBand Verbs Layer**: The *verbs* layer is the lowest access layer to InfiniBand. There are no intermediate copies in the OS. The verbs interface follows Queue Pair (QP) communication model. Each QP has a certain number of work queue elements. Upper-level software using verbs can place a work request on a QP. The work request is then processed by the Host Channel Adapter (HCA). When work is completed, a completion notification is placed on the completion queue. Upper level software can detect completion by polling the completion queue or by asynchronous events (interrupts). Polling often results in the lowest latency. The OpenFabrics interface [13] is the most popular verbs access layer due to its applicability to various InfiniBand vendors.
(b) **InfiniBand IP Layer**: InfiniBand software stacks, such as OpenFabrics, provide a driver for implementing the IP layer. This exposes the InfiniBand device as just another network interface available from the system with an IP address. Typically, Ethernet interfaces are presented as `eth0`, `eth1`, etc. Similarly, IB devices are presented as `ib0`, `ib1` and so on. It does not provide OS-bypass. This layer is often called "IP-over-IB" or IPoIB for short. We will use this terminology in the paper. There are two modes available for IPoIB. One is the datagram mode, implemented over Unreliable Datagram (UD), and the other is connected mode, implemented over Reliable Connection (RC).

2) 10 Gigabit Ethernet: 10 Gigabit Ethernet was standardized in an effort to improve bandwidth in data center environments. It was also realized that traditional sockets interface may not be able to support high communication rates [23], [24]. Towards that effort, iWARP standard was introduced for performing RDMA over TCP/IP [25]. iWARP is very similar to the verbs layer used by InfiniBand, with the exception of requiring a connection manager. In addition to iWARP, there are also hardware accelerated versions of TCP/IP available. These are called TCP Offload Engines (TOE), which use hardware offload. The benefits of TOE are to maintain full socket streaming semantics and implement that efficiently in hardware. We used 10 Gigabit Ethernet adapters from Chelsio Communications in our experiments.

C. Unified Communication Runtime (UCR)

The low-level InfiniBand Verbs (IB Verbs) can provide better performance without intermediate copies overhead, but the programming for IB Verbs is complicated. Therefore, we developed a light weight, high-performance communication runtime named Unified Communication Runtime (UCR). It abstracts the communication APIs of different high-performance interconnects like InfiniBand, RoCE, iWARP, etc. and provides a simple and easy to use interface. UCR was proposed initially to unify the communication runtimes of different scientific programming models [26], such as MPI and Partitioned Global Address Space (PGAS). In [17], it was used as the middleware for Memcached, and significant performance improvements were observed in terms of operation latency and throughput. UCR is designed as a native library to extract high performance from advanced network technologies.

D. Yahoo! Cloud Serving Benchmark (YCSB)

Yahoo! Cloud Serving Benchmark (YCSB) [27] is a framework for evaluating and comparing the performance of different "Key/Value" and "cloud" serving stores. It defines a core set of benchmarks for four widely used systems: HBase, Cassandra [28], PNUTS [29] and a simple shared MySQL implementation. It also provides the flexibility for adding benchmarks for other data store implementations.

There are six core workloads and each of these represents different application behaviors. For example, the typical application example for workload C is user profile cache, Zipfian and Uniform distribution modes are used in YCSB for record selection in database. Besides that, we can also define our own workloads. In addition to these different workloads, there are three runtime parameters defined in YCSB to adjust the workload on the client side, like number of clients, target number of operations per second, status report modes, etc. In our experiments, we used workload A (50% Read, 50% Update), workload C (100% Read) and a modified version of workload A (100% Update).

III. Integrating InfiniBand Network into HBase

In this section, we briefly describe the design of HBase with traditional networks and identify the potential performance penalties in this design. Then we introduce our design using UCR and show how these overheads can be avoided using UCR-based design.

A. Performance Penalty with Socket-based Interface

Existing HBase makes use of Java Socket Interface for communication. As depicted in the left part of Figure 3, practical applications call HBaseClient APIs to access HBase. These APIs internally invoke Java Socket Interface for transferring data. As discussed in section I, socket-based data send/receive incurs significant overheads. In order to understand the performance penalties, we conduct detailed profiling analysis to measure the time spent in different stages of a query processing operation. The major steps involved in an operation are: (1) "Communication" for data transfer, (2) "Communication Preparation" for putting the serialized data into socket buffer, (3) "Server Processing" for HRegionServer to process a request, (4) "Server Serialization", (5)"Client Serialization" to serialize a Java object into byte format on HRegionServer and HBaseClient sides, respectively and (6) "Client Processing" to read reply from server and de-serialization data. The profiling results are presented in Figure 2.

In the procedure to perform a HBase *Put* with 1 GigE network, socket transmission and receipt ("Communication") is responsible for 60% of the total latency seen by the application. As a result, a bulk portion of the latency is ascribed to the inefficient socket transmission. This ratio is 41% and 34% for IPoIB and 10 GigE, respectively. Due to its internal driver stack overhead, IPoIB is not able to minimize the communication cost. Even though 10 GigE has a much higher raw bandwidth than 1 GigE, it is still not capable enough to substantially cut down the communication cost.

The "Communication Preparation" stage copies a serialized object into Java Socket buffer which involves Java software stack overhead. This part consumes 6.1%, 8.3% and 9.6% of total time for 1 GigE, IPoIB and 10 GigE networks, respectively. The "Communication" and "Communication Preparation" stages together are responsible for the high communication cost in existing HBase design. Both stages can be improved by advanced network technologies and RDMA capability. The profiling results indicate the potential performance benefits that are achievable by reducing the communication time.

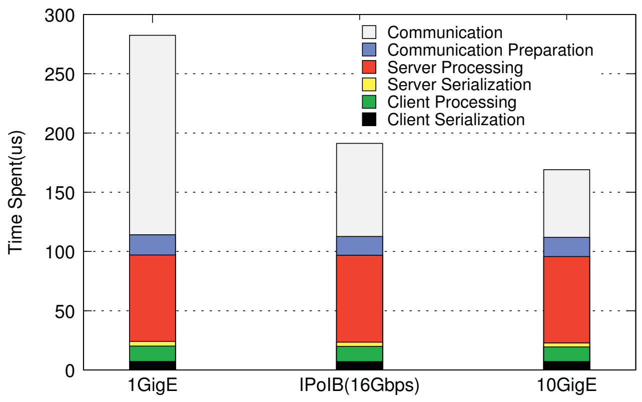


Figure 2. Breaking Down *Put* Latency into Different Components. Socket Transmission is Responsible for a Substantial Part in the Overall Cost.

B. Integrating InfiniBand into HBase: A Hybrid Approach

High performance networks, such as InfiniBand and iWARP with their RDMA capabilities, have the potential to substantially reduce the latency. Compared with Sockets over RDMA and 10Gigabit Ethernet networks, the native InfiniBand network provides the lowest latency [17]. Thus, we focus on integrating InfiniBand network into HBase to achieve better performance.

However, HBase is written in Java for portability, while the interfaces for RDMA programming over InfiniBand are implemented in C language. Therefore, Java Native Interface (JNI) is introduced to work as a bridge between HBaseClient APIs and UCR as shown in Figure 3.

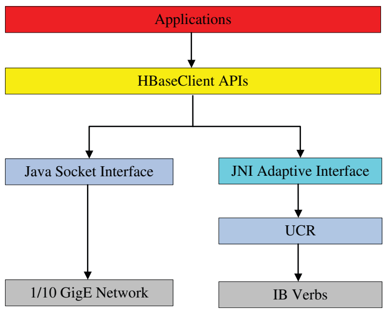


Figure 3. General Design

In particular, our new design aims to accomplish the following goals: 1) Simple yet efficient - Clear and simple interfaces shall be provided on different layers to satisfy the needs from the adjacent upper layer; 2) Open - It shall be easy to be applied to other similar middlewares used in data centers. Our design shall support both conventional and new clusters with advanced network interconnect features.

With all these goals in mind, we propose a hybrid approach to incorporate both socket interface and InfiniBand transport into the same framework. Our extended HBase design supports conventional socket-based networks and also leverages the advanced features in InfiniBand interconnect to deliver the best performance. Several challenges must be addressed for such a design in order to achieve an ideal performance:

1) How to bridge the portable Java-based HBase design and the native high performance communication primitives in an efficient manner to reduce the intermediate overhead?
2) How to extend the multi-threaded HBase into the native data transmission environment to extract the full potentials of high performance networks?
3) How to manage the buffer used in the native communication primitives in order to reduce memory footprint?

IV. Detailed Design

We present our hybrid design for HBase to incorporate both socket and advanced networks such as InfiniBand in this section. First, we briefly walk through the existing socket-based HBase communication flow. Then we describe the new hybrid architecture and explain its key components. Our design extends HBase communication management policies to provide hybrid communication support. It makes use of UCR for InfiniBand communication via JNI Adaptive Interface.

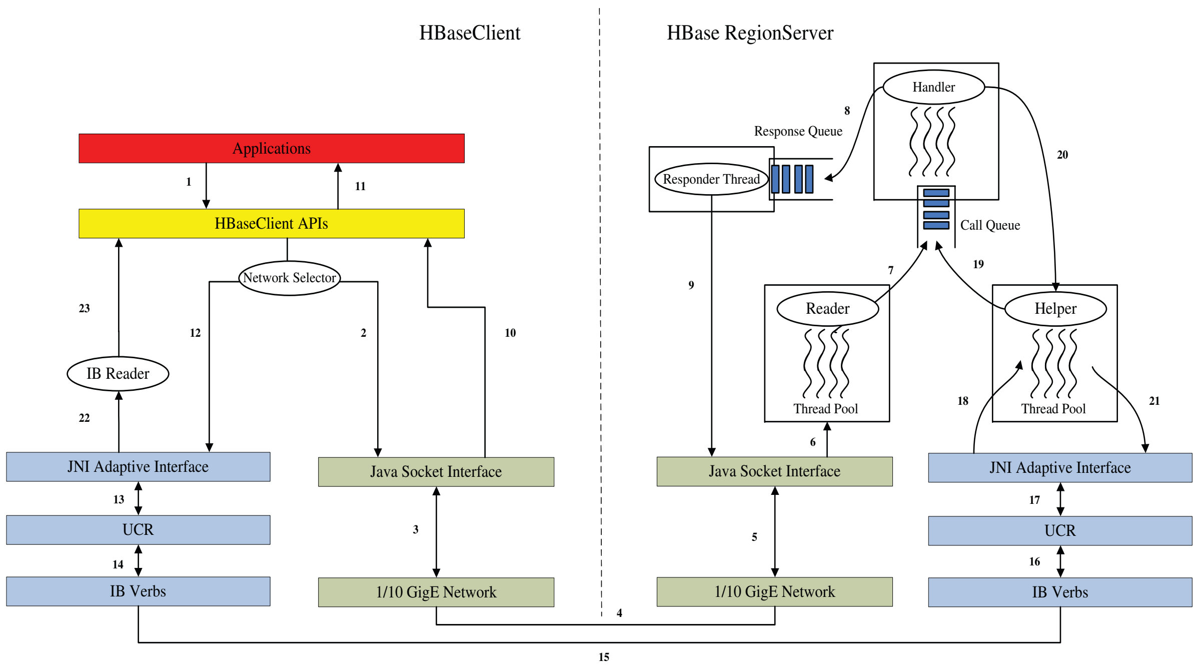


Figure 4. Architecture and Workflow of HBase Design with Hybrid Approach

A. Socket-based HBase Communication Flow

In our hybrid framework, the original components in HBase software stack are kept intact to support the conventional sockets interface. We briefly outline some key components in HBase before explaining our design.

On HBaseRegionServer, there is a ***Listener*** thread to monitor Java Sockets. When incoming data is detected, the Listener picks a thread from the ***Reader Thread Pool*** to process the data. Once a *Reader Thread* is awaken by the *Listener*, it reads data from the socket, de-serializes the data into a Java `Call` object defined in HBase, and pushes it into a queue associated with a ***Handler Thread Pool*** (path 7 in Figure 4). One of the *Handler Threads* claims the `Call` object and performs the actual processing. Once the processing is completed, the *Handler Thread* puts the `Call` object together with the serialized reply data into the response queue which is maintained by the ***Responder Thread*** (path 8 in Figure 4). The *Responder Thread* sends the reply data back to the client (path 9 in Figure 4).

B. Hybrid Communication Framework to Support High Performance Networks

We adopt a hybrid approach by extending the existing HBase communication framework to support InfiniBand and RDMA capability using the UCR library.

As shown in Figure 4, the new design includes two parts: the HBaseClient side, and the HRegionServer side. We introduce four new components in HBase client side: (1) ***Network Selector*** assists in selecting networks according to operation types and pre-defined rules; (2) ***IB Reader***, is responsible for receiving data sent from the HRegionServer and de-serializing into Java objects; (3) ***JNI Adaptive Interface***, enables upper-layer Java code to invoke our native UCR library; and (4) ***UCR*** enables RDMA-based data communication over InfiniBand network. On the HRegionServer side, *Helper Thread Pool* co-operates with the UCR for data communication. Since the connection management is different between HBaseClient and HRegionServer, we chose to implement the JNI Adaptive Interface separately on both sides to operate in a more efficient manner. We discuss these components in detail in the following sections.

C. Communication Flow over InfiniBand via UCR library

Figure 4 highlights the key components involved in data communication. *Network Selector* enables the hybrid communication mode. It assigns operations to socket helper threads or UCR helper threads based on the operation type. *Put* and *Get* operations are assigned to UCR threads, where as all other operations are assigned to socket threads. For clarity, we explain the communication work flow of HBase *Get* operation here. Application calls the HBaseClient *Get* API (path 1 in Figure 4). *Network Selector* then passes the *Get* operation (path 12 in Figure 4) to the *JNI Adaptive Interface*. This interface is the glue between HBase and UCR, it not only enables HBase to access the UCR, but also manages memory sharing between Java and C. *JNI Adaptive Interface* invokes *UCR* communication API (path 13 in Figure 4) to send the request via InfiniBand.

UCR is an end-point based communication library. End point is analogous to sockets; client and server use this as a communication end-point. In HBase, clients create end-point and connect to the region server using this end-point. When an operation is issued, it first checks if there is an end-point already with the respective HRegionServer. If end-point exists, it is re-used; otherwise, a new end-point is created. New end-point requests are assigned to *Helper Threads* in a round-robin manner. *Helper Threads* deal only with InfiniBand communications, just as *Reader Threads* deal only with sockets. When a *Helper Thread* receives a message on one of its assigned end-points (path 18 in Figure 4), it de-serializes the request into a Java `Call` object and puts it into a call queue (path 19 in Figure 4) for further processing. This is similar to what the *Reader Threads* do in case of sockets. In the original HBase design, after a Handler thread finishes processing a request, it pushes the `Call` object into a response queue, from where a Responder thread writes it back out to the HBaseClient via sockets. In the IB module, the *Handler Thread* just sets the corresponding end-point's response status (path 20 in Figure 4).

D. Connection Management

1) Extending HBase Connection Management: HBase uses `Connection` object to represent a communication channel between HBaseClient and HRegionServer. Multiple threads multiplex a socket in the `Connection` to talk with the other side. We extended the HBase connection management to encapsulate both conventional socket interface and the end-point based high performance networks at the same time. The HBase `Connection` object now encloses a mapping to keep track of all active end-points as a resource pool. When communication is needed to the other side of the `Connection`, a usable end-point is selected from the pool to perform the data transfer. This end-point pool improves the connection resource usage by multiplexing end-points among many concurrent threads. It also ensures that a request can be issued without any delay.

2) Multi-threading Support in HBaseClient: Most of the practical applications deploy multi-threaded design, to increase concurrency and achieve maximum throughput. YCSB benchmark used in our experiments launches multiple clients as multiple threads. threads for its workloads. As we discussed in previous section IV-D1, a *Connection* object is created in HBaseClient when it establishes a connection with the target HRegionServer. Each `Connection` object maintains a hash table to store all the requests (`Call` objects) issued by applications. For the single HRegionServer case, all the threads in the application layer share the same connection since their target HRegionServer addresses are the same. Also, in a typical deployment scenario, an HBaseClient needs to communicate with multiple HRegionServers. Therefore, HBaseClient needs to build multiple connections with multiple HRegionServers at the same time. Our HBase design with RDMA over InfiniBand takes this also into consideration. In our JNI Adaptive Interface library, a list of end-points could be created for each connection. Each time, when a thread in the application issues a request, it can get a free end-point to send its request to the target HRegionServer. If there is no available end-point for that connection, a new end-point is created and added into the end-point list. When the request gets reply from HRegionServer, the end-point will be returned back to the end-point list for future use. In the unmodified HBase software, when the `Call` object is sent out to HRegionServer, `call.wait()` is called until client receives data from HRegionServer and finishes the de-serialization of the returned data. If the returned data size is too large, the overhead caused by the de-serialization could be considerable, and this could block other threads, if there are multiple threads sharing the same connection. In our design, we removed the de-serialization part from the critical path. Once the IB Reader thread gets notified that one end-point's result is back, it will notify the corresponding `Call` object to get the returned data in the registered buffer.

E. Communication Buffer Management

In HBase, Key/Value pairs are organized as Java objects, while the underlying RDMA layer provides memory-based semantics. To bridge the gap between the upper Java object semantics and underlying memory-based semantics, we used Java direct byte buffer in our design. This enables our design to take advantage of zero-copy techniques offered by lower layer communication library.

As we discussed in Section IV-D, each connection handles a list of end-points on HBaseClient side, and all these end-points in the same connection share the same chunk of registered buffer for keeping low memory footprint. All the end-points share the same registered buffer chunk and this avoids memory footprint explosion. To avoid interference among these end-points, each end-point maintains a dynamic pointer and size to keep track of the specific region in the registered buffer currently associated with the end-point. Buffer management scheme is the same in both client and server sides.

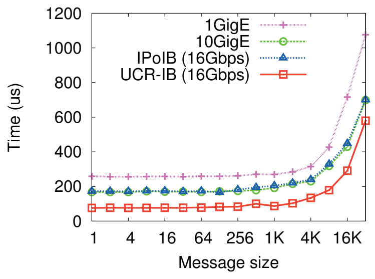
V. Performance Evaluation

In this section, we present the detailed performance evaluation results of our design, as compared to the traditional socket based design over 1 GigE, IPoIB and 10 GigE networks. We conduct micro-benchmark level experiments as well as YCSB benchmark experiments in our evaluations.

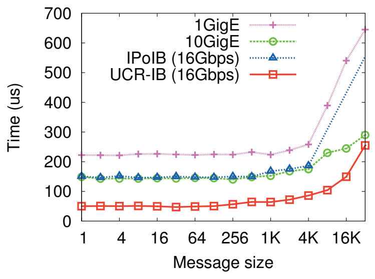
(1) Micro-benchmark Experiments: In this set of experiments, we evaluate the latency of HBase *Put* and *Get* operations. We also present detailed profiling analysis of HBase *Put*/*Get* operations and identify different factors contributing to the overall latency.

(2) Synthetic Workload using YCSB (Single server - Multi-clients): In this experiment, we keep one HRegionServer and vary the number of HBase clients. We use 1 KB payload for the multi-client experiments. This is the default payload in YCSB benchmark.

(3) Synthetic Workload using YCSB (Multi-servers - Multi-clients): In this experiment, we deploy multiple HRe-

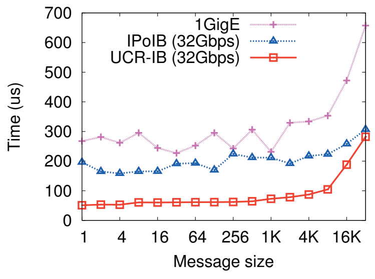


(a) Put Latency

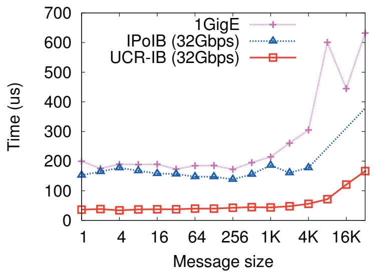


(b) Get Latency

Figure 5. Single Server, Single Client (Cluster A)



(a) Put Latency



(b) Get Latency

Figure 6. Single Server, Single Client (Cluster B)

gionServers and vary number of HBase clients. This workload is similar to real HBase deployment workloads.

A. *Experimental Setup*

We use two different clusters in our evaluations.

(1) Intel Clovertown Cluster (Cluster A): This cluster consists of 64 compute nodes with Intel Xeon Dual quad-core processor nodes operating at 2.33 GHz with 6 GB RAM. Each node is equipped with a ConnectX DDR IB HCA (16 Gbps data rate) as well as a Chelsio T320 10GbE Dual Port Adapter with TCP Offload capabilities. Each node runs Red Hat Enterprise Linux Server release 5.5 (Tikanga), with kernel version 2.6.30.10 and OpenFabrics version 1.5.3. The IB cards on the nodes are interconnected using a 144 port Silverstorm IB DDR switch, while the 10 GigE cards are connected using a Fulcrum Focalpoint 10 GigE switch.

(2) Intel Westmere Cluster (Cluster B): This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. Each node runs Enterprise Linux Server release 6.1 (Santiago) at kernel version 2.6.32-131 with OpenFabrics version 1.5.3. This cluster is less than two years old and it represents the leading edge technologies used in commodity clusters.

We use HBase version 0.90.3 and Hadoop version 0.20.2 in all our experiments. We use the following configuration: Master node and HDFS name-node run on one compute node, while HBase clients and HRegionServers run on different nodes. In this paper we focus on the performance potentials of high performance network to accelerate HBase data transmission. So we choose a small data set size in the experiments such that all data to be accessed is completely stored in server memory. By doing this we preclude the possible performance anomalies caused by accessing data from disks. Nowadays, it is common that a server is equipped with 64 GB or more main memory, and the aggregated memory size of many HBase servers is big enough to hold majority of the working set data [30]. Therefore we feel our assumption to cache data in memory is fair, and it can capture the essence of a production HBase deployment. In our future study we will expand our investigations to take disk access cost also into account.

B. *Micro-benchmark Evaluations*

We design micro-benchmarks to evaluate the latency of HBase *Put* and *Get* operations. The benchmark issues *Put* or *Get* operations of a specified size and measures the total time taken for the operation to complete. We run this micro-benchmark in single region server - single client configuration. We keep only one record in the region server, so that all the accesses are serviced from server's memory store and no disk accesses are involved. The experiment is performed on both Cluster A and Cluster B. Due to space limitations, we report only the results of record sizes from from 1 byte to 64 KB. It is to be noted that, most of the *Put*/*Get* payload sizes in a typical Cloud application fall within this range [27]. We also present a detailed profiling analysis of both HBase client and HRegionServer during a *Put*/*Get* operation. It provides insights to what all factors contribute to the overall operation latency. We did the profiling for multiple networks and the results are shown in Section V-B3.

1) Results on Cluster A: As we can observe from Figure 5 that, the UCR based design (denoted as UCR-IB) outperforms the socket based design for all the data sizes. For 1 KB *Put* operation, the latencies observed for IPoIB, 1 GigE and 10 GigE are 204.4 μs, 269.1 μs and 191.4 μs, respectively. Our design achieves a latency of 86.8 μs, which is 2.2 times faster than 10 GigE. The same level of improvements is observed for *Get* operations as well. This demonstrates the capability of native InfiniBand RDMA to achieve low latency.

2) Results on Cluster B: The same experiment is repeated on Cluster B and the results are shown in Figure 6. Cluster B does not have 10 GigE network cards. For both *Get* and *Put* operations, UCR based design outperforms socket-based channels by a large margin. For *Get* operation of 1 KB message size, latency observed with our design is 43.7 μs, where as the latencies are 185.8 μs and 214.4 μs for IPoIB and 1 GigE, respectively. This is a factor of four improvement. For *Put* operations, the performance improvement over IPoIB and 1 GigE networks is 2.9 X and 3.2 X times, respectively. It is to be noted that UCR based design in Cluster B is faster than in Cluster A, and this is due to the higher InfiniBand card speed (32 Gbps in Cluster B vs 16 Gbps in Cluster A).

These results, illustrated in Figure 5 and Figure 6, clearly underline the performance benefits that we can gain through OS-bypass and memory-oriented communication offered by RDMA semantics.

3) Detailed Profiling Analysis: We have profiled HBase client and HRegionServer to measure the cost at different steps during a *Put*/*Get* operation. The profiling results, shown in Figure 7, indicate the time taken at different steps for *Put*/*Get* operation with 1 KB payload.

During a *Put* operation, the client first serializes the request and puts it into a communication buffer. These are indicated as "Client Serialization" and "Communication Preparation," respectively. The serialized object is then sent to the server, where it is de-serialized and processed. This step is indicated as "Server Processing." After handling the request, the server sends back the response to the client. The time taken for serializing the response is indicated as "Server Serialization." The Client receives the response and processes it (indicated as "Client Processing"). The overall communication time (client to server and server to client) is denoted as "Communication Time." The actual number of bytes written on wire for *Put* operation of 1 KB are 1296 (request) and 85 (response). For 1 KB *Get* operation, these are 136 (request) and 1091 (response) bytes. It is to be noted that the communication time decreases considerably in the UCR-based HBase design. Communication preparation time is also reduced in this design. In socket based design, the serialized object has to be copied into Java socket buffer. But our design bypasses this, which avoids the copy overhead.

As we can observe from Figure 7, communication cost is one of the major factors contributing to the overall operation latency. The total communication time for 1 KB *Put* operation over UCR is 8.9 μs, where as it is 168.3 μs, 78.5 μs and 57.1 μs for 1 GigE, IPoIB and 10 GigE networks, respectively. Our design achieves a performance improvement of 6 X times over 10 GigE (fastest among the socket versions). Similar trend is observed for *Get* operation, as indicated in the figure. Overall, UCR-based design decreases the communication time substantially and reduces the overall latency perceived by end user.

C. YCSB (Single server and Multiple clients):

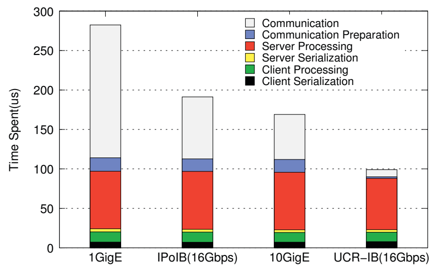
In this experiment, we use the YCSB benchmark and measure the HBase *Get* operation latency. We use single region server - multiple clients configuration for this experiment. All the clients issue *Get* operation on a same record. We restrict the server working set to just one record, so that all the queries are serviced from the server memory, avoiding hard disk access anomalies.

Figure 8(a) shows the average latency for a *Get* operation for varying number of clients. As it can be observed from the figure, UCR-based design substantially reduces the operation latencies as compared to socket-based transports. For 8 clients, our design reduces the latency by around 26% as compared to 10 GigE network. As the number of clients increase, the operation latency rises due to heavier workload on the server. The performance improvement using UCR can be observed even with increased number of clients.

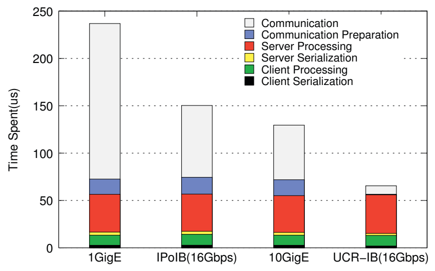
Figure 8(b) denotes the transaction throughput for the same experiment. The total throughput increases as more clients are added because of higher level of concurrency in request handling. It is to be noted that UCR based design achieves very good throughput scalability with increasing number of clients. For 16 clients, our design achieves a throughput of 53.4 K ops/sec, which is 27% higher than 10 GigE network.

D. YCSB: Multi-servers and Multi-clients

In this experiment we use four region server nodes and vary the number of client nodes from 1 to 16. We prepare client nodes in two configurations, 1 client thread per node and 8 client threads per node. For each of these configurations, we use different workloads: 1) 100% *Get* operations (Read Workload), 2) 100% *Put* operations (Write Workload), and 3) 50% *Get* + 50% *Put* operations (Read-Write Workload). We disable caching at the client side by deleting the following statements in YCSB benchmark - `_hTable.setAutoFlush(false);` `_hTable.setWriteBufferSize(1024*1024*12);` This models the benchmark behavior to that of real world workloads. We use four HRegionServers in this experiment, with 320,000 1 KB records (320 MB data) evenly distributed

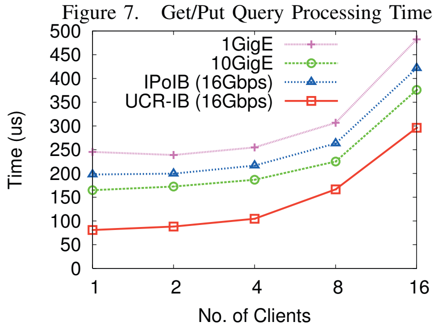


(a) Put 1 KB

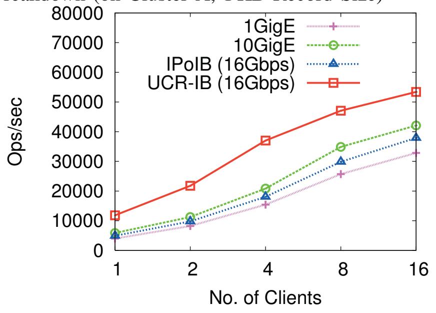


(b) Get 1 KB

Figure 7. Get/Put Query Processing Time Breakdown (on Cluster A, 1 KB Record Size)



(a) Latency



(b) Throughput

Figure 8. Single Server, Multiple Clients, HBase *Get* on Cluster A, 1 KB Message Size.

among all four servers. Each HRegionServer hosts 80 MB data which is completely cached in memory. The client threads perform queries according to either Zipfian distribution or Uniform distribution. We do not find any perceivable difference in terms of performance between these two distributions. Therefore only the results from Zipfian distribution are reported due to space constraints. Also, because of limited number of 10 GigE adapters available, we have only 12 client nodes in the experiments with 10 GigE networks.

1) Read Workload, One Thread per Client Node: In this experiment, we vary the number of client nodes from 1 to 16, with each node running one YCSB thread. Each YCSB client issues *Get* operations to the HRegionServers. Similar to previous results, UCR-based design significantly reduces the query latency as indicated in Figure 9(a). For 8 clients, the operation latency using our design is 174.6 μs. This is 43% lower than the 10 GigE operation latency. As the latency is reduced, UCR-based design almost doubles the aggregated operation throughput as compared to the 10 GigE network.

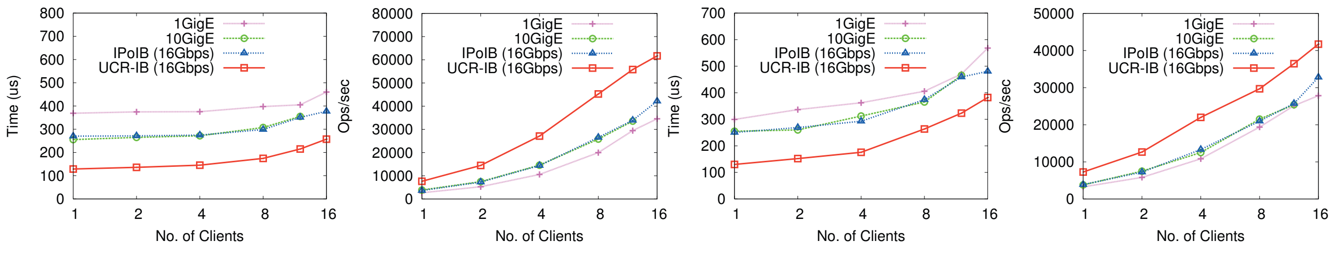
2) Write Workload, One Thread per Client Node: This experiment is similar to the previous experiment. Each client node runs one YCSB thread that issues *Put* operations to the HRegionServers. As shown in Figure 9(c), significant performance improvement is observed for UCR-based design. For 16 clients, UCR design achieves a throughput of 41.7 K ops/sec, which is 20% higher than IPoIB network.

3) Read-Write Workload, One Thread per Client Node: In this workload each YCSB thread issues equal amounts of *Get*/*Put* operations. Among the three socket-based transports 10 GigE performs the best. The results are depicted in Figure 10. Compared to 10 GigE, UCR reduces the read and write latency by 24% and 15%, respectively for 8 clients. Our design also boosts up the overall transaction throughput by up to 23%.

4) Read Workload, Eight Threads per Client Node: In this experiment, each client node runs 8 YCSB threads to perform *Get* operations. We vary the number of client nodes from 1 to 16 so that the total number of client threads ranges from 8 to 128. Figure 11(a) depicts that all the socket-based transports perform almost analogously, with 10 GigE slightly better. The operation latency rises with increased workload from more client threads. More client threads also lead to higher throughput. Across all the ranges, UCR-based design always yields the lowest latency and highest throughput. For 64 clients, our design reduces the latency by around 26% as compared to 10 GigE network. Our design improves the operation latency by around 25% as compared to IPoIB for 128 clients.

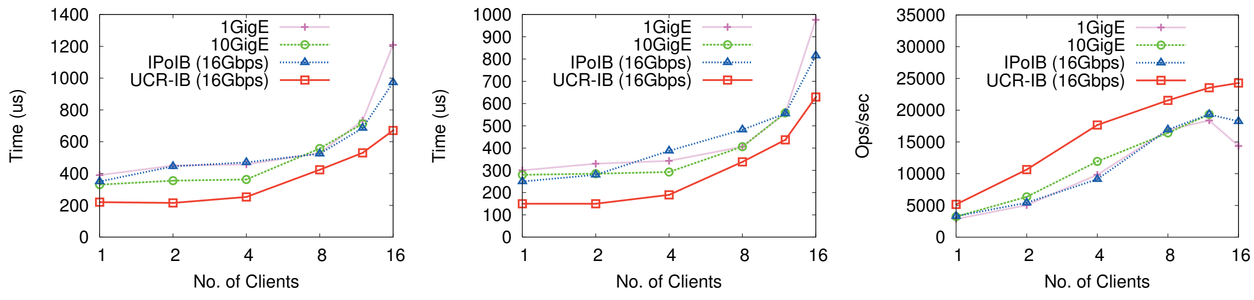
5) Write Workload, Eight Threads per Client Node: In this workload, each client thread issues *Put* operations to the HBase HRegionServers. The total number of client threads are varied from 8 to 128. The latency and throughput results are given in Figure 11(c). The write operation latency is reduced by around 15% as compared to 10 GigE, for 64 clients. For 128 clients, the write throughput is boosted up by 22% over the throughput obtained using IPoIB network.

6) Read-Write Workload, Eight Threads per Client Node: In this workload illustrated in Figures 11, each client thread performs both *Put* and *Get* operations. We disable client side query batching in YCSB to force each client request directly



(a) Read: Latency (b) Read: Throughput (c) Write: Latency (d) Write: Throughput

Figure 9. Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster A (1 thread/node, 1 KB message)



(a) Read Latency (b) Write Latency (c) Throughput

Figure 10. Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (1 thread/node, 1 KB message)

go to HBase server without being cached. We observe very high latency with 10 GigE and 1 GigE. For 64 clients, the throughput improvement is around *3X* times over 1 GigE and 10 GigE networks.

VI. Related Work

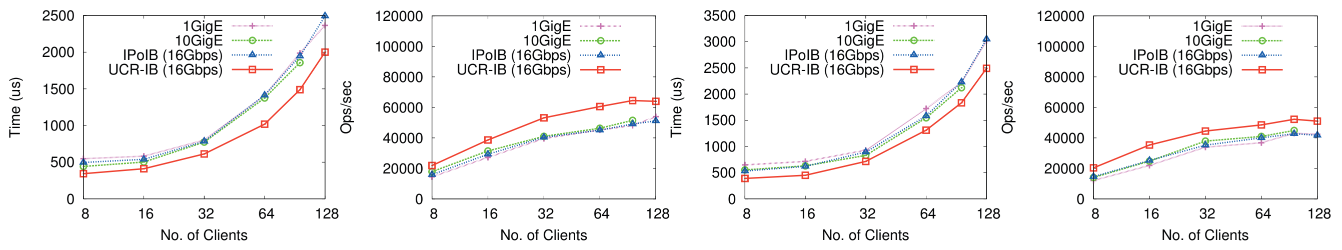
Distributed file systems and databases are active research topics. Cooper et al. [27] presented YCSB framework to compare the performance of new generation cloud database systems. Patil et al. [31] presented YCSB++ to extend YCSB to support complex features in cloud database, such as the ingest-intensive optimizations. Dory et al. [32] presented the cloud database elasticity to measure the behavior when adding or removing data nodes. Shi et al. [33] also proposed benchmarks to evaluate HBase performance and provided some insights to optimize cloud database system. As illustrated in this work, HBase has better scalability and elasticity compared with other cloud database systems. However, HBase does not support many advanced features, such as the server-side filtering and the transactional guarantee for multi-key accessing. When the researchers extended HBase to support those advanced features in MD-HBase [34] and G-Store [35], they found HBase performance became the bottleneck for the extended cloud database systems. So, it is very important to improve HBase performance. In this paper, we focused on HBase performance improvement using high performance network, InfiniBand. We compared HBase performance over IPoIB, 1 GigE and 10 GigE networks, and proposed a new communication layer design for HBase using InfiniBand. To the best of our knowledge, this is the first work to enhance HBase using InfiniBand.

In recent times, HBase and InfiniBand are gaining attraction in data-center deployments. Facebook deployed 'Facebook Messages' using HBase. Their engineers indicate that network I/O plays an important role in the whole system performance [36]. Oracle designed 'Oracle Exalogic Elastic Cloud' to integrate business applications, middlewares, and software products [16]. The storage and compute resources in this cloud are integrated using its high-performance I/O back-plane on InfiniBand. However, these systems support protocols such as Sockets Direct Protocol (SDP), IP-over-InfiniBand (IPoIB), and Ethernet-over-InfiniBand (EoIB), and these do not make use of native InfiniBand. In our work, we rely on native InfiniBand using UCR. In our previous work [37], we evaluated HDFS performance on InfiniBand and 10 G Ethernet networks using different workloads. In this paper, we enabled RDMA communication channel in a Java based system.

VII. Conclusion and Future Work

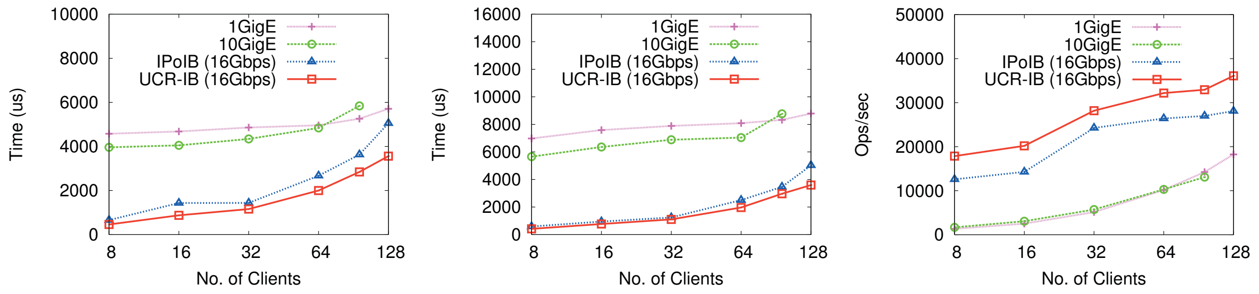
In this paper, we took on the challenge to improve HBase data query performance by improving the communication time. We performed detailed profiling to reveal the communication overheads caused by Java sockets, and showed that communication played an important role in the overall operational latency. We identified this as an opportunity to take advantage of high performance networks such as InfiniBand to improve HBase operation performance.

In order to integrate high performance networks into HBase, we designed a hybrid hierarchical communication mechanism to incorporate both conventional sockets and high performance InfiniBand. By leveraging the high-throughput and low-latency InfiniBand network, we were



(a) Read: Latency (b) Read: Throughput (c) Write: Latency (d) Write: Throughput

Figure 11. Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster A (8 threads/node, 1 KB message)



(a) Read Latency (b) Write Latency (c) Throughput

Figure 12. Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (8 threads/node, 1 KB message)

able to substantially drive down HBase data query latency and boost up transaction throughput. We exploited the InfiniBand RDMA capability and adopted it to match with the object delivery model used by HBase. Our design achieves a latency of as low as 43.7 μs for a 1KB payload Get operation, which is around *3.5 X* times better than 10 GigE sockets. Our design also substantially increases the operation throughput. In YCSB 50%-read-50%-write experiment with 64 clients, our design delivers a throughput *3 X* times higher than 10 GigE.

As part of future work, we plan to run comprehensive benchmarks to evaluate the performance of HBase and expose further potential performance bottlenecks. We also plan to analyze the software overheads of HBase in detail and come up with a highly optimized and scalable design.

VIII. Acknowledgments

This research is supported in part by a grant from IBM OCR Program, U.S. Department of Energy grant #DE-FC02-06ER25755, National Science Foundation grants #CCF-0702675, #CCF-0833169, and #CCF-0916302; and an equipment grant from Wright Center for Innovation #WCI04-010-OSU-0.

References

[1] R. Cattell, "Scalable SQL and NoSQL Data Stores," *SIGMOD Record*, vol. 39, pp. 12–27, May 2011.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *The Proceedings of the Seventh Symposium on Operating System Desgin and Implementation (OSDI'06)*, WA, November 2006.

[3] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," in *The Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, NY, USA, October 19-22 2003.

[4] Apache HBase, http://hbase.apache.org.

[5] Apache Hadoop, http://hadoop.apache.org/.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *The Proceedings of the 26th IEEE International Symposium on Mass Storage Systems and Technologies (MSST 10)*, Incline Village, NV, May 3-7 2010.

[7] J. Shafer, S. Rixner, and A. L. Cox, "The Hadoop Distributed Filesystem: Balancing Portability and Performance," in *The Proceedings of the Internation Symposium on Performance Analysis of Systems and Software (ISPASS'10)*, White Plains, NY, March 28-30 2010.

[8] J. Seligsein, "Facebook Messages," http://www.facebook.com/blog.php?post=452288242130.

[9] N. O'Neil, "Facebook Social Inbox," http://www.allfacebook.com/facebook-social-inbox-always-on-messaging-with-people-you-care-about-2010-11.

[10] H. O. M. Baker and A. Shafi, "A Study of Java Networking Performance on a Linux Cluster," *Technical Report*.

[11] P. W. Frey and G. Alonso, "Minimizing the Hidden Cost of RDMA," in *The Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Montreal, Quebec, Canada, June 22-26 2009.

[12] Infiniband Trade Association, http://www.infinibandta.org.

[13] OpenFabrics Alliance, http://www.openfabrics.org/.

[14] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.

[15] D. Abts, M. Marty, P. Wells, P. Klausler and H. Liu, "Energy Proportional Datacenter Networks," *International Symposium on Computer Architecture (ISCA)*, 2010.

[16] Oracle, "Oracle Exalogic Elastic Cloud: System Overview," *White Paper*, 2011.

[17] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *International Conference on Parallel Processing (ICPP)*, Sept 2011.

[18] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg, "Providing a cloud network infrastructure on a supercomputer," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 385–394. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851534

[19] "Memcached: High-Performance, Distributed Memory Object Caching System," http://memcached.org/.

[20] Top500 Supercomputing System, http://www.top500.org.

[21] X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. K. Panda, "High Performance Pipelined Process Migration with RDMA," in *The Proceedings of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CA, USA, May 23-26 2011.

[22] H. Subramoni, P. Lai, M. Luo, and D. K. Panda, "RDMA over Ethernet - A Preliminary Study," in *Proceedings of the 2009 Workshop on High Performance Interconnects for Distributed Computing (HPIDC'09)*, 2009.

[23] P. Balaji, H. V. Shah, and D. K. Panda, "Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth Analysis of the Memory Traffic Bottleneck," in *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT), in conjunction with IEEE Cluster*, 2004.

[24] G. Liao, X. Zhu, and L. Bhuyan, "A New Server I/O Architecture for High Speed Networks," in *The Proceedings of 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, San Antonio, Texas, February 12-16 2011.

[25] RDMA Consortium, "Architectural Specifications for RDMA over TCP/IP," http://www.rdmaconsortium.org/.

[26] J. Jose, M. Luo, S. Sur, and D. K. Panda, "Unifying UPC and MPI Runtimes: Experience with MVAPICH," in *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2010.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *The Proceedings of the ACM Symposium on Cloud Computing (SoCC 2010)*, Indianapolis, Indiana, June 10-11 2010.

[28] Apache Cassandra, http://cassandra.apache.org/.

[29] B. F. Cooper, R. Ramakrishnan, R. Sears, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," in *34th International Conference on Very Large Data Bases*, 2008.

[30] R. Stevens, A. White, P. Beckman, R. Bair, J. Hack, J. Nichols, A. Geist, H. Simon, K. Yelick, J. Shalf, S. Ashby, M. Khaleel, M. McCoy, M. Seager, B. Gorda, J. Morrison, C. Wampler, J. Peery, S. Dosanjh, J. Ang, J. Davenport, T. Schlagel, F. Johnson, and P. Messina, "A Decadal DOE Plan for Providing Exascale Applications and Technologies for DOE Mission Needs," 2010.

[31] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. Lopez, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *The Proceedings of the ACM Symposium on Cloud Computing (SOCC 2011)*, Cascais, Portugal, October 26-28 2011.

[32] T. Dory, B. Mejias, P. V. Roy, and N.-L. V. Tran, "Measuring Elasticity for Cloud Databases," in *The Proceedings of the Second International Conference on Cloud Computing, GRIDs, and Virtualization*, Rome, Italy, September 25-30 2011.

[33] Y. Shi, X. Meng, J. Zhao, X. Hu, B. Liu, and H. Wang, "Benchmarking Cloud-based Data Management Systems," in *The Proceedings of the 2nd International Workshop on Cloud Data Management*, Toronto, Ontario, Canada, October 30 2010.

[34] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services," http://www.cs.ucsb.edu/ sudipto/papers/md-hbase.pdf.

[35] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud," in *The Proceedings of the ACM Symposium on Cloud Computing (SOCC 2010)*, Indianapolis, Indiana, June 10-11 2010.

[36] D. Borthakur, J. S. Sarma, J. Gray, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apapche Hadoop Goes Realtime at Facebook," in *The Proceedings of the International Conference on Management of Data (SIGMOD'11)*, Athens, Greece, June 2011.

[37] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda, "Can High Performance Interconnects Benefit Hadoop Distributed File System?" in *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds, in Conjunction with MICRO 2010*, Atlanta, GA, December 5 2010.