2011 International Conference on Parallel Processing

Memcached Design on High Performance RDMA Capable Interconnects

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

Department of Computer Science and Engineering, The Ohio State University

{jose, subramon, luom, zhanminj, huangjia, rahmanmd, islamn, ouyangx, wangh, surs, panda}@cse.ohio-state.edu

***Abstract*—Memcached is a key-value distributed memory object caching system. It is used widely in the data-center environment for caching results of database calls, API calls or any other data. Using Memcached, spare memory in data-center servers can be aggregated to speed up lookups of frequently accessed information. The performance of Memcached is directly related to the underlying networking technology, as workloads are often latency sensitive. The existing Memcached implementation is built upon BSD Sockets interface. Sockets offers byte-stream oriented semantics. Therefore, using Sockets, there is a conversion between Memcached's memory-object semantics and Socket's byte-stream semantics, imposing an overhead. This is in addition to any extra memory copies in the Sockets implementation within the OS.**

Over the past decade, high performance interconnects have employed Remote Direct Memory Access (RDMA) technology to provide excellent performance for the scientific computation domain. In addition to its high raw performance, the memory-based semantics of RDMA fits very well with Memcached's memory-object model. While the Sockets interface can be ported to use RDMA, it is not very efficient when compared with low-level RDMA APIs. In this paper, we describe a novel design of Memcached for RDMA capable networks. Our design extends the existing open-source Memcached software and makes it RDMA capable. We provide a detailed performance comparison of our Memcached design compared to unmodified Memcached using Sockets over RDMA and 10 Gigabit Ethernet network with hardware-accelerated TCP/IP. Our performance evaluation reveals that latency of Memcached Get of 4KB size can be brought down to 12 μs using ConnectX InfiniBand QDR adapters. Latency of the same operation using older generation DDR adapters is about 20 μs. These numbers are about a factor of *four* better than the performance obtained by using 10 GigE with TCP Offload. In addition, these latencies of Get requests over a range of message sizes are better by a factor of *five* to *ten* compared to IP over InfiniBand and Sockets Direct Protocol over InfiniBand. Further, throughput of small Get operations can be improved by a factor of *six* when compared to Sockets over 10 Gigabit Ethernet network. Similar factor of *six* improvement in throughput is observed over Sockets Direct Protocol using ConnectX QDR adapters. To the best of our knowledge, this is the first such memcached design on high performance RDMA capable interconnects.

***Index Terms*—Memcached, RDMA, Infiniband, INCR, UCR**

I. Introduction

There has been a tremendous increase in interest in interactive web sites that offer social networking, and e-commerce during the last several years. Social networks and financial markets have led the way in generating a lot of dynamic data. Typically, such dynamic data is stored in databases for future retrieval and analysis. Database lookups using languages such as SQL are very expensive. Data-centers that host popular web sites must be prepared to serve millions of visitors to the web site. Due to the dynamic nature of data, it is inevitable that data that is stored in database will need to be accessed. It is hard to improve the rate of response of each database query while keeping ACID (Atomicity, Consistency, Isolation and Durability) guarantees, especially when reads are mixed with writes. A new memory caching layer, `memcached` was proposed to cache the results of previous database queries. In an environment dominated by read operations, such caching can prevent expensive database queries in the critical path. Memcached has quickly gained popularity for a diverse set of applications, including distributed file systems, such as `memcachefs` [1]. The extremely popular social networking web site Facebook is a heavy user and frequent contributor to Memcached. It was reported in 2008 that Facebook has 800 Memcached servers with up to 28 Terabytes of data in their cache [2]. It can be expected that since 2008, these numbers have gone up. The reasons behind the popularity of Memcached is its generic nature and its open-source distribution [3].

The existing open-source Memcached implementation is designed using the traditional BSD Sockets interface. While the Sockets interface provides a great degree of portability, the byte-stream model within Sockets interface entails additional processing to be able to incorporate Memcached's memory-object model. This can impose overheads. In addition, Sockets implementations internally can copy messages, resulting in further loss of performance. High performance networks and their software APIs, such as OpenFabrics [4], provide RDMA capability with memory-based semantics that fits very well with the Memcached model.

Scientific and parallel computing domains use the Message Passing Interface (MPI) as the underlying basis for most applications. Implementations of MPI, such as MVAPICH and MVAPICH2 [5] can achieve very low one-way latencies in the range of 1-2μs. Whereas, even the best implementation of Sockets on InfiniBand achieve 20-25μs one-way latency. MVAPICH and MVAPICH2 are specifically designed for leveraging RDMA for InfiniBand and 10 Gigabit Ethernet/iWARP networks using the OpenFabcrics APIs.

In this paper, we propose a novel design of Memcached for RDMA capable networks. Our design extends the existing open-source Memcached software and makes it RDMA capable. We provide a detailed performance comparison of our Memcached design compared to unmodified Memcached using Sockets over RDMA and 10 Gigabit Ethernet network

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.37

IEEE computer society

with hardware-accelerated TCP/IP.

A. Motivation

During the last decade, commodity High-performance RDMA capable interconnects have emerged in the scientific computation domain. InfiniBand [6] and 10 Gigabit Ethernet have led the resurgence of commodity interconnects. MPI software already utilizes the strengths of RDMA to a great extent. At the same time, applications using Memcached are still relying on Sockets. Performance of Memcached is critical to most of its deployments. For example, there has been significant work by Facebook engineers to improve its performance [7].

A question that naturally arises is: *Can Memcached be re-designed from the ground up to utilize RDMA capable networks?*

As data-center middle-ware and runtimes become widely used, there is a considerable amount of standardization in the industry. For example, the Apache Hadoop project [8] is an outstanding example of a software stack that is standardized and widely used by many applications. In the past, there was no such standardized software. Leveraging RDMA in data-center middle-ware meant that *every* application needed to be re-designed to use new networking APIs instead of Sockets (and lose portability). This was prohibitive to the adoption of RDMA. However, the adoption of standardized software such as Memcached has opened new doors for RDMA to be adopted in the data-center environment. Most applications now depend on standardized middle-ware APIs such as Memcached, Hadoop etc., not Sockets API directly. If key performance critical run-times can be re-designed to leverage RDMA internally, while keeping the same external APIs; then many applications can immediately benefit from it without requiring any modifications. We believe that this is a strong motivation to re-design Memcached to leverage RDMA and pass on the benefit to latency critical applications.

B. Contributions

There are several contributions of this work:

1) A thorough performance evaluation of existing Memcached software on High Performance networks, such as InfiniBand and 10 Gigabit Ethernet using IP-layer over InfiniBand, Sockets on InfiniBand, and TCP-offload engine (TOE) for 10 Gigabit Ethernet.
2) A novel communication library called Unified Communication Runtime (UCR) to enable data-center middleware to effectively use features of High performance RDMA capable networks.
3) A new design of Memcached using UCR to dramatically improve performance when using RDMA capable networks. This new design of Memcached is based on existing open-source distribution, and extends it for commodity high performance networks. We believe that this is the first known design of Memcached that leverages RDMA from high performance commodity interconnects.

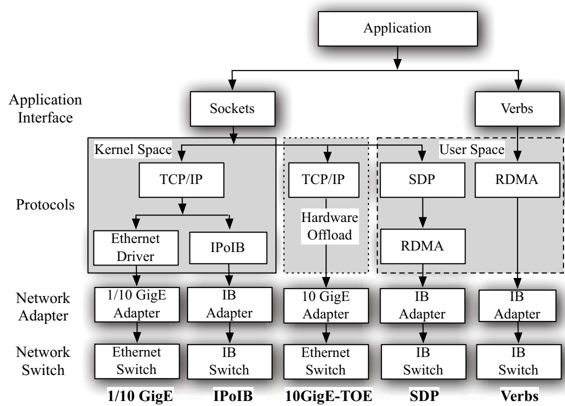
Our work is based on the Unified Communication Runtime (UCR). UCR was first proposed in [9], (previously called INCR). The goal of the UCR project is to design a communication runtime that can support parallel programming models such as MPI, Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC), in addition with distributed communication models, such as Memcached. We believe that there is significant overlap between the high performance computing domain and distributed data-center runtime requirements when it comes to leveraging RDMA technologies.

The UCR project draws from the design of MVAPICH and MVAPICH2 software. MVAPICH and MVAPICH2 [5] are very popular implementations of the MPI-1 and MPI-2 specifications. They are being used by more than 1,580 organizations in 61 countries. In addition, they are distributed by popular InfiniBand software stacks, such as OpenFabrics Enterprise Distribution (OFED) as well as Linux distributions like RedHat and SuSE. It is used on top InfiniBand clusters, such as 7th ranked NASA Pleiades (111,104 cores) and 17th ranked TACC Ranger (62,976 cores). The design of UCR reuses previous research findings from [10], [11] among many others. These are the same scalability enhancements that have enabled MVAPICH and MVAPICH2 to scale to very large InfiniBand clusters by reducing buffer consumption.

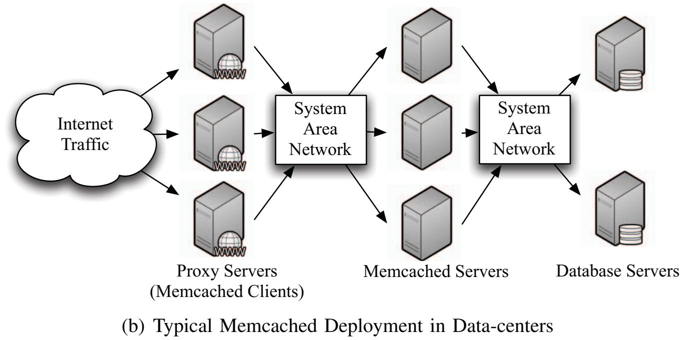
UCR exposes an easy to use Active Message based API that can be used by Memcached, without re-implementing performance critical logic (like buffer management, flow control, etc.). These are shared with other high-performance runtimes, such as those of MPI or PGAS.

We provide a detailed performance comparison of our Memcached design compared to unmodified Memcached using Sockets over RDMA and 10 Gigabit Ethernet network with hardware-accelerated TCP/IP. Our performance evaluation reveals that latency of Memcached Get of 4KB size can be brought down to 12 μs using ConnectX InfiniBand QDR adapters. Latency of the same operation using older generation DDR adapters is about 20 μs. These numbers are about a factor of *four* better than the performance obtained by using 10 GigE with TCP Offload. In addition, these latencies of Get requests over a range of message sizes are better by a factor of *five* to *ten* compared to IP over InfiniBand and Sockets Direct Protocol over InfiniBand. Further, throughput of small Get operations can be improved by a factor of *six* when compared to Sockets over 10 Gigabit Ethernet network. Similar factor of *six* improvement in throughput is observed over Sockets Direct Protocol using ConnectX QDR adapters.

The rest of the paper is organized as follows. In Section II, we provide the necessary background for this work. Section III describes previous work in this domain and distinguishes contributions of our work. The design of UCR is presented in Section IV and the new design of Memcached using UCR are described in Section V. Performance results are presented in Section VI. The paper concludes in Section VII.



(a) Networking Layers, OS-Bypass and Hardware Offload, courtesy [12]



(b) Typical Memcached Deployment in Data-centers

Fig. 1. Overview of High-Performance Networking Stacks and Memcached

II. Background

In this Section, we provide a "bottom-up" overview of networking technologies that can be utilized in a data-center for high-performance communication. We also provide background on Memcached.

A. *InfiniBand Overview*

InfiniBand [6] is an open industry standard switched fabric that is designed for interconnecting nodes in HEC clusters. It is a high-speed, general purpose I/O interconnect that is widely used by scientific computing centers world-wide. The recently released TOP500 rankings 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 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. InfiniBand has started making inroads into the commercial domain with the recent convergence around RDMA over Converged Enhanced Ethernet (RoCE) [13]. InfiniBand offers various software layers through which it can be accessed. They are described below:

1) InfiniBand Verbs Layer: The *verbs* layer is the lowest access layer to InfiniBand. It is illustrated in the Figure 1(a) (extreme right). Verbs are used to transfer data and are completely OS-bypassed, i.e. there are no intermediate copies in the OS. The verbs interface follows the Queue Pair (or communication end-points) model. Each queue pair has a certain number of work queue elements. Upper-level software using verbs can place a work request on a queue pair. 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 [4] is the most popular verbs access layer due to its applicability to various InfiniBand vendors.

2) 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. This interface is presented in Figure 1(a) (second from the left, named IPoIB). 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). The connected mode offers better performance since it leverages reliability from the hardware. In this paper, we have used connected mode IPoIB, which has better point-to-point performance.

3) InfiniBand Sockets Direct Protocol: Sockets Direct Protocol (SDP) is a byte-stream transport protocol that closely mimics TCP sockets stream semantics. It is illustrated in Figure 1(a) (second from the right, named SDP). It is an industry-standard specification that utilizes advanced capabilities provided by the network stacks to achieve high performance without requiring modifications to existing sockets-based applications. SDP is layered on top of IB message-oriented transfer model. The mapping of the byte-stream protocol to the underlying message-oriented semantics was designed to transfer application data by one of two methods: through intermediate private buffers (using buffer copy) or directly between application buffers (zero-copy) using RDMA. Typically, there is a threshold of message size above which zero-copy is used. Using SDP, complete OS-bypass can be achieved. For the sake of simplicity, the figure only shows the RDMA implementation of SDP. SDP can also be used in buffered mode for small messages. While SDP offers better performance than IPoIB due to its capability to leverage RDMA, it still does not match the performance of verbs. This is due to the mismatch of semantics between the sockets interface and the much lower-level, but higher performance verbs.

B. 10 Gigabit Ethernet Overview

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 [14]. Towards that effort, iWARP standard was introduced for performing RDMA over TCP/IP [15]. iWARP is very similar to the verbs layer used by InfiniBand, with the exception of requiring a connection manager. In fact, the OpenFabrics [4] network stack provides a unified interface for both iWARP and InfiniBand. 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. Figure 1(a) shows this option (in the middle, named 10 GigE-TOE). The benefits of TOE are to maintain full socket streaming semantics and implement that efficiently in hardware. We used 10 Gigabit Ethernet adapters (T3) from Chelsio Communications for this study.

Convergence of Fabrics: Recently, 10 Gigabit Ethernet and InfiniBand are witnessing a convergence. In particular, InfiniBand adapters from Mellanox can be configured to run on Ethernet networks. The software stack by OpenFabrics also provides a unified environment for both networks to support same applications without any code changes. Protocols focusing on this convergence will be part of our future study.

C. Memcached Overview

During the past several years, there has been a tremendous increase in Internet traffic that relates to dynamic data. Social networks and financial markets have led the way in generating this kind of dynamic data. Typically, such dynamic data is stored in databases for future retrieval and analysis. It is hard to scale databases while keeping ACID guarantees. A new memory caching layer, `memcached` was proposed by Fitzpatrick [16] to cache database request results. Memcached was primarily designed to improve performance of the web site LiveJournal. However, due to its generic nature and open-source distribution [3], it was quickly adopted in a wide variety of environments. Using Memcached, spare memory in data-center servers can be aggregated to speed up lookups of frequently accessed information, be it database queries, results of API calls or web page rendering elements. A typical deployment of Memcached is shown in Figure 1(b). Generally, the system area network used in current-generation Memcached deployments is Ethernet. However, using either IPoIB or Sockets Direct Protocol (described in Section II-A2 and II-A3), it can be used over InfiniBand without any modifications.

Memcached is a key-value distributed memory store. Memcached clients can store and retrieve items from servers using "keys." These keys can be any character strings. Typically, keys are MD5 sums or hashes of the objects being stored/fetched. The identification of the destination server is done at the client side using a hash function on the key. Therefore, the architecture is inherently scalable as there is no "central server" to consult while trying to locate values from keys.

The performance of Memcached is directly related to that of the underlying networking technology. Workloads that depend on Memcached are extremely latency sensitive and need to serve many clients simultaneously. Even though the underlying architecture of Memcached is scalable, the implementation needs careful attention to be able to scale to thousands of clients accessing data simultaneously (i.e. data that is hot). Previous work relating to scaling up Memcached is described in Section III.

III. Relation to Previous Work

In this Section, we compare our work with previous work on this topic. Facebook has been a very important user of Memcached software for its back-end services. As Facebook is an extremely popular web site, the traffic handled by its data-centers is very high. In particular, the ability of Memcached servers to handle large numbers of clients is critical. In order to scale up the number of clients and performance of Memcached, Facebook proposed a UDP version of Memcached [7]. Using their changes, Memcached was able to handle up to 200,000 UDP requests per second with an average latency of 173 μs. The maximum throughput can be up to 300,000 UDP requests/s, but the latency at that request rate is too high to be useful in their system. No details as to request size were released. In this work, we show that using our version of Memcached on RDMA capable networks, the latency is around 12 μs and request rates are in Millions per second.

Appavoo et. al. have presented their work on providing cloud computing services on the Blue Gene supercomputer [17]. In their work, they have re-designed memcached to use RDMA features from the Blue Gene network. Our work differs significantly from their effort. First, our work is based on the open-source version of Memcached and can work on cheap commodity data-centers exploiting RDMA. Secondly, their work assumes that Memcached clients know the virtual address of data on the server and can use RDMA to fetch it. They accomplish this by splitting the server's hash table and storing it in the client. However, in a real deployment, it is not possible to do this even when memory consistency models allow it. This is due to the fact that the server can consolidate data into memory slabs to avoid fragmentation (without informing clients). Also, items in Memcached servers have expiration time limits. Directly reading data may result in corrupt data. Finally, and most importantly, there may be billions of items in the aggregate Memcached store. Clients are typically run on low-end commodity machines with low amount of memory available. Memory is one of the most expensive resources in a data-center environment. It is impossible to meaningfully cache all the item addresses without wasting a lot of memory. We believe that our design of Memcached using UCR is a generalized extension of the Memcached software that is designed to work on commodity platforms across a wide range of workloads.

Balaji et. al. have worked on improving the performance of SDP on InfiniBand [18]. They report that the latency

and bandwidth benefits of InfiniBand are beneficial in the data-center environment. At the same time, Dror et. al. from Mellanox have proposed asynchronous zero-copy SDP [19]. The software from this work is distributed in the OpenFabrics stack. We use this software as a part of our evaluation. We have in the past explored the benefits of SDP on Hadoop software [12]. Hadoop is an open-source implementation [8] of the very popular Map-reduce programming model proposed by Dean et. al. of Google Inc. [20].

Even though SDP provides some degree of higher performance to Memcached, the fundamental limitation is that of the mismatch of Sockets interface and the underlying hardware for high performance RDMA capable networks. Our work not only presents a native verbs based design of Memcached, but through our evaluations, we prove that using verbs, much better performance can be achieved through Sockets over InfiniBand.

Vaidynathan et. al. have proposed various high-performance distributed computing substrates [21]. They concluded that using verbs, data-center middle-ware can more effectively leverage RDMA capable network architectures. However, their work did not target Memcached, and designed low-level substrates, like caching and distributed lock management.

IV. Designing Unified Communication Runtime (UCR) for Data-Center Environments

The Unified Communication Runtime (UCR) aims to unify the communication runtime requirements of scientific parallel programming models, such as MPI and Partitioned Global Address Space (PGAS) along with those of data-center middleware, such as Memcached. Our approach is to re-use the best communication runtime designs across these domains.

The requirements imposed by these disparate models are quite challenging. For example, in the parallel computing domain, there is a notion of a parallel "job." Processes belonging to a job may communicate with others using a rank or a thread id (in case of PGAS). This is quite different from the Memcached model, where clients maintain a list, or pool, of available Memcached servers. Based on the data they want to access, keys are assigned to the data and a server is chosen using a hash function. Another important distinction is that of fault tolerance. In MPI or PGAS, when a process belonging to a job unexpectedly fails, the entire job fails. However, in the data-center domain, failure of one Memcached server or client must be tolerated.

There are several new contributions to UCR made as a part of this work. They include a fault-tolerant model (one failing process doesn't fail others), a revamped active message API, and active message counters. They are described in detail below:

A. Communication Model and Connection Establishment

As discussed above, the communication model for Memcached does not allow for one failing process to impact others. Based on this requirement, we have re-designed the connection establishment process in UCR. Previously, UCR only accepted destination ids (such as ranks in MPI or thread ids in PGAS). The new version of UCR follows an end-point model. The client must establish an end-point with the server before communication begins. Each end-point is bi-directional, i.e. the server can talk back to the client using the same end-point. The client has a choice of the type of end-point that can be used (reliable vs. unreliable). This is similar to either TCP or UDP sockets. For the purposes of this paper, we restrict our discussion to reliable connections only due to space constraints.

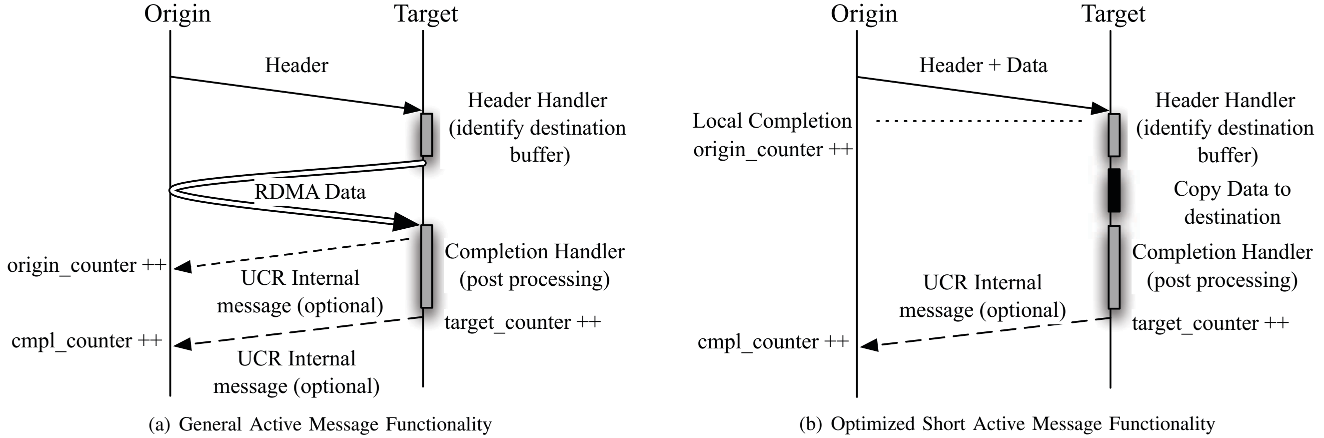
Another addition to the UCR API is that of synchronization with timeouts. In MPI or PGAS, an erroneous process (either software or hardware failure) can result in the job hanging. This is not permitted in the data-center environment. Accordingly, we have introduced UCR methods to wait for events, but with specified timeout values. If events do not occur within that time limit, then Memcached can choose to take corrective action. For example, a client may decide that a server has gone down. While this is possible in the UCR model, in this paper, however, we do not evaluate such situations due to space constraints. This paper focuses on the core issue of performance improvements in Memcached using UCR.

B. Active Messaging in UCR

Active messages were first introduced by Von Eicken et. al. [22]. They have proven to be very useful in many environments, such as PGAS library design, e.g. GASNet project from U. C. Berkeley [23] and MPI design using the Low-Level API (LAPI) from IBM [24]. They were never considered in the data-center environment in the past. To the best of our knowledge, this is the first application of Active Messaging in commodity RDMA capable networks in the data-center domain.

UCR provides interfaces for Active Messages as well as one-sided put/get operations. The basic operation of Active Messages in UCR is illustrated in Figure 2. The process that initiates communication, is called the *Origin*, the destination is called the *Target*. An example of the general working of active messages in UCR is shown in Figure 2(a). An active message consists of two parts: **header** and **data**. When an active message arrives at the target, a 'header handler' is run. The header handler is a short function that can perform some limited logic and identify the destination buffer of the data. Then, UCR can transfer the data to the identified location using RDMA operations. Once the data has been put, a 'completion handler' is run at the target, which can perform any operation on the data. It should be noted that the choice to run completion handler is also optional and depends on the logic in the header handler. The associated counters and 'optional messages' are explained in the following section.

As an optimization for short messages, header and data may be combined into one network transaction. In that case, instead of RDMA operation, the data can be simply copied off the network buffer into destination location. An example of the active Message API in UCR is as shown below:



(a) General Active Message Functionality

(b) Optimized Short Active Message Functionality

Fig. 2. Active Messaging in UCR

```
int ucr_send_message( ucr_ep_t *ep, int msg_id, void *hdr,
size_t hdr_len, void *data, size_t data_len, ucr_cntr_t *ori-
gin_counter, ucr_cntr_t *target_counter, ucr_cntr_t *comple-
tion_counter )
```

Arguments:`ep` identifies the target, `msg_id` identifies the header handler to be run on the target, `hdr, hdr_len, data, data_len` indicate the header and data to be sent. The counters are explained in the following section.

C. Active Message Counters

Counters provide a generalized and clean way to track progress of active messages. Counters are objects with monotonically increasing values. There are three types of counters: `origin_counter`, `target_counter` and `completion_counter`. The `origin_counter` and `completion_counter` belong to the origin, whereas the `target_counter` belongs to the target.

Origin counter: It is incremented at the origin when the memory containing header and data can be re-used by the application (e.g. Memcached). In the case of large active messages, once the RDMA of the data is complete, a message is sent to the initiator side that results in the counter being updated. This is shown in Figure 2(a). If the origin counter supplied to ucr_send_message is `NULL`, then this extra message is not issued by UCR. For short messages, UCR can just look for local completion and update the counter. Local completion here means, when the network layers indicate that buffer can be safely re-used. This may mean that the message is internally copied into a buffer. However, it is safe to re-use the application level buffer. **Target counter**: It is incremented at the target when the data has completely arrived and completion handlers have run. **Completion counter**: It is incremented at the origin when the completion handler at the target side is complete. Note that if the supplied value of completion_counter is `NULL`, then UCR will not issue the optional internal message.

V. Designing Memcached Using UCR

In this section, we describe our design of Memcached using the UCR active message APIs in detail. The active message concepts in UCR were explained in the previous section. In this section, we use the active message concepts to illustrate the design of Memcached. We focus on the *set* and *get* operations. These are the fundamental operations included in Memcached APIs. Other operations such as: mget, increment, decrement and delete can be implemented using these principles. We used Memcached server version 1.4.5 [3] and client C library (libmemcached) version 0.45 [25].

A. Connection Establishment

Memcached relies on `libevent` for detecting sockets which have become ready for communication. Based on which socket is ready for communication, it is handed over to a server thread. We maintain this overall model with some modifications. The goal of our design is to maintain compatibility of the existing Memcached server to work with both Sockets based clients and UCR based clients. In our version of Memcached, the connection establishment process begins as normal. However, the UCR clients indicate UCR compatibility. UCR client request causes a libevent notification at the server side and a worker thread is assigned (in a round-robin manner) for this client. Client and worker thread both create end points and start communication using active messages over UCR. All further requests from client will be served by this worker thread. Note that a worker thread can handle several clients at a time. The number of worker threads can be set using a runtime parameter. The underlying UCR calls are always non-blocking and asynchronous.

B. Set Operations

Using the set operation, a Memcached client can put an object (item) in Memcached server memory. Each item is assigned a key, which can be a hash value computed from its contents. However, there is no restriction on how the key is chosen: it is up to the client. Once the key is computed, using another hash operation, a target server is chosen from the pool of available servers. Then, the set operation is issued.

First, the client issues an active message to the server with the set request (AM 1). The basic operation of active messages in UCR was described in Section IV. This active message can be large (depending on size of item being stored). The

client also indicates a counter C. After the server parses the command from the client, it identifies where it wants to store the item. Then, it issues an RDMA Read to that destination memory location. When the store is complete, it indicates the status of the completion back to the client using another active message (AM 2). Meanwhile, the client is waiting on counter C to be incremented. This is a blocking call with client specified timeout. The server indicates this counter C as the "target counter" in AM 2. AM 2 is typically a short message, as it only contains the status. When it completes at the target (which is the Memcached client), counter C is incremented. The Memcached client then knows that the server has responded with an answer. It then inspects the contents of the message received and takes appropriate actions.

C. Get Operations

Using the get operation, a Memcached client can get an object (item) from a Memcached server. The key of the item indicates which server it may be stored in (through a hash function). Once the server is identified, the get operation is issued.

As with the case in set, the client indicates the counter C it is waiting on in the first active message (AM 1). Once the server gets the command from the client, it finds the requested item. In standard Memcached API, the length of the item is not known by the client before-hand. It is known only at the server. The server then responds with active message AM 2 with the Memcached client as the target and counter C as the "target counter." Once the client learns the size of the item being sent by the server, it allocates required memory (it may implement a local buffer pool). Then, the item is read into the destination buffer. When data is available, UCR increments the target counter, C. The client then knows that the get operation is complete.

Note on Small Set/Get operations: As described in Section IV, UCR is optimized for small data. In the case the amount of data being set or get fits into one network buffer (8 KB), it is packaged within one transaction. In this case, no RDMA calls are used, and memcpy is used at the target to copy data into destination buffers.

VI. Performance Evaluation

We describe the experimental setup and the results of our performance analysis of Memcached in this section. Our benchmarks are inspired by the popular `memslap` benchmark that is distributed along with the Memcached client library. This benchmark measures latency and operations per second for varying mixes of Set and Get operations. Unfortunately, this benchmark does not utilize the libmemcached API itself. Rather, it directly sends messages using Sockets. Therefore, we created our suite of benchmarks that perform similar evaluation, but use the standard libmemcached C API. We used Memcached server version 1.4.5 and client (libmemcached) version 0.45. The client behavior was set using: `memcached_behavior_set(memc, MEMCACHED_BEHAVIOR_TCP_NODELAY, 1)`. This resulted in better and more predictable latency performance.

A. Experimental Setup

We used two different clusters for our evaluation.

Intel Clovertown 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, a PCIe 1.1 interface. The nodes are 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. The operating system used is Red Hat Enterprise Linux Server release 5.5 (Tikanga), with kernel version 2.6.30.10 and OpenFabrics version 1.5.1. 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.

Intel Westmere 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 (36 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with kernel version 2.6.18-164.el5 and OpenFabrics version 1.5.1.

Note Regarding Interpreting Results: It is to be noted that the performance improvements in Memcached performance seen with our UCR based design are mainly due to OS-bypass and the match in memory-oriented nature of RDMA semantics as opposed to stream oriented socket semantics. UCR provides a good design of active messaging on top of OpenFabrics verbs. We may expect to see good gains in performance with the iWARP/RoCE implementations of UCR that will run over a 10 GigE network. We are currently in the process of designing the iWARP and RoCE versions of UCR.

For all SDP related experiments zero-copy was turned off as the existing sockets based Memcached code requires sockets to be in non-blocking mode while the default SDP implementation shipped with OFED does not work in non-blocking mode with zero copy [19]. Without turning off zero copy, Memcached with SDP crashes with an error.

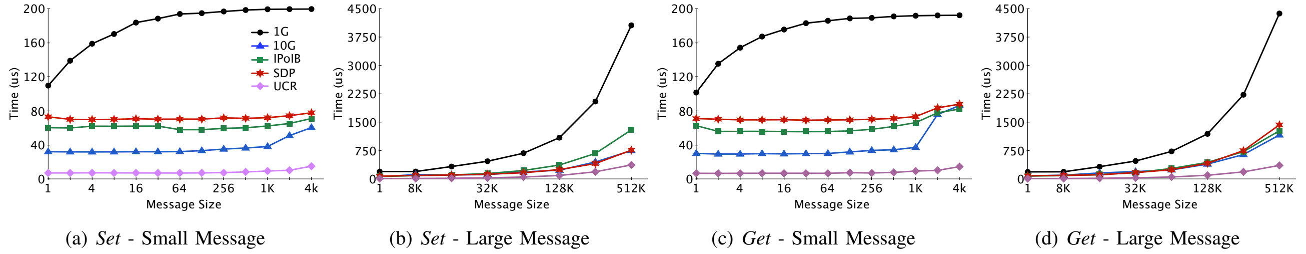
B. Performance of Memcached with Single Client

In this section, we look at the single client performance with Memcached for various operations over different network interconnects.

1) Performance of Set and Get: The first set of experiments measure the performance for set and get operations. In these experiments, the Memcached client repeatedly sets (or gets) a particular size of item from the cache. i.e. the Memcached instruction mix is either 100% Set or 100% Get.

Performance on Cluster A: Data for Set and Get operations are shown in Figure 3. We observe that UCR based design outperforms 10 GigE with TCP offload by at least a factor of *four* for all message sizes. We can also see that the UCR based design outperforms SDP and IPoIB by a factor of *eight* or more for small to medium message sizes and by a factor of *five* for large message sizes.

Performance on Cluster B: Performance of Set and Get operations are shown in Figure 4. We observe that, the UCR



(a) *Set* - Small Message (b) *Set* - Large Message (c) *Get* - Small Message (d) *Get* - Large Message

Fig. 3. Latency of *Set* and *Get* Operations on Cluster A

based design outperforms SDP and IPoIB based options by at least a factor of *ten* for all small message sizes and up to a factor of *four* for large message sizes. Due to lack of 10 GigE cards on this cluster, we were not able to present the corresponding 10 GigE numbers. We observed that the SDP results on this cluster were noisy. We made several attempts to reduce the jitter by increasing the number of samples taken in the experiments (up to 10,000). However, the jitter did not subside. We conclude that this must be an implementation artifact of SDP on QDR adapters. The fact that IPoIB and UCR results are jitter free indicates that it is not a problem of the underlying network itself.

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

C. Performance of Mixed Set and Get

The second set of experiments mix the instruction set of Memcached operations. We use a mix of 10% Set operations and 90% Get operations. The pattern of access is 10 Sets followed by 90 Gets. We call this pattern non-interleaved. We also evaluate an interleaved pattern in which the instruction mix is 50% Set operations and 50% Gets. In this pattern, 1 Set is followed by 1 Get. We restrict the presented data to small messages due to space limitations.

Performance on Cluster A: Data for mixed operations is shown in Figures 5(a) and 5(c) for the non-interleaved and interleaved instruction mix described above. We can see that these mixed operations follow the same trends as the basic Set and Get operations. We see that the UCR based design outperforms 10 GigE with TCP offload by at least a factor of *four* for all message sizes. We can also see that the UCR based design outperforms SDP and IPoIB by a factor of *seven* or more for small to medium message sizes and by a factor of *four* for large message sizes.

Performance on Cluster B: Data is shown in Figures 5(b) and 5(d). We can see that these mixed operations follow the same trends as the Set and Get operations seen above. We see that the UCR based design outperforms the SDP and IPoIB based designs by a factor of 10 or more for small to medium message sizes and by a factor of four for large message sizes.

D. Performance of Memcached with Multiple Clients

In this section, we look at the performance of Memcached with multiple clients over different network interconnects. We restrict our evaluation to Get operations due to space limitations. The multi-client benchmark is an extension of the single client benchmark discussed in Section VI-B. All the clients are started simultaneously. Instead of latency, we report the total number of transactions that were executed per second. The number reported is the aggregate transactions per second observed by all the clients. This benchmark is expected to simulate the case where many clients access the same Memcached server simultaneously.

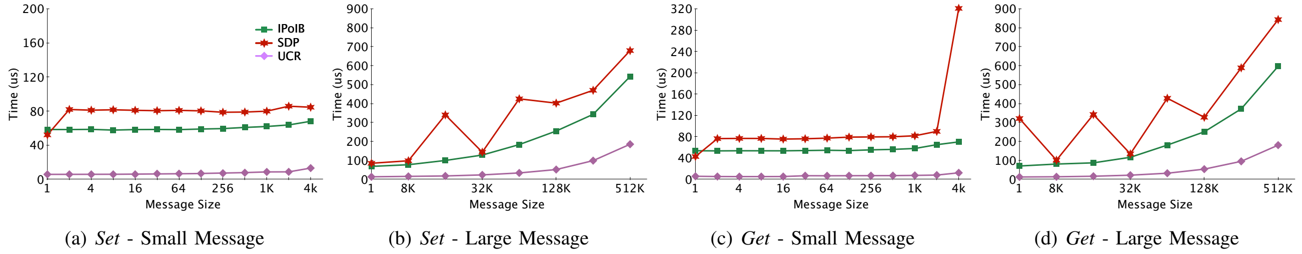
We vary the clients from eight to sixteen. The clients are all located on different nodes than the Memcached server. The results for two message sizes (4 Byte and 4KB) are presented in Figure 6. On Cluster A, we observe that the transactions per second achieved by Memcached over UCR is a factor of *six* better than that of 10 Gigabit Ethernet with TCP acceleration for small transactions. The 10 Gigabit TOE outperforms SDP over InfiniBand. On Cluster B, we can observe that the improvements in transactions per second is quite significant, a factor of *six* over the corresponding SDP version for 4 byte messages. The transactions per second is around 1.8 Million operations/sec with UCR on QDR adapters. We observe that the SDP performance is lower than that of IPoIB. This is similar to our latency observations in Figure 4. We believe that this could be a software issue with SDP.

VII. Conclusion and Future Work

In this paper, we have described a novel design of Memcached for RDMA capable networks. Our design extends the existing open-source Memcached software and makes it RDMA capable.

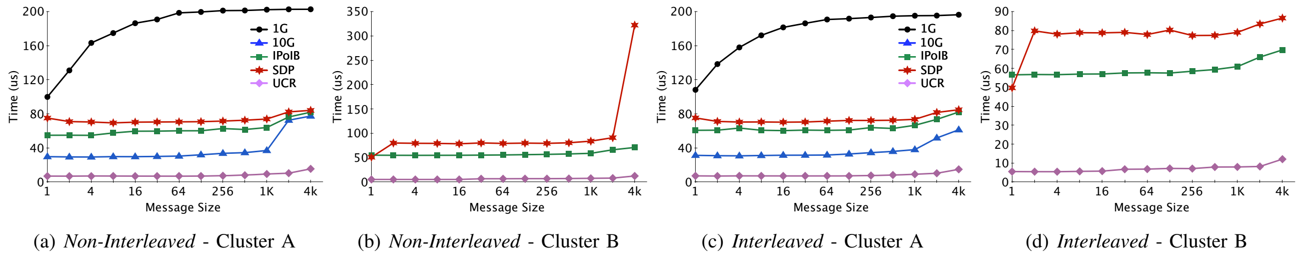
We provided design details of our Memcached implementations underlying communication layer – Unified Communication Runtime (UCR). UCR provides an easy to use Active Message API that matches very well with requirements of Memcached. We have significantly enhanced our UCR design to suit a wide range of needs from high performance communication runtimes.

We have provided a detailed performance comparison of our Memcached design compared to unmodified Memcached using Sockets over RDMA and 10 Gigabit Ethernet networks with hardware-accelerated TCP/IP. Our performance evaluation shows that latency of Memcached Get of 4KB size can be brought down to 12 μs using ConnectX InfiniBand QDR adapters. Latency of the same operation using older generation DDR adapters is about 20 μs. These numbers are about a factor of *four* better than the performance obtained by using 10 GigE with TCP Offload. In addition, these latencies of Get requests over a range of message sizes are better by a factor of *five*



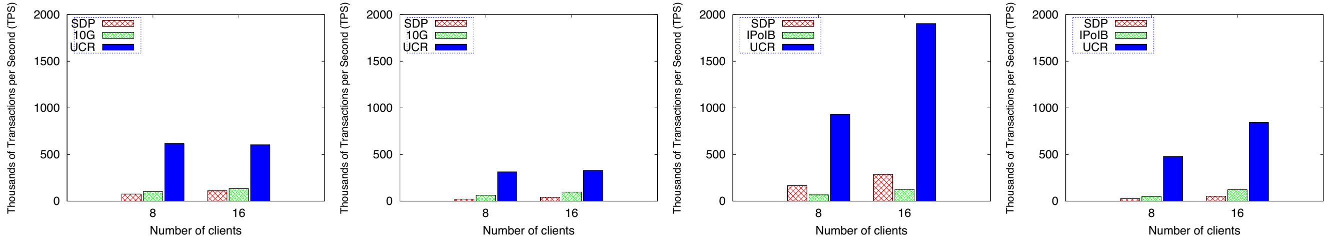
(a) *Set* - Small Message (b) *Set* - Large Message (c) *Get* - Small Message (d) *Get* - Large Message

Fig. 4. Latency of *Get* and *Set* Operations on Cluster B



(a) *Non-Interleaved* - Cluster A (b) *Non-Interleaved* - Cluster B (c) *Interleaved* - Cluster A (d) *Interleaved* - Cluster B

Fig. 5. Latency of Small Messages for Non-Interleaved (Set 10% Get 90%) and Interleaved (Set 50% Get 50%) Operations



(a) Message size: 4 byte - Cluster A (b) Message Size: 4096 byte - Cluster A (c) Message size: 4 byte - Cluster B (d) Message Size: 4096 byte - Cluster B

Fig. 6. Number of Transactions per Second for *Get* Operation

to *ten* compared to IP over InfiniBand and Sockets Direct Protocol over InfiniBand. Further, throughput of small Get operations can be improved by a factor of *six* when compared to Sockets over 10 Gigabit Ethernet network. Similar factor of *six* improvement in throughput is observed over Sockets Direct Protocol using ConnectX QDR adapters.

We plan to continue working along this line in the future. Our plans are to improve the UCR runtime further by taking into account the many features provided by OpenFabrics API, especially for InfiniBand. For example, we aim to leverage the Unreliable Datagram transport to scale up the total number of clients that can be handled by a single server. Apart from this, we are currently in the process of designing the iWARP and RoCE versions of UCR.

VIII. Acknowledgments

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0621484, #CCF-0833169, #CCF-0916302; #OCI-0926691 and #OCI-0937842; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from IBM, Intel, Mellanox, AMD, Appro, Chelsio, Dell, Fujitsu, Fulcrum, Microway, Obsidian, QLogic, and Sun Microsystems.

The authors would like to thank Dr. Chet Murthy of IBM Research for many useful discussions on this topic and his constant encouragement.

References

[1] "memcachefs: a memcache filesystem using FUSE," http://memcachefs.sourceforge.net/.

[2] Todd Hoff, "Facebook's Memcached Multiget Hole: More Machines != More Capacity," http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html.

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

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

[5] MVAPICH2: High Performance MPI over InfiniBand, iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.

[6] InfiniBand Trade Association, http://www.infinibandta.org.

[7] Paul Saab, "Facebook Engineering Notes: Scaling Memcached at Facebook," https://www.facebook.com/note.php?note_id=39391378919.

[8] The Apache Software Foundation, "The Apache Hadoop Project," `http://hadoop.apache.org/`.

[9] 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 '10)*.

[10] M. Koop, S. Sur, Q. Gao, and D. K. Panda, "High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters," in *21st ACM Int'l Conference on Supercomputing (ICS07)*, June 2007.

[11] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda, “Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters,” in *IEEE Int’l Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006.

[12] 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*, Atlanta, GA, 2010.

[13] 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)*.

[14] 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.

[15] RDMA Consortium, “Architectural Specifications for RDMA over TCP/IP,” http://www.rdmaconsortium.org/.

[16] B. Fitzpatrick, “Distributed caching with memcached,” *Linux Journal*, vol. 2004, August 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1012889.1012894

[17] J. Appavoo, A. Waterland, and e. a. Da Silva, “Providing a cloud network infrastructure on a supercomputer,” in *Proceedings of the 19th ACM Int’l Symposium on High Performance Distributed Computing*, ser. HPDC ’10. New York, NY, USA: ACM, 2010, pp. 385–394.

[18] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, “Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?” in *The Proceedings of the IEEE Int’l Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 10-12 2004.

[19] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, “Transparently Achieving Superior Socket Performance using Zero Copy Socket Direct Protocol over 20 Gb/s InfiniBand Links,” in *in 2005 IEEE Int’l Conference on Cluster Computing (Cluster)*, 2005, pp. 1–10.

[20] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*. USENIX Association, 2004.

[21] K. Vaidyanathan, S. Narravula, P. Balaji, and D. K. Panda, “Designing Efficient Systems Services and Primitives for Next-Generation Data-Centers,” in *Workshop on NSF Next Generation Software(NGS) Program; held in conjunction with IPDPS*, 2007.

[22] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: a mechanism for integrated communication and computation,” *SIGARCH Comput. Archit. News*, vol. 20, pp. 256–266, April 1992. [Online]. Available: http://doi.acm.org/10.1145/146628.140382

[23] Editor: Dan Bonachea, “GASNet specification v1.1,” U. C. Berkeley, Tech. Rep. UCB/CSD-02-1207, 2008.

[24] IBM, “Message Passing Interface and Low-Level Application Programming Interface (LAPI),” `http://www-03.ibm.com/systems/software/parallel/index.html`.

[25] “libmemcached: Open Source C/C++ Client Library and Tools for Memcached,” http://libmemcached.org/.