Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store

Christopher Mitchell Yifeng Geng* Jinyang Li
New York University **Tsinghua University*
{cmitchell, ygeng, jinyang}@cs.nyu.edu

Abstract

Recent technological trends indicate that future datacenter networks will incorporate High Performance Computing network features, such as ultra-low latency and CPU bypassing. How can these features be exploited in datacenter-scale systems infrastructure? In this paper, we explore the design of a distributed in-memory key-value store called Pilaf that takes advantage of Remote Direct Memory Access to achieve high performance with low CPU overhead.

In Pilaf, clients directly read from the server's memory via RDMA to perform `gets`, which commonly dominate key-value store workloads. By contrast, `put` operations are serviced by the server to simplify the task of synchronizing memory accesses. To detect inconsistent RDMA reads with concurrent CPU memory modifications, we introduce the notion of *self-verifying* data structures that can detect read-write races without client-server coordination. Our experiments show that Pilaf achieves low latency and high throughput while consuming few CPU resources. Specifically, Pilaf can surpass 1.3 million ops/sec (90% `gets`) using a single CPU core compared with 55K for Memcached and 59K for Redis.

1 Introduction

The network implementations found in High Performance Computing (HPC) clusters have historically differed from those in datacenters in a few key aspects: low latency, low CPU overhead, and high cost. Recent trends in the networking world indicate that these distinctions are beginning to disappear as HPC network prices drop and datacenter network equipment begins to adopt features previously found only in HPC clusters. Products are already being offered that implement kernel or CPU bypassing (two common HPC network features) over 10Gbps Ethernet [30, 24], while the prices for the popular Infiniband HPC interconnect have dropped dramatically and are now competitive with 10Gbps Ethernet hardware. For example, a Mellanox 40Gbps Infiniband adapter costs ~$500, while 10Gbps Ethernet cards range in price from ~$300 to $800. Surprisingly, low-latency Infiniband switches are now less expensive than their 10Gbps Ethernet counterparts. Given these changes, it is important that we understand how to leverage the features of these high-performance networks to build general-purpose applications. In this paper, we focus on how to effectively use Remote Direct Memory Access (RDMA), a common component of high performance networking fabrics.

RDMA operations allow a machine to read (or write) from a pre-registered memory region of another machine without involving the CPU on the remote side. Compared to traditional message passing, RDMA achieves the smallest round-trip latency (~3μs), highest throughput, and lowest (zero) CPU overhead. These advantages are offset by the difficulty of incorporating RDMA into distributed system designs. In a traditional design, the server processes all service requests from clients and thus acts as a single point of coordination for memory accesses. With RDMA, clients can directly access the server's memory to implement a service request without any involvement by the server. However, without the server's coordination, races in memory accesses by different machines become a serious concern.

In this paper, we present Pilaf, a distributed in-memory key-value store that leverages RDMA to achieve high throughput with low CPU overhead. We argue that the sweet spot in the design space is to restrict the use of RDMA to read-only service requests, namely `gets`, while letting the server handle all other requests via traditional messaging. As practical key-value workloads tend to be dominated by read operations [1], this approach can capture most of RDMA's performance benefits while facilitating a much simpler design than using RDMA for all types of requests. In particular, this approach restricts the class of memory access races that can occur: clients might read inconsistent data while the server is concurrently modifying the same memory addresses.

We use *self-verifying* data structures to address read-write races between the server and clients. A self-

verifying data structure consists of checksummed root data objects as well as pointers whose values include a checksum covering the referenced memory area. Starting from a set of root objects with known memory locations, clients are guaranteed to traverse a server's self-verifying data structure correctly, because the checksums can detect any inconsistencies that arise due to concurrent memory writes done by the server. When a race is detected, clients simply retry the operation.

Other projects have also used RDMA to enhance the performance of Memcached-like key-value stores [28, 14, 13]. In these designs, RDMA is treated simply as a means for accelerating standard message-passing; the server still handles all service requests. In contrast, in Pilaf, clients can process `get` requests without involving the server process at all, resulting in optimal (zero) CPU overhead. To the best of our knowledge, Pilaf is the first system design where clients can completely bypass the server's CPU for processing read requests.

We have implemented Pilaf on top of Infiniband, a popular HPC network interconnect. Our experiments on a cluster of machines equipped with 20Gbps Infiniband cards show that Pilaf achieves high performance with very low CPU overhead. In a workload consisting of 90% `get`s and 10% `put`s, Pilaf achieves 1.3 million ops/sec while utilizing only a single CPU core, compared to 55K for Memcached and 59K for Redis.

2 Opportunities and Challenges

This section gives an overview of RDMA and other HPC networking features and discusses how they might impact the design of distributed systems. Our discussion of the performance implications is based on Infiniband, a popular HPC interconnect.

Manufactured by Intel and Mellanox, Infiniband hardware provides 10, 20, or 40 Gbps of bandwidth in each direction. Applications running on top of Infiniband have several communication options:

IP over Infiniband (IPoIB) emulates Ethernet over Infiniband. As with normal Ethernet, the kernel processes packets and copies data to application memory. IPoIB allows existing socket-based applications to run on Infiniband with no modification.

Send/Recv Verbs provide user-level message exchange: these Verbs messages pass directly between user space applications and the network adapter, bypassing the kernel. Send/Recv Verbs are commonly referred to as two-sided operations since each Send operation requires a matching Recv operation at the remote process. Unlike IPoIB, applications must be rewritten to use the Verbs API.

RDMA allows full remote CPU bypass by letting one machine directly read or write the memory of another machine without involving the remote CPU. Unlike Send/Recv Verbs, RDMA operations are one-sided, since an RDMA operation can complete without any knowledge of the remote process. RDMA is technically a type of Verbs. In this paper, we use the term RDMA specifically to refer to RDMA Verbs and the phrase verb messages to refer to Send/Recv Verbs, both of which we use in reliable mode.

We note that Infiniband is not the only network to support RDMA and user-level networking. Similar features have recently been made available in 10 Gbps Ethernet environments. For example, both Myricom and Solarflare offer 10GE adapters that support kernel bypass, and Intel offers 10GE iWARP adapters capable of RDMA over Ethernet. Although it remains unclear which specific hardware proposal will dominate the datacenter market, one can realistically expect future datacenter networks to support some form of CPU bypassing.

2.1 Performance Benefits of RDMA

How fast and efficient is RDMA? How does its performance compare to alternatives such as verb messages or traditional kernel-based TCP/IP transport? We answer these questions by benchmarking the various Infiniband communication options.

Our experiments were run on a small cluster of machines equipped with Mellanox ConnectX-2 20Gbps Infiniband cards. For RDMA experiments, each client node performs RDMA reads on the server. For verb message experiments, each client node issues a request (as a verb message in reliable mode) to which the server responds immediately with a reply. The IPoIB and Ethernet experiments are similar except that we use TCP/IP for exchanging requests and replies. We vary the size of the RDMA read or the request message while fixing the reply size at 10 bytes.

Figure 1 shows the roundtrip latencies of different communication methods. For small operations (< 256 bytes), a verb message exchange takes less than 7μs, while the RTT of IPoIB or Ethernet is over 60 μs. Our Infiniband switch imposes a lower delay than our Ethernet switch, but the IPoIB latency is similar to that of Ethernet, suggesting that packet processing through the kernel adds significant latency. RDMA achieves the lowest RTT ($\sim 3\mu$s), half that of verb messages. This is because the request/reply pattern of traditional messaging involves two underlying Verbs exchanges. By contrast, an RDMA operation involves only one underlying Verbs exchange, thereby halving the latency.

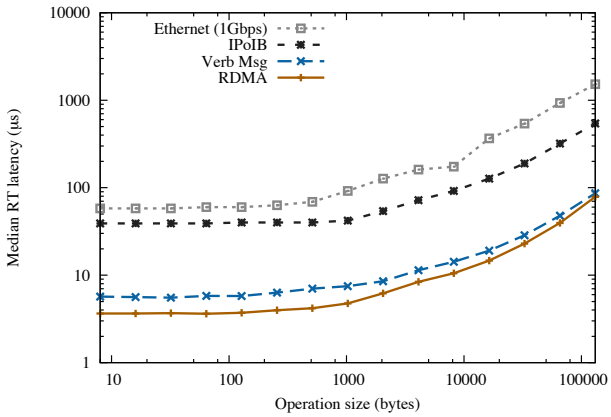


Figure 1: Median round-trip latency. The error bars depict 1% and 99% latency.

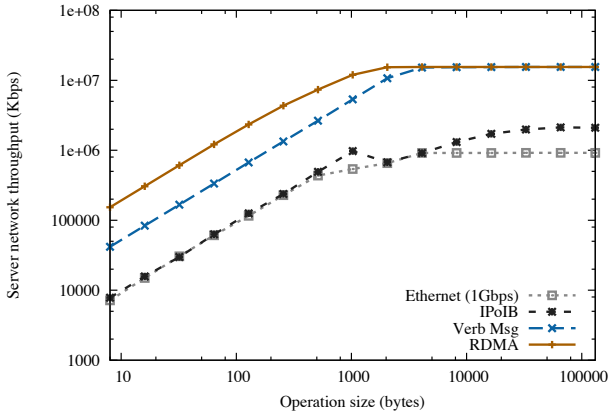


Figure 2: Server's network throughput under different communication methods.

Transport	Throughput (M ops/sec) 16-byte	1024-byte
RDMA	2.449	1.496
Verbs Message	0.668	0.668
IPoIB	0.126	0.122
Ethernet (1Gbps)	0.120	0.068

Table 1: Throughput (in million operations/sec) for 16 byte and 1Kbyte operations.

Figure 2 shows the throughput (in Kbps) achieved by the server. Since different communication methods incur varying CPU overhead, we limit the server's CPU consumption to a single core (AMD Opteron 6272) in all experiments. In Figure 2, large operations (>1024 bytes) over all communication methods except IPoIB can saturate their respective network's peak throughput. For smaller operations, both RDMA and Verbs messages are able to saturate the Infiniband network card's capacity when running the server on a single CPU core. By contrast, kernel-based transports require more than one core to saturate the network card, hence the much lower throughputs achieved in IPoIB and Ethernet experiments.

RDMA not only incurs zero CPU overhead on the server, it also saturates the network card at the highest operation throughput. As shown in Table 1, a server can sustain 2.45 million operations/second with 16-byte RDMA reads. By contrast, the server can only achieve 0.668 million operations/sec when exchanging Verbs request/reply messages. There are two reason for this performance gap. First, each request/reply exchange uses two underlying Verbs exchanges compared to one for RDMA. Second, because the card does less bookkeeping for RDMA, the achieved RDMA throughput is more than twice that of of just sending or receiving verb messages.

2.2 Opportunities for System Builders

As we have seen, bypassing the kernel and CPU allows for reduced latency and CPU overhead. Of these two, CPU bypass via RDMA is particularly powerful in that it achieves the highest throughput while incurring zero CPU overhead. As future datacenter networks embrace RDMA, how should we design datacenter infrastructures such as distributed storage systems? To better understand the ensuing opportunities and challenges, we have chosen to build a distributed key-value store to exploit RDMA. We decided to use the key-value store as a case study system because it is a popular infrastructural service with demanding performance requirements [21]. Key-value stores are also used as a building block for other more sophisticated storage systems (e.g. BigTable [2], Spanner [4], Cassandra [17]) or distributed computation frameworks (e.g. Piccolo [23]).

Our experience in exploring the design space for a key-value store leads to two observations, both of which are applicable to other distributed systems besides key-value stores.

High performance is feasible with fewer CPU resources. With traditional Ethernet-based distributed systems, the performance bottleneck is often the CPU despite the availability of multiple cores [20]. With kernel and CPU bypass, servers can saturate the network using many fewer cores. The improvement in CPU efficiency is particularly notable with RDMA, which potentially allows clients to process service requests without involving the server at all. Efficient CPU usage is crucial in datacenters, which often operate a shared environment by running multiple applications on a single machine [6]. With less CPU overhead, one can pack more applications onto each machine, use fewer machines, rely on wimpier cores [28] and yet achieve the same or better performance.

Multi-round operations are practical. Because the roundtrip latency on Ethernet is substantial, traditional systems designs aim to minimize the rounds of communications required to complete an operation. For example, existing key-value stores process each `get` or `put` operations in one roundtrip. With RDMA's ultra-low latency, it becomes feasible to use multi-round protocols without adversely affecting end-to-end operation latency. For example, each `get` operation in Pilaf requires at least two roundtrips, yet the end-to-end latency is less than 30μs.

Challenges. It is technically challenging to fully exploit RDMA's performance advantage in a system design. The common existing practice is to use RDMA to optimize verb message exchange [19, 14, 12]. Specifically, in order to send (or receive) a large message, a client first transmits some control information to the server using a verb message. The server then performs an RDMA read (or write) to the client to fetch (or store) the actual payload. This design maintains the traditional request/reply communication pattern, but does not fully exploit the benefits of RDMA since the overall latency and throughput is still bottlenecked by sending/receiving verb messages.

A more efficient system design is one in which all or a large fraction of the existing request/reply traffic is *replaced* (instead of supplemented) by RDMA operations. However, letting clients directly perform RDMA on the server's memory introduces serious synchronization problems: not only can multiple clients' concurrent RDMA accesses cause races, but the server can also simultaneously perform local memory accesses that race with remote accesses. While there are some hardware mechanisms for synchronizing RDMA accesses, there exist no efficient capability at all for coordinating local and remote memory accesses.

3 Pilaf Design

This section traces the evolution of Pilaf's design up to its current form. We first motivate Pilaf's overall architecture, which processes write operations at the server and uses RDMA for read-only operations (Section 3.1). We then explain how clients perform `gets` using RDMA reads and discuss how Pilaf synchronizes clients' RDMA accesses with the server's local memory writes. Last, we describe the Cuckoo hashing optimization that reduces the number of required roundtrips in the worst case.

3.1 Overview

The most straightforward design would be to take a traditional key-value store and re-implement its messaging layer using verb messages instead of TCP sockets. However, this design fails to reap the benefits of RDMA, which has much lower latency and CPU overhead than verb messages. Therefore, our goal is to find a system design that can exploit one-sided RDMA operations without adding too much complexity.

A key-value store has two basic operations: $V \leftarrow get(K)$ and $put(K, V)$, where both the key K and value V are strings of arbitrary length. In our initial design iterations, we tried to use one-sided RDMA operations for both `gets` and `puts`. In other words, each client performs RDMA reads to implement `gets` and RDMA writes to implement `puts`.

We quickly discovered that using RDMA for all operations leads to complex and fragile designs. First, clients must synchronize their RDMA writes so as not to corrupt the server's memory. The Infiniband card supports atomic operations (such as compare-and-swap) on top of which one could build an explicit locking mechanism or implement a lock-free data structure [10]. Locking introduces the complication of clients failing while holding a lock. On the other hand, lock-free implementations are complex and require remote dynamic memory allocation. Second, a `put` operation requires memory allocation to store key-value strings of arbitrary length; such memory management becomes unwieldy in the presence of remote writes. Having clients implement memory management remotely is expensive, with excessive locking and round trips required. On the other hand, letting the server perform memory management introduces write-write races between the server and clients. Unfortunately, there exists no efficient hardware mechanism to synchronize memory accesses initiated by the CPU and the network card. Last but not least, by making all operations transparent to the server, debugging becomes painful, as race conditions involving remote accesses are much more difficult to find and reproduce than those involving local accesses.

Our first major design decision is to have the server handle all the write operations (i.e. `put` and `remove`) and have the clients implement read-only operations (i.e. `get` and `contains`) using one-sided RDMA reads. Since real-world workloads are skewed towards reads (e.g., Facebook reported read-to-write ratios ranging from 68%-99% for its active key-value stores [1]), this design captures most of the performance benefits of RDMA while drastically simplifying the problem of synchronization. In fact, the beauty of this design is that it incurs no write-write races, but only read-write races between RDMA reads and the server's local memory writes, Write-write races are the main source of design complexities since they must be avoided at all costs to prevent memory corruption. In contrast, read-write races can be made harmless by simply detecting the presence of such races and re-trying the affected operation. Thus, no fragile and expensive locking or lock-free protocol is

needed.

Figure 3 shows Pilaf's overall architecture. Using verb messages, clients send all `put` requests to the server, which inserts them in its in-memory hashmap before sending the corresponding replies. By contrast, `gets` are transparent to the server in that the clients perform RDMA reads over multiple roundtrips to directly fetch data from the server's memory.

As in other key-value store designs [20, 25], the Pilaf server has the option to asynchronously log updates to its local disk.

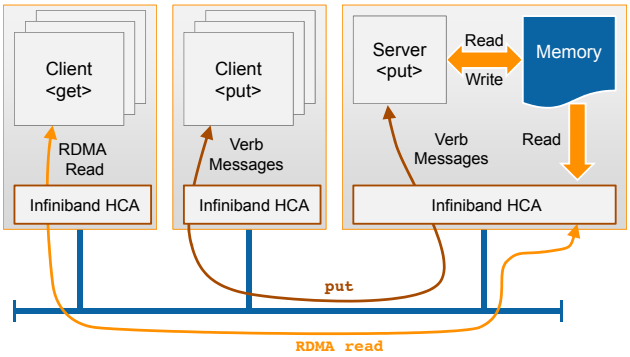


Figure 3: Pilaf restricts the clients' use of RDMAs to read-only `get` operations and handles all `puts` at the server using verb messages.

3.2 Basic `get` Operation Using RDMA

We first explain how Pilaf performs `gets` without involving the server's CPU. We defer the challenge of coping with concurrent `puts` and `gets` to Section 3.3.

To allow RDMA reads, the server must expose its data structure for storing the hash table, as shown in Figure 4. There are two logical memory regions: an array of fixed size hash table entries and an extents area for storing the actual keys and values, which are strings of arbitrary length. The server registers both memory regions with the network card, and clients obtain the corresponding registration keys of these two memory regions (as well as the size of the hash table array) when they first establish a connection to the server. Subsequently, clients can issue RDMA requests to any memory address in these two regions by specifying the memory's registration key and an offset.

In the basic design, a client looks up a key in the hash table array using linear probing [26]. Each probe involves two RDMA reads. The first read fetches the hash table entry corresponding to the key. If the entry is currently filled (indicated by an `in_use` bit), the client initiates a second RDMA read to fetch the actual key and value strings from the extents region according to the address information stored in the corresponding hash table entry. The client checks whether the fetched key string matches the requested key. If so, the `get` operation finishes. Otherwise, the client continues with the next probe.

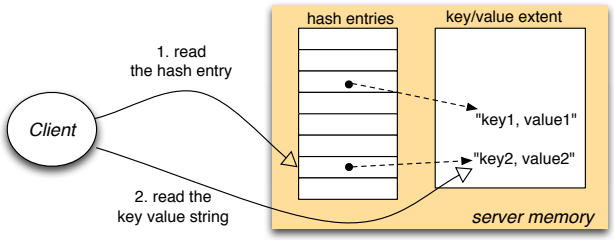


Figure 4: The memory layout of the Pilaf server's hash table. Two memory regions are used, one contains an array of fixed-size hash table entries, the other contains variable sized key-value strings (extents). To perform a `get`, clients probes the server's hash table using two RDMA reads, first fetching a hash table entry, then using the address information in that entry to fetch the associated key-value string.

3.3 Coping with Read-Write Races

The Pilaf server handles all `put` operations. Thus, local memory writes performed by the server's CPU can create read-write races with concurrent RDMA reads done by clients. This is a challenge as there exists no efficient hardware mechanism to coordinate the CPU and the network card. To inhibit RDMA reads during a write, the server could resort to resetting all existing connections, or temporarily de-register memory regions with the network card. However, both mechanisms are far too expensive to be used for every `put` operation. Therefore, we must be able to cope with the presence of read-write races.

To implement a read operation, clients need to traverse the server's data structure. The traversal starts from a set of "root" objects with known memory locations and recursively follows pointers read previously. In the context of Pilaf, we can view each hash table entry as a "root" object which points to additional key-value information. Read-write races introduce the possibility that clients can traverse the server's data structure incorrectly.

Two scenarios can result in incorrect traversal. First, a root object can be corrupted. In Pilaf, this happens when the server modifies a hash table entry while a client is reading that entry. Consequently, the client will read a partially-modified (and thus incorrect) hash table entry, potentially causing it to read the key-value string from a wrong memory location. Second, a client's pointer reference can become invalid. For example, in Pilaf, the server may delete or modify an existing key/value pair while a client is holding a pointer reference to the old string from its first RDMA read of the hash table entry. Thus, during its second RDMA access, the client might read garbage or an incorrect key-value string.

To permit correct traversal in the face of read-write races, we introduce the notion of a *self-verifying* data structure by making both root objects and pointers self-verifying. For a root object, we append a checksum that covers the object's entire contents. Thus, any ongoing modification on the root object results in a checksum failure. To make a pointer self-verifying, we store it as a tuple combining a memory location, the size of the memory chunk being referenced, and a checksum covering the contents of the referenced memory. Therefore, a client can detect inconsistencies between a pointer's intended memory reference and the actual memory contents. For example, if the server de-allocates the memory chunk being referenced and re-uses parts of it later while a client is still holding a pointer to it, the client will fail to verify the checksum when it retrieves the memory contents using the old pointer. Figure 5 shows Pilaf's self-verifying hash table. As a root object, each hash table entry contains a checksum covering the whole entry. The pointer stored in each hash table entry contains a checksum verifying the key-value string being referenced.

Self-verifying data structures ensure correct traversal starting from a set of known root object locations. On rare occasions, the server may need to change the root object locations. This can be accomplished correctly by having the server reset all its existing RDMA connections to clients to inhibit clients from reading stale root object locations. In Pilaf, whenever the server needs to resize its hash table array, it disconnects all clients. Clients reconnect once the resize operation is complete to obtain up-to-date information about the location and size of the hash table array. Since hash table resizing is infrequent, there is a minimal performance penalty from resetting connections.

A self-verifying data structure allows clients to perform consistent reads in the face of concurrent writes. In addition, the Pilaf server uses a memory barrier to force any updates from the CPU cache to the main memory before replying to a `put` request. Doing so ensures that a subsequent `get` always reads the effect of any completed `puts`. As a result, Pilaf provides the strongest consistency semantics, i.e. linearizability [11].

3.4 Improving a Hash Table's Memory Efficiency

In the basic design, a client performs linear probing to look up a key in the server's hash table array. This simple hash scheme does not achieve a good tradeoff between memory efficiency and operation latency. For example, when the hash table is 60% full, the maximum number of probes required can be as high as 70. To achieve good memory efficiency with fewer probes, Pilaf uses n-way Cuckoo hashing [22, 16]. This hashing scheme uses n orthogonal hash functions, and every key is either at one

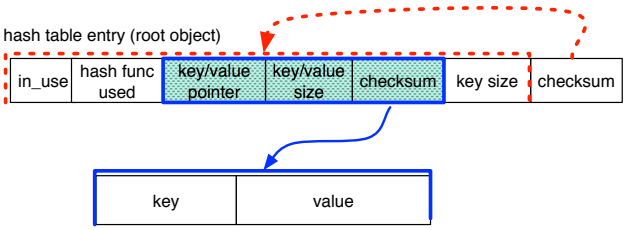


Figure 5: *Self-verifying* hash table structure. Each hash table entry is protected by a checksum. Each entry stores a self-verifying pointer (shown in shaded fields) which contains a checksum covering the memory area being referenced.

of n possible locations or absent. If all n possible locations for a new key are filled, the key is inserted anyway, kicking the resident key-value pair to one of that key's alternate locations. That operation may in turn kick out another pair, ad infinitum. The table is resized when a limit is reached on the number of kicks performed or when a cycle is detected.

The main challenge in using Cuckoo hashing for Pilaf lies in the process of moving an existing entry to a different hash table location. Ordinarily, bulk key movements such as resizing the hash table requires that the server reset all existing RDMA connections. This is not desirable, as the need to move a key occurs much more frequently than table resizing with Cuckoo hashing. Without resetting connections, there is the danger that a key-value pair might appear to be "lost" to the clients while the server is moving it to a new location. To address this issue, during a `put` operation the server first calculates the new locations of every affected key without actually moving the keys. Then, starting from the last affected key, the server shifts each key to its new location, thereby ensuring that a key is always stored in at one or two (instead of zero or one) hash table entries during movement.

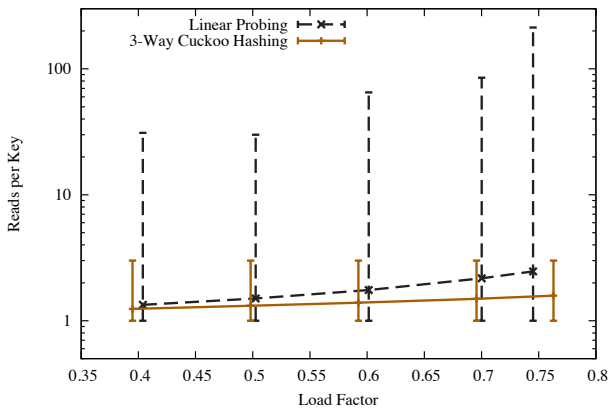


Figure 6: The average number of probes required during a key lookup in 3-way Cuckoo hashing and Linear probing. The error bars depict the median and maximum values.

We explored different parameter values for n and determined that 3-way Cuckoo hashing achieves the best memory efficiency with few hash entry traversals per read. As Figure 6 shows, at a fill ratio of 75%, the average and maximum number of probes in 3-way Cuckoo hashing is 1.6 and 3, compared to 2.5 and 213 respectively for linear probing.

4 Implementation

We implemented Pilaf in C++. Pilaf uses the `libibverbs` library from the OpenFabrics Alliance, which allows user-space processes to use verb messages and RDMA directly. The Pilaf server continuously polls the network card for new events, including the reception of verb messages or the completion of recently-sent RDMA operations or verb messages. Since Pilaf is able to saturate the network card's performance using a single thread, our implementation uses the same polling thread to process `put`s as well.

RDMA-Friendly Extents: The server must register a region of memory and gives clients the registration key for that memory before clients can perform RDMA on the region. This process is relatively expensive and should be made infrequent. Therefore, Pilaf allocates and registers a large contiguous address space for the key-value extent. We ported the `mem5` memory management unit from SQLite to C++ to "malloc" and "free" strings in the key-value extent. Whenever the extents region becomes full, the server resets all existing connections, expands the extents, and then allows clients to re-connect and obtain new registration keys. As with hash table resizing, extents-resizing is also an infrequent event.

Self-Verifying Data Structures: Our implementation uses CRC64 as the checksum scheme for our self-verifying data structures. CRCs are not effective for cryptographic verification. Instead, they were originally intended to detect random errors, making them ideal for our application. The ideal n-bit CRC will fail to detect 1 in 2^n message collisions. Although 32-bit CRC is popular (e.g. for Ethernet and SATA checksums), we believe that CRC32 is insufficient for Pilaf. Every `put` incurs two CRC updates, one on the hash table entry and one on the key-value string. As will be shown in Section 5.2, Pilaf can process 663K `put`s per second. Therefore, up to 1.326 million CRCs may be calculated per second. Since each CRC32 incurs a collision with probability 1 in 2^{32}, we expect a collision once every 3239 seconds (54 minutes). We find this rate to be unacceptably high. Using CRC64, we can expect a collision once every $1.35 * 10^{13}$ seconds, or once per 428 millenia.

CRC64 is fast. Our implementation consumes about a dozen CPU cycles for each checksummed byte, and incurs the same overhead as CRC32 when running on 64-bit CPUs.

Logging: By default, Pilaf server asynchronously logs all `put` and `delete` operations to the local disk, similar to the logging facility in other key-value stores including Redis [25], Masstree [20] and LevelDB [8]. Using a single solid state disk, Pilaf is able to log 663K (our peak `put` throughput) writes per second if the average key-value size is smaller than 500 bytes. Should one desire a high logging capacity, multiple SSDs must be used.

5 Evaluation

We evaluate the performance of Pilaf on our Infiniband cluster. The highlights of our results are the following:

- Pilaf achieves high performance: its peak throughput reaches 1.3 million ops/sec. The end-to-end operation latency is very low with a 90-percentile latency of ~30μs.
- Pilaf is CPU-efficient. Even when running on a *single* CPU core, Pilaf is able to saturate the network hardware's capacity to achieve 1.3 million ops/sec. By comparison, Memcached and Redis achieve less than 60K ops/sec per CPU core, so they require at least 20× the CPU resource to match Pilaf's performance.
- Self-verifying data structures are effective at detecting read-write races between the clients' RDMA operations and the server's local memory accesses.

5.1 Experimental setup

Hardware and configuration. Our experiments are run on a cluster of ten machines, each with two AMD or Intel processors and 32GB of memory. Each machine is equipped with a a Mellanox ConnectX-2 20 Gbps Infiniband HCA as well as an Intel gigabit Ethernet adapter. The machines run Ubuntu 12.10 with the OFED 3.2 Infiniband driver.

For each experiment, we run a server process on one physical machine, while the clients are distributed among the remaining machines to saturate the server. By default, we restrict the server process to run on one CPU core. For Ethernet experiments, we configure the kernel's network interrupt processing to trigger on the same core used by the server process.

We disable Pilaf's asynchronous logging in the experiments. With logging turned on, Pilaf incurs no measurable reduction in achieved throughput for key-value sizes less than 500 bytes. With larger operations, the I/O bandwidth of the server's single local SSD becomes the bottleneck.

Workload. We use the YCSB [3] benchmark to generate our workloads. YCSB constructs key-value pairs with variable key and value lengths, modelled on the statistical properties of real-world workloads. Furthermore, with YCSB, the keys being accessed follow a long-tailed zipf distribution. The original YCSB software is written in Java. We ported it to C so that we could saturate the server with fewer client machines.

In all experiments, we vary the size of the value string from 16 to 4096 bytes while keeping the average key size at 23 bytes, the default value in YCSB. We use two mixed workloads, one consisting of 10% `puts` and 90% `gets`, the other 50% `puts` and 50% `gets`. Since Facebook has reported that most of their Memcached deployments are read-heavy [1], our mixed workloads give reasonable representations of real workloads.

Points of comparison. We compare Pilaf against Memcached [7] and Redis [25] (with logging disabled). Additionally, we also compare Pilaf to an alternative implementation of itself, which we refer to as Pilaf-VO (short for Pilaf using Verb messages Only). In Pilaf-VO, clients send all operations (including `gets`) to the server for processing via verb messages. The performance gap between Pilaf and Pilaf-VO demonstrates the importance of bypassing the CPU using RDMA.

5.2 Microbenchmarks

The microbenchmarks measure the throughput and latency of individual `get` and `put` operations.

Throughput: Figure 7 shows Pilaf's peak operation throughput, achieved with 40 concurrent clients. Pilaf can perform 1.28 million `get` and 663K `put` operations per second for small key-values. Of note is that Pilaf's high throughput is achieved using a single CPU core, which saturates the Infiniband card in most cases.

Performing `get` operations via RDMA incurs zero CPU overhead on the server. Furthermore, `get` operations also have the highest throughput. As shown in Table 1 (Section 2), the card's peak RDMA throughput is much higher than that of verb messages, especially for small messages. In particular, the card can satisfy 2.45 million RDMA reads per second for small reads. Since each `get` requires at least two RDMA reads, the overall throughput is approximately half of the raw RDMA throughput at 1.28 million `gets`/sec. By contrast, the peak verb throughput is 667K request/reply pairs/sec for small messages, resulting in 667K ops/sec for `puts`.

For larger key-value pairs, the throughputs of `get` and `put` converge as they both approach the network bandwidth. For example, for 4096-byte key-values, Pilaf consumes 11.7Gbps of the 16Gbps data bandwidth supported by the network card. Interestingly, we find that when processing `puts` with large values, the Pilaf server becomes CPU-bound when using a single core. Specifically, for 1024-byte value size, Pilaf achieves 75.4% of its network-bound `put` throughput (543K ops/sec) with one core and 100% (663K ops/sec) with two cores.

We also measure the throughput of Pilaf-VO's `get` operation, which is processed by the server instead of by the clients using RDMA. As Figure 7 shows, the throughput of performing `gets` using verb messages is similar to that of `puts` and is much smaller than the throughput of `gets` done via RDMA for small key-value pairs.

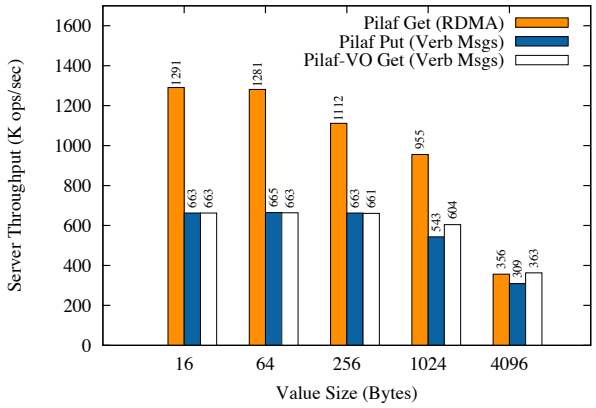


Figure 7: Server throughput for `put` and `get` in Pilaf and `get` in Pilaf-VO. All tests are performed with 40 connected clients.

Latency: Figure 8 shows the average latency of `get` and `put` operations with 10 concurrent clients. With a larger number of clients (e.g. 40), the latency becomes mostly determined by queuing effects and thus is much higher. With a single client (not shown in the figure), the latency of `get` is slightly more than 2 RDMA roundtrips and is twice the latency of `put`. With more clients and thus more load, we found that the RDMA latency scales better than that of verb messages. As Figure 8 shows, for 10 clients, average latency for small `gets` is 10μs, while small `puts` take around 8μs. For large key-values, the latencies of `get` and `put` are similar and are both bounded by the packet transmission time.

5.3 Performance of self-verifying data structure

Pilaf uses a self-verifying hash table structure to detect read-write races during concurrent `gets` and `puts`. We expect such races to be rare in a normal workload. To artificially vary the conflict rate, we inject the maximum achievable `get` and `put` loads, simultaneously reading and writing a varying number of unique key-value pairs. Therefore, the probability of races increases as the `gets` and `puts` are restricted to fewer and fewer unique keys.

Figure 9 shows the probability of detecting a read-write race as measured by the fraction of `gets` that need

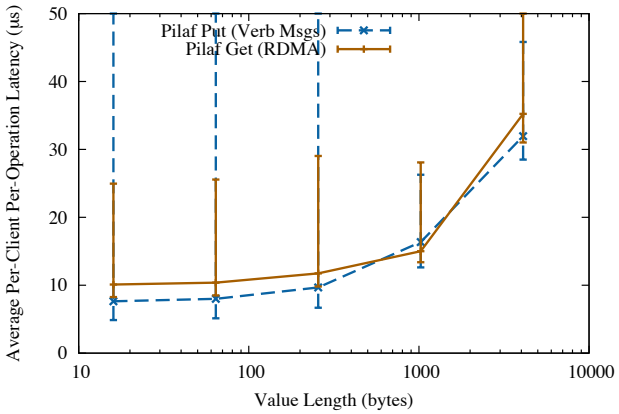


Figure 8: Average operation latency for `put` and `get` operations as the average value size increases. All tests are performed with 10 connected clients; though not pictured, we observe a linear relationship between the number of connected clients and latency due to queuing effects.

to be re-tried. The two lines in Figure 9 illustrate the probabilities of a retry due to a race when reading the hash table entry or when reading the key-value extent. The figure shows a non-negligible race probability only when the hash table is extremely small. When the hash table contains more than 20,000 keys, the probability of racing is less than 0.01% even under peak `put` and `get` loads.

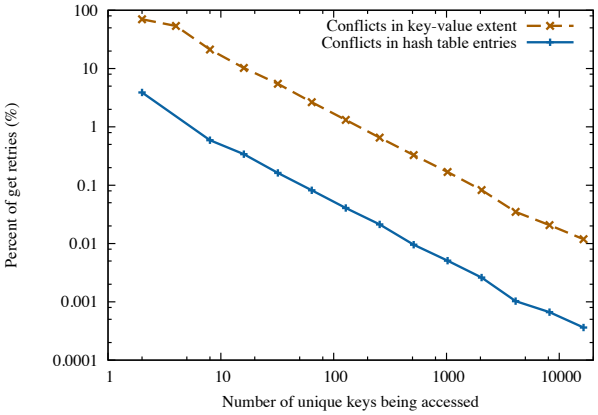


Figure 9: Percentage of re-reads of extents and hash table entries due to detected read-write races. We control the likelihood of races by varying the number of unique keys being read or updated. The Pilaf server is operating under peak throughput.

5.4 Pilaf versus Memcached and Redis

We compare Pilaf to two existing popular key-value systems, Memcached [7] and Redis [25]. Both systems are widely deployed in the industry, including Facebook [1], YouTube [5], and Instagram [15]. Memcached is commonly used as a database query cache or a web cache to speed up the server's generation of a result web page and improve throughput. Low operation latency is vital in such a usage scenario: the faster the key-value cache can fulfill each request, the faster a page involving many dependent cache lookups can be returned to the client. High throughput and low CPU overhead are also crucial, since these properties allow more clients can be served with fewer server resources.

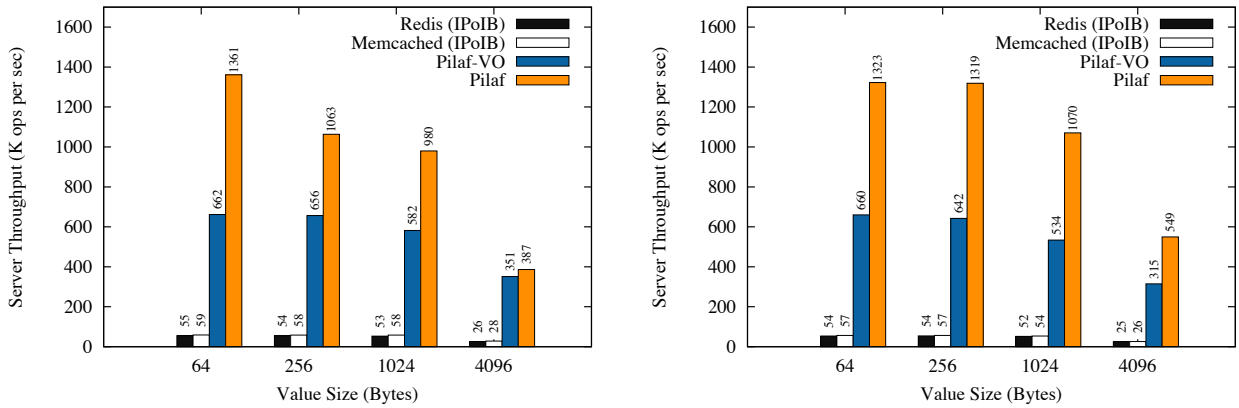
Because Memcached and Redis are written to use TCP sockets, we run them on our Infiniband network using IPoIB. It's important to note that we do not batch requests for any of the systems, unlike in [20].

Throughput Comparison: In our experiments, the peak throughput of each system is achieved when running 40 concurrent client processes. Figure 10 shows the achieved operation throughput using a single CPU core for various value sizes in a mixed workload with 90% `get`s. We can see that the performance of Pilaf far exceeds that of Redis and Memcached running on top of IPoIB. For small operations (64-byte values) Pilaf achieves 1.3 million ops/sec compared to less than 60 Kops/sec for Memcached and Redis. Both Memcached and Redis are bottlenecked by the single CPU core and are unable to saturate the Infiniband card's performance. Because of the CPU bottleneck, their single core performance is the same when running on 1 Gbps Ethernet. We elided those numbers from Figure 10 for clarity.

The throughputs of Memcached and Redis can be scaled by devoting more CPU cores to each system. For example, both systems can saturate the 1Gbps Ethernet card when running on four CPU cores. We were not able to scale Memcached and Redis' performance on IPoIB using more CPU cores because the IPoIB driver is unable to spread network interrupts across multiple cores. Nevertheless, even if we optimistically assume perfect scaling, Memcached and Redis would require 17× CPU cores to match the performance of Pilaf running on a single core for small key-values. In reality, these systems do not exhibit perfect scaling. For example, [20] reported a 11× throughput improvements for non-batched Memcached `put`s when scaling from 1 to 16 cores.

When comparing against Pilaf-VO, we see that Pilaf also achieves substantially better throughput across all operation sizes. In particular, the throughput of Pilaf is 2.1× that of Pilaf-VO for 64-byte values and 1.1× for 4096-byte values. The shrinking performance gap between Pilaf and Pilaf-VO for larger values reflects the increasingly dominant network transmission overhead for large messages.

Figure 10(b) shows the peak throughput of different systems in a second workload with 50% `get`s and 50% `put`s. Not surprisingly, the performance of Memcached and Redis are similar under both workloads.



(a) Peak throughput (90% `gets`, 10% `puts`)

(b) Peak throughput (50% `gets`, 50% `puts`)

Figure 10: Throughput achieved on a single CPU core for Pilaf, Pilaf-VO, Redis, and Memcached.

We were surprised to see that Pilaf achieves identical and sometimes better throughput in the second workload compared to the first. Since RDMA-based `get` operation has much higher performance than verb message-based `put` (Figure 7), we initially expected the second workload to achieve worse throughput since it contains a larger fraction of `puts`. On further investigation, we found that our Infiniband cards appear to be able to process verb messages and RDMA operations somewhat independently. Quantitatively, the card can reach ~80% of its peak RDMA throughput while *simultaneously* sending and receiving verb messages at ~95% of the peak verb throughput. This explains why the second workload has better throughput. For example, with 256-byte values, the first workload achieves 0.9 million `gets`/sec (80% of peak RDMA performance) and 0.1 million `puts`/sec (far less than the card's verb message sending capacity). By contrast, the second workload produces 0.65 million ops/sec for both `get` and `put` which represents 60% of the card's peak RDMA performance and 94% of the card's verb message performance. Thus, the second workload has a total throughput of 1.3 million ops/sec, better than that achieved by the first workload.

Latency: Figure 11 shows the cumulative probability distribution of operation latencies under different systems in the workload with 90% `gets`. The underlying experiments involved 10 concurrent clients issuing operations with 1024-byte values as fast as possible.

In Figure 11, Pilaf's median latency is 15μs which is determined by the `get` operation latency. From the earlier experiments in Section 2 (Figure 1), we know the average RDMA roundtrip latency is 4μs for 1024-byte reads with a single client. With an average of 1.45 probes (each involving two RDMA reads) to find a particular key-value in a 65%-filled 3-way Cuckoo hash table, the ideal `get` latency would be 11.2 μs. The extra 4μs reflects the overhead in calculating CRCs on the clients' side plus the queuing effects incurred by having ten connected clients. The latency tail in Figure 11 is very short.

As expected, IPoIB also maintains lower latency than Ethernet for both Memcached and Redis. Median Ethernet latency is 209μs for Redis and 230μs for Memcached. Pilaf beats Redis' and Memcached's median Ethernet latency by 14×-15×, and their median IPoIB latency by 9×-11×. The experiments for Figure 11 involve ten clients connected to a single server. In these experiments, Pilaf-VO reaches 95% of its peak throughput, Memcached is at 75% of its maximum throughput, and Redis and Pilaf at half their peak throughput. Therefore, queuing effects are uneven for these systems in Figure 11. When tested under light loads (e.g. using a single client), Pilaf-VO and Pilaf have similar latency while Memcached and Redis running on IPoIB also have similar latency.

6 Related Work

There has been much work in the HPC community to exploit performance critical features like kernel and CPU bypassing. Many MPI implementations, e.g. MPICH/MVAPICH [18, 19] and OpenMPI [27], supports an Infiniband network layer, leveraging both verb messages and RDMA to reduce latency and increase bandwidth.

RDMA as a powerful HPC networking feature has been recognized in the system community in several works. Due to the perceived cost of specialized HPC hardware, some have advocated software RDMA over traditional Ethernet. Soft-iWARP is a version of the iWARP protocol implemented entirely in software [29]; it reduces TCP latency by 5%-15% by minimizing data copying and limiting the number of context switches re-

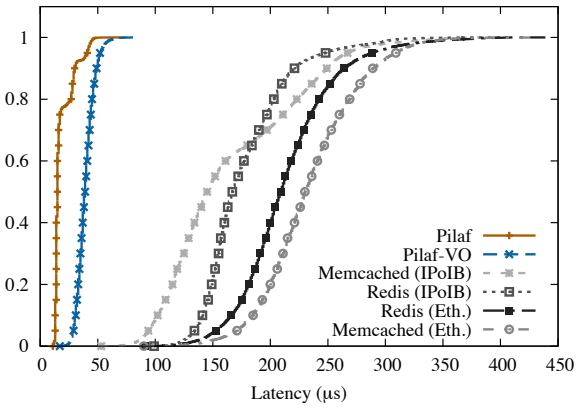


Figure 11: CDF of Pilaf latency compared with Memcached, Redis and Pilaf-VO in a workload consisting of 90% `gets` and 10% `puts`. The average value size is 1024 bytes. The experiments involved 10 clients.

quired. Another project later used soft-iWARP to realize a 20% reduction in per-`get` CPU load for Memcached without Infiniband hardware [28].

Many have leveraged RDMAs to improve the throughput and reduce the CPU overhead of existing networked systems such as PVFS [31], NFS [9], Memcached [14, 13, 28], and HBase [12]. All of these works re-use existing system designs on top of a modified communication layer which utilizes RDMA within a traditional request/reply message exchange. In other words, RDMAs are used as a supplemental mechanism to optimize data transfer; each RDMA access is always preceded by verb or other messaging mechanisms that signal control information for that RDMA. As an example, a client sends a verb message to instruct the server to perform an RDMA read (or write) to the client. When the server completes the RDMA operation, it replies with another verb message informing the client that the transfer is complete. This strategy uses RDMA effectively only for large messages since the throughput and CPU overhead of processing small messages are still bounded by the verb message performance. By contrast, our work aims to replace a large fraction of the request/reply message exchanges with RDMA reads by the clients, thereby significantly reducing the server's CPU overhead.

The three projects that implement Memcached over RDMA on Infiniband [14, 13] or soft-iWARP [28] also adopt the usual combination of control messages plus RDMAs to process `gets` and `puts` at the server. In [14], the client uses a verb message to send the server a local buffer address, which the server then copies data into using an RDMA write. `Put` operations also involve two verb messages and one RDMA read, wherein the client gives the server an address, from which the server pulls a key-value pair from the client via an RDMA read. Both `put` and `get` include short-operation optimizations that combine the data normally read or written via RDMA into one of the verb messages exchanged. Compared to Pilaf, this design achieves much lower throughput. Their reported throughput in an Infiniband cluster similar to ours is 300 Kops/sec for small operations, significantly lower than that achieved by Pilaf (1.3 million ops/sec). The other Memcached over Infiniband project [13] combines Infiniband's Reliable Connection (RC, with guarantees similar to TCP) and Unreliable Datagram (UD, resembling UDP) modes. The resulting performance is also lower than achieved by Pilaf, despite running on a QDR Infiniband cluster which is twice as fast as ours (DDR).

7 Conclusion

As future datacenter networks move towards incorporating HPC network features, it is time to rethink networked system designs that can fully exploit powerful features like RDMA. We have demonstrated such a design by building a high-performance key-value store with very low CPU overhead. Pilaf replaces the usual request/rely messaging pattern for read-only operations by having the clients directly read from the server's memory using RDMA. It uses *self-verifying* data structures to detect read-write races in the face of concurrent RDMA reads from the clients and local memory writes from the server. Pilaf is able to achieve a peak throughput of over 1.3M ops/sec with a single CPU core, outperforming existing systems running over Ethernet or IPoIB by more than an order of magnitude.

Acknowledgements

Members of the NeWS group – Yang Zhang, Russell Power, and Aditya Dhananjay – gave valuable feedbacks that helped improve this work. Our special thanks go to Yang Zhang, who first suggested using CRCs to check for data inconsistency. Frank Dabek suggested useful experiments to evaluate Pilaf's self-verifying data structure. This work was partially supported by NSF Award CSR-1065114 and a Google Research Award. Yifeng Geng was supported by a Tsinghua visitor scholarship.

References

[1] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[3] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[4] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, pp. 251–264.

[5] CUONG, C. Youtube scalability. In *Google Seattle Conference on Scalability* (2007).

[6] DEAN, J. Software engineering advice from building large-scale distributed systems. Slides.

[7] FITZPATRICK, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[8] GHEMAWAT, S., AND DEAN, J. Leveldb, 2011.

[9] GIBSON, G., AND TANTISIRIROJ, W. Network file system (nfs) in high performance networks. Tech. rep., Carnegie Mellon University, 2008.

[10] HERLIHY, M. Wait-free synchronization. *Transactions on Programming Languages and Systems (TOPLAS) 13*, 1 (1991), 124 – 149.

[11] HERLIHY, M., AND WING, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[12] HUANG, J., OUYANG, X., JOSE, J., UR RAHMAN, M. W., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with rdma over infiniband.

[13] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[14] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[15] KRIEGER, M. What powers instagram: Hundreds of instances, dozens of technologies.

[16] KUTZELNIGG, R., AND DRMOTA, M. *Random bipartite graphs and their application to Cuckoo Hashing*. PhD thesis, PhD thesis, Vienna University of Technology, 2008.

[17] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[18] LIU, J., JIANG, W., WYCKOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (april 2004), p. 16.

[19] LIU, J., WU, J., KINI, S., BUNTINAS, D., YU, W., CHANDRASEKARAN, B., NORONHA, R., WYCKOFF, P., AND PANDA, D. Mpi over infiniband: Early experiences. In *Ohio State University Technical Report* (2003).

[20] MAO, Y., KOHLER, E., AND MORRIS, R. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 183–196.

[21] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcached at facebook. In *Proceedings of USENIX NSDI 2013* (2013).

[22] PAGH, R., AND RODLER, F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[23] POWER, R., AND LI, J. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2010).

[24] RASHTI, M., AND AFSAHI, A. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), pp. 1–8.

[25] SANFILIPPO, S., AND NOORDHUIS, P. Redis.

[26] SEDGEWICK, R., AND WAYNE, K. *Algorithms*. Addison-Wesley, 2011.

[27] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. Infiniband scalability in open mpi. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), pp. 10–pp.

[28] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of USENIX Annual Technical Conference* (2012).

[29] TRIVEDI, A., METZLER, B., AND STUEDI, P. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), p. 17.

[30] VIENNE, J., CHEN, J., WASI-UR-RAHMAN, M., ISLAM, N., SUBRAMONI, H., AND PANDA, D. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on* (aug. 2012), pp. 48 –55.

[31] WU, J., WYCKOFF, P., AND PANDA, D. Pvfs over infiniband: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on* (2003), pp. 125–132.