Nessie: A Decoupled, Client-Driven, Key-Value Store using RDMA

Tyler Szepesi, Benjamin Cassell, Bernard Wong, Tim Brecht, Xiaoyi Liu
Cheriton School of Computer Science, University of Waterloo
{stszepesi, becassel, bernard, brecht, x298liu}@cs.uwaterloo.ca

Abstract

The increasing use of key-value storage systems in performance-critical data centre applications has motivated new storage system designs that use Remote Direct Memory Access (RDMA) to reduce communication overhead. However, existing approaches that achieve low latency and high throughput do so by dedicating entire cores to RDMA message handling, including polling local memory for incoming RDMA messages.

In this paper we describe and demonstrate why polling-based RDMA is not suitable for many data centre applications with significant data processing and data storage requirements. We then propose, design, implement and evaluate an alternative communication approach that strictly uses one-sided RDMA operations, which eliminates polling on the server-side and is therefore a serverless (client-driven) design. This approach is used to build Nessie, a distributed client-driven RDMA-enabled key-value store that uses native RDMA compare-and-swap operations to coordinate conflicting operations between clients. Nessie further reduces communication requirements by decoupling the key-value location from the key lookup mechanism, which enables local write operations, thus avoiding comparatively expensive RDMA operations. Nessie's one-sided design ensures that only the clients in the system consume any CPU, which in turn allows it to maintain high performance despite the need for computation by the application utilizing the key-value store or contention from other applications.

1. INTRODUCTION

Distributed in-memory storage systems, such as memcached [12], Redis [5], and Tachyon [17], have become a critical component in many datacenter and enterprise applications. By eliminating slow disk or SSD accesses, they can reduce an application's request service time by more than an order-of-magnitude [19]. In the absence of disks and SSDs, however, an operating system's network stack is often the next largest source of throughput reduction and latency increase for applications. With network operations taking tens or hundreds of microseconds to complete, the performance bottleneck of large-scale applications has shifted, creating a need for better networking solutions.

The ever increasing drive for lower latencies and higher throughput have led recent in-memory storage systems to use remote direct memory access (RDMA) [18, 11, 14], which provides kernel bypass and zero-copy data transfers. RDMA's ability to access memory on other nodes, and provide over-the-network atomic operations, make it perfectly suited for integration into distributed in-memory storage systems. These RDMA-based systems are able to provide far higher throughput than those using traditional networking protocols like TCP and UDP.

Most of the recently proposed RDMA-based in-memory storage systems provide a simple key-value store interface, but they make different design decisions based on, or resulting in, different CPU, latency and throughput tradeoffs. For example, for `GET` operations, Pilaf [18] leverages one-sided RDMA `READ` operations to eliminate processing-related delays on the server. FaRM [11] reduces `PUT` operation latency and increases throughput by introducing a dedicated process on each server that polls on a circular buffer to determine when a `PUT` request has arrived. This allows FaRM to avoid system call and process scheduling-related delays for `PUT` operations. In both Pilaf and FaRM, servicing a `GET` request requires two back-to-back one-sided `READ` operations; the first to perform a key lookup on the index table, and the second to fetch the key's associated value from the data table. HERD [14] forgoes one-sided `READ` operations in order to serve `GET` requests by using two RDMA `WRITE` operations. A HERD `GET`

consists of a WRITE from the client to the server that carries the request, followed by a WRITE in the opposite direction that carries the result. In this design, the polling process must service both GET and PUT requests.

Although these systems provide significant improvements over traditional in-memory storage system designs, FaRM and HERD's reliance on dedicated CPU resources can make them unsuitable for multi-tenant deployments, deployments with low-power "wimpy" CPUs (increasingly common in cloud environments), or for applications where the storage servers are running on the same machines as the compute nodes, such as Spark and Hadoop. Without careful consideration for the placement of polling threads, interference between processes that share a CPU can dramatically reduce the throughput of an RDMA-enabled key-value store (RKVS). Furthermore, the reliance on dedicated CPU resources requires a time consuming tuning period in order to determine the optimal number of CPUs to dedicate to server-side polling and request processing in order to obtain peak performance. In this paper, we present the design of Nessie, a distributed high-performance RKVS which, in contrast to existing RKVS designs, uses exclusively client-driven operations to avoid CPU-intensive polling that is characteristic of server-driven designs.

Another characteristic of modern servers that an RKVS must account for is the dramatic difference in performance between main memory accesses, which incur latencies on the order of tens or hundreds of nanoseconds [16], and RDMA operations, which incur latencies on the order of microseconds. This issue may be exacerbated as the size of data values being stored in an RKVS increase, thus incurring even higher latencies when retrieved over the network. The ability of an RKVS to exploit data access locality can significant reduce the number of remote operations that are required. Nessie adopts a distributed design which completely decouples its key indexing data structure from its storage data structure, improving locality.

This work builds upon our previous workshop paper describing a client-driven, RDMA-enabled key-value store [22]. In addition to extending the design described in our previous work, we also provide a full implementation and evaluation of our prototype system. We furthermore introduce an analysis to compare client and server-driven systems. Finally, this work also introduces new optimizations that are critical to providing high performance for client-driven systems. Our contributions are as follows:

- We explain and demonstrate why server-driven RDMA operations, while appropriate for low-latency, high-throughput operations, may not be suitable for many data center applications with significant data processing and storage requirements.
- We design, implement and evaluate Nessie, a distributed RKVS that exclusively uses one-sided RDMA operations to make it more suitable for use with such applications.
- We describe how our Nessie design completely decouples the index table from data table, thus permitting values to be stored on different nodes than their keys. We demonstrate that this permits data values to be obtained from local memory without requiring RDMA operations, thus obtaining significant performance benefits.

2. BACKGROUND AND RELATED WORK

In this section, we provide an overview of RDMA technologies and survey current in-memory key-value stores. We provide the background necessary to compare and contrast existing designs for in-memory, RDMA-based key-value storage systems, and to motivate our design for Nessie.

2.1 RDMA

Remote Direct Memory Access (RDMA) can be used as an alternative to traditional networking protocols such as TCP or UDP. RDMA, with its ability to bypass the kernel and provide zero-copy data transfers between local and remote memory regions, has lower overhead than traditional protocols and, as a result, achieves lower latency and higher throughput. These improvements allow an application to perform remote memory accesses within a matter of several microseconds. Although RDMA is most closely associated with Infiniband networks, two additional protocols have been developed to support RDMA over traditional Ethernet networks: iWARP [6] performs RDMA over the TCP/IP network stack with processing offloaded to the NIC, and RDMA over Converged Ethernet (RoCE) [4] uses the Infiniband transport protocol over Ethernet. Although converged Ethernet, a lossless enhancement to standard Ethernet, provides performance benefits to RoCE deployments, RoCE is also compatible with traditional Ethernet environments.

Communication over RDMA is performed using the verbs interface, which consists of two-sided and one-sided verbs. Two-sided verbs follow a message-based model where one process sends a message using the SEND verb and the other process receives the message using a RECV verb, a process which involves CPU interaction on both the sending and receiving side. The contents of the sent message are passed directly into an address in the receiving application's memory, specified by the RECV verb. Therefore, a SEND verb is indirectly manipulating the receiving application's memory.

One-sided verbs allow direct access to pre-designated regions of a remote application's address space, without interacting with the remote CPU. The READ and WRITE

verbs can be used to retrieve and modify the contents of these regions, respectively. RDMA also provides two atomic one-sided verbs: compare-and-swap (`CAS`), and fetch-and-add (`FAA`). The atomic verbs both operate on 64 bits of data.

RDMA supports an asynchronous programming model in which verbs are posted to a send and receive queueing pair. By default, completed operations post an event to a completion queue when finished. An application may block or poll on this queue while waiting for operations to finish. Alternatively, as noted by the authors of FaRM [11], most RDMA interfaces guarantee that an operation's bytes are read or written in address order. Applications may take advantage of this fact to poll on the last byte of a region being used to service RDMA operations, as once the last byte has been updated, the operation is complete.

2.2 Key-Value Stores

Key-value stores are increasingly popular storage solutions that, at a high level, map input identifiers (keys) to stored content (values). They are frequently employed for large web applications and distributed computation frameworks. A typical key-value store uses some form of data structure, such as a hash table or tree, to provide efficient key lookups and updates. Most key-value stores provide an interface consisting of `GET`, `PUT` and `DELETE` operations. `GET` accepts a key and obtains it's associated value. `PUT` inserts or updates the value for a key, and `DELETE` removes a key and any associated values from the store.

Some key-value stores, such as Redis [5], provides persistent storage while others, including memcached [12], are used for lossy caching. Column storage systems, such as BigTable [8] and Cassandra [15], are primarily used to store large volumes of structured data such that most requests are served from disk, whereas systems such as RAMCloud [19] store data entirely in main memory. There are also specialized key-value stores that provide workload-specific performance optimizations. For example, Dynamo [10] provides high availability in the event of node failures by using quorums and hinted-handoffs, and MicroFuge [21] provides performance isolation through the use of request-specific deadlines.

2.3 RDMA Key-Value Stores

RDMA has recently been employed as a means of increasing the performance of key-value storage systems. We differentiate the dimensions of RDMA-enabled key-value stores (RKVSes) and characterize existing RKVSes according to these dimensions.

2.3.1 Design Space

The RKVS design space spans two primary dimensions: communication mechanisms for performing remote operations, and system components for indexing and storing key-value pairs. In this section, we will describe different points in the design space for both attributes.

Communication Mechanisms: RKVSes can use a client-driven (CD) communication mechanism where remote operations are initiated and performed entirely by the client using one-sided verbs; a server-driven (SD) communication mechanism where clients instruct remote servers to perform operations on their behalf; or a combination of the two.

TCP/IP-based key-value stores are server-driven (SD) because they require server-side interaction in order to process requests. Server-driven mechanisms are attractive because they reduce the complexity of synchronizing resource access from multiple clients. The server determines the request order and provides mutual exclusion for key accesses. However, the SD mechanism requires processing by the server's CPU to handle each request. For high-performance systems that require low packet processing latency, one or more dedicated CPUs are often needed to poll for incoming requests to minimize response latency [14]. However, this reduces the CPU resources available on the server.

In contrast to key-value stores that use TCP/IP, RKVSes can take advantage of RDMA's one-sided verbs to perform client-driven communication. CD operations are entirely processed by the server's NIC rather than the server's CPUs; the NIC performs the memory read or write operation. Therefore, the server's CPU does not participate in the request handling and can instead be used entirely for user applications. Furthermore, by bypassing the operating system on the server side, CD mechanisms have the added benefit of lower latencies than SD mechanisms. However, without a server to coordinate concurrent operations, CD mechanisms must provide per-key mutual exclusion which requires distributed coordination between concurrent clients.

System Components: In order to provide efficient key operations, most RKVSes organize their keys (or the hash of their keys) using an O(1)-lookup data structure, such as a hash table. The lookup data structure, also known as an index table, maps each key to a location in a secondary storage structure. This storage structure contains the associated value for each key and, depending on the design, can be on the same server as the index table or on a policy-determined server in the system. We refer to these designs as partially-coupled and decoupled, respectively. Alternatively, some systems combine the index and storage structures into a single structure in order to avoid a second lookup operation per key access. We refer to this design as having complete coupling.

2.3.2 Existing Designs

Pilaf [18] is a partially-coupled RKVS that uses a cuckoo hash table [20] to store its index. A Pilaf client uses one-sided RDMA `READ` verbs to perform `GETs`, and `SEND` verbs to instruct the server to perform a `PUT` or `DELETE` on its behalf. Combining client and server-driven mechanisms can lead to clients concurrently performing conflicting operations with the server. To address this issue, Pilaf uses consistency verification after each `GET` operation: hash table entries contain checksums which allow the client to determine whether or not it has retrieved a stale value.

FaRM [11] presents an RKVS design that uses a variant of hopscotch hashing [13] to store its index. FaRM can operate using either completely or partially-coupled design, depending on the configured size of its key-value pairs. FaRM clients use one-sided RDMA `READs` to satisfy `GET` requests, and RDMA `WRITE` verbs to insert `PUT` and `DELETE` requests into a server-polled circular buffer. Once a `PUT` or `DELETE` is serviced, the server responds to the client using RDMA `WRITEs`. FaRM combines both client and server-driven mechanisms and must therefore manage the possibility of conflicting simultaneous operations. This synchronization is provided through the use of version numbers on entries, which are atomically updated using RDMA `CAS` verbs.

HERD [14] provides the design of a partially-coupled RKVS that uses set-associative indexing. HERD is fully server-driven, with clients using RDMA `WRITE` verbs to post `GET`, `PUT` and `DELETE` requests to a server-polled memory region. The HERD server responds to requests using connectionless RDMA `SEND` verbs, which retain no state while completing and therefore aid with scalability. Because HERD uses only SD communication, synchronization is inherent in the server-based design. Furthermore, HERD uses RDMA optimizations to reduce the number of required round trips and request latency. These optimizations include inlining data into RDMA message headers, and using "unreliable" RDMA connections which do not require acknowledgment packets to be sent for successful operations.

In the following section we motivate the design of Nessie, a fully client-driven RKVS that completely decouples the index table from the storage structure.

3. THE CASE FOR Nessie

Table 1 summarizes the existing systems described in Section 2.3.2 and outlines where they fit within the RKVS design space. For simplicity, and because each system uses the same operation for `PUTs` and `DELETEs`, we merge both operations into the `PUT` column. Notably, all of the previously-existing systems use server-driven operations for at least their `PUTs`, and none of them are completely decoupled. In this section we show how Nessie uses unexplored areas of the design to motivate its design. In particular, Nessie uses wholly client-driven operations to provide high performance and low CPU utilization. Additionally, it completely decouples the index table from the storage structure to better exploit data locality by avoiding remote memory accesses when possible.

	`GET`	`PUT`	Coupling
Pilaf	CD	SD	Partial
FaRM	CD	SD	Complete
HERD	SD	SD	Partial
Nessie	CD	CD	Decoupled

Table 1: Systems in the design space.

3.1 Distributed Design

We have designed Nessie to operate in a distributed environment. Borrowing from a peer-to-peer design each node acts as both a client and storage node. We believe this provides for easy integration with applications that perform distributed computation while using the RKVS to store data.

3.2 Client-Driven Operations

Despite performing well, high-performance server-driven designs for RKVSes, like HERD, rely on polling for incoming requests in order services them with the lowest latencies possible. Interference from other threads will increase latencies and decrease throughput. To demonstrate the benefits of Nessie's client-driven design, we conduct a series of microbenchmarks to contrast client-driven operations and server-driven operations under a variety of CPU load conditions. The details of the machines and network used to conduct these microbenchmarks are provided in Section 6 with the exception that these microbenchmarks we use a two nodes, one generating load and the other servicing requests (this node uses 6 cores).

We start by using a single remote thread (the server worker) containing a memory region that several local threads (client workers) wish to access. We perform several microbenchmarks in which the client workers use client-driven `READ` verbs to retrieve data of various sizes and the server worker sits idle Results obtained using this approach are labeled "READ" in Figure 1. We then examine a server-driven system, where client workers make requests using eight byte `WRITE` verbs to the server worker. The server worker polls on incoming requests, and responds to them using `WRITE` verbs using various sizes. In Figure 1 these results are labeled "WR/WR" for write/write because they are implemented using two `WRITE` verbs.

Figure 1 shows the throughput for our microbenchmarks as the data size retrieved per request is in-

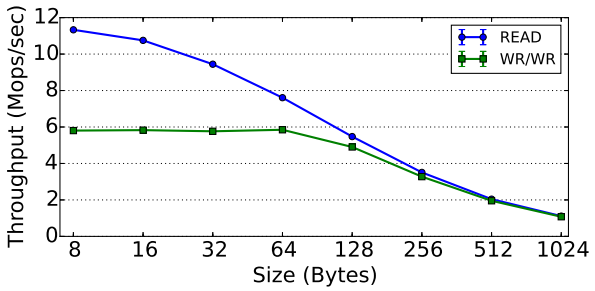

Figure 1: Throughput for a single server node.

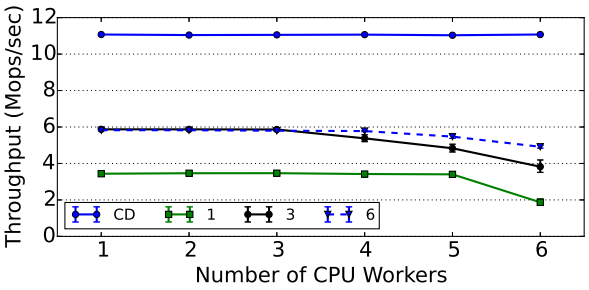

Figure 2: RDMA throughput for a single server node. The numbers in the legend denote the number of server worker threads responding to requests (with CD representing a client-driven experiment).

creased. The number of client workers for each data point is hand-tuned to provide the result with maximum throughput. With small data sizes, the throughput in millions of requests per second of the client-driven design (READ) is significantly higher than that of server-driven design (WR/WR). As the request size grows, both techniques converge to the same throughput. In Section 6.2 we demonstrate the benefits of Nessie's decoupled design by exploiting data locality. Nessie leverages the high performance of small RDMA reads to access remote index tables and then finding and accessing the associated values in the local data table.

We now demonstrate how server-driven mechanisms are impacted by CPU contention. We repeat our previous microbenchmarks with an eight-byte response size and add threads (CPU workers) to the server node. Each CPU worker continuously computes a SHA-256 hash on a small buffer. We vary the number of CPU workers from one to six (the number of cores in the machine), and the number of threads servicing requests (server workers) for server-driven runs. The number of threads generating requests (client workers) is tuned to provide the best throughput for each data point.

Figure 2 shows the total maximum throughput, in millions of requests per second, achieved by all client workers as CPU workers (and thus CPU contention) increase. Each line represents a different number of server workers. The line labeled CD uses no server workers (i.e. it is client-driven), while the lines labeled 1, 3, and 6 use the specified number of server workers to service the load. The client-driven case, shows that CPU contention has no effect on a client-driven workload. The remaining lines, exhibit declining throughput as CPU contention increases. Once server workers suffer from interference from CPU workers (when the sum of the server workers and CPU workers is greater than six in this microbenchmark) throughput begins to decline. This can be seen, for example, with 1 server worker and 6 CPU workers. Distributing request load across multiple server workers helps mitigate the impact of shared CPU resources on client worker throughput (for example, line 6 with 6 CPU workers), but does not eliminate it. We further discuss the benefits of using a client-driven system when evaluating Nessie's performance in Section 6.1.

3.3 Decoupling and Data Locality

The second motivating aspect of Nessie's design is the stark contrast in performance between RDMA operations and local memory accesses. RDMA operations, although much faster than traditional network operations, are still up to 23 times slower than accesses to local memory [11]. With such a large gap in performance between the two types of operations, the advantages of intelligently placing data to exploit local data access are important.

Table 1 shows that previously-existing systems have implemented partially or completely-coupled indexing and storage data structures. Nessie, on the other hand, uses a decoupled design. This approach provides Nessie with complete flexibility with respect to the location of stored values. Nessie is not bound by a launch-time partitioning scheme for its keys. Rather, it is free to place data table entries on whichever nodes it wishes. In Section 6.2 we demonstrate how Nessie exploits localized data access to increase throughput.

In addition to the direct benefits of data locality, Nessie is also able to use its decoupled design to implement a variety of other optimizations. By employing a least recently used (LRU) cache to store recently accessed data table entries, and using filter bits on index table entries, Nessie can further eliminate an operation's data table accesses. These optimizations are described in detail in Section 4.1. For some workloads, these optimizations, significantly reduce the number of remote read operations required to obtain data values. In these cases Nessie only requires `GET` and `PUT` operations to perform low latency remote read operations of relatively small index table entries, which as shown in Figure 1 are extremely efficient.

4. Nessie DESIGN

In this section, we discuss the design of Nessie, a

high-performance RDMA-enabled key-value store. Unlike other RDMA-enabled key-value stores, Nessie is an entirely client-driven design that can leverage data locality to reduce network resource requirements and is also more resistant to performance degradation that occurs as a result of server CPU contention.

4.1 System Components

The Nessie architecture consists of two main components: Index tables, which contain the mapping of key hashes to the location of the data, and data tables, which store the key-value pairs. Each node in the system contains both an index table and a data table. Nessie's decoupled approach allows index table entries to refer to any data table entries in the deployment. Figure 3 provides an overview of Nessie's architecture.

An index table is implemented as an N-way cuckoo hash table [20], where we refer to N as the hash dimensionality. Each index table entry is a 64-bit integer. The choice of using a single 64-bit integer for each entry is to facilitate the use of atomic RDMA `CAS` verbs during writes in our protocols. These RDMA `CAS` verbs prevent inconsistency that could arise from simultaneous reads and writes to the same index table entry. Because `CAS` verbs are only atomic relative to other RDMA operations, all index table reads, whether local or remote, are performed using the RDMA `READ` verb.

Of an index table entry's 64 bits, the first 15 bits are used as a data table identifier, which uniquely identifies a specific data table in the deployment. The next 32 bits are used to determine the offset of a key-value pair within its data table. An offset of zero is used to represent empty index table entries. Following the offset, the next 16 bits are used as a version number to determine if a set of operations are performed atomically (explained in Section 4.2 and Section 4.3). The last bit is used as a watermark for determining when Nessie's RDMA events have completed. By taking advantage of the property that RDMA operations update their bytes in address order, the client process can determine if an RDMA operation has completed by polling on the watermark bit. This provides a lower-latency alternative to the event-raising model also offered by the RDMA interface.

For small to medium size deployments where the total number of nodes can be described in fewer than 15 bits, we provide an optimization that repurpose bits from the data table to reduce the number of data table accesses. By storing the trailing bits from a key's hash, which we refer to as filter bits, in an index table entry, we can determine that an index table entry is not for a particular key if the filter bits do not match. Filter bits can help improve throughput when Nessie's index tables are heavily populated, especially for large hash dimensionalities or when Nessie is storing large key-value pairs.

A data table in Nessie is implemented as a simple array of entries, each containing a key-value pair. The maximum key and value sizes in Nessie are user-configurable at startup, and may be set to suit the storage needs of a specific workload. The last byte of a data table entry is reserved, with 7 bits used as a valid flag and the final bit used for watermark polling. The valid flag is used to denote that an data table entry belongs to an in-progress `PUT` operation. Upon receiving an invalid entry, the request must be retried until in-progress `PUT` operation completes, in which case the valid flag will have been set to true.

Data table entries are accessed remotely using RDMA `READ` verbs, and local memory read operations when the data table entry is on the same node as the client. A key concept for Nessie is that a data table entry is not modified once it has been written, unless it is being deleted and recycled for use with a new value. This ensures that multiple concurrent updates to a value will operate on different data table entries, which prevents data items from being corrupted due to simultaneous writes. Furthermore, by making data entries immutable, Nessie only needs to check the index table to ensure that an entry has not been updated. This enables Nessie to cache remote data entries once they have been read without synchronizing the cache copies. To facilitate efficient caching, Nessie includes a small least recently used (LRU) cache in order to reduce the number of remote data table entry accesses. This optimization improves performance for workloads that either exhibit skewed popularity distributions of their keys, such as Zipf-distributed workloads, or have data table entries that are much larger than 64-bit index entries.

4.2 GET

During a `GET` operation, Nessie first determines which index table entries the key hashes to. These index entries are determined by the N different hash functions used to determine the N slots for the cuckoo hash table. The hash functions are ranked to determine the ordering of entries to access during `GET`, `PUT` and `DELETE` operations. The N index entries can span multiple

Nessie retrieves the computed index table entries using RDMA `READs` sequentially in order of their rank (Figure 3, step 1). Nessie will discard empty index table entries, namely those with a data table entry offset of zero. For non-empty entries, Nessie will determine if there is a mismatch between the filter bits of the index and key hash, treating mismatched entries as empty (Figure 3, step 2). If the data table entry referenced by the retrieved index table entry is on the same node as the client, the data is read directly from memory. For data stored in a remote data table entry, Nessie checks the local data cache for a data item (Figure 3, step 3). On a cache hit, the data is directly retrieved from the

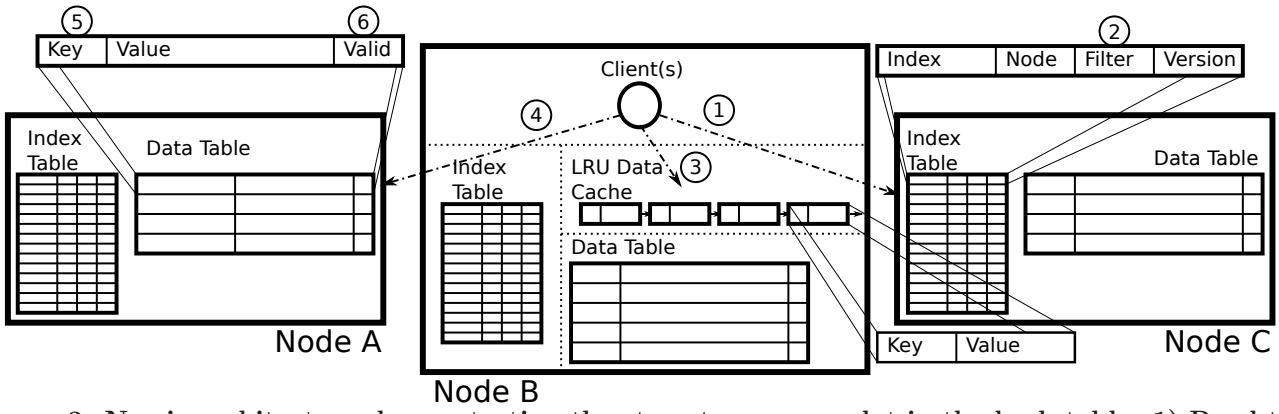

Figure 3: Nessie architecture demonstrating the steps to access a slot in the hash table. 1) Read the index table entry determined by the hash of the key. 2) Check if the filter bits match the desired key. 3) Check if the local data cache contains the current version of the key. 4) Read the data table entry according to the index table entry contents. 5) Check if the data table entry contains the value of the desired key. 6) Check if the data table entry is valid.

cache. Otherwise, Nessie retrieves the data from the remote node using an RDMA `READ` (Figure 3, step 4) and caches it locally if its valid flag is true.

After retrieving the data table entry, its key is compared against the `GET`'s requested key (Figure 3, step 5). On a key mismatch, Nessie examines the next ranked index table entry. If all of the index table entries for the key have been examined, Nessie informs the user that the requested value is not present in the system. In the case where the keys match, Nessie must check the valid flag of the data table entry (Figure 3, step 6). If the entry is invalid, Nessie informs the user that an operation on the requested key is in progress. Otherwise, Nessie returns the value to the user.

It is possible that, due to a cuckoo hash collision from a concurrent `PUT` operation, the requested key's index table entry is migrated from a lower to higher ranked location (described in Section 4.3.1). This can lead to `GET` operation erroneously informing the user that the key is not in the system because it scans index locations from high to low rank. To prevent this, `GETs` record the version numbers of the index table entries as they are retrieved. After examining all the index table entries and finding no matches, Nessie iterates backwards over the index table entries, retrieves them with a second RDMA `READ`, and compares their current version numbers against the recorded ones. If no changes are observed, then no concurrent `PUT` operations have occurred, and the key is not resident in Nessie. Otherwise, Nessie informs the user of the concurrent operation, and the user can choose to retry the `GET` operation.

4.3 PUT

Nessie's `PUT` operation, like its `GET` operation, is client-driven. Because of Nessie's decoupled design, the client issuing the `PUT` operation can decide where the

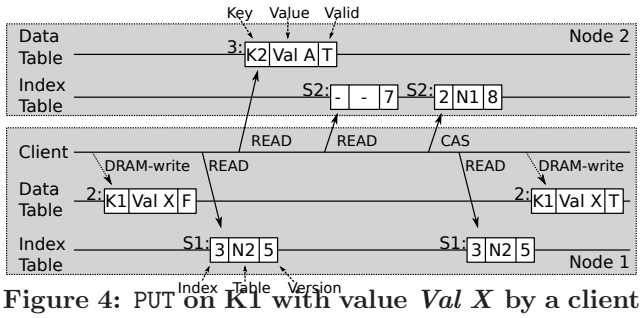

Figure 4: `PUT` on K1 with value *Val X* by a client on Node 1: 1) Write the data to the local Data Table. 2) Read the first hash slot (S1). 3) Read the Data Table entry referenced by S1. 4) Read the second hash slot (S2). 5) `CAS` S2 to reference the new Data Table entry. 6) Verify that S1 has not changed. 7) Make the data valid.

key-value pair is stored. A simple strategy to minimize network traffic is to place the key-value pair in the local data table if space is available. Following this strategy, a `PUT` operation begins by writing the requested key-value pair into a new data table entry in the local data table with the valid flag set to false. The client must then insert a reference to the new data table entry in an index table. To determine which index table entry is updated, the client must iterate through the N index table entries for the key in the order of decreasing rank.

The client determines the status of a index table entry by examining index value using an RDMA `READ`. In the case where the index is empty, the client updates the index entry to reference the new data table entry. If the index is not empty, the client must determine if the index entry references a data table entry whose key matches that of the `PUT` operation. If filter bits are used and they do not match the trailing bits of the key hash, then the keys cannot be a match and the index entry

can be skipped. Otherwise, the client must read the key of the referenced data table entry.

If the referenced data table entry is on the same node as the client, then the key is read directly from memory. For cases where the data entry table is on a remote node, the key is retrieved through the local cache if possible, or from the appropriate data table using an RDMA `READ`. The data table entry's key is checked against the `PUT` operation's key. On a match, the index table entry is updated with a reference to the new data table entry. After examining all N of the index table entries, if no available entry is found, then Nessie must perform a `MIGRATE` operation that reorganizes the occupied entries in the index tables. We discuss migration in detail in Section 4.3.1.

Updating an index table entry requires the use of an RDMA `CAS` verb. `CAS` ensures that the entry is atomically updated, which prevents clients from reading or writing to it at the same time that it is being changed. Additionally, Nessie increments an index table entry's version number each time that it is updated. This is used to avoid potential ABA problems [1].

Once an index table entry has been updated, Nessie must ensure that there are no duplicate lower-ranked index table entries which also refer to a data table entry for the `PUT` request's key. Using the same methodology to retrieve index table and data table entries as before, Nessie continues iterating over the key's N index table entries, comparing the keys of the data table entries against the `PUT` request's key. If a key match is found, Nessie uses an RDMA `CAS` verb to overwrite the relevant index table entry with an empty index value.

Before a `PUT` can report a successful completion to the user, it must make a final check to ensure that its own operations have not been superseded by another more recent `PUT` request, and that it has not been interfered with by a concurrent `MIGRATE`. As with a `GET` request, Nessie does this by iterating over the index table entries it examined in order of reverse rank, reading them using RDMA `READ` verbs. It compares the index table entries it retrieves against those that it retrieved originally. If a mismatch is found, Nessie informs the user that there is a conflicting concurrent operation, and the user may then re-attempt the `PUT` request. Otherwise, Nessie sets the valid flag in the new data table entry to true, and informs the user that the operation completed successfully.

A timeline for a sample `PUT` across two nodes is shown in Figure 4. For simplicity, a hash dimensionality of two is used, and both filter bits and local caching are disabled. In the sample, Nessie writes the new key-value pair, with key K1 and value X, to the local data table. The client uses an RDMA `READ` to examine the first index table entry, which is local, and sees it is non-empty. It uses an RDMA `READ` to retrieve the index table entry's referenced data table entry from the remote data table, and finds a key mismatch. The client then examines the second index table entry, which is remote, using an RDMA `READ`. It sees that this index table entry is empty, and updates it to point to the new data table entry using an RDMA `CAS`. Finally, the client uses an RDMA `READ` to retrieve the first index table entry again. The first index table entry has not changed since it was last examined, and thus the client knows that no concurrent operations have interfered with the ongoing `PUT`. The client therefore sets the new data table entry's valid flag to true, and returns.

4.3.1 MIGRATE

If a `PUT` determines that all of a key's index table entries are occupied by entries for other keys, a `MIGRATE` operation must occur. An index table entry is selected as a migration source, and the data table entry it refers to is copied to a new local data table entry with the valid flag set to false. The key associated with the source entry has one of its alternate index table entries selected as a migration destination. If an unoccupied destination exists, an RDMA `CAS` is used to associate it with the copied data table entry, after which the source entry is cleared using another RDMA `CAS`. The valid flag on the copied data table entry is then set to true. Figure 5 illustrates a sample `MIGRATE` operation.

Once an index table entry has been freed through migration, the `PUT` that incurred the `MIGRATE` is re-attempted. If no destination entry is available, Nessie picks an occupied destination and recursively calls migrate on it before proceeding. At a certain recursive depth, the `PUT` is aborted, and the user is informed that the system's index tables must be made larger before attempting further operations on the same key.

As with `GETs` and `PUTs`, `MIGRATEs` may be affected by concurrent operations. Figure 6 demonstrates such an occurrence where, during a `MIGRATE`, a concurrent `PUT` overwrites an index table entry which Nessie is attempting to migrate. The example shows how Nessie detects and gracefully reverts the failed `MIGRATE` operation, before re-attempting the migration.

4.3.2 DELETE

A `DELETE` operation iterates over a key's N index table entries, retrieving them using RDMA `READs`. For non-empty index table entries, the client retrieves the corresponding data table entry. If the data table entry contains the requested key, the client updates the index table entry to be empty using an RDMA `CAS` operation. If a concurrent `PUT` has updated a high-ranked index table entry, but has not yet cleared a low-ranked index table entry for the requested key, a `DELETE` could reveal stale data to concurrent `GETs`. To prevent this, `DELETEs` iterate over index table entries from lowest rank to high-

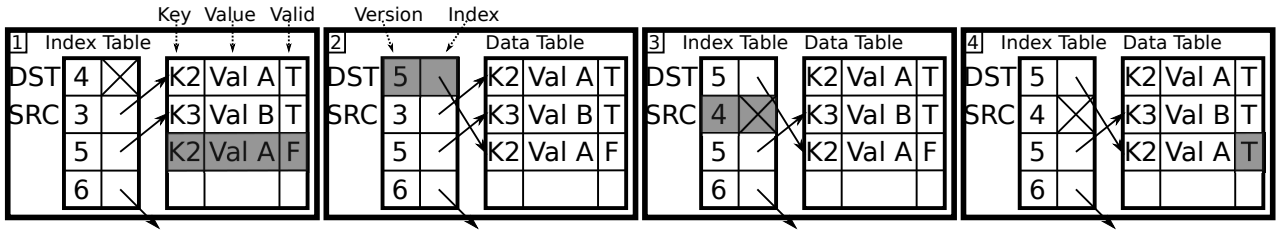

Figure 5: Migration of K2's value from the SRC slot in the index table to the DST slot: 1) Create a copy of the data table entry. 2) Change DST to refer to the new data table entry. 3) Clear the SRC slot. 4) Make the new data valid.

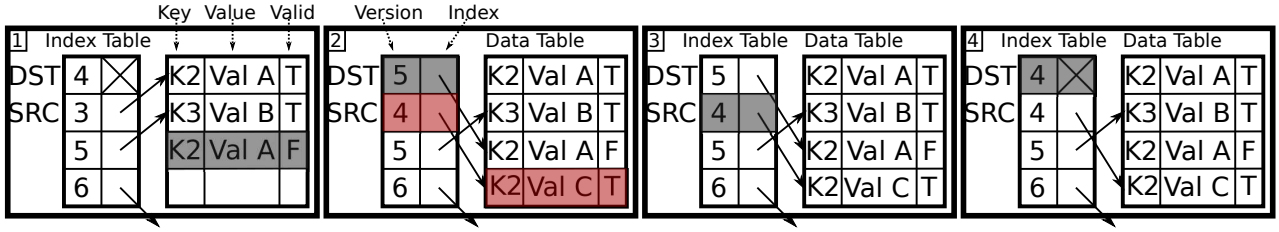

Figure 6: A migration of K2's value from the SRC slot to the DST slot that fails due to a conflict: 1) Create a copy of the data table entry. 2) Change DST to refer to the new data table entry, at the same time another client updates K2's value. 3) Attempt, and fail to clear the SRC slot (using a `CAS`). 4) Restore the old DST slot.

est.

4.3.3 *Data Table Management*

Nessie's decoupled design allows it to place new key-value pairs in any data table in the system. For simplicity, and to maximize Nessie's ability to take advantage of local memory accesses, our current data table management scheme always places new key-value pairs inside a node's local data table. Future modifications could allow Nessie to place the key-value pair in a data table on a user-specified node.

A data table entry is considered in-use if an index table entry contains a reference to it. When a reference to a data table entry is removed, that data table entry can be re-used by a future `PUT`. The node that removes the reference to a data table entry lazily propagates a notice to the data table entry's node. However, if a particularly slow operation reads a reference to a data table entry, and that entry is subsequently reused by another operation, the slow operation would retrieve stale data. We solve this issue by delaying the reuse of a data table entry by a specified amount of time. We further stipulate that successful operations must complete within this time frame, aborting if they exceed it. This allows Nessie to guarantee that, once the timeout has expired, no outstanding operations will contain references to the data table entry, and it can be safely reused.

5. IMPLEMENTATION

We have constructed a working prototype of Nessie on top of a high-performance RDMA networking framework. Nessie allows for quick and easy deployment across a cluster of RDMA-enabled nodes. It supports changes to the sizes of keys, values, the system's hash dimensionality, networking settings, data table sizes, and index table sizes. RDMA functionality for our networking framework is provided through the Libibverbs citelibibverbs and RDMA CM [3] libraries. Nessie uses RDMA reliable connections (RC) to provide network connectivity between nodes in the system. RC connections, which guarantee reliable transmission, are necessary for Nessie's protocol as it requires RDMA `READ` and `CAS` verbs. They are also appropriate for a distributed Nessie cluster, as they prevent the need to implement an extra, external layer of reliability on top of our RDMA networking framework. To provide cuckoo hashing functionality and filter bits for keys, Nessie uses Google's CityHash [2], with different seeds providing the equivalent of separate hashing functions.

In order to compare our client-driven Nessie prototype against an equivalent server-driven approach, we have also used our networking framework to implement a server-driven version of Nessie, called NessieSD. NessieSD's server-driven communication design is inspired by HERD [14], as it uses RDMA `WRITE` verbs to insert requests into a remote server's request memory region. These regions are polled by server worker threads, which service both `GET` and `PUT` requests, and respond to them using RDMA `WRITE` verbs. An exception to this occurs when the data a client is requesting is managed by the same node that the client is running on. In this case, the request and response are written directly, without using RDMA. Although HERD is

described and evaluated within the context of a single server node, we expand the design of NessieSD to run with multiple nodes. NessieSD employs a partially-coupled index and data table design. Like Nessie, NessieSD's index table is implemented as a cuckoo hash table. Unlike Nessie, however, an index table entry in NessieSD only ever refers to data table entries on the same node.

Despite NessieSD being inspired by HERD, we have eschewed some of the low-level RDMA optimizations examined in HERD, including using RDMA unreliable connections (UC) and data inlining (which allows small messages to avoid a copy over the PCIe bus). While we do understand and value the performance benefits of using unreliable connections and inlining demonstrated in the HERD paper, we avoid using unreliable connections to focus on environments that require *guaranteed* reliable delivery. Additionally, our goal is to build a storage system that is not designed specifically for very small sizes and as a result we have not been concerned with obtaining the benefits of inlining. In some regard, such optimizations are orthogonal to our exploration of the design space, in particular the need to use CPUs for other computation besides the continuous polling required by low-latency, high-throughput server-driven designs.

The implementations of Nessie and NessieSD share several portions of the same code base. Together with the underlying RDMA networking framework, they were implemented using 11,000 lines of C++11.

6. EVALUATION

We conduct experiments using 10 nodes, where each node is a Supermicro SYS-6017R-TDF server containing one Mellanox 10GbE SFP port, 64 GB of RAM, two Intel E5-2620v2 CPUs, each containing 6 cores with a base frequency of 2.1 GHz and a turbo frequency of 2.6 GHz. To simplify experiments and to ensure repeatability we have disabled hyperthreading. This also ensures that we avoid situations where threads responsible for handling server-driven requests might be scheduled onto hyperthreads of the same core, thus significantly reducing the processing power available. Each node is connected to a Mellanox SX1012 10/40 GbE switch. All nodes run an Ubuntu 14.04.1 server distribution with Linux kernel version 3.13.0.

All results presented in this section use 2 million key-value pairs with 128 byte keys and 879 byte values. Together, the key, value and ancillary data add up to 1024 bytes, which is the size of a `PUT`, to match the YCSB [9] workload specification. Additionally, roughly 1 million index table entries are provisioned on each node; this achieves a load factor of about 20% which works well with cuckoo hashing. We also provision enough data table entries to avoid reuse during experiments.

Because the performance of server-driven systems relies on being able to use a CPU to poll for requests, their performance is sensitive to other demands on those CPUs. We conducted several experiments, the results of which are not shown, to determine the number of server-worker threads that maximizes the performance of NessieSD. We found that 3 server workers, each pinned to their own dedicated core, and 9 client workers pinned to the remaining 9 cores provided the highest throughput.

6.1 Client-Driven Operations

We first isolate the performance properties of a client-driven design versus a server-driven design. While a client-driven protocol requires more network operations, all of these extra network operations are small `READs` from the index table (64 bits), and eliminate the need to have active server workers for handling requests. To demonstrate the trade-offs, we exercise the RDMA key-value store in the presence of computation, both in separate processes that share the machines, as well as processing within the client workers (emulating an application that, in order to obtain efficient access to data, is executing on the same nodes that are used for the RKVS).

We first consider the impact of external processing on the performance of the key-value store. To do so, we introduce CPU workers, which are external to the client workers and server workers but run on the same machine and consume CPU for unrelated tasks. These CPU workers may be from another application sharing the same machine or other tenants entirely in a data center environment. In the case of a client-driven design, the CPU workers interfere with the client worker's ability to issue `GETs` and `PUTs`.

In contrast, in a server-driven design, a CPU worker may compete with both client and server worker threads. To demonstrate why a server-driven design is not viable in a shared computing environment, Figure 7 shows a comparison of the throughput achieved by Nessie and NessieSD when there is contention for CPU resources. As shown by the line labeled "NessieSD– Unpinned", although there are three server workers, the performance of NessieSD is substantially impacted with as few as one competing process. This is because when server workers do not have full access to CPUs, the performance of every client in the cluster that needs to communicate with that particular server worker is inhibited. In contrast the throughput of the client-driven design degrades much more slowly as competition for CPUs increases.

In order to prevent the severe degradation that occurs when server workers have to compete for CPU resources, we sequester CPUs for exclusive use by server

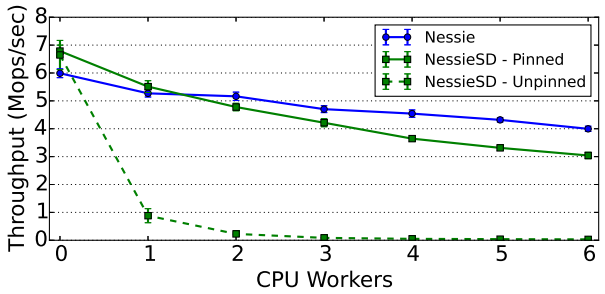

Figure 7: 90% GETs, under uniform random access

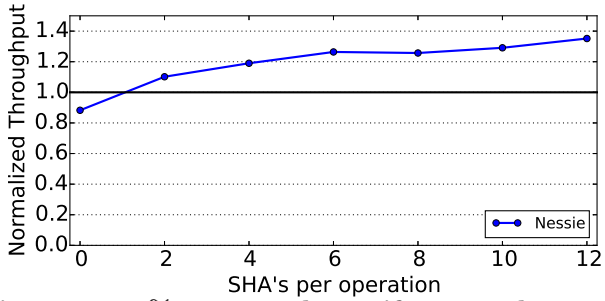

Figure 8: 90% GETs, under uniform random access

workers. This is done by pinning each server worker to its own core and pinning the client workers to the remaining cores. The line labeled "NessieSD– Pinned" in Figure 7 shows that this significantly improves the throughput of the server-driven approach. However, as the number of CPU workers increases, the throughput of the client-driven design approaches 1.33 times that of NessieSD. This is because the client-driven design is able to make use of all 12 cores to perform client work, while the server-driven design has only 9 cores to perform client work (because 3 cores are reserved for the server workers to handle client requests).

We next consider the case where client workers, in order to efficiently access data used in computations, are performing computation on the same nodes that are used to host the RKVS. Figure 8 shows that Nessie provides as much as a 33% improvement over a server-driven approach when processing is introduced in the client workers. Again, this is the expected performance improvement, as the server-driven approach has 9 of 12 cores available to perform computations (3 are reserved for server workers), while the client-driven approach can utilize all 12 cores.

In our environment, with 10 Gbps network links, 3 server workers are necessary to achieve the peak throughput in the server-driven design. However, a deployment with faster NICS (e.g., 40 Gbps) would require more server workers to fully utilize the network. For example, HERD requires 6 server workers to fully saturate a 56 Gbps link [14]. This would increase the ratio of cores dedicated to server workers, which further increases the benefits obtained when comparing a client-driven and server-driven design.

In summary, high throughput in a server-driven environment requires dedicated CPU resources for the server workers. Dedicating CPUs to server workers reduces the CPU resources available for other computation which results in reduced performance for applications that require those resources.

6.2 Leveraging Local Memory

RDMA is a low-latency, high-throughput mechanism for accessing memory from any machine in a cluster. However, despite the order of magnitude improvement over alternative remote data access mechanisms such as TCP, it still has much higher latency compared to local memory access. By decoupling the data table entries from the index table entries, and adding a small cache, we are able to demonstrate substantial performance improvements by reducing network transfers wherever possible without introducing inconsistency.

The first scenario that we consider is when groups of client workers have a shared working set. That is, groups of clients perform `GETs` and `PUTs` to a subset of keys with greater frequency than all of the other keys in the system. An example of such a situation is a graph clustering algorithm, where each vertex is a key, and the access and manipulation is localized by the connectivity in the graph.

To evaluate the performance of workloads that exhibit working sets, we divide the key space into 10 equal part (one for each node) and have client workers on a node access or modify their subset of keys with greater probability than the remaining keys in the system. Figure 9 presents the throughput at various `GET` percentages and an increasing portion of access from the working set. In this graph, a locality value of 0 represents a fully uniform access pattern and 80 represents a workload where 80% of the key accesses come from the working set.

Nessie leverages the fact that `PUTs` place the data table entry into the local data table, enabling subsequent `GETs` to retrieve the value from local memory. Due to the relatively large size of data table entries compared to index table entries, avoiding the network overhead of retrieving a large data table entry significantly reduces the pressure placed on the network, and allows for more small index table lookups to be handled. The result is a 1.8 times increase in overall throughput of Nessie when 80% of the data accesses are from the working set.

A portion of the performance comes from using the filter bits to avoid data table accesses. If a hash of the key does not match the filter bits stored in the index table entry, then the key can not reside in the associated data table entry and therefor does not need to be fetched. This optimization is very similar to accessing

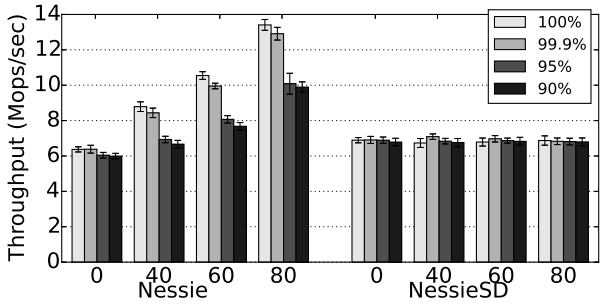

Figure 9: Increasing access from working sets.

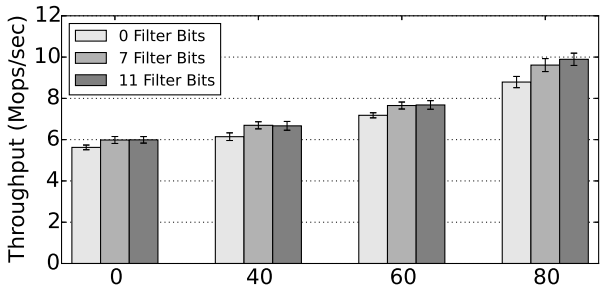

Figure 10: Impact of filter bits at 90% `GETs`, with increasing access from working sets.

the data table entries from local memory, but does not rely on the data table entry being on a particular node.

Figure 10 shows performance as an increasing number of index table entry bits are reserved for filtering, demonstrating an increase in throughput of up to 1 Mops/sec. Notably, there is no statistically significant gain from reserving 11 bits instead of 7 bits for filtering. Because the bits of the index table are a limited resource, it is advantageous that only a small number of bits are needed to provide the benefits, so that the rest can be used to maximize the number of data entries that can be indexed.

Many workloads have a small subset of keys that are globally more popular than all of the other keys in the system. The most common distribution is Zipf, which can be found prominently in web applications. Figure 11 presents the throughput of Nessie and NessieSD under 0.7 and 0.9 Zipf workloads. Due to the skew in popularity, a few nodes see more network traffic than all of the others in both the client-driven and server-driven scenarios, which results in an overall decrease in performance when compared to the uniform random workload.

Caching is a well-understood technique for improving the performance of a storage system that is accessed with a Zipf distribution [7]. By introducing an LRU cache, which stores a copy of up to 1% of the data entries on each node, the performance of the key-value store improves dramatically in the case of 100% and 99.9% `GETs`, where 99.9% `GETs` is typical of a workload distribution for Wikipedia [23]. With a 99.9% `GET` workload and a Zipf parameter of 0.9, Nessie performs over 200% better than NessieSD. However, as the number of `PUTs` are increased, Nessie must perform an increasing number of back-offs due to contention over updates to the most popular keys. The clients become bottlenecked on `PUTs`, and so caching has little impact on performance in these cases.

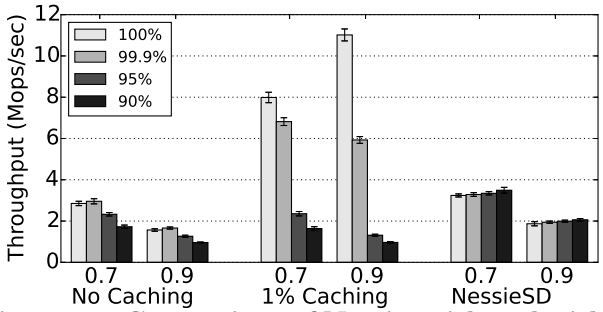

Figure 11: Comparison of Nessie, with and without caching, to NessieSD on a Zipf distribution.

7. CONCLUSIONS

In this paper we design, implement and evaluate the performance of Nessie, a high-performance key-value store that uses RDMA. The novelty of this work derives from its exclusively client-driven operations, in addition to its completely decoupled indexing and storage data structures. Nessie's client-driven architecture eliminates the heavy loads placed on CPUs by the polling threads used for low latency server-driven designs. The decoupling of the location of the index table and data table allows Nessie to perform write operations to local memory, significantly reducing the costs of key-value pair writes and subsequent reads.

8. ACKNOWLEDGMENTS

Funding for this project was provided in part by the University of Waterloo President's scholarships, GO-Bell scholarships, and scholarships and grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada. Additionally, this work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

9. REFERENCES

[1] ABA problem. `http://en.wikipedia.org/wiki/ABA_problem`.
[2] Cityhash. `http://code.google.com/p/cityhash/`.
[3] Rdma cm. `http://linux.die.net/man/7/rdma_cm`.
[4] RDMA over converged ethernet (RoCE). `http://www.mellanox.com/page/products_dyn?product_family=79`.
[5] redis. `http://redis.io/`.
[6] Understanding iWARP. `http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf`.

[7] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE International Computer and Communications Societies* (New York, NY, March 1999).

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

[9] 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* (2010), ACM, pp. 143–154.

[10] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 205–220.

[11] Dragojević, A., Narayanan, D., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (Seattle, WA, April 2014).

[12] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (2004), 5.

[13] Hendler, D., Incze, I., and Tzafrir, M. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing* (2008).

[14] Kalia, A., Kaminsky, M., and Andersen, D. G. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM* (Chicago, IL, August 2014).

[15] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[16] Levinthal, D. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide* (2009).

[17] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–15.

[18] Mitchell, C., Geng, Y., and Li, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference* (San Jose, CA, June 2013).

[19] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[20] Pagh, R., and Rodler, F. Cuckoo hashing. *Journal of Algorithms 51*, 3 (May 2004), 122–144.

[21] Singh, A. K., Cui, X., Cassell, B., Wong, B., and Daudjee, K. MicroFuge: A middleware approach to providing performance isolation in cloud storage systems. In *Proceedings of the International Conference on Distributed Computing Systems* (Madrid, Spain, June 2014).

[22] Szepesi, T., Wong, B., Cassell, B., and Brecht, T. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *Proceedings of the Workshop on Rack-scale Computing (WRSC)* (Amsterdam, The Netherlands, April 2014).

[23] Urdaneta, G., Pierre, G., and Van Steen, M. Wikipedia workload analysis for decentralized hosting. *Computer Networks 53*, 11 (2009), 1830–1845.