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

Benjamin Cassell*, Tyler Szepesi*, Bernard Wong*, Tim Brecht*, Jonathan Ma*, Xiaoyi Liu*
{becassel, stszepes, bernard, brecht, jonathan.ma, x298liu}@uwaterloo.ca

Abstract—Key-value storage systems are an integral part of many data centre applications, but as demand increases so does the need for high performance. This has motivated new designs that use Remote Direct Memory Access (RDMA) to reduce communication overhead. Current RDMA-enabled key-value stores (RKVSes) target workloads involving small values, running on dedicated servers on which no other applications are running. Outside of these domains, however, there may be other RKVS designs that provide better performance. In this paper, we introduce Nessie, an RKVS that is fully client-driven, meaning no server process is involved in servicing requests. Nessie also decouples its index and storage data structures, allowing indices and data to be placed on different servers. This flexibility can decrease the number of network operations required to service a request. These design elements make Nessie well-suited for a different set of workloads than existing RKVSes. Compared to a server-driven RKVS, Nessie more than doubles system throughput when there is CPU contention on the server, improves throughput by 70% for `PUT`-oriented workloads when data value sizes are 128 KB or larger, and reduces power consumption by 18% at 80% system utilization and 41% at 20% system utilization compared with idle power consumption.

Index Terms—Computer networks, data storage systems, distributed computing, distributed systems, key-value stores, RDMA

1 Introduction

By eliminating slow disk and SSD accesses, distributed in-memory storage systems, such as memcached [17], Redis [6], and Tachyon [24], can reduce an application's request service time by more than an order of magnitude [27]. In the absence of disks and SSDs, however, an operating system's network stack is often the next largest source of application latency. 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 drive for increasing performance has led recent in-memory storage systems to use Remote Direct Memory Access (RDMA) [14], [21], [26], [30]. RDMA enables direct access to memory on a remote node, and provides additional performance benefits such as kernel bypass and zero-copy data transfers. These RDMA-based systems are able to provide lower latencies and higher throughput than systems using TCP or UDP.

Although existing RDMA-enabled key-value stores (RKVSes) perform well on workloads involving small data sizes and dedicated servers, we have identified three environments for which these designs are not well-suited. The first environment includes those where CPUs can not be dedicated for exclusive use by the RKVS. This includes applications where it is advantageous for distributed computation and storage to be integrated into the same node, or where physical CPUs are shared by multiple virtual machines. Existing RKVSes achieve low latency by using dedicated CPUs on the server to poll on incoming requests [14], [21]. We refer to these systems, in which client requests are forwarded to a dedicated server thread for processing, as server-driven. This approach is susceptible to performance degradation in environments with multiplexed resources, as an increase in CPU usage from a co-resident application or VM can cause context switching that delays the processing of requests, and significantly reduces the throughput of the RKVS. This delay is amplified when a server-thread is preempted, preventing progress for *all* clients communicating with that server thread.

The second environment that current designs have not focused on are those where the RKVS is required to store large data values. Existing RKVSes use designs that couple indexing and data storage, resulting in rigid data placement and unavoidable data transfers for `PUTs` [11]. As data value sizes grow, network bandwidth becomes a bottleneck for performance. This presents issues for a variety of workloads, for example when using an RKVS as a memory manager for a distributed computation framework such as Apache Spark [31] which manages data units that are hundreds of megabytes in size. When indexing and data storage are coupled, writing new data of this size results in significant overhead as the data is sent over the network to the node responsible for its key.

The third environment includes those where reducing energy consumption is important. As previously explained, server-driven RKVSes use polling threads alongside RDMA to decrease operation latency. With the current speed of network interfaces, however, a single polling thread per machine is often insufficient to saturate the link. Instead, these RKVSes must dedicate multiple polling threads to handling requests. While this is useful when operating at peak load, polling during off-peak periods is energy-inefficient. In some parallelized data centre applications, it is possible to consolidate resources by shutting down VMs during periods of inactivity. For a big memory storage sys-

* *Cheriton School of Computer Science, University of Waterloo*

tem, however, this means frequently reorganizing hundreds or possibly thousands of gigabytes of data.

In this paper, we present the design of Nessie, a new RKVS that addresses some of the limitations of current RKVSes. Nessie is fully client-driven, meaning requests are satisfied entirely by the client using one-sided RDMA operations that do not involve the server process' CPU. One-sided reads and writes allow Nessie to eliminate the polling which is essential for providing high throughput in server-driven RKVSes. This makes Nessie less susceptible to performance degradation caused by CPU interference in shared CPU environments such as in the cloud. Furthermore, Nessie only consumes CPU resources when clients are actively making requests in the system, reducing Nessie's power consumption during non-peak loads. Finally, Nessie's decoupled indexing and data storage allows indices to refer to data stored on different nodes in the system. This enables more flexible data placement, for instance local writes, and can benefit write-heavy workloads by exploiting locality and reducing the volume of remote operations.

This paper makes three main contributions:

- We describe the design of Nessie, and show how its decoupled, fully client-driven design allows it to operate without server-side interaction.
- We evaluate a prototype of Nessie. For comparison purposes, we also evaluate NessieSD, a server-driven version of Nessie that uses polling, similar to HERD [21]. We futhermore evaluate NessieHY, a hybrid version of Nessie that uses client-driven `GETs` and server-driven `PUTs` similar to FaRM [14].
- We identify three problematic workloads for existing RKVSes: Workloads in shared CPU environments, workloads with large data values, and workloads where energy consumption is important. We show that Nessie performs significantly better than server-driven approaches for these workloads.

2 Background and Related Work

This work is a follow-up to our previous position paper [29]. We extend our preliminary design and introduce new mechanisms and performance optimizations. Additionally we provide a full system implementation and evaluation. In this section, we provide an overview of RDMA technologies and survey other in-memory key-value stores.

2.1 RDMA

Remote Direct Memory Access (RDMA) is an alternative to networking protocols such as TCP or UDP. RDMA can bypass the kernel, providing zero-copy data transfers between local and remote memory. Therefore, RDMA has less overhead than traditional protocols, with network operations completing in several microseconds. RDMA is often associated with Infiniband hardware deployments, but is also compatible with traditional Ethernet networks using the iWARP [7] or RDMA over Converged Ethernet (RoCE) [5] protocols. RDMA-enabled network interface cards (NICs) are becoming increasingly commonplace in data centre settings as costs continue to fall. In fact, RDMA was competitively priced as far back as October 2013 [14]. RDMA can even be used in virtualized environments. RDMA inside a VM has been shown to, for larger data transfer sizes like those we are focused on, provide performance comparable to that of using RDMA directly on hardware [13].

Communication over RDMA is performed using the verbs interface, consisting of one-sided and two-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. This exchange 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.

One-sided verbs allow direct access to pre-designated regions of a remote server'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`). These verbs both operate on 64 bits of data, and are atomic relative to other RDMA operations on the same NIC.

RDMA supports an asynchronous programming model in which verbs are posted to a send and receive queuing pair. By default, operations post an event to a completion queue when they have finished. An application may block or poll on this queue while waiting for operations to complete. Alternatively, as noted by the authors of FaRM [14], 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 (KVSes) are increasingly popular storage solutions that map input identifiers (keys) to stored content (values). They are frequently employed as tools in application infrastructure and as components of other storage systems [26]. Most KVSes 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 KVSes, such as Redis [6], provide persistent storage while others, including memcached [17], are used for lossy caching. Column storage systems, such as BigTable [10] and Cassandra [23], are used to store large volumes of structured data. Systems such as RAMCloud [27] serve a similar purpose but were designed with the intention of storing data entirely in main memory.

2.3 RDMA Key-Value Stores

RDMA has recently been employed as a means of increasing the performance of KVSes. We first explore the dimensions of the design space of RDMA-enabled key-value stores (RKVSes). We then 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 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 performed entirely by the client using one-sided verbs, a server-driven (SD) communication mechanism where a remote server is instructed to perform an operation, or a combination of the two. The main distinction between CD and SD mechanisms is what type of RDMA operations are being used and, as a result, which CPUs are responsible for processing requests.

TCP/IP-based KVSes use SD communication mechanisms, requiring server-side interaction in order to service requests. This can reduce the complexity of synchronizing resource access from multiple clients, as the server determines the request order and provides mutual exclusion for key accesses. However, SD mechanisms require server CPU processing to handle each request. For high-performance systems that require low packet processing latency, one or more dedicated CPUs are needed to poll for incoming requests to minimize response latency [21], reducing CPU resources available for other purposes on the server. Although it is possible for a server-driven system to use a mechanism whereby the server reverts to a higher latency blocking model during periods of low activity, in practice existing systems such as FaRM and HERD have rejected the use of blocking communication mechanisms as they introduce unacceptable latency overhead.

RKVSes can take advantage of RDMA's one-sided verbs to perform CD 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. The server's idle CPU can instead be used for other purposes, or powered down to conserve energy. Furthermore, by bypassing the operating system and application on the server side, CD mechanisms have the added benefit of lower latencies than SD mechanisms [21]. However, without a server process 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 using an O(1)-lookup data structure. 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, which we refer to as complete coupling. Coupling's effect on data placement can impact the efficiency of data lookups for a system. Whereas systems with tight coupling have limited options in terms of placement, a decoupled system can place data to exploit locality and migrate data to suit dynamic access patterns.

2.3.2 Existing Designs

Pilaf [26] is a partially-coupled RKVS that stores its index in a cuckoo hash table [28]. 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. To avoid consistency issues caused by simultaneously issuing `PUTs` and `GETs` for a key, Pilaf's hash table entries contain checksums which allow the client to determine whether or not a `GET` has retrieved a corrupted value.

FaRM [14], alongside its distributed memory model, presents an RKVS design that uses a variant of hopscotch hashing [19] to store its index. FaRM operates either completely or partially-coupled, 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 CD and SD communication mechanisms and must therefore manage the possibility of conflicting concurrent operations. This synchronization is provided through the use of version numbers on entries, which are protected through the use of local `CAS` operations [15].

HERD [21] is a partially-coupled RKVS that uses set-associative indexing. HERD is fully SD, 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 a number of specific 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 omit sending acknowledgment packets for successful operations.

DrTM [30] is an RDMA-enabled system that executes transactions over large data sets. DrTM distinguishes between storing unordered data and ordered data, and furthermore decomposes `PUT` operations into inserts (for creating new data) and updates (for modifying existing data). All DrTM operations performed on ordered data are server-driven, and use RDMA `SEND` and `RECV` verbs to pass requests to the appropriate remote nodes. For unordered data, `GETs` and updates both use client-driven RDMA operations, whereas `DELETEs` and inserts use server-driven RDMA operations. DrTM uniquely incorporates hardware transactional memory (HTM) into its design using it for operations on data stored in local memory to increase throughput. These techniques restrict DrTM to deployments that support HTM, and require DrTM to be implemented as a partially-coupled system.

The approaches discussed here may not be appropriate for certain workloads. In Section 3 we discuss alternative design principles to those of existing RKVSes, and how they allows a system such as Nessie to handle workloads that can be problematic for existing RKVSes.

3 MOTIVATION

Existing RKVSes use server-driven operations for at least some aspect of their `PUT`s, and none are decoupled. Certain workloads cause performance problems for existing RKVS designs, and can be better addressed by a client-driven, decoupled system. In this section, we touch on the high-level design principles of a client-driven, decoupled system, and outline workloads for which such a system would outperform existing RKVSes.

3.1 Design Principles

Deployment Model: A client-driven, decoupled system is designed to operate in a distributed environment, with each machine acting as a request-issuing client and as a storage node. This provides for easy integration with distributed computation frameworks such as Apache Spark [31]. More generally, any workload containing collocated CPU-bound and memory-bound tasks is well-suited to Nessie. This includes web servers, which often consist of a CPU-bound process serving small amounts of pages from memory, as well as a in-memory cache or storage system, for example memcached [3], that requires little in terms of CPU utilisation but consumes large amounts of memory.

Client-Driven Operations: Operations in a client-driven, decoupled system are exclusively one-sided, comprised primarily of RDMA `READ`s and RDMA `CAS`es. While such a system does use one-way RDMA `WRITE`s as well, no active polling is involved on the server-side. This adds complexity to the protocol, but provides unique opportunities for performance improvements, as we explain in Section 3.2. We provide a comprehensive breakdown of a client-driven protocol in Section 4.

Decoupled Data Structures: Although RDMA operations are quite fast relative to traditional network operations, they are still up to 23 times slower than a local memory access [14]. Therefore, taking advantage of local memory is an important part of maximizing performance in a distributed environment. However, the ability of systems that use complete or partial coupling to leverage local memory is limited by relying on a static partitioning of the data that is determined ahead-of-time. By using fully decoupled data structures, the system avoids a static launch-time partitioning scheme for its keys and values. A decoupled data structure is free to dynamically partition the data at run time, allowing it to exploit local memory for performance purposes. Furthermore, the system can implement other optimizations such as a local cache to store recently accessed remote data table entries, and bits from the key hash, which we call filter bits, in index table entries to reduce unnecessary remote memory accesses. These techniques and optimizations are described in detail in Section 4.

3.2 Targeted Environments

Shared CPU Environments: For modern computing workloads, it is often impossible to fully control the environment in which a KVS runs. This is especially common when using a cloud or shared cluster deployment. In the cloud, a physical machine is typically shared between multiple VMs, and its resources are oversubscribed. As we will demonstrate in Section 6, an oversubscribed CPU leads to contention between multiple guest VMs, and as a result the throughput of a polling server worker thread suffers.

A common strategy for dealing with CPU contention is to isolate processes through tuning or pinning processes to particular CPUs, but this is difficult or impossible to do in a shared CPU environment such as a cloud VM. Every time a polling server thread is pre-empted in order to share the CPU with another thread, *all clients* that have outstanding requests for that server to process experience a substantial delay. The magnitude of this delay can be quite severe given that RDMA operations typically complete on the order of microseconds, whereas timeslices are often in the millisecond range. As a result, if polling servers do not execute on dedicated CPUs, the throughput of the system can be dramatically reduced. A client-driven protocol eliminates server worker threads as potential sources of latency for operations. This leads to a greater degree of resilience against the effects of CPU contention.

Large Data Values: Previous RKVSes were designed primarily to provide high performance for operations involving small data values. FaRM [14] and HERD [21] in particular demonstrate significantly better performance when data values are small. Small data values allow FaRM, for instance, to inline data values into its indexing data structure, which is an example of a complete coupling design. HERD, on the other hand, is explicitly designed with only small values in mind, and does not consider workloads with larger data values.

Many applications require storage for, and low latency access to, larger objects, including web pages, photos, and distributed data. Apache Spark [31], for example, operates on objects divided into units called partitions, where each partition is typically 64 MB to 128 MB in size. Another system, Redis [6], is an in-memory data structure store popular with web server operators for its ability to cache complex structures such as lists and sorted sets. For example, Pinterest, a content sharing service, uses Redis to store social data structures, including lists of users and pages followed by other users [9]. Given the complexity of the data structures stored by Redis, they can grow to large sizes. Nessie's use of decoupled indexing and storage structures, together with other optimizations that reduce data transfer during `GET` and `PUT` operations by encouraging the use of local memory access, helps to improve performance for workloads involving large data values.

Energy Consumption: The polling nature of server-driven RKVSes implies that for these systems, CPU on certain cores of a node will always be 100% busy. This is appropriate for workloads that constantly saturate the system, but many workloads, including those found in data centres, exhibit bursty traffic patterns, with occasional busy peaks but also periods of little or even no traffic. Heller et al. [18] showed, for instance, that traffic exhibited diurnal behaviour in both a large e-commerce application and a production Google data centre. Berk et al. [8] likewise showed diurnal and weekly traffic patterns for requests to Facebook's memcached deployment. During any period of downtime for a bursty or light workload, a server-driven RKVS would

continue using relatively large amounts of CPU, wasting electricity. The client-driven design, which precludes the need for polling server threads, results in lower power consumption for these same types of workloads.

4 Nessie Design

4.1 System Components

Nessie is an RDMA-enabled key-value store that decouples indexing metadata, which is stored in data structures we refer to as *index tables*, and key-value pairs, which are stored in data structures we refer to as *data tables*. Each instance of Nessie contains an index table, a data table, and a small local cache of remote key-value pairs. Figure 1 depicts a sample Nessie deployment across three nodes, with one Nessie instance per node.

4.1.1 Index Table

Index tables are implemented as N-way cuckoo hash tables [28], with each key hashing to N entries across the index tables in a deployment. Cuckoo hashing was chosen because it only needs to examine a maximum of N entries on each read operation, and is simple to implement without requiring coordination between nodes. Other approaches would yield different performance properties. For example, hopscotch hashing may require fewer network operations under certain scenarios. We plan to explore other hashing alternatives in future work. Each index table entry (ITE) is a 64-bit integer, allowing ITEs to be operated on by atomic RDMA `CAS` verbs. This atomic access eliminates inconsistencies that could result from simultaneous writes to the same ITE. Because `CAS` verbs are only atomic relative to other RDMA operations on the same NIC, all ITE accesses (even for local ITEs) are managed using RDMA. This means that each NIC may be responsible for one or more index tables, but no index table may have entries that are being managed by more than one NIC. The small size of index table entries allows Nessie's index tables to have a very large number of entries without using much memory.

A possible concern with using RDMA atomic operations is that they can negatively affect performance [22]. Nessie reduces the performance impact of using these operations by using them judiciously. RDMA `CASes` only occur during `PUTs` and `DELETEs`, and `PUTs` will typically only require a single `CAS`. Additionally, because Nessie is concerned primarily with large values, the overhead of working with these values tends to dominate workloads. Moreover, on some RDMA-enabled hardware which we did not have access to, proper placement of data can significantly improve performance when working with atomic operations. Kalia et al. [22] demonstrate that some NICs use buckets to slot atomic operations by address. Nessie's ITEs are stored in memory that is completely separate from data table entries (DTEs), meaning that on such hardware, an RDMA `CAS` on an ITE would not conflict with concurrent access to DTEs, and the large size and low load factor on the index table could also be exploited to further reduce contention. Even without special hardware support, Nessie has the ability to use multiple NICs by creating multiple index tables and data tables and partitioning them between NICs. This can further reduce contention for the NIC by different requests, ensuring that NIC resources do not bottleneck the system.

The 64 bits of an ITE are flexibly divided between several fields, as illustrated in Figure 1. The first field is a data table identifier (DTI), which uniquely identifies a data table in the deployment. The next field is an identifier that determines a particular DTE in a given data table. Taken together, a DTI and DTE identify a spatially unique portion of memory in the cluster. This spatial uniqueness is combined with an expiration-based system in order to resolve concurrency and caching issues (see Section 4.3.1). A third field, called the empty bit, is a single bit wide and is used to represent whether or not a particular ITE is empty. Because RDMA operations update their bytes in address order, the last bit of an ITE is used as a watermark which clients poll on to determine if an operation has completed [14]. This provides a lower-latency alternative to RDMA's event-raising model [14], [21]. Finally, any remaining bits which are not needed for the DTI and DTE fields may be used for a field that we call the filter bits, which helps reduce network accesses to DTEs.

4.1.2 Data Table

Nessie's data tables are simple arrays of key-value pairs and metadata. Key and value sizes are configurable. DTEs contain a valid bit, denoting whether or not a DTE belongs to an in-progress `PUT` (see Section 4.2.2). Like ITEs, DTEs also contain a watermark bit on which clients poll to determine whether or not an operation has completed. The fields of a DTE are arranged such that keys and metadata may be read without reading the value field. Furthermore, DTEs also contain an eight-byte timestamp and a recycle bit, which are used to implement a simple expiration-based system for reclaiming DTEs that are no longer in use (see Section 4.3.1), and an eight-byte previous version index, which is used to reduce the abort rate of `GETs` (see Section 4.3.2).

Once a DTE is assigned to an ITE, that ITE has ownership over the DTE. Apart from DTEs moving from invalid to valid, DTEs are not modified after being written. Nessie's protocol ensures that it is impossible for operations to read DTEs which are semi-complete or are in-between valid and invalid. DTEs may therefore be accessed remotely using RDMA `READ` verbs, and locally using simple memory reads.

4.1.3 Local Cache

The immutable nature of valid DTEs allows them to be locally cached, using their spatially unique ITE as a cache key. This caching reduces network load, particularly for workloads with skewed popularity distributions. Our current implementation of Nessie uses a small least recently used (LRU) cache on each node. When a key's associated entries are modified, its newly assigned ITE will be spatially unique from the key's previous ITE, and the key's old cache entry will be evicted over time as further entries are accessed. The same expiration-based strategy used to guarantee correctness when accessing DTEs remotely is also used with the cache in order to prevent stale entries from being returned in the case where an application waits for a long duration before accessing the cache again.

4.2 Protocol Overview

In this section, we present a protocol overview of each Nessie operation. We later introduce additional mechanisms that are used to ensure correctness and improve performance. For additional details about these operations, we direct readers to Appendix A, which provides step-by-step breakdowns of each operation in addition to detailing how they interact in terms of consistency.

4.2.1 Get

At a high level, Nessie's GET protocol consists of a forward pass searching for a DTE containing the requested key, and a reverse pass to verify consistency. During the forward pass, Nessie computes a list of possible ITEs for the requested key using its cuckoo hashing functions. Nessie iterates over these ITEs, retrieving them using RDMA READs. ITEs that are empty or have filter bits that do not match those of the requested key are skipped, as they cannot possibly refer to a DTE that contains the requested key. ITEs that are not skipped are used to retrieve candidate DTEs, either from the local cache, directly in memory from a local data table, or using an RDMA READ from a remote data table. The key in a retrieved DTE is compared against the requested key, and if they do not match then Nessie continues to look for a match by iterating over any remaining ITEs.

Pending PUT operations set a valid flag in DTEs to false. To avoid returning inconsistent data, Nessie checks this flag when retrieving DTEs and returns a concurrent operation error if the DTE is invalid. A GET that fails in this manner may retry after a back-off period. The use of previous versions, an optimization described in Section 4.3.2, ensures that back-offs for GETs happen infrequently. If a valid DTE is found containing a matching key, it is cached and the value is returned. Figure 1 illustrates the basic steps involved in a sample GET operation. The labelled steps in the figure are explained in the caption.

If the requested key is not found in any of the candidate DTEs, another pass is made over the ITEs to account for an edge case. During the forward pass, a concurrent MIGRATE could have moved a yet-unexamined ITE into an already-examined ITE's slot. Therefore, a GET re-examines the ITEs in reverse order to determine if any have changed since they were last read. If a change is detected, a concurrent operation error is returned, and the GET is re-attempted after a back-off period. Otherwise, Nessie returns that the requested key is not in the system. Given that a key-value store is typically used to access elements that are known to be present, performing this reverse pass is rare in practice. Finally, additional precautions are taken to prevent Nessie from returning recycled DTEs and stale cache values. These preventive measures are described in Section 4.3.1. More details on GETs can be found in Appendix A.

4.2.2 Put

Abstractly, Nessie's PUT protocol performs a forward pass over a requested key's possible ITEs to insert a new entry. After inserting a new entry, the forward pass continues iterating over any remaining ITEs checking for conflicting entries to remove. The PUT then performs a reverse pass and aborts if any conflicting PUTs or MIGRATEs are detected. These forward and reverse iterations ensure that only one operation in a set of conflicting operations completes successfully.

During the forward pass, Nessie iterates over a requested key's possible ITEs, searching for a candidate ITE that is empty or refers to a DTE whose key matches the requested key (and can therefore be overwritten). This iteration is identical to that used by GETs. When retrieving DTEs during a PUT, values are excluded as they are not needed. If the forward pass finds that all ITEs for a requested key are non-empty and do not refer to matching DTEs, then a MIGRATE is performed to free one of the ITEs.

With a candidate ITE chosen, Nessie must allocate a DTE for the PUT's key-value pair. Nessie's decoupled design allows this DTE to be placed on any node. In our prototype, the DTE is placed on the same node as the client making the request. This placement scheme is useful for a variety of workloads, as it avoids network roundtrips and provides locality for subsequent operations on the key from the same node. We discuss alternate placement schemes in Section 7, as well as the selection of empty local DTEs and their re-use in Section 4.2.5 and Section 4.3.1. The DTE contains a valid bit that is initially set to false, signifying that it belongs to an in-progress operation.

Once a new DTE has been written, the candidate ITE is atomically updated using a CAS verb to contain the data table identifier and index of the new DTE. If the CAS on the candidate ITE fails, an error is returned and the caller may re-attempt the PUT. If the CAS succeeds and the candidate was initially non-empty, the candidate's original DTE is staged for recycling once the PUT completes. After updating the candidate ITE, the PUT continues its forward pass, iterating over any unchecked ITEs to ensure there are no duplicate entries for the key. Any duplicate entries that are found are marked as empty using a CAS, and their DTEs are staged for recycling once the PUT completes.

After the forward pass, a reverse pass is used to check for interference from concurrent operations. The PUT iterates backwards over the key's possible ITEs and verifies that none of them have changed since they were last read. If a discrepancy is detected, a concurrent operation error is returned, and the caller must retry the PUT. Otherwise, the new DTE's valid bit is flipped to true. Any values that have been staged for recycling are marked with expiration times using RDMA WRITEs, although for latency purposes Nessie may also delay this step until the PUT is complete and the system is not busy. DTE expiration is further discussed in Section 4.3.1. The PUT then returns successfully.

An example of a PUT can be seen in Figure 2 (details for each numbered step are described in the figure caption). For simplicity, the figure uses two cuckoo hashing functions, ignores filter bits, and assumes that no interference occurs. In the sample, the first checked ITE refers to a DTE with a key mismatch, and the second ITE is found to be empty. A detailed breakdown of PUTs in Nessie is provided in Appendix A.

4.2.3 Migrate

If a PUT determines that all of a key's candidate ITEs point to DTEs containing other keys, one of the candidates must be freed by migrating its contents to an alternate location.

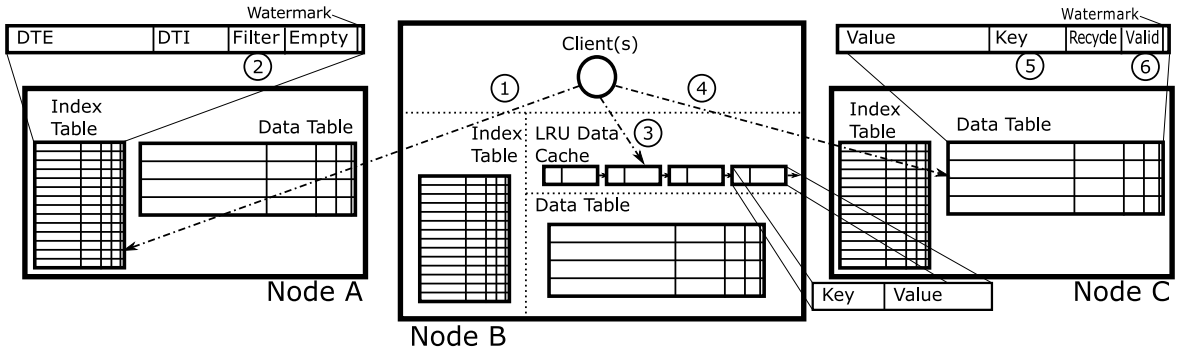


Fig. 1: System architecture overview and sample GET operation on node B. Each node contains a Nessie instance, index table, and data table (for simplicity, previous versions are omitted). (1) A node and ITE is determined by the key's hash and is read with RDMA from node A's index table. (2) The ITE's filter bits match the key's filter bits. (3) No entry is found for the ITE in node B's local cache. (4) The ITE's DTI and DTE are used for an RDMA read from node C's data table. (5) The DTE's key matches the requested key. (6) The DTE is valid, and the value is returned retrieved.

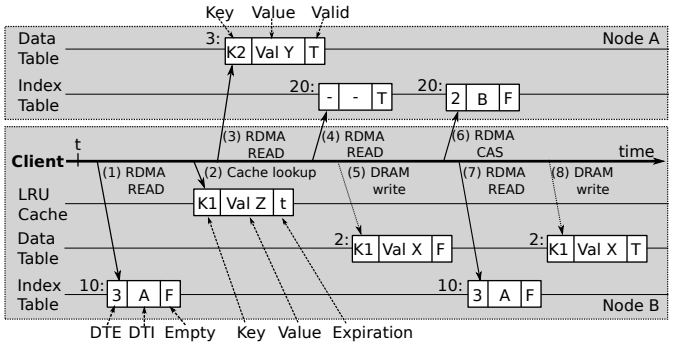


Fig. 2: Sample PUT on key "K1", value "Val X" by a client on node B: (1) Starting at time t, read the ITE determined by the first hash function on "K1" (ITE 10 on node B). (2) Local cache lookup on key "K1" returns a match with expiration time t. This expired cache entry is evicted. (3) Read the referenced DTE (DTE 3 on node A). The contents are for a different key. (4) Read the ITE determined by the second hash function on "K1" (ITE 20 on node A). (5) ITE 20 on node A is a valid (unused) candidate. Create a new local DTE. (6) CAS ITE 20 on node A to refer to the new DTE. (7) Verify that ITE 10 on node B has not changed. (8) Make the new DTE valid.

A candidate ITE is selected as a *source*. In our prototype implementation, we choose the first candidate ITE as the source. The MIGRATE is aborted if the source refers to an invalid DTE to avoid interfering with an in-progress operation. Otherwise, the DTE referred to by the source is copied to a new local DTE, and the new DTE's valid bit is set to false. The list of candidate ITEs is computed for the key in the newly-copied DTE, and one of these candidates is selected as a *destination*. In our prototype implementation, the destination is chosen to be the first candidate which is different from the source.

If the destination is not empty, Nessie frees it by performing a recursive MIGRATE using the destination as a source. Once the destination is empty, an RDMA CAS overwrites it to point to the newly-copied DTE, and the source ITE is marked as empty using another CAS. The newly-copied DTE's valid bit is set to true, and the ITE originally referred to by the source is marked for recycling. The use of a new DTE for each operation, in combination with the DTE expiration times and operation timeouts discussed in Section 4.3.1, ensures that it is impossible for an operation, such as a GET, to miss an existing value due to concurrent operations. For example, if a GET reads an ITE during its forwards pass, and a MIGRATE subsequently moves an existing value into that ITE, the ITE now contains indexing data that is different from what was recorded by the GET. Even if further operations change the ITE, the GET will detect a mismatch on its backwards pass and return a concurrent operation error. Figure 3 illustrates an example MIGRATE operation (step by step descriptions are provided in the figure caption).

In theory, multiple recursive MIGRATEs may be needed to free a destination ITE. Therefore, at a configurable recursive depth the PUT is aborted and an error is returned, informing the caller that the system's index tables must be made larger or data must be deleted before re-attempting the PUT. In practice, because ITEs in Nessie are only 64 bits, Nessie is able to use large index tables with low load factors without consuming much memory. This makes even non-recursive MIGRATEs extremely rare. A comprehensive examination of MIGRATEs is provided in Appendix A.

4.2.4 Delete

DELETEs in Nessie are identical to PUTs, but instead of inserting a DTE with a specified value, a DELETE inserts a DTE with an empty value. Just as with PUTs, any operations running concurrently to a DELETE on the same key will encounter an invalid DTE, allowing them to abort, back-off, and retry. Unlike a PUT, a DELETE never marks its inserted DTE as valid. Instead, the DELETE performs one last CAS on the ITE referring to the DTE, marking the ITE as empty. The DELETE then marks the DTE for recycling and returns. Detailed information about DELETEs can be found in Appendix A.

4.2.5 Data Table Management

Each node maintains a list of data table entries which may be allocated during PUTs or MIGRATEs. A DTE is in-use if an ITE contains a reference to it. When an ITE is changed to be empty or to reference a different DTE, the original DTE it referred to can be re-used after an expiration period. Because ITEs are updated atomically using RDMA CAS verbs, and only the thread that clears an ITE is allowed to recycle a DTE, it is impossible for multiple operations to simultaneously mark a DTE for recycling. Therefore, direct memory writes and RDMA WRITE verbs (depending on

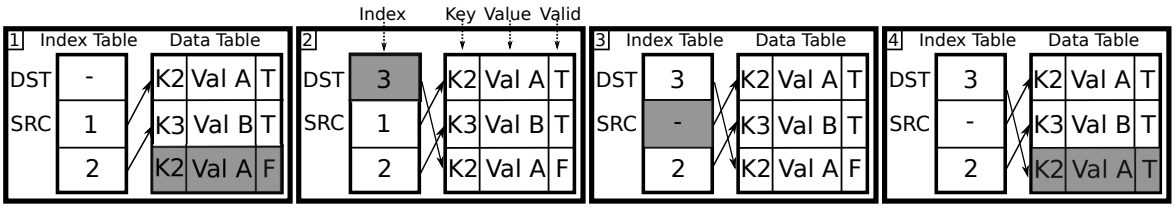


Fig. 3: Migration from ITE "SRC" to ITE "DST". For simplicity, the index table's DTI and DTE are combined into a single "Index" field. The shading highlights the component being acted on in each step: 1) Create an invalid copy of SRC's DTE. 2) Change DST to refer to the new DTE. 3) Clear SRC. 4) Make the new DTE valid.

locality) are sufficient to set the recycle bit and expiration timestamp of a DTE.

When a node runs out of available DTEs, it scans its DTEs for entries that have been marked for recycling, and adds them back into its list of available entries. This scan is performed when available DTEs are exhausted, but to reduce the frequency of this occurring on the critical path, it may be performed lazily by a background thread at scheduled intervals or when the CPU is idle. Likewise, the act of marking DTEs for recycling can also be moved into a background thread, removing the need to do so from an operation's critical path. To avoid the case where a particularly slow `GET` or `PUT` operation reads an ITE, waits for a long duration, and then erroneously reads a DTE which has been recycled, we use a simple time-based expiration system. This is explained in greater detail in Section 4.3.1.

4.3 Protocol Modifications

In addition to the basic components of the Nessie protocol described in Section 4.2, several other techniques are used by Nessie to achieve better performance for certain workloads, and to avoid consistency issues when recycling memory. In this section, we examine these additional mechanisms.

4.3.1 Correctness

Aborting migration: `MIGRATEs` may be interrupted by encountering an invalid source ITE, or by failing a `CAS` on the destination or the source ITE. If a `MIGRATE` encounters an invalid source, or is unable to `CAS` the destination ITE, it may safely abort as it has made no changes to the system. If a `MIGRATE` fails to `CAS` the source ITE then it must mark the destination copy it has created as empty. This is done using a `CAS`. If marking the destination as empty is successful, the DTE referred to by the destination is marked for recycling and the `MIGRATE` returns a concurrent operation error. If marking the destination as empty fails, this means that another operation has already marked the destination as empty. The `MIGRATE` may then safely return a concurrent operation error.

DTE timestamping and expiration: Nessie does not allow DTEs to be immediately recycled when they are replaced by `PUT`, `MIGRATE` or `DELETE` operations. This is to prevent a case where a slow operation reads an ITE and then delays long enough before reading the associated DTE that the DTE has been recycled and re-used by concurrent operations. To enforce this, DTEs contain a recycle flag, denoting whether or not they are in the process of being recycled, and an 8-byte expiration timestamp. When the recycle flag is set, the timestamp is updated to a time in the future after a configurable expiration period. A Nessie client attempting to re-use freed DTEs cannot re-use a DTE until it has fully expired. Furthermore, a DTE is only timestamped for recycling after the completion of an event that has marked its ITE as empty. Thus, any node that reads a valid ITE is guaranteed to receive a consistent value if they subsequently read the associated DTE within one expiration period. We therefore enforce the condition that every Nessie operation returns an error once a single expiration period of time has elapsed, and must be re-attempted. Although a timeout at the operation level is more strict than is necessary to maintain consistency, it is easy to implement and is unlikely to be exceeded in practice.

Configuring the DTE expiration period: The expiration period should be chosen so that it is long enough to not cause unnecessary operation aborts, but also short enough to keep the data table from running out of entries. It is possible to determine a safe upper bound on the expiration period using the rate of `PUTs`, and the amount of overflow capacity available in the data table. For example, if the system is expected to handle 600,000 `PUTs` per second, which is more than the highest rate of `PUTs` we reached in any experiment, and has 600,000 data table entries of overflow capacity, which in our experimental set up would be a data table operating at 80% of capacity, then the expiration period should be no longer than 1 second. In less `PUT` heavy workloads, a rate of 100,000 `PUTs` per second would be expected, and would require an expiration period of 6 seconds. As is shown in Section 6.1, the latency of `GETs` and `PUTs` are on the order of microseconds. Even when operations have tail latencies in the millisecond range, this would still be an order of magnitude smaller than an expiration period on the order of seconds. Thus, a 1 second expiration period would ensure DTE availability without causing unnecessary aborts.

Avoiding the ABA problem: If a slow-running Nessie client were to read an ITE and then stall while other nodes continued to run, it could be possible for the other nodes to write over the ITE many times in sequence. If the original DTE referred to by the ITE were to be recycled and then used again for the same ITE, the slow-running Nessie client might re-read the ITE and assume it has not changed. This is known as the ABA problem [1]. DTEs already have expiration times, and therefore as long we continue to enforce timeouts on operations, it is impossible for an operation to use a recycled DTE. Thus, timestamps and expiration times also eliminate the ABA problem.

Cache consistency: Nessie stores cached DTEs using their

associated ITEs as keys, and looks up cached entries using ITEs retrieved during the course of an operation. As discussed previously, a DTE retrieved using a valid ITE is also guaranteed to be valid for a full expiration period from the time the ITE was read. To maintain cache consistency, Nessie therefore stores items together with their expiration times in the local cache. If a cache lookup yields an expired ITE-DTE pair, Nessie treats it as a cache miss, and evicts the entry from the cache. When an ITE results in an unexpired hit in the cache, the DTE in the entry is again guaranteed to be consistent for a full expiration period from the time at which the ITE was read. This is because reading the DTE from the cache, so long as it has not expired, is no different than fetching it from a remote node. This allows Nessie to extend the expiration time of entries on a cache hit. Because Nessie is a big memory system, the size of the dataset implies that the local cache primarily benefits workloads with highly skewed access patterns. Therefore, Nessie continues to receive most of its caching performance benefits even with a relatively short expiration time.

Synchronizing DTE timestamps: Using timestamps for expiration times requires the clocks of nodes in the cluster to be loosely synchronized. The configurable expiration time used by a Nessie cluster will typically be on the order of seconds, whereas operations in the system occur on the order of microseconds. This means that the simple use of Network Time Protocol (NTP), which is able to achieve better-than-millisecond accuracy in data centre environments [4], provides sufficient synchronization accuracy for Nessie's purposes. This minor variance is then accounted for by operations when they check if their timeout period has expired.

Consistency and availability during node failure: Nessie does not provide replication, and as a result a node failure leads to the loss of indices and data. Additionally, portions of the keyspace for which the lost node shared responsibility or was operating on may be unavailable until the node is restored. There are potential challenges involved in adding fault tolerance to Nessie. This could involve the implementation of a client-driven version of a consensus protocol or the use of external failure detection. We intend to explore providing a fault-tolerant client-driven system in future research.

4.3.2 Performance

Filter bits: Nessie uses filter bits in ITEs to prevent the unnecessary retrieval of DTEs that would not match the requested key. This is similar to hash table optimizations used by MemC3 [16], MICA [25], and MemcachedGPU [20]. This optimization is particularly important for workloads with densely populated index tables, and workloads with large DTEs. When a new DTE is being inserted, the trailing bits of its key hash are stored in the ITE that references the DTE. Upon reading an ITE, the filter bits may be compared against the filter bits of the requested key. If these bits do not match, then it is guaranteed that the ITE's referenced DTE does not contain an entry for the requested key.

Previous versions: If clients attempt to `GET` a key for which a `PUT` is simultaneously occurring, the clients performing the `GETs` would be unable to complete as they would see an invalid DTE. This could cause large latencies for `PUT`-heavy workloads with a small subset of popular keys. We avoid this issue by taking advantage of the fact that Nessie retains removed DTEs for a timeout period. While a `PUT` has not yet completed and therefore has not officially occurred, as it could still be interrupted, a previous DTE for the same key is, by design, still in the system. The previous DTE is only invalidated and set to be recycled and given an expiration timeout once the `PUT` has completely finished. To allow other clients to access the previous DTE, when a `PUT` modifies a non-empty ITE, it copies the ITE being replaced into the DTE associated with the current operation. When another client accesses the DTE of the in-progress operation, the client can then use the previous ITE field to retrieve the previous DTE. If this previous DTE is valid, and matches the requested key, the `GET` successfully returns it. The result is a significant improvement in throughput, which is demonstrated in Section 6.2.

Back-offs and avoiding retries: The use of previous versions prevents Nessie's `PUTs` from interfering with its `GETs`. However, it is still possible for `PUTs` on colliding keys to interfere, and in these circumstances, one of the `PUTs` must be aborted and retried after a back-off period. To avoid a scenario where newly arriving `PUTs` continually interrupt in-progress `PUTs`, Nessie aborts `PUTs` that encounter an invalid DTE while choosing a candidate ITE, as this implies another `PUT` is already occurring. We use a standard exponential back-off for retrying `PUT` operations.

Multiple reads: When the values stored by Nessie become large, filter bits may not be enough to eliminate the overhead of unnecessarily reading invalid or non-matching DTEs during a `GET`. To account for this, we allow `GETs` to split RDMA `READs` from DTEs into two, one `READ` to fetch all non-value fields, and another `READ` to fetch the DTE with value, if the key matches and the DTE is valid. Despite incurring an additional round trip, only transmitting necessary values decreases completion times for Nessie operations with large values, as we demonstrate in Section 6.

5 Implementation

We have constructed a working prototype of Nessie consisting of about 4,400 lines of C++, which sits on top of networking, configuration and benchmarking infrastructure consisting of another 8,500 lines of C++. Nessie allows for quick and easy deployment across a cluster of RDMA-enabled nodes. It supports configurable numbers of cuckoo hashing functions, networking parameters, data table sizes, index table sizes, and more. Our prototype supports a key size and data size configurable at launch time. Future implementations could be made to partition data tables into entries of different sizes in order to accommodate more efficient storage of variably-sized data. This is similar to the approach used by memcached, which divides slabs of memory into pieces using configurable chunk sizes [3]. Our current implementation of Nessie supports partitioning across a fixed number of machines. For a discussion of supporting dynamic node join and leave, please refer to Section 7. Nessie uses RDMA reliable connections (RC) to provide network connectivity between nodes in the system, as they are required in order to support Nessie's use of

RDMA READ and CAS verbs. RC connections also guarantee transmission, and therefore prevent the need to implement an extra, external layer of reliability for Nessie's networking protocol.

In order to compare our fully decoupled, client-driven Nessie prototype against equivalent approaches that use other communication mechanisms, we have also implemented a server-driven version of Nessie, called NessieSD, and a hybrid version of Nessie, called NessieHY. Similar to HERD [21], NessieSD uses RDMA WRITE verbs to insert requests into a 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. In the case where a request is made by and serviced by the same node, the request and response are optimized to be 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. Like Nessie, NessieSD's index tables are implemented as cuckoo hash tables. Unlike Nessie, however, an ITE in NessieSD only ever refers to DTEs on the same node (making it partially coupled instead of decoupled). This design decision also emulates HERD, limiting the number of network operations necessary for NessieSD to complete a single round trip, but potentially limiting opportunities for locality-based optimizations.

Inspired by the RKVS demonstrated in FaRM [14], NessieHY employs a hybrid communication system that uses client-driven GETs and server-driven PUTs. For its protocol, NessieHY combines techniques from both Nessie and NessieSD. As in NessieSD, NessieHY uses RDMA WRITE verbs to push PUT requests to remote servers. These servers use polling server worker threads to service and respond to the PUT requests using RDMA WRITE verbs. NessieHY's GET protocol is largely similar to that of Nessie, described in in Section 4.2.1. For comparison with Nessie and NessieSD, index tables for NessieHY are implemented as cuckoo hash tables. Similar to the index tables of NessieSD, NessieHY's index tables are partially coupled. This allows NessieHY to minimize the number of round trip operations required to service a PUT. Additionally, NessieHY is optimized to use direct memory accesses in place of RDMA operations whenever possible when dealing with same-node data, for example when reading DTEs. This helps to reduce unnecessary consumption of NIC resources.

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 UC and inlining demonstrated in the HERD paper, we avoid using UC 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 value sizes and as a result we have not been concerned with obtaining the benefits of inlining. Likewise, although NessieHY draws inspiration from the hybrid design of FaRM, it does not include other features that FaRM's hashtables do, for example the use of chained associative hopscotch hashing, the ability to perform multi-item transactions, and the use of data replication. Rather, we have provided a system that closely resembles Nessie and NessieSD apart from communication mechanisms, allowing for easy comparison between the three. Furthermore, we have not implemented several of Nessie's optimizations in NessieHY, including filter bits, caching, previous versions, and multiple reads. This is primarily because these feature are not present in other hybrid systems, furthermore implementing these features in a NessieHY requires significant engineering effort which is outside the scope of our work.

The NessieSD and NessieHY implementations are about 3,600 lines of C++ and 4,200 lines of C++ respectively, and employ the same underlying infrastructure, configuration and deployment options as Nessie.

6 Evaluation

To evaluate Nessie, we use a cluster of 15 nodes, where each node is a Supermicro SYS-6017R-TDF server containing one Mellanox 10 GbE ConnectX-3 NIC, 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. 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 experiments use an index table load factor of 0.4. This load factor is small enough that cuckoo hash collisions occur infrequently. We believe it is reasonable to use a lower load factor because each index table entry is 64 bits, meaning that the index table can be sparse without wasting memory. For example, with 1.2 billion keys and a load factor of 0.4, only 1.5 GB are required per node to store the index table. Index table entries in all Nessie experiments are configured to use seven filter bits. Due to the low load factor used in our experiments, these filter bits do not play a major role in the results that we present. Similar to the results presented by Pilaf [26], we empirically determined that using three cuckoo hash functions struck a good balance between memory efficiency and performance and therefore, we use this value for all of our experiments.

All experiments use a key size of 128 bytes, and we vary the size of the value from 256 bytes to 128 KB. The number of keys used per experiment is chosen such that, when all key-value pairs are present in the system, each node should contain 20 GB worth of data on average. As a result, experiments with smaller data values have a larger number of key than experiments with larger data values. Since each update requires a new DTE, for workloads with enough PUTs, all DTEs will eventually become full and further additions will require recycling. To prevent excessive recycling we use data table sizes of 30 GB per node. Additionally we use a cache size that is able to accommodate data for 0.5% of the total number keys on each node. The cache provides negligible benefits for experiments with uniformly distributed access patterns. Therefore, the cache size was determined experimentally by its impact on Zipfian-distributed workloads. The nature of the Zipfian distribution means that the cache provides diminishing benefits as it continues to grow.

Our Nessie experiments use 12 client worker threads per node (one for each core on our nodes) which drive load by performing operations in parallel, as we found that exceeding one worker per core caused contention which reduced throughput. Similarly, NessieSD and NessieHY were tuned to have one worker thread per core, however these are divided between server workers handling requests and client workers making requests. For these systems, tuning the number of client and server workers presents a trade-off between throughput and latency. In our experiments, we adjust these values for maximum system throughput. Through manual tuning, we determined that NessieSD's peak performance is obtained by having 3 server workers and 9 client workers per machine when value sizes are less than 1 KB. If value sizes are greater than or equal to 1 KB, we found that 2 server workers and 10 client workers per machine obtained the peak performance for NessieSD. With NessieSD, server worker threads service all operations, and therefore tuning their number is a simple matter of trying to accommodate the load being generated by the system. With NessieHY, however, client workers service `GETs`, and server workers service `PUTs`. Tuning the number of server workers versus client workers depends on additional factors such as the ratio of an experiment's `GETs` to `PUTs`. Through manual tuning, we discovered that NessieHY's best throughputs for 50% `GETs` workloads were achieved with 3 server workers and 9 client workers for data sizes less than or equal to 32 KB, and 2 server workers and 10 client workers for larger data sizes. For 90% and 99% `GETs` workloads, NessieHY performed best with 1 server worker and 11 client workers for most data sizes, with results at 2 server workers and 10 client workers providing comparable throughput as individual operations are serviced with lower latency but less load is driven by the system overall.

While it is possible to create server-driven mechanisms that reduce CPU usage by alternating between polling and blocking modes during periods of high and low activity, blocking communication mechanisms have been avoided by systems such as FaRM and HERD due to the dramatic increase in operation latency that they cause. We do not compare against such an approach, as the results would be difficult to interpret meaningfully due to the unacceptably high service latency when serving requests using blocking communication mechanisms.

6.1 Shared CPU Environments

Our first set of experiments emulate a shared CPU environment as might be found in a cloud data centre. In such environments, external processes compete for the CPU, causing contention. In this experiment, each server worker thread is pinned to its own core, all client worker threads are pinned to a separate set of cores that does not include those being used by the server worker threads, and background threads are not pinned to any CPUs. The background threads mimic the behaviour expected in a cloud computing environment where external processes can not be controlled. Figure 4 shows an experiment on a 15 node cluster with an increasing number of background CPU-consuming processes, in which key accesses are randomly distributed in a uniform fashion across the entire keyspace and the size of a data value is

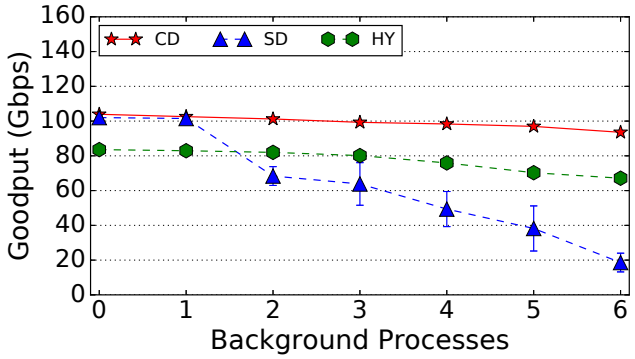


Fig. 4: Throughput for Nessie (CD), NessieSD (SD), and NessieHY (HY) as other processes are introduced, using 16 KB values and 90% `GETs`.

16 KB. Repeating this experiment with smaller data sizes yielded data demonstrating similar trends. The ratio of `GET` to `PUT` operations in the experiment is 90 to 10. The figure shows that, with no background processes (i.e., no CPU contention), Nessie and NessieSD both perform equivalently well, while the goodput for NessieHY is slightly lower. However, as CPU contention is introduced in the form of external processes, NessieSD's throughput declines dramatically, NessieHY's throughput declines somewhat but not nearly as dramatically, while Nessie remains relatively unaffected.

NessieSD's performance decline occurs because its server workers are unable to maintain the same level of high performance when faced with CPU contention. In NessieSD clients must wait for a server worker to respond when a server worker thread is unable to access the CPU due to contention. When this occurs, all clients sending them requests are unable to make any progress. In contrast, Nessie clients operate independently so if a single individual client worker thread is unable to access the CPU it only impacts that operation. NessieHY's performance similarly declines as CPU workers are introduced, however this rate of decline is lower than that of NessieSD as NessieHY only uses its server worker threads for `PUTs`. CPU contention is therefore less impactful on NessieHY than on NessieSD, and more impactful of NessieHY than Nessie. It is worth noting that the impact of competing background threads would decrease for NessieHY with a lower percentage of `PUTs`, and increase for a higher percentage of `PUTs`.

6.2 Large Data Values

Unlike existing RKVSes, Nessie is designed to efficiently support large data values. We now evaluate Nessie's performance using a range of data sizes and `GET`/`PUT` ratios under both uniform and Zipf distributions. Unless otherwise stated, all experiments use 15 nodes and throughput results are presented as goodput, which is defined by the number of operations performed per second multiplied by the size of the data value.

Figures 5, 6 and 7 show the throughput of Nessie, NessieSD and NessieHY using random, uniformly distributed keys with 50%, 90%, and 99% `GETs`, respectively. The results show that, although NessieSD performs better than Nessie for small data sizes, the opposite is true as data sizes increase. With a 2 KB data size and 50% `GETs`,

Nessie performs 30% worse than NessieSD. However, for the 50% GET workload at 8-KB data values and beyond the performance of Nessie begins to outpace that of NessieSD. For data values of 128 KB, Nessie's throughput is about 70% higher than NessieSD's throughput. For GET-heavy workloads, both Nessie and NessieSD perform equally well at 16 KB and beyond, with Nessie making small relative gains for GET-intensive workloads thereafter. The throughput of NessieHY is consistently lower than both Nessie and NessieSD, primarily due to its lack of filter bits which we will discuss next.

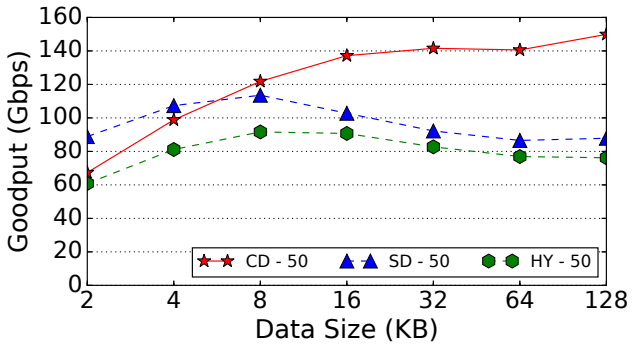


Fig. 5: Uniform random key access using 15 nodes, for 50% GETs.

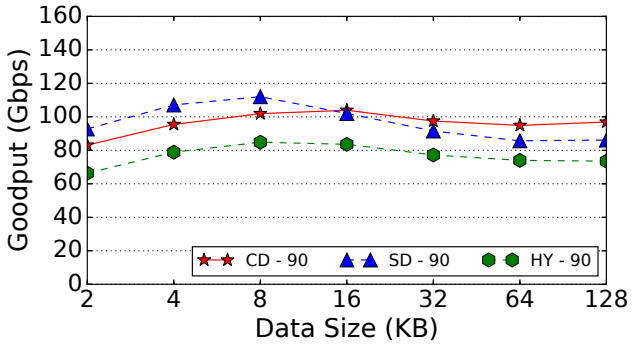


Fig. 6: Uniform random key access using 15 nodes, for 90% GETs.

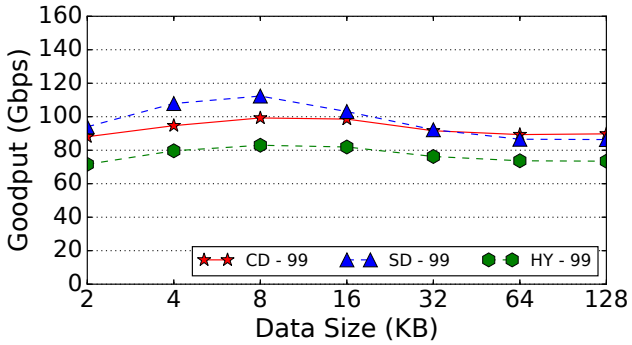


Fig. 7: Uniform random key access using 15 nodes, for 99% GETs.

To explain the differences between performance for each system, we need additional information about how the systems operate. Figure 8 shows a CDF of the average number of RDMA operations required during GETs and Figure 9 shows a CDF for the average number required by PUTs in Nessie (labelled CD), NessieSD (labelled SD) and NessieHY (labelled HY). Similarly, Figures 10 and 11 show a CDF for the average number of bytes sent over the network by both systems during individual GETs and PUTs, respectively. All four figures are presented using data from the 50% GETs, 50% PUTs workload using 16 KB DTEs and uniform key access across 15 nodes. 16 KB DTEs are used because at this point Nessie begins to trend away from NessieSD and NessieHY in Figure 5.

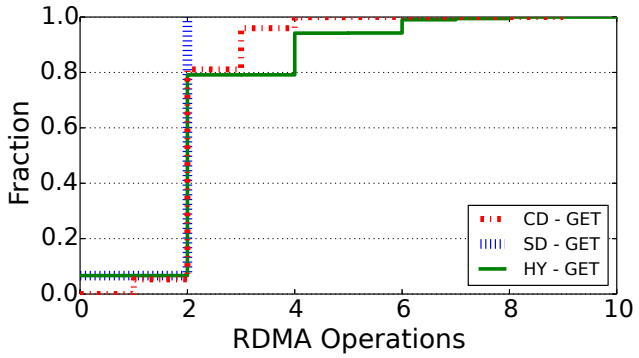


Fig. 8: CDF of RDMA operations required for GETs in a uniform random workload, with 50% GETs on 16 KB data values across 15 nodes.

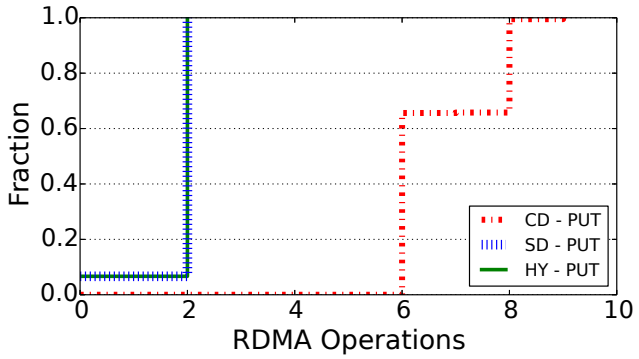


Fig. 9: CDF of RDMA operations required for PUTs in a uniform random workload, with 50% GETs on 16 KB data values across 15 nodes. SD-PUT and HY-PUT are overlapping.

Figure 8 and 9 show that the server-driven aspects of NessieSD and NessieHY allow them to complete a small number of operations without using RDMA, as about 6% of ITE and DTE lookups would be hosted on the node making a request. This benefit would shrink as the size of a cluster scales. Non-local NessieSD operations and non-local NessieHY PUTs use two RDMA WRITEs. Comparatively, Nessie GETs and PUTs must always read at least one ITE using RDMA. A Nessie GET uses no other RDMA operations if data is local or can be serviced from the cache, but otherwise data is serviced remotely using RDMA. When the load factor of the system's hash tables increases, a small number of Nessie and NessieHY GETs require more than two RDMA operations as they iterate over cuckoo hash indices. In these cases, Nessie is often able to avoid DTE lookups using filter bits. NessieHY, which does not use filter bits, incurs additional operations on each cuckoo hash iteration. Nessie's PUTs require a minimum of 5 RDMA READs to ITEs and an RDMA CAS on an ITE. Despite using more RDMA operations, as values grow larger performs increasingly better than NessieSD and NessieHY. This can be attributed to Nessie's ability to exploit data locality and

prevent network usage through local writes. We believe that introducing filter bits into NessieHY would bring its throughput in line with some Nessie results at small data sizes and NessieSD results at large data sizes. Note that filter bits are not available on any current hybrid systems.

Examining the number of bytes transferred during GET and PUT operations in Figures 10 and 11 reveals that, although Nessie and NessieSD transfer similar amounts of bytes on average during GETs, Nessie transfers far fewer bytes than NessieSD and NessieHY during PUTs due to its local DTE placement scheme. For PUTs, NessieHY on average transfers the same number of bytes as NessieSD. However, for GETs a significant fraction of the NessieHY 16 KB GETs require two or three cuckoo hash iterations before successfully retrieving the data. Unlike Nessie, which uses filter bits to prevent data transfer in this specific case, NessieHY requires a data transfer in these circumstances (as can be seen by the fraction of operations requiring 32 or 48 KB transfers) and therefore obtains lower throughput in Figures 5, 6 and 7.

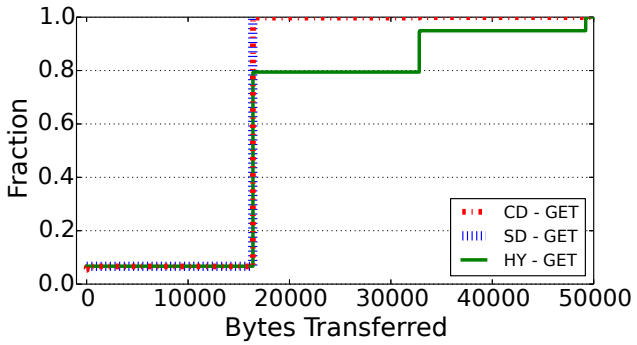


Fig. 10: CDF of bytes transferred for GETs for a uniform random workload, with 50% GETs on 16 KB data values across 15 nodes.

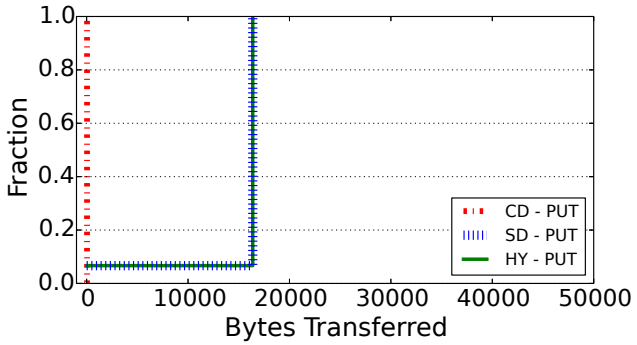


Fig. 11: CDF of bytes transferred for PUTs for a uniform random workload, with 50% GETs on 16 KB data values across 15 nodes. SD-PUT and HY-PUT are overlapping.

Table 1 contains data showing the GET, PUT and overall average latencies incurred for different data sizes, in addition to the total number of bytes transferred in each case. The total number of bytes includes not only goodput, but also bytes caused by overhead such as cuckoo hashing. Minimum values for each grouping are shown in bold. As seen in Table 1, when values grow large transfer times and network contention begins to impact performance, causing increased average operational latencies. This table also shows, however, that Nessie sends substantially fewer total bytes over the network compared to NessieSD and NessieHY for all data sizes. By eliminating unnecessary data transfers with filter bits, and by preventing GETs from needing to compete with PUTs for network resources through the use of local writes, Nessie continues to operate at lower levels of latency than NessieHY and NessieSD at large data sizes. Interestingly, average PUT latencies are minimized for data sizes larger than 16 KB when using NessieHY because its server workers, which only process PUTs, are more lightly loaded than those of NessieSD.

Data Size (KB)	2	4	8	16	32	64	128
CD GET Lat. (µs)	**19**	**27**	**46**	**86**	**172**	**391**	**811**
SD GET Lat. (µs)	25	43	82	184	412	875	1732
HY GET Lat. (µs)	24	47	109	231	524	1193	2455
CD PUT Lat. (µs)	61	83	133	233	447	858	1533
SD PUT Lat. (µs)	**25**	**41**	77	171	381	816	1602
HY PUT Lat. (µs)	42	42	**69**	**129**	**271**	**709**	**1388**
CD All Lat. (µs)	40	55	89	**159**	**310**	**624**	**1172**
SD All Lat. (µs)	**25**	**42**	**80**	178	397	846	1667
HY All Lat. (µs)	33	45	89	180	398	951	1922
CD Sent (GB)	**514**	**729**	**877**	**975**	**999**	**989**	**1053**
SD Sent (GB)	1245	1503	1590	1439	1291	1212	1230
HY Sent (GB)	957	1287	1461	1445	1314	1219	1208

TABLE 1: Latency averages for different operations, and total bytes sent over the network for Nessie (CD) NessieSD (SD) and NessieHY (HY) for uniform random key access using 15 nodes and 50% GETs.

Data access patterns are also important to consider when values become large. Many applications exhibit skew in the access patterns for keys. In order to highlight some of the features of Nessie that have been explicitly designed to optimize for such cases, we next examine the performance of Nessie and NessieSD under these circumstances. NessieSD is, by its design, not able to make use of these optimizations. We do not compare against NessieHY as it lacks the optimizations necessary for it to achieve acceptable performance for this workload. We consider a workload that uses a Zipfian distribution with an alpha of 0.99, following the guidelines of the YCSB benchmark [12], which shows the percentage difference in goodput between Nessie and NessieSD. The results of these experiments are shown in Figure 12. case for workloads that use a uniform distribution of keys, Nessie provides its peak performance for large value workloads with 50% GETs (an improvement of over 50% compared to NessieSD). However, unlike the uniform random workload, Nessie also performs well for large values with a large percentage of GETs, outperforming NessieSD by over 40% for workloads with 99% GETs.

In addition to showing results for large data values, Figure 12 also shows the relative performance between Nessie and NessieSD for small data values of 256 bytes. Because Nessie's operations require more roundtrips to complete, smaller data values are negatively impacted by Nessie's design. However, GET-heavy workloads with data sizes greater than 2 KB, or balanced workloads with data sizes greater than 16 KB, are able to take advantage of Nessie's design, which is specifically targeted at those workloads.

Nessie's performance results with a Zipf workload are obtained through a collection of optimizations. Figure 13 shows the breakdown of each optimization's contribution for 50% GETs and 99% GETs with 16 KB data values. In its most basic form (labelled basic), Nessie is not able to provide competitive throughput because it uses a form of optimistic concurrency control. Our protocol assumes that

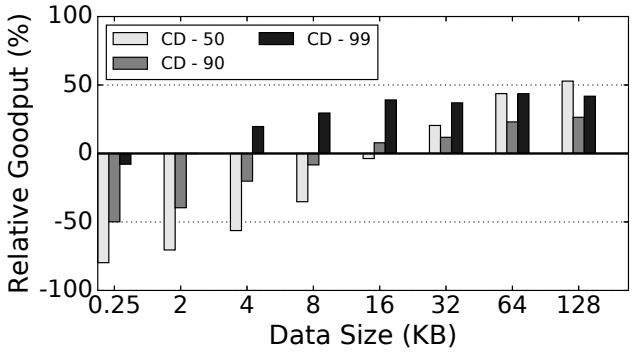


Fig. 12: Zipf random key access using 15 nodes, for 50%, 90%, and 99% GETs. Relative goodput is the percentage improvement in goodput between Nessie and NessieSD.

no other clients are trying to manipulate the same data at the same time and then checks for conflicts at the end of the operation. Such a design is not naturally suited for high levels of contention.

The first optimization we introduce for Nessie is filter bits (labelled + filter) which allow GETs and PUTs to skip non-matching DTE lookups during cuckoo hashing by examining hash bits stored in ITEs. Normally filter bits have a moderate impact on throughput by reducing unnecessary bytes sent over the network, especially as data sizes grow or index table load factor increases. In this Zipf workload it appears that the most popular keys, which are accessed by Nessie far more often than other entries, do not require multiple lookups as filter bits have little impact. By comparison, the uniform workload results seen in Figure 8 and Figure 11 show that Nessie with filter bits eliminates a value-sized RDMA READ in 20% of GETs and another value-sized READ in 5% of GETs, as well as two round-trips in 35% of PUTs. NessieHY, which does not use filter bits but otherwise uses almost the same protocol, does not receive these benefits.

The second optimization we introduce is caching (labelled + caching), which shortcuts the GET protocol if the key has not changed since its last update. As one might expect, caching is not useful for PUT-heavy workloads because the majority of key accesses are for the keys that change the most frequently. In contrast, GET-heavy workloads make good use of caching by avoiding large data accesses, thereby freeing up network bandwidth for other operations.

While caching does increase Nessie's throughput, it does not solve the fundamental problem of high contention on the most popular keys. The problem occurs when a PUT is in progress for a key and other clients are trying to acquire the value for the same key. Once the PUT changes the ITE to reference its new, but still invalid, data table entry, all of the GETs are forced to back-off and retry until the PUT is able to finish. This is particularly problematic because each time a GET reads the invalid DTE, it is consuming a large portion of the network resources and thereby potentially blocking the PUT from finishing.

To alleviate this contention, DTEs contain a copy of the old ITE so that the GET can read the previous value referenced by the ITE (labelled + prev. vers.). Importantly, this allows clients to read a valid DTE even after the PUT has replaced the candidate ITE. However, it is wasteful to read the entire DTE when the large value portion is only used if the DTE is valid. Therefore, we add our final optimization which uses multiple READs to access a DTE to avoid unnecessary data transfers (labelled + multi. read). The first READ of a DTE is for the key and metadata only, and then a second READ is for the value field only if it the value is useful. Taken together, with 16 KB data values, the protocol optimizations produce a 60% improvement for workloads with 50% GETs, and a 90% improvement for workloads with 99% GETs compared to Nessie without any optimizations.

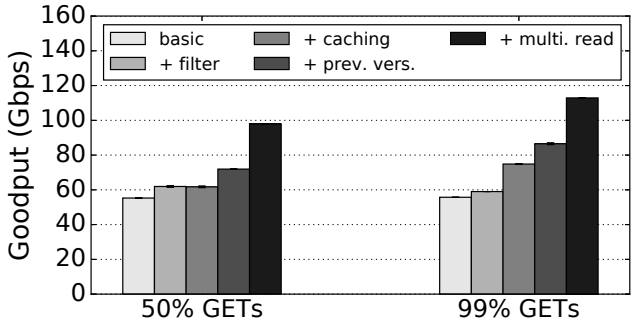


Fig. 13: Optimization impacts for a Zipf random workload, with 50% and 99% GETs on 16 KB data values across 15 nodes.

6.3 Energy Consumption

Our last set of experiments seek to determine the effects of server-driven versus client-driven RKVSes on power consumption. We use IPMI [2] to measure the energy consumption of the machines hosting these instances throughout the duration of the experiments. Figure 14 shows average power consumption across all nodes given varying levels of total system load (measured as a percentage of maximum possible throughput) for experiments using 15 nodes, uniform key access distribution, a 16 KB data value size, and a 90 to 10 GET to PUT ratio. The data point for 100% maximum throughput represents Nessie, NessieSD and NessieHY when they are generating requests as fast as possible. We use this value to determine a mean for a Poisson distribution that produces interarrival rates for workloads operating at 5%, 20%, 40%, 60% and 80% of maximum throughput.

NessieSD and NessieHY expend energy on server worker threads which poll and process requests, and client worker threads which make requests and poll for responses. Nessie's energy consumption derives entirely from worker threads making requests and polling for responses. At 100% of maximum throughput, each system is saturated with requests. Because every thread in each system is busy either servicing a request or polling, the energy consumption of the systems is the same, at around 139 W.

As the percentage of maximum throughput decreases, some client worker threads in each system idle when no work is available. This translates into energy savings for all three systems. Nessie, however, decreases its energy consumption at a more rapid rate than NessieHY and NessieSD, thanks to its strictly client-driven design. NessieHY and NessieSD both contain server worker threads which always run at 100% capacity. These systems therefore consume more energy than Nessie at lower percentages of maximum throughput. NessieSD, which requires more server workers

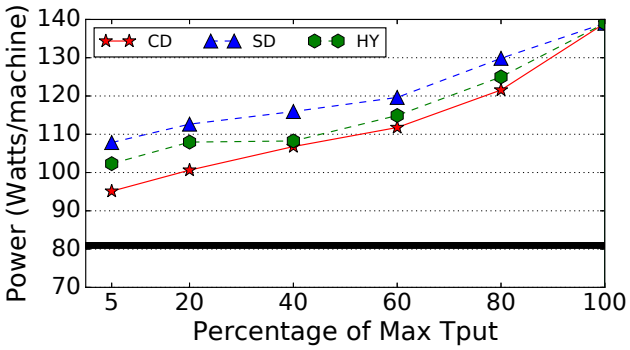


Fig. 14: Energy consumption for Nessie (CD), NessieSD (SD), and NessieHY (HY) for varying percentages of maximum throughput. The dark line at 81 Watts is the power consumption of an idle machine.

at the same level of throughput than NessieHY due to its strictly server-driven design, likewise consumes more energy than NessieHY. At 80% maximum throughput, the difference between the worst performer, NessieSD, at 130 W and the best performer, Nessie, at 121 W is 18% over the system's idle wattage of 81 W. At 20% maximum throughput this difference increases to 113 W for NessieSD and 100 W for Nessie, or 41% over the idle wattage. In both cases, NessieHY falls between the other systems. We therefore conclude that for realistic data centre workloads where inactivity can fluctuate, such as the workloads described in Section 3.2, a client-driven approach provides significant energy savings over a server-driven approach.

7 Discussion

One concern with Nessie's design is potential contention on NIC resources from multiple requests, from collocated computation, or from other tenants in a cloud environment. This can result in the NIC becoming the system's performance bottleneck. However, in many environments including the cloud, servers are equipped with more than one NIC. Providing Nessie with a dedicated NIC would ensure that it consumes network resources separate from those used by other applications. Additionally, if NIC saturation becomes an issue, a single Nessie node is able to make use of more than one NIC by creating multiple index and data tables, and partitioning them across the NICs. This can help to ensure that NIC resources do not bottleneck the system.

In this paper we have primarily focused on workloads where data is generated roughly equally across all nodes, memory is not fully utilized, or minor imbalances can be remedied by migrating data between nodes during off peak periods. These workloads benefit from Nessie's local writes in order to improve performance. Because of this focus, we have not added support for remote writes to our Nessie prototype, and as a result it is not possible for us to isolate the performance improvements Nessie receives from writing locally. In future work, Nessie's protocol could be updated to support the placement of DTEs non-locally by replacing direct memory writes with RDMA writes and adding a background mechanism to lazily request lists of empty DTEs on remote nodes. This would allow us to support other workloads that do not share these characteristics and to place data according to other criteria, for example by using application-speficic knowledge of data locality. Nessie's performance would be different under these circumstances, as the amount of data written over the network would change for `PUTs` based on workload-dependant factors.

The current Nessie design does not support dynamic group membership changes. Failed nodes can be re-added or replaced, but the number of nodes in the system cannot be changed. This is something that we intend to examine in future work, perhaps through the implementation of client-driven consistent hashing for data partitioning.

8 Conclusions

In this paper, we design, implement and evaluate the performance and energy consumption of Nessie, a high-performance key-value store that uses RDMA. The benefits of this work derive from Nessie's client-driven operations, in addition to its 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. This allows Nessie to perform favourably in shared CPU environments compared to an equivalent server-driven approach, more than doubling system throughput. Additionally, the decoupling of index tables and data tables allows Nessie to perform write operations to local memory, which is particularly beneficial in workloads with large data values. This provides Nessie with a 60% throughput improvement versus a fully server-driven equivalent when data value sizes are 128 KB or larger. Nessie also demonstrates throughput wins against a hybrid system in this case. Furthermore, Nessie's client-driven approach allows it to consume less power than a fully server-driven system during periods of non-peak load, reducing power consumption by 18% at 80% system utilization and 41% at 20% system utilization when compared with idle power consumption. Nessie likewise improves power consumption over a hybrid system to a lesser extent.

9 Acknowledgments

Funding for this project was provided by the University of Waterloo President's Scholarship, the David R. Cheriton Graduate Scholarship, the GO-Bell Scholarship, the Natural Sciences and Engineering Research Council of Canada (NSERC) PGS-D and CGS-D, the Ontario Graduate Scholarship, as well as NSERC Discovery Grants and an NSERC Discovery Accelerator Supplement. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo. Finally, we wish to thank the Canada Foundation for Innovation and the Ontario Research Fund for funding the purchase of equipment used for this research.

References

[1] ABA problem. http://en.wikipedia.org/wiki/ABA_problem.
[2] Intelligent Platform Management Interface (IPMI). http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html.
[3] MemCached - A distributed memory object caching system. http://memcached.org.
[4] Network Time Protocol (NTP). http://en.wikipedia.org/wiki/Network_Time_Protocol.

[5] RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.
[6] redis. http://redis.io/.
[7] Understanding iWARP. http://www.intel.com/content/dam/doc/technology-brief/iwarp-brief.pdf.
[8] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proc. ACM SIGMETRICS PER* (2012), vol. 40, ACM, pp. 53–64.
[9] BLOOM, A. Using Redis at Pinterest for billions of relationships. https://blog.pivotal.io/pivotal/case-studies/using-redis-at-pinterest-for-billions-of-relationships, 2013.
[10] 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 TOCS 26*, 2 (2008), 4.
[11] CHOWDHURY, M., KANDULA, S., AND STOICA, I. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. ACM SIGCOMM* (New York, NY, August 2013), ACM.
[12] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proc. ACM SoCC* (2010), ACM, pp. 143–154.
[13] DAVDA, B., AND SIMONS, J. RDMA on vSphere: Update and future directions. In *Proc. OFA Workshop* (2012).
[14] DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. FaRM: Fast Remote Memory. In *Proc. USENIX NSDI* (Seattle, WA, April 2014), USENIX.
[15] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. ACM SOSP* (2015).
[16] FAN, B., ANDERSEN, D., AND KAMINSKY, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proc. USENIX NSDI* (Lombard, IL, April 2013), USENIX.
[17] FITZPATRICK, B. Distributed caching with MemCached. *Linux Journal 2004*, 124 (2004), 5.
[18] HELLER, B., SEETHARAMAN, S., MAHADEVAN, P., YIAKOUMIS, Y., SHARMA, P., BANERJEE, S., AND MCKEOWN, N. ElasticTree: Saving energy in data center networks. In *Proc. USENIX NSDI* (2010), vol. 10, USENIX, pp. 249–264.
[19] HENDLER, D., INCZE, I., AND TZAFRIR, M. Hopscotch hashing. In *Proc. DISC* (2008).
[20] HETHERINGTON, T. H., O'CONNOR, M., AND AAMODT, T. M. MemcachedGPU: Scaling-up scale-out key-value stores. In *Proc. ACM SOCC* (2015), ACM, pp. 43–57.
[21] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM* (Chicago, IL, August 2014), ACM.
[22] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *Proc. USENIX ATC* (2016), USENIX.
[23] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS OSR 44*, 2 (2010), 35–40.
[24] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. ACM SoCC* (2014), ACM, pp. 1–15.
[25] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. USENIX NSDI* (2014), vol. 15, USENIX, p. 36.
[26] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX ATC* (San Jose, CA, June 2013), USENIX.
[27] 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 OSR 43*, 4 (2010), 92–105.
[28] PAGH, R., AND RODLER, F. Cuckoo hashing. *Journal of Algorithms 51*, 3 (May 2004), 122–144.
[29] SZEPESI, T., WONG, B., CASSELL, B., AND BRECHT, T. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *Proc. WRSC* (Amsterdam, The Netherlands, April 2014).
[30] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proc. ACM SOSP* (2015), ACM, pp. 87–104.
[31] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI* (2012), USENIX, pp. 2–2.

AUTHOR BIOGRAPHIES

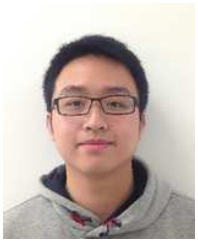
Benjamin Cassell Ben is a PhD student in computer science at the University of Waterloo. He has worked for Acpana Business Systems, the Ontario Teachers' Pension Plan, Public Safety Canada, and BMO Nesbitt Burns. His research interests include RDMA-enabled systems, distributed systems and storage, improving system performance, and privacy controls for IoT devices and cyberphysical systems.

Tyler Szepesi Tyler received his Masters in computer science at the University of Waterloo. His research interests included distributed systems, data centre networking, and reconfigurable optical networks. As of 2017, Tyler is a senior platform developer at Symantec.

Bernard Wong Bernard is an Associate Professor in the Cheriton School of Computer Science at the University of Waterloo. His research interests span distributed systems and networking, with particular emphasis on problems involving decentralized services, self-organizing networks, and distributed storage systems.

Tim Brecht Tim is an Associate Professor in the Cheriton School of Computer Science at the University of Waterloo. He has held positions while on sabbatical at Netflix, École Polytechnique Fédérale de Lausanne, and HP Labs. His research interests include: empirical performance evaluation, Internet systems and services, wireless networking, operating systems, parallel and distributed computing, and IoT infrastructure.

Jonathan Ma Jonathan is a fourth year undergraduate computer science student at the University of Waterloo. He has interned at Citadel and LinkedIn as a software engineer working in infrastructure, and at Facebook as a data scientist. He has additionally worked as an Undergraduate Research Assistant, with a focus on networks and distributed systems.



Xiaoyi (Eric) Liu Eric received his B.CS from the University of Waterloo in 2015. A portion of that time was spent as a part-time and full-time Undergraduate Research Assistant in the networks and distributed systems group. As of 2017, Eric is a member of the Google Technical Infrastructure team, where he performs C++ tooling and code analysis.