Vrije Universiteit Amsterdam

Bachelor Thesis

PERK: a pretty efficient distributed key-value store using RDMA

Author: Louk Onkenhout (2619109)

1st supervisor: dr. ir. Animesh Trivedi
2nd reader: ir. Jesse Donkervliet

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

June 24, 2021

Abstract

In modern data-heavy society, applications and websites need to process and access exponentially growing amounts of data. To provide high performance, these services rely on fast networks and a lot of processing power. Although network speeds have been improving rapidly, CPU improvements have been slowing down. In order to keep up with the increasing demand, applications like key-value stores must utilize the available resources as optimally as possible. Unfortunately, traditional networking technologies are CPU-intensive and are therefore unable to take advantage of these new network speeds. As a result, alternative networking technologies have to be explored as replacements. Remote Direct Memory Access (RDMA) is a prominent network technology that has gained much traction over the last decade. It allows bypassing this CPU usage by offloading processing to special network hardware and consequently, presents an excellent opportunity to do more efficient network communication.

Using RDMA, this thesis describes the design and implementation of a new key-value store, with the goal of delivering high-performance distributed storage: PERK. Several metrics such as latency, operations per second, and CPU usage are used to determine its performance. These are then compared with Memcached, a TCP/IP-based key-value store used in real production systems. The benchmarks show that PERK performs significantly better for smaller-sized key-value pairs (<512 bytes). It is able to process up to 400% more operations per second using a single process and achieves stable low latencies up to 4.8 times shorter than Memcached. For larger sizes, however, inefficiencies in the design impact CPU-usage negatively. The source code for this project can be found at: https://github.com/lonkenhout/perk.

Contents

1 Introduction

Data is an asset in modern society. People produce data at an unprecedented pace, and by 2025, IDC expects the average connected person to have some form of digital engagement over 4,900 times every day [28]. Where this data is stored and processed depends strongly on its intended usage. Processing vast amounts of data requires immense computational power. Therefore, most data that requires intensive processing will be stored in some HPC (High-performance computing) setting, e.g. a data center or other cluster computers. Here, applications can utilize massive storage, high network speeds, and many cores.

Large corporations store data for a variety of purposes: big data, machine learning, generic storage (backups), and special storage for services (distributed applications, websites, etc.). For data that services websites, it is particularly important that data is served quickly. Users continuously interact with this data, i.e. this data is frequently updated and requested. To be able to do this for many users at the same time, a system that has good computational resources can spread the workload by having multiple processes share storage, as displayed in Figure 1. This imaginary system uses multiple processes to serve some set of users with data stored in some shared storage.

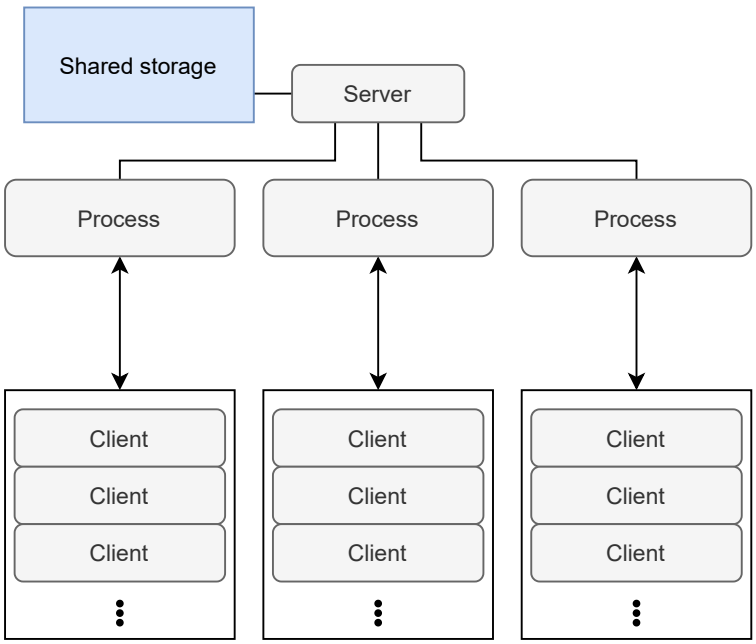


Figure 1: Example of shared storage. Here, three processes on the server side serve groups of clients, who are interacting with some application that requires access to the same resource.

One example of shared storage is the distributed key-value store. These are general-purpose distributed storage systems that maintain some data in fast computer memory. Key-value stores can store data persistently. In this thesis, when referring to key-value stores, we mean an object caching system, that is, it only stores data temporarily. They are quite popular with several large tech companies, e.g. Voldemort at LinkedIn [1], dynamo at Amazon [10]. Redis at Github, Twitter, Pinterest, Snapchat, and more [2]. Memcached

at Facebook [23]. These applications are implemented using traditional networking technologies: sockets. Sockets allow programs to communicate through network protocols such as TCP/IP. Almost always, operating systems provide methods for this by means of a programming application interface (API). For a long time, this was deemed sufficient. However, dramatic increases in network speeds overshadowing CPUs improvements [30] led to sockets becoming a bottleneck for network communications. And although some solutions have been developed to reduce the impact of the overhead [23, 13], RDMA has become an attractive substitute for key-value stores over the last decade.

Remote Direct Memory Access (RDMA) is a network technology that shows significant performance improvements over TCP [29]. The reason is two-fold. First, RDMA can (as the name suggests) perform write and read operations directly on remote memory. Doing that allows the elimination of extraneous copying operations. Second, RDMA moves the processing of network traffic into RDMA-enabled network card hardware (RNIC). Because traffic is normally processed on the CPU, this relieves it significantly, as the CPU will be able to do something else. Besides these properties, RDMA hardware generally allows for TCP (and other) traffic to be sent as well.

1.1 Context

By now, it is clear that high-performance applications that require a lot of network communication, and run in distributed environments, are probably better off using RDMA over TCP. However, to get the most out of RDMA, one needs special hardware. Although it is possible to emulate RDMA in software and still see considerable performance improvements [31], this still uses the CPU, as this emulated protocol has to be executed somewhere. Until recently, adding RDMA-capable hardware to an environment required significant changes to the infrastructure, mostly in cabling (RDMA used to require Infiniband cables). RoCE (RDMA over Converged Ethernet) has made it much easier to add RDMA to existing infrastructure, by transporting RDMA traffic over Ethernet [29].

Since a KV store requires lots of network I/O, it makes sense to use RDMA. KV stores are currently still able to keep up with workloads. For example, an extensive study was done into the performance of Memcached (a KV store) at Facebook's servers [5]. This research reported that Memcached regularly had to process between 70,000 and 140,000 requests per second, on their busiest store. This store stores user-account status information. Now, this was research from 2012, and given the estimated growth of the datasphere, it would be wise to adapt applications before the workload becomes unmanageable.

For the remainder of this paper, we use TCP and sockets interchangeably. When referring to TCP, we mean an implementation of TCP in an interface called sockets.

1.2 Problem Statement

Due to the increasing network speed and lack thereof for CPUs, as well as the fact that the traditional networking stack requires a lot of processing on the CPU, the network performance bottleneck has shifted from the network to the CPU. Although we cannot force CPUs to improve faster, we can utilize the network more effectively by reducing CPU usage, and offloading expensive processing to specialized hardware units. Because socket-based key-value stores are still the most common in practice, it is important to understand

the performance trade-offs that various RDMA configurations provide for the transition that will at some point occur. This study aims to analyze several (mostly) unoptimized configurations, to generate an overview of interesting performance metrics. This should give readers a detailed understanding of several benefits that RDMA provides over TCP.

1.3 Research Questions and contributions

The objective of this study is to understand the applicability of RDMA for key-value stores and give an overview of its performance benefits, that is:

RQI. **What performance benefits does RDMA provide over sockets (using TCP)? How do different configurations (combinations of RDMA verbs) affect the performance of applications using it, and is there one 'best' configuration?** With the intention of gaining a deeper understanding of RDMA and its applicability in certain circumstances, a design, abstraction, and prototyping methodology is used [16, 12, 26]. For example, certain verbs (methods for sending/receiving data) allow the remote side to be completely idle, whereas others require the remote side to stay alert and continually check for activity. How does this compare with a regular TCP-based approach? The decision for using certain combinations strongly influences measured performance. Depending on the requirements of the environment (e.g. reducing CPU usage or highest possible throughput), one needs to choose the used verbs carefully. To allow the widest range of verbs, we limit this study's scope to the usage of RC (Reliable Connection) exclusively.

RQII. **How does one properly evaluate the performance of a distributed system such as a distributed key-value store?** Answering this question requires setting up benchmarks for a complex system [17, 14, 24]. At what depths should the performance be measured to gain comprehensive insights into whether RDMA does indeed perform 'better' than TCP? It is important to evaluate the performance from different perspectives. In particular: latency, CPU usage, operations performed per second (throughput), and scalability are important parameters of a system that needs to run in a distributed environment with many resources.

The contributions, that result from answering these questions are as follows:

CI. Provide a basic understanding of RDMA and its implications for distributed systems.

CII. Insight into performance trade-offs for various RDMA configurations.

CIII. Performance evaluation of several designs for key-value stores.

CIV. Open-source code that can be used to help guide programmers to create RDMA-based key-value stores.

1.4 Thesis structure

The rest of this thesis is structured as follows. Section 2 contains the relevant knowledge to understand the design and implementation of PERK. Anyone experienced with RDMA,

key-value stores, and hashing may choose to skip this section and proceed to Section 3. Section 3 elaborates on the choices made in the design of PERK and explains key trade-offs involved in different designs. Section 4 discusses the implementation details of PERK. Then, Section 5 presents the evaluation of the PERKs performance from various dimensions. Section 6 discusses other research done specifically on key-value stores using RDMA, and Section 7 draws relevant conclusions, answers the research questions previously posed, indicates limitations of this study, and provides several ideas for future research. At the end of the thesis, Appendix A provides steps for reproducing any experiments used for the evaluation of PERK.

2 Background

In this section, a variety of knowledge is provided that is required for the understanding of the system design presented in section 3. Firstly, basics on KV stores and hashing are given. Then, the fundamentals and subtleties of RDMA are explained. This ought to give most readers unfamiliar with this technology a reasonable grasp of its inner workings.

2.1 Key-Value Store

KV stores are general-purpose storage systems, that use a computer's main memory for storing a smaller set of data points that are frequently requested or modified. By main memory, we mean cache memory and RAM. This memory is fast because it is close to the CPU. In turn, this means it does not require expensive disk operations to read or write like with SSD or HDD. Persistent storage systems like normal data stores and databases are slow for this precise reason because they require disk access to persistently store their data. Frequently accessed data from these slower storage systems can instead be moved into a caching system like a KV store, which can then provide fast intermediate access [32]. They generally support at least GET and SET operations, that is, operations to read and modify the data. Each entry in a KV store is a key-value pair. For each key, there is one, and only one value stored in the store. If a new value is given for the same key, the old value is simply overwritten.

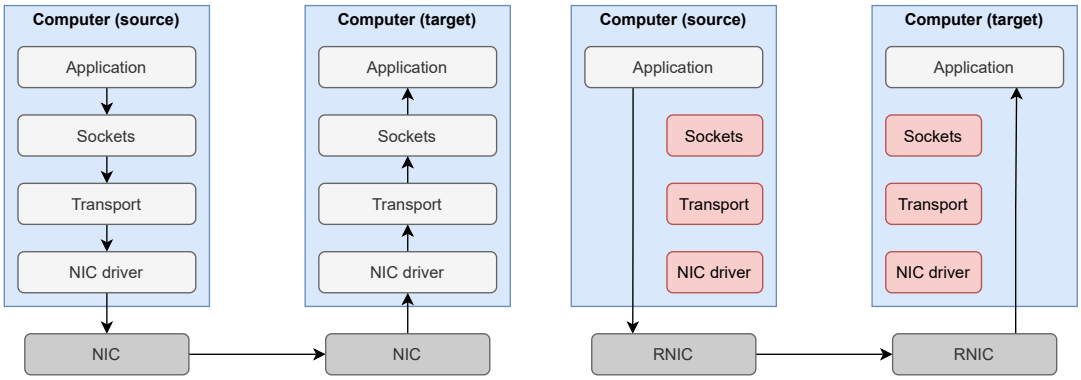
2.1.1 Choice of Data Structure

Because KV stores do not need ordered or strongly structured data, they can use simple, but fast data structures to hold the data. In practice, hash tables are a common choice [9, 11, 22]. A hash table is a data structure that maps keys to values. It uses a hash function on a key to generate a number, which is then used as the index into an array. At this index's location, the key-value pair can be found. A good hash function will generate a unique value so that the pairs can be spread uniformly across the array. This way, when a new pair is inserted, the probability is much higher that the pair holds a unique location in the array, and subsequent lookups will immediately find it at this same location. This type of lookup functionality is generally faster than traditional lookup methods, such as search trees, because these require multiple comparisons per lookup.

2.2 RDMA

What exactly makes RDMA so attractive? As mentioned previously, it moves processing into hardware and skips many copying operations (zero-copy networking). Figure 2 shows how RDMA manages to skip these operations by using the specialized hardware, from an applications point of view. By removing layers of interactions with the operating system, the path of the data is shortened drastically [29]. Hardware systems have had DMA (Direct Memory Access) available for quite some time. It is used to reduce the amount of work the CPU has to do for access to main system memory. The CPU only tells the DMA engine what to transfer and where to transfer it, and the DMA engine takes care of the rest. Once the DMA engine is done it generates an interrupt, which tells the CPU that the transfer is complete. RDMA extends DMA by allowing this mechanism to be

performed on remote computer memory. This conveniently leaves the CPU to do other work.



(a) TCP/IP (b) RDMA

Figure 2: The difference between using TCP and RDMA, TCP performing several actions inside the OS (on CPU), whereas RDMA uses the RNIC to transmit and receive data directly from the application.

2.2.1 Network awareness

RDMA provides two types of communication: one-sided and two-sided. One-sided means only one side performs some action to do the operation, i.e. only one side is 'aware' of the fact that the communication occurred. Two-sided means both sides are aware of the communication and are required to participate in order for the communication to complete successfully. Both of these types use DMA to perform the actual memory transaction. RNICs use DMA to write and read this memory directly. Then, only if the operation is two-sided, the application with which this memory is associated is explicitly notified.

2.2.2 Verbs and work requests

These two types consist of four distinct operations, called verbs: WRITE, READ, SEND, and RECV. WRITE and READ both perform remote memory operations without the remote systems knowledge (one-sided). SEND and RECV, however, are coordinated (two-sided). To SEND to a remote system, the remote system must have first initiated a RECV. The SEND then 'consumes' the first available RECV found at the remote system.

Unlike in TCP, which sends byte streams, RDMA works with the notion of work requests. A work request represents one operation, that is, one READ, WRITE, SEND, or RECV. To set up a work request, it is 'posted' to the queue pair (QP). The QP consists of two queues, one for send work requests and one for receive work requests. When work requests (of either queue) are completed, a work completion is placed on a completion queue, which is associated with that QP. The completion queue can then be queried for these completions, allowing the applications to check their status.

Because this QP is used for any network I/O, it can be viewed as the connection in RDMA. In order to send anything, however, a memory region will have to be explicitly pointed out for usage. This memory 'pinning' is what allows the hardware to perform

network I/O to this user-specified memory. The RNIC maintains a table of all memory pinned by the application in its cache. The application that wants to WRITE to or READ from some remote memory, will first need some information about this memory. It needs to know the remote address, which has to be registered with the remote RNIC. It also requires a security key, without which, the WRITE or READ will be rejected. Such information can be exchanged beforehand during the establishment of the connection. More detailed information on RDMA can be found in [3, 27].

2.2.3 Transport types

RDMA supports three transport types to perform the verbs on. Reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD). These different transport types all support a different subset of verbs, and only RC supports all verbs. An overview of which verbs are supported by which transports is given in Table 1.

Verb	RC	UC	UD
SEND/RECV	✓	✓	✓
WRITE	✓	✓	-
READ	✓	-	-

Table 1: Overview of transport types RDMA and available verbs for each type

3 Designing PERK: a distributed RDMA key-value store

In this section, the design of PERK is discussed. First, relevant factors for the design are discussed. Then, possible design choices are touched on, and reasons for their inclusion or exclusion are explained.

3.1 Trade-offs

Broadly, the goal of the application is to minimize CPU usage (mostly server-side), and get higher performance compared to systems currently in production that use sockets, i.e. more GET/SETs per second.

Because one-sided and two-sided verbs operate differently, the more abstract decision for using either can be dependent on how long the connection lives. SENDs and RECVs do not require prior exchange of memory information (address and security key) because they use pinned memory that has been specified in the respective work requests. A SEND cannot choose what memory to place the data in, but instead places the data in the memory associated with the first available RECV in the remote queue pair. Because the local and remote queue pairs are linked through special identifiers, the SEND cannot consume RECVs from any other queue pair. If the remote queue pair does not contain a RECV work request with similar size specifications, the SEND is rejected. Due to the dynamic usage of SENDs and RECVs (i.e. not having to specify a remote address to perform the operation), they are simpler in usage. However, they are also slower, as they require some extra processing on both the sending and receiving side, i.e. uses more CPU. WRITE and READ, on the other hand, require the exchange of information, because the remote memory's address has to be attached to the WRITE or READ work request explicitly. Thus, their usage is slightly more complex. However, these operations are performed entirely by the RNICs and DMA, leaving the remote CPU to do other things. Of course, this also means the remote side is unaware and needs a mechanism to actually find out the memory has changed.

Using one-sided RDMA, it is possible to alleviate the majority of compute power required server-side. By exposing the internal data structures used to store the key-value pairs, the clients can effectively READ pairs directly from the remote store [9, 22, 31]. Unfortunately, RDMA only allows for reading of raw memory, and not for advanced programming concepts, such as pointer dereferencing. This limitation usually means retrieving a pair requires multiple READs, for reading metadata of the structure to locate the pair of interest, and one for reading the desired pair from memory. For PERK, this approach was avoided due to several more complex synchronization problems that have to be solved to ensure data correctness.

3.2 PERK's request cycle

PERK uses a server-client model, in the sense that in order to distribute memory of the system, a main computer (the server) creates and maintains the memory and a set of other computers (the clients) connect with the main computer to indirectly perform operations on this memory. In systems like Memcached, this communication is done using sockets, PERK uses RDMA for communicating any data. Regardless of which verb is used, the request cycle itself is straightforward:

1. The client reads a request, either from a file or through input.
2. The client parses the request into a request structure that holds the type of the request, the key, and the value.
3. The client sends this request to the server by using a RDMA verb of choice.
4. The server detects the new request and start processing depending on the type of request:
    - If the request is a GET request, it looks up the value for the given key in the hash table. Then, if the pair was present, the value is copied into pinned memory and a flag is set indicating whether the response is empty or not.
    - If the request is a PUT request, it inserts the given pair into the hash table. Here, it also sets the flag upon completion.
5. The server sends back the response to the client using some RDMA verb or the client retrieves this response itself.
6. The client detects the response, and can do any processing with the data..

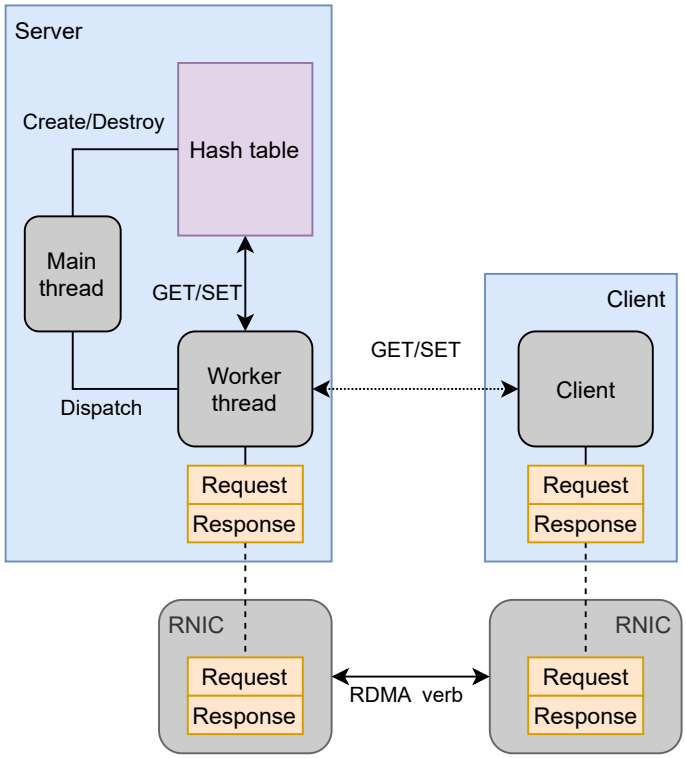


Figure 3: Abstract overview of how PERK works.

3.3 PERK verbs

The design choices in PERK can be aptly summarized by the location in the applications request-cycle where RDMA was used. PERK uses a design that, similar to Herd [20], substitutes the network I/O normally performed through TCP for RDMA-based operations. This implies, the server and client still perform a regular set of tasks, i.e. the client sends requests over and the server processes them. Unlike designs that avoid CPU usage server-side [31, 22, 9], in this design the server still performs tasks such as adding entries to the store and retrieving them.

Using this design allows for a generic setup, where the network operation can be abstracted away from the application, without introducing any overhead. This abstraction also allows for various configurations in the same implementation. Here, configuration means different combinations of RDMA verbs for the request and response mentioned earlier. Combinations have been chosen mostly for diversity, to provide a good sense of the differences in performance, and to give an elaborate evaluation of the 'best' verbs over RC in the context of a key-value store. Therefore, one double two-sided (SEND/SEND), two with both types (WRITE/SEND, WRIMM/SEND), and two with double one-sided (WRITE/WRITE, WRITE/READ) are integrated in PERK. Table 2 provides an overview of the combinations supported by PERK, where the operation for the request and response is indicated for the client-side and server-side. More combinations could have been added but were mostly excluded due to time constraints. Using READ server-side has been excluded due to the nature of its usage. Having a server READ a client's memory until a request is found potentially requires multiple READs before the client completes filling in the full request. Thus, to preserve resources server-side, this is avoided. The same applies to a SEND/WRITE configuration, as two-sided verbs mostly cause overhead for the receiving side [20]. Therefore, any combinations that use SENDs (except for SEND/SEND), use the SEND server-side, to move the overhead of posting the RECV to the client-side. The WRITE with immediate data additionally consumes a RECV on the remote side, this can handily be used to signal the remote application of the WRITE's completion without it having to check the memory manually. This immediate data will be very small (1 byte) to limit extra overhead caused by having to transfer data.

Configuration	Request (C)	Response (C)	Request (S)	Response (S)
sd_sd	SEND	RECV	RECV	SEND
wr_sd	WRITE	RECV	-	SEND
wrimm_sd	WRITE IMM	RECV	RECV	SEND
wr_wr	WRITE	-	-	WRITE
wr_rd	WRITE	READ	-	-

Table 2: Overview various RDMA configurations used in PERK

Because every transport except RC only permits a subset of the verbs, this study focuses on RC exclusively. It is worth noting that combinations of different transports are perfectly viable. In fact, combinations of UC and UD have shown excellent performance [20, 18]. Despite their ominous names, RDMA traffic tends to be very reliable, and any lost packets can be taken care of using application-level resends. For PERK, this will not be necessary, as using RC enforces hardware-level resends. It can thus assume that any sent packet is

always received.

Figure 4 illustrates the idea of how each verb is used in PERK. It shows the interactions that may take place at any given time in the request cycle for the client-side. Depicted in Figure 4a, is SEND-based communication. This requires a RECV request to be available on the remote side. Therefore, for any configuration involving SEND/RECV, the client or server will pre-post a RECV to ensure availability. The WRITE-based communication shown in Figure 4b is a bit more tricky. As the server is unaware of the occurrence of the operation, it will have to periodically check the contents of the memory associated with a particular client. By doing this, the server can determine whether a request was received. Figure 4c shows how a READ-based approach operates. The local process continually retrieves the contents of the remote memory and checks those contents to determine if the request was completed. For the READ-based communication, there is a chance one READ retrieves the contents right before the request is completed. The local process will then have to perform one/multiple more READs until the request is indeed finished.

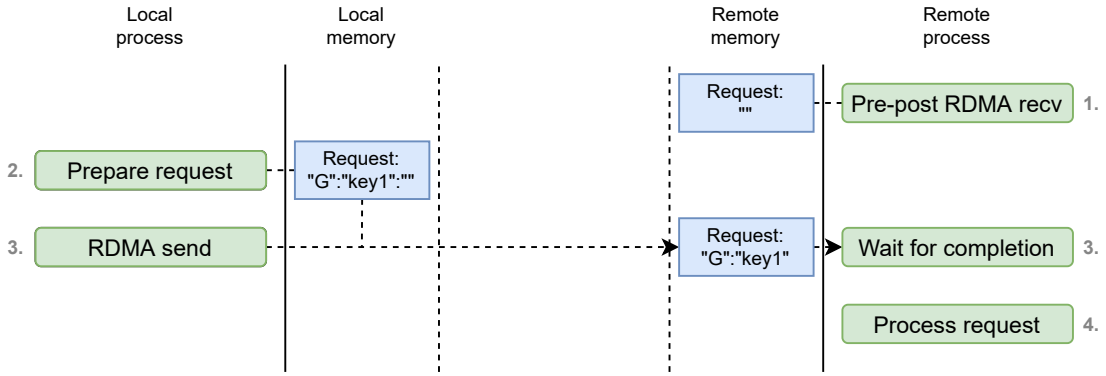
3.4 PERK's hashing backend

Various types of hashing algorithms have been researched in the setting of key-value stores. Cuckoo hashing often uses multiple hash functions to provide several possible locations in an array for a key-value pair to reside [25]. This can be beneficial because it guarantees the key will be found in K lookups (for a scheme using K hash functions) or is not in the table. Cuckoo hashing has been used successfully for improving the performance of Memcached [11], as well as other memory-efficient key-value store design [21]. Hopscotch hashing uses a slightly modified approach. It uses fixed-size 'virtual buckets' to store the pairs. A single hash function provides the index of the bucket, after which the bucket is linearly probed until empty entries are found (for PUTs) or the desired entry is found (for GETs) [15]. PERK chose a simplified design similar to a generalization of cuckoo hashing: blocked cuckoo hashing [8]. PERK uses a single hash function to determine a pair's location, and fixed-size buckets (two pairs per bucket). It uses two consecutive arrays, of which each entry i of one array belongs to the same bucket as the ith entry of the other array. In the case of a hash collision, and a full bucket, the last entry is evicted.

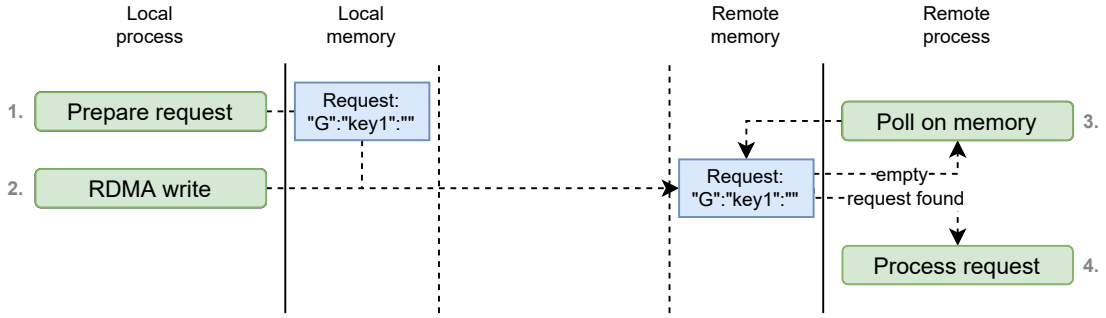
Although this makes the scheme intolerant of hash collisions, we argue that for the scope of this project this is sufficient, and provides some performance-wise favorable properties. First, it guarantees the search of any key to complete with the computation of one hash function, followed by one or two array comparisons. Second, it does not require resizing, as adding newer entries into a full bucket results in the eviction of the oldest pair, avoiding the displacement cycle that can normally occur with cuckoo-hashing, and potentially lead to a failure anyways. Memory-wise, however, this is quite inefficient, as when only a smaller set of popular pairs is stored, a large portion of the array will remain empty. More limitations of this design are discussed in Section 7.

3.5 Multithreading in PERK

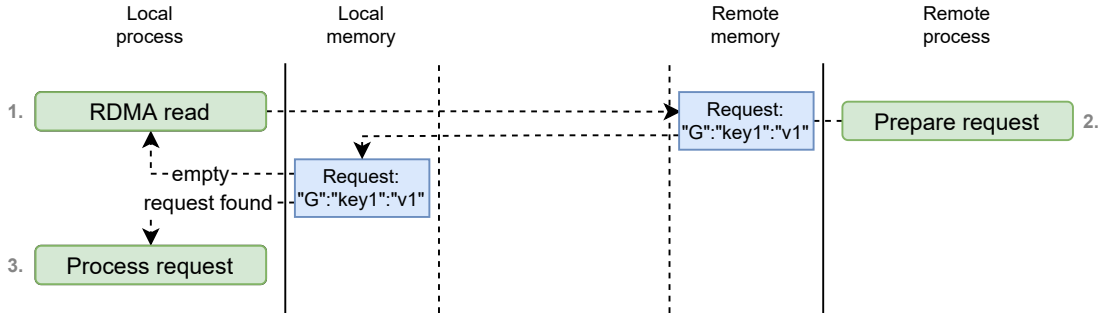
To process as many requests as possible, the server will dispatch a thread for each active client. Each thread can access the hash table that stores all the key-value pairs. If threads could access this at will, they might access a pair that is being changed, causing



(a) Send-based communication



(b) WRITE-based communication



(c) READ-based communication

Figure 4: Forms of communication used in PERK. Every form is applicable to both client- and server-side, except the READ-based communication, which is exclusively used client-side. Here, each form is displayed from a client's perspective.

a dirty read (reading of memory in an inconsistent state). To prevent this, a lock on the entire hash table can be used. Then, when a thread wants to read some entry from the store, it has to obtain the lock first. This adequately protects the integrity of the store, however, does not allow concurrent access. Because while some thread holds the lock for an expensive operation, like the insertion of a new pair, no other thread will be able to read any data. This idea can be improved slightly by using read/write locks. These locks allow simultaneous reads of the data, but in the case of a write, the other threads must still wait. This approach is better but still suffers from contention caused by modifications. Therefore, the solution that PERK uses, is a lock on individual buckets in the hash table. Now, when a thread wants to modify some entry in the table, it acquires a lock only for that bucket. Every other bucket can then still be written to and read from. This comes at the cost of extra memory usage, as each entry now additionally requires the existence of a lock, but allows faster access.

4 Implementation of PERK

In this section, the implementation details of PERK are discussed, dependencies of PERK are touched on, as well as the libraries used. A common theme in this section is race conditions, and how they are prevented.

4.1 Dependencies

PERK is written in C. C is often used for high-performance applications. Its lower-level interface allows for many code optimizations because the machine code is simpler. It provides relatively few libraries (compared to languages such as Python, Java, etc.) and is a rather fault-intolerant language, in the sense that it will not provide many of the utilities other modern programming languages provide, such as garbage collection or array boundary checks. The responsibility of writing a correct program lies entirely with the programmer. Everything used PERK, is native C. For multithreading, PERK uses pthreads.

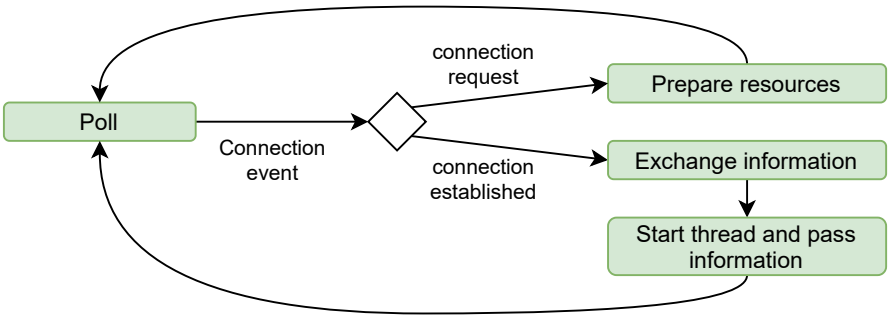


Figure 5: The connection flow when a client connects. First the request is accepted, and some resources are allocated. Then the connection is established, resources are exchanged and a thread is started for the client.

4.2 PERK core

The process of setting up a client-server connection is split up into several steps. RDMA conveniently provides connection management primitives that can be used to establish connections in two phases. The flow is shown in Figure 5. The client initiates the connection by sending a connection request to the server's connection management channel. As soon as the server accepts this request, both sides prepare a set of general resources (queue pair, completion queue) required for later communication. The client then sends a message indicating the connection is established client-side and, after the server confirms, any additional information about remote memory regions is exchanged. Establishing connections in split phases, allows multiple clients to connect at the same time. The server maintains a table of connection slots, where each entry holds all relevant information for a single client. On a connection request, the client is added to an empty slot. Afterward, when a client indicates the connection establishment, the server finds the slot for this client using a connection identifier that is uniquely associated with that client. After this client has gone through both phases of the connection flow, a thread is dispatched and all

the data stored in the slot is passed to it. Now, it freely communicates with the client until the client terminates the connection.

In Figure 4, the basis of communication was presented for each communication primitive. Now, an example for a full request-cycle is given. In Figure 6, one can observe the request-cycle of the WRITE/SEND configuration. This configuration uses a WRITE for sending the request from the client to the server. The server polls this block of memory until the request was fully written, processes the request, and then SENDs back the response to the client. Listing 1 shows a main loop with correct usage of this request cycle.

```
while(not_done){
    prepare_request(request);
    post_recv(response);
    rdma_write(request);
    wait_for_recv_completion();
}
```

Listing 1: Example main loop flow client-side for wr_sd configuration with no optimization.

The client prepares the request, then pre-posts the RECV right before sending the request to the server. Pre-posting the RECV ensures that when the server quickly SENDs back the response, the RECV is ready. Although this approach is correct, after the client sends the request to the server, it waits idly until the result is sent back. Instead, the client can use this otherwise wasted time to post the RECV, this approach is shown in Listing 2.

```
while(not_done){
    prepare_request(request);
    rdma_write(request);
    post_recv(response);
    wait_for_recv_completion();
}
```

Listing 2: Example main loop flow client-side for wr_sd configuration with race condition as result of attempted optimization.

Unfortunately, this scenario is problematic, as the RECV costs more time to set up than performing a SEND, the server may attempt to SEND before the RECV is ready. If the RECV is found by the NIC in time, the SEND can complete, otherwise, the SEND will retry some number of times before failing. Accordingly, this causes the server to crash. To avoid this, an extra RECV can be posted in advance, i.e. before the main loop starts. After, whenever the client waits, it can prepare a RECV for the next request, this is shown in Listing 3.

```
post_recv(response);
while(not_done){
    prepare_request(request);
    rdma_write(request);
    post_recv(response);
    wait_for_recv_completion();
}
```

Listing 3: Example main loop flow client-side for wr_sd configuration with simple optimization. Posting the RECV before the loop starts to ensure availability, and using idle time to post the next RECV.

Apart from saving time otherwise spent performing no tasks, this has an additional lower-level benefit. NICs cache some resources to prevent having to retrieve them over PCIe (which costs time) [18]. Those resources include queue pairs and work queue elements. Posting the RECV far enough in advance of the request, allows the NIC to retrieve and store the RECV in its cache. As a result, when the server initiates the SEND, the client's NIC can instantaneously consume the RECV.

The configuration that uses WRITE with immediate data functions essentially the same as WRITE/SEND version. There is one crucial difference, which is that the client's WRITE also consumes a RECV server-side. This is exclusively to create a simpler way to signal completion on the server-side. Rather than manually inspecting the memory associated with the request, the server can simply check whether the RECV has completed, and then read the request memory. With a general sense of the communication involved for configurations involving SENDs, the remainder of this section focuses on complications and solutions for the WRITE/WRITE and WRITE/READ versions and briefly explains the disconnection of clients at the end.

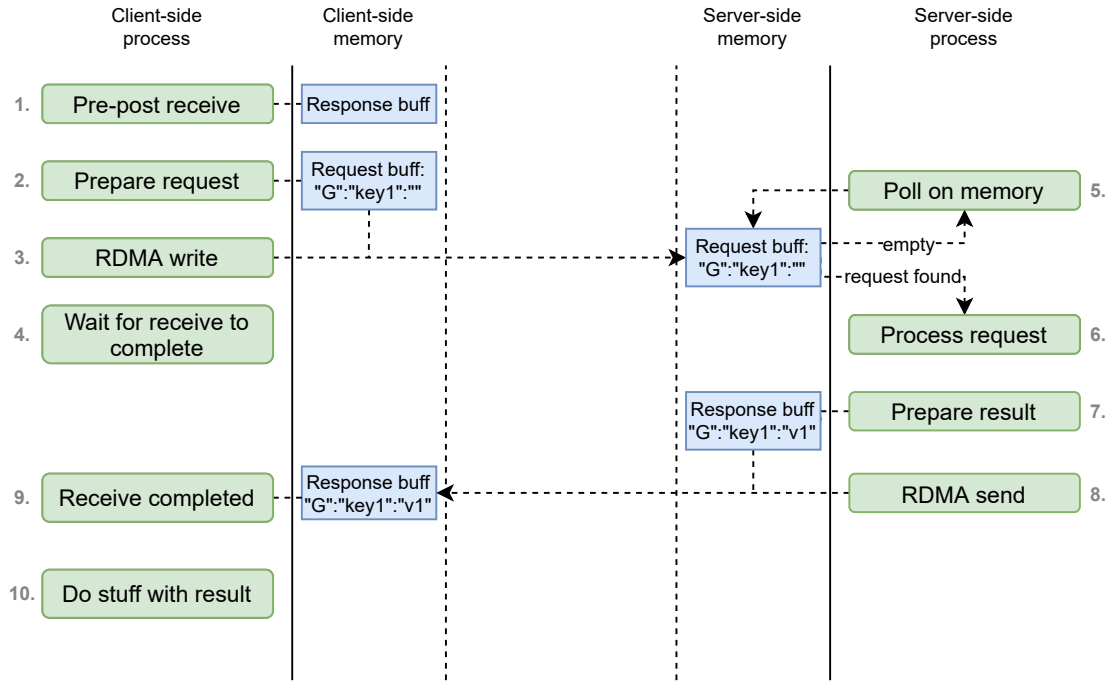


Figure 6: Example request cycle for the wr_sd configuration in PERK.

Using WRITE twice has one significant advantage over SENDs: it does not require any coordination (after the exchange of memory information). A WRITE can be performed, and the assumption is made that the remote side will simply check the memory and pick up the request/response. The only thing either side using a WRITE has to be mindful of, is to create a way for the receiving end to distinguish requests from previous cycles with new requests. This is done by simply modifying the flag indicating the type of the request, then the receiving end will only continue once the flag has a particular value. The type of the request is purposefully placed at the end of the structure to prevent processing of partially written requests because WRITEs are performed in increasing address order

[9]. Even though this is unlikely to occur because a WRITE for small objects is very fast. This only applies to memory in which messages are received, so for the client-side response memory, and the server-side request memory.

The WRITE/READ configuration allows the server to perform less work by having the client read back the results of its request. For the client, this means more CPU usage, but saving CPU usage server-side enables it to process more requests. There is no guarantee that the response will be ready when the client attempts to read from server memory though. Thus, the client will perform READs until the response has been completed server-side. There are still some race conditions when carelessly implemented. Say the client successfully completes a single request-cycle and WRITEs a new request to the server. Because these operations are fast, the client may READ the response of the last operation. Normally this could be prevented by having the server set the memory to zero, however, it has no way of knowing whether the client read the result yet or not. Hence, this is not an option, instead, the same memory in which the request arrived is reused. Then, the client will overwrite the previous response with the new request, and when READ back, will find a flag indicating the resulting response or the request it just sent.

Finally, once a client has completed all the requests in wanted to have processed, it sends a normal request, but with an exit flag. The worker thread will find this flag, clean up, and exit. Once the worker thread and client have disconnected through the message exchange. The client sends the server a disconnection event. Once again, this provides the main thread with a connection identifier, which it can use to find the entry in the client table, and clean up any further resources.

4.3 Interfacing with Memcached

To properly compare PERK with Memcached, we need a client that can communicate with Memcached servers. Memcached provides a library for doing so. By reusing the same structure of PERK's client for reading and parsing requests, it is relatively easy to create a Memcached client. Using the same backend for both will also ensure the performance is not dependent on other optimizations on the client-side.

5 Performance Evaluation

In this section, PERKs performance is evaluated based on the following metrics: operations per second for increasing numbers of clients, CPU usage, latency. Aside from these benchmarks, this section will explain the experimental setup on which the benchmarks were performed. First, we provide a brief summary of the findings:

1. In terms of scalability, PERK scales well for all payload sizes to up to 16 clients, showing almost linearly increasing performance with respect to the number of clients. However, only for smaller sizes does PERK outperform Memcached in terms of operations done per second, with over 400% more ops/sec for a single client, for 32 byte payloads.

2. PERK has low latency for smaller payload sizes: the round trip time of PERK is about 4.8x shorter than that of Memcached. This latency grows little for larger payload sizes and is very stable, showing little variability.

3. CPU-usage is 35% less for PERK's best configuration, whereas PERK's worst performing configuration uses roughly equal CPU, but still providing 3 times higher throughput. Due to inefficient input functions, the performance for larger impacted is drastically reduced and uses over 150% more CPU.

5.1 Experimental setup: DAS5

All experiments for the performance of PERK were run on DAS5, the Distributed ASCI Supercomputer. More specifically, on the cluster in the domain of VU [6]. DAS5 has 68 nodes available, most of which have a dual 8-core Intel E5-2630v3 CPU. The nodes are internally connected using Infiniband FDR interconnects that have a bandwidth of 48 Gb/s. To run any benchmark, a special infrastructure was built to allow CMake to conditionally add extra preprocessor directives to the project. This makes it so macros can be defined to print benchmarking information, or alternatively, to leave the definitions empty. As a consequence, the compiler can create machine code that does not contain unnecessary print statements. By adding conditional definitions for payload size, this system can modify the constant length of key-value pairs. Using the reservation system of DAS, a single node is reserved for the server, which is then started on that node in the background. Then, some number of clients is started in parallel, ranging from 1 to 16.

Because the workload is very important for the performance of key-value store , this study chooses to generate a very common type: read-intensive. Read-intensive workloads generally contain mostly GET requests (+/-95%) [5]. To give the workload some weight, a relatively small range of keys was used: max 999 different keys. This ensures the store will at some point contain live key-value pairs. For the value, a random string of characters was generated. The length of the values is calculated to match a required payload size. Because the requests are sent a single packed structure, the total size of the payload is predictable. We have $payload\_size = (1 + 8) + x$, with x the payload size in bytes, and 1 and 8 the sizes of the request type, and key, respectively. The actual range of $payload\_size$ is from 32 to 2048 bytes. In total, for all the payload sizes, 16 sets of 3,000,000 random GET/SET requests were generated

Memcached was used as a baseline for each of the following experiments. It is a commonly used key-value store in large corporations, e.g. Facebook [5, 23]. It is a long-running project with many years of development. This makes it an interesting baseline to compare against.

5.2 Scalability

By measuring the number of GET/SET operations performed per second, for both PERK and Memcached, we can get a basic idea of whether using PERK improves performance. Operations performed per second also have a strong relationship with the number of clients connected. That is, if PERK scales well, for increasing numbers of clients there should be an increase in throughput (ops/sec). Ideally, for each doubling of clients, the number of ops/sec should also double. The experiment for ops/sec was repeated five times for each configuration and Memcached, and all payload sizes. Figure 8 shows this performance for 1, 2, 4, 8, and 16 clients, for four distinct payload sizes. The performance will be evaluated as follows. Single client performance is explained briefly, then scalability is discussed.

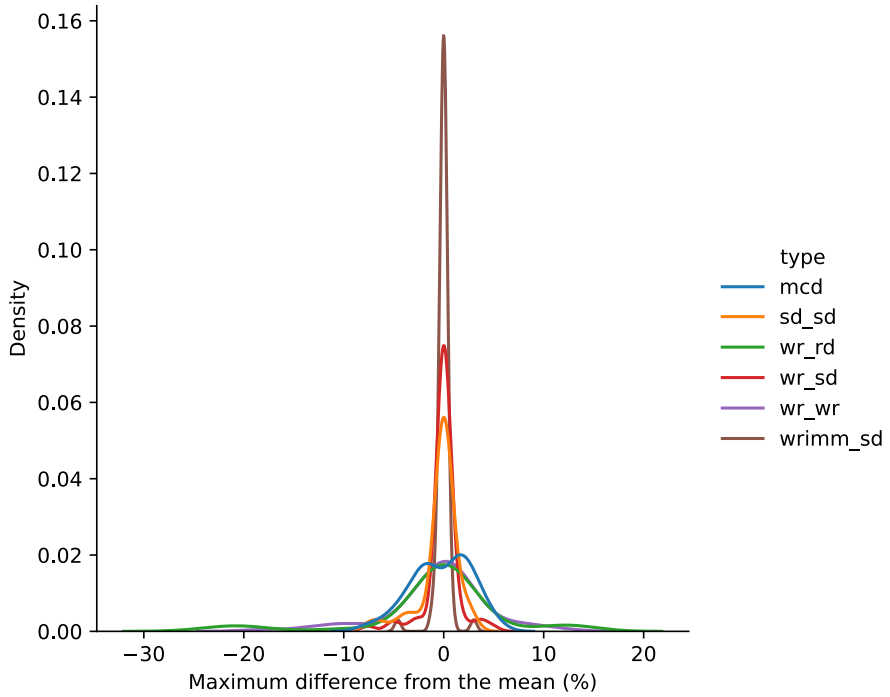
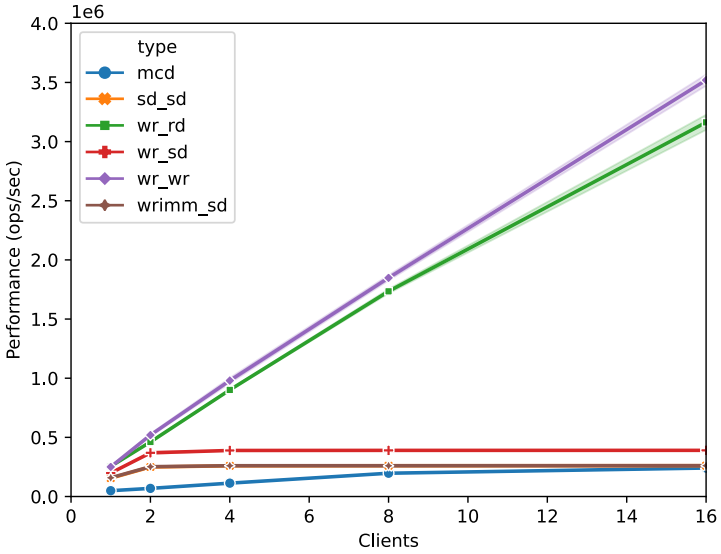
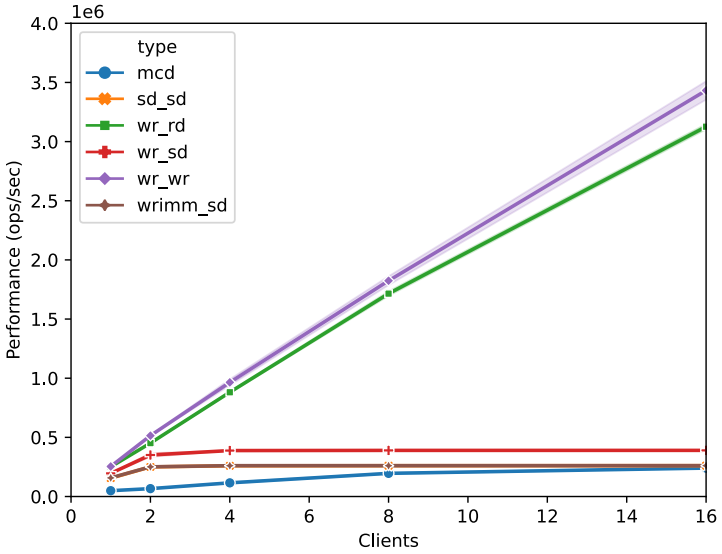


Figure 7: Distributions of difference between a configurations (including all payload sizes) min/mean and max/mean. For higher variability in performance like WRITE/READ and Memcached, the distribution is more flat.

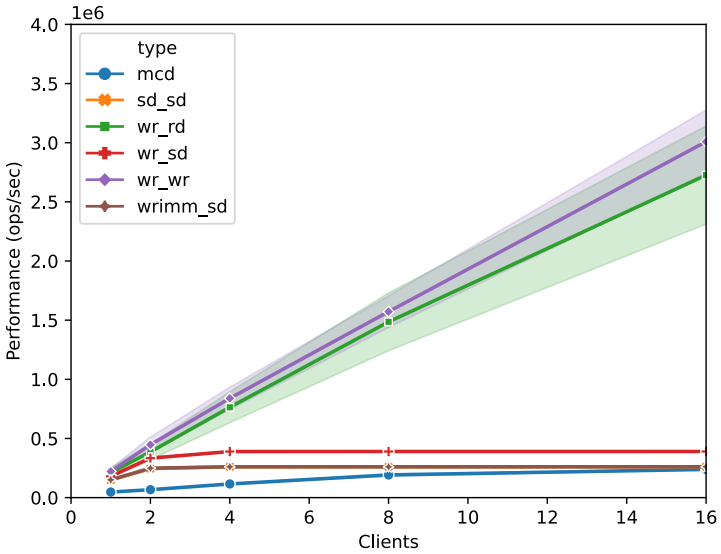
Any single-client verb configuration of PERK outperforms Memcached significantly for small payload sizes. The worst-performing combination (SEND/SEND) performs >2 times as many operations per second, whereas the best (WRITE/WRITE) performs 4 times as many operations, reaching up to 250K ops/sec. With increasing sizes, however, PERK's performance slowly degrades, until, at 2048 byte payloads, it is roughly equal to the



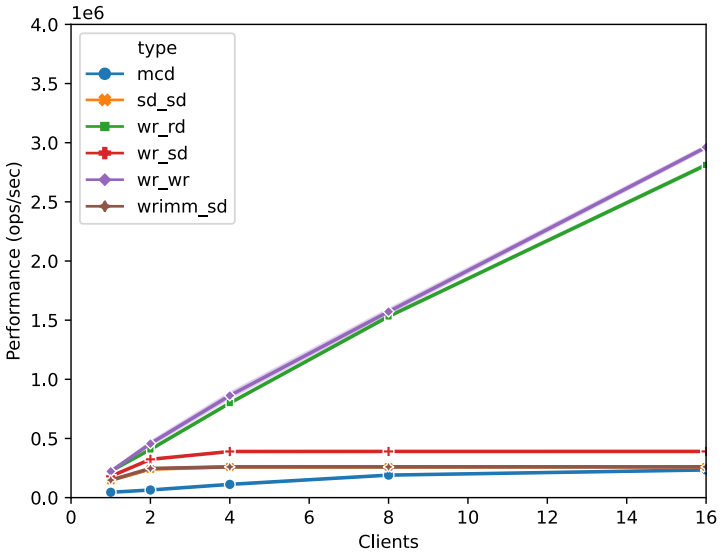
(a) 32 bytes



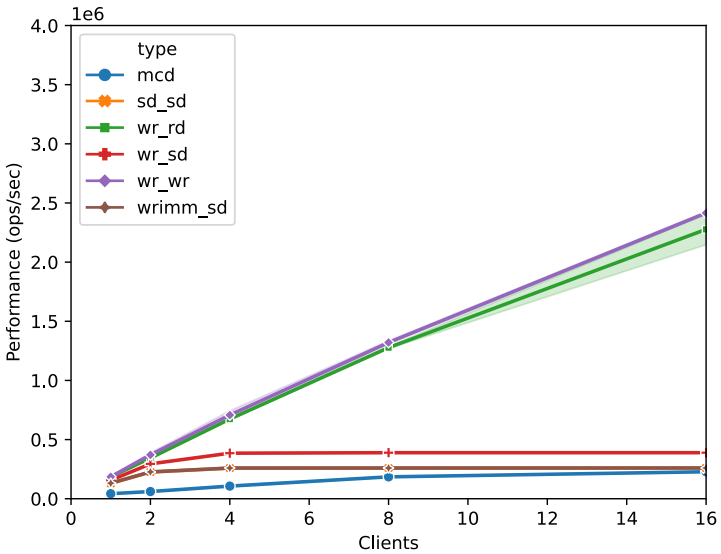
(b) 64 bytes



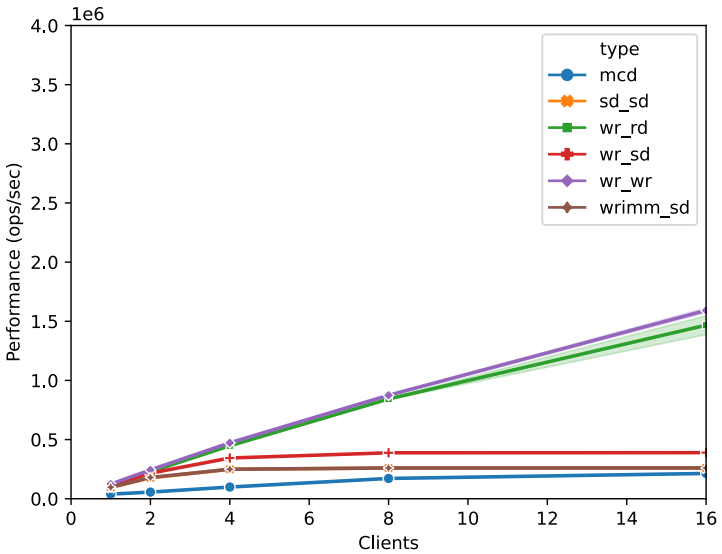
(c) 128 bytes



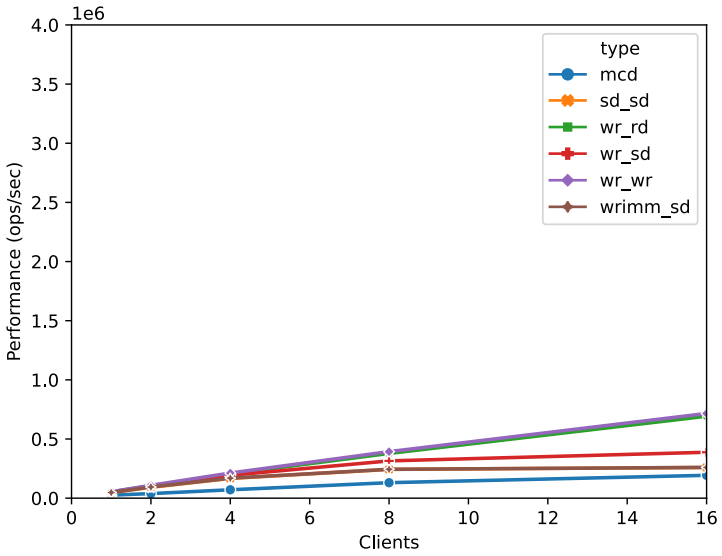
(d) 256 bytes



(e) 512 bytes



(f) 1024 bytes



(g) 2048 bytes

Figure 8: Scalability of PERK for various payload sizes (see a-g). The y-axis gives operations per second on a scale of 1e6 (millions).

performance of Memcached. This is caused by inefficiencies in PERK's client and will be discussed in more detail when evaluating CPU utilization. Even though the different combinations vary in their performance, at a single client, each configuration performs consistently. Over five runs, the greatest variability in performance was found for the WRITE/READ configuration at 128-byte payloads, where the greatest difference from the mean had 23.3% fewer ops/sec. This can be explained by randomness in the number of reREADs that have to be performed. When the server completes the response right after the client READs the response memory, the client must READ the memory again, which for smaller payload sizes, is still the majority of the request cycle time spent.

Figure 7 shows an overview of this same variability for 1-16 clients and all payloads, both for PERK's configurations, as well as Memcached. It shows Memcached, PERK's WRITE/READ and WRITE/WRITE configuration to have the greatest variability, having the largest difference between the mean ops/sec and minimum/maximum ops/sec for 128 bytes. Configurations using SEND-based communication appear most stable, but this is partially due to these configurations reaching a plateau after adding only a few clients. This plateau is more apparent in Figure 8a, where the initial increase (from 1 to 2 clients) for both the WRIMM/SEND and SEND/SEND configurations is from roughly 150K ops/sec to 250K. Then, for 2 to 4 clients, both increase only slightly more to around 260K. After 4 clients, neither of these versions scale any further, nor does the WRITE/SEND version, which reaches initially higher throughput, but then suffers from the same issue that other configurations using SEND-based communication suffer from. The exact cause of this is unknown, and relatively little literature is available on RC SENDs and RECVs, as most research focuses on either one-sided operations over RC/UC [31, 9, 22], or two-sided operations over UD [7, 19, 20, 18]. We therefore refrain from further speculation on the performance degradation of multi-client two-sided verbs over RC.

Regardless of the shortcomings in scalability for SEND-based communication configurations, both the WRITE/WRITE and WRITE/READ versions scale well, scaling almost linearly with the number of clients up to 8 clients. Followed by a slightly shallower slope from 8 to 16 clients: 91% increase for WRITE/WRITE and 70% for WRITE/READ. This difference can be attributed to the client performing multiple READs to retrieve the result, which can happen when the client READs the remote memory right before the server

completes setting up the response. The WRITE/WRITE version is constantly polling the memory and is essentially instantly aware once the response has been transmitted by a server-side WRITE. The number of READs per request were measured for both 32 and 2048 bytes. Table 3 shows these READ counts for the WRITE/READ version, on 16 randomly generated sets of 3,000,000 operations.

	32 bytes		2048 bytes	
	GET	PUT	GET	PUT
Min	1	1	1	1
Max	114	22	181	231
Mean	1.0013	1.0012	1.0019	1.0029

Table 3: Average number of READs performed for WRITE/READ version's client.

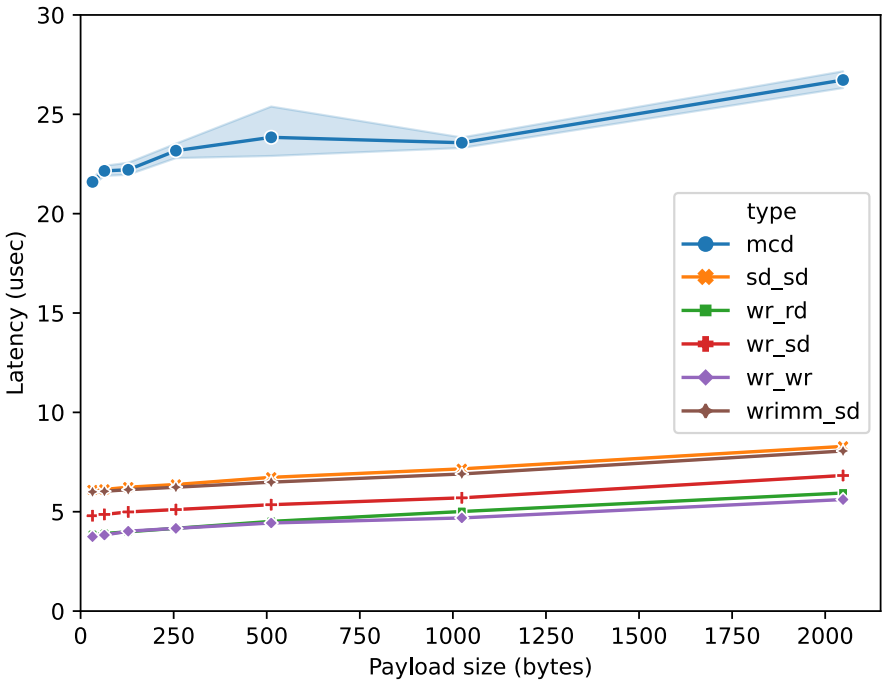


Figure 9: Overview of latency (RTT) for a request in PERK and Memcached for various payload sizes.

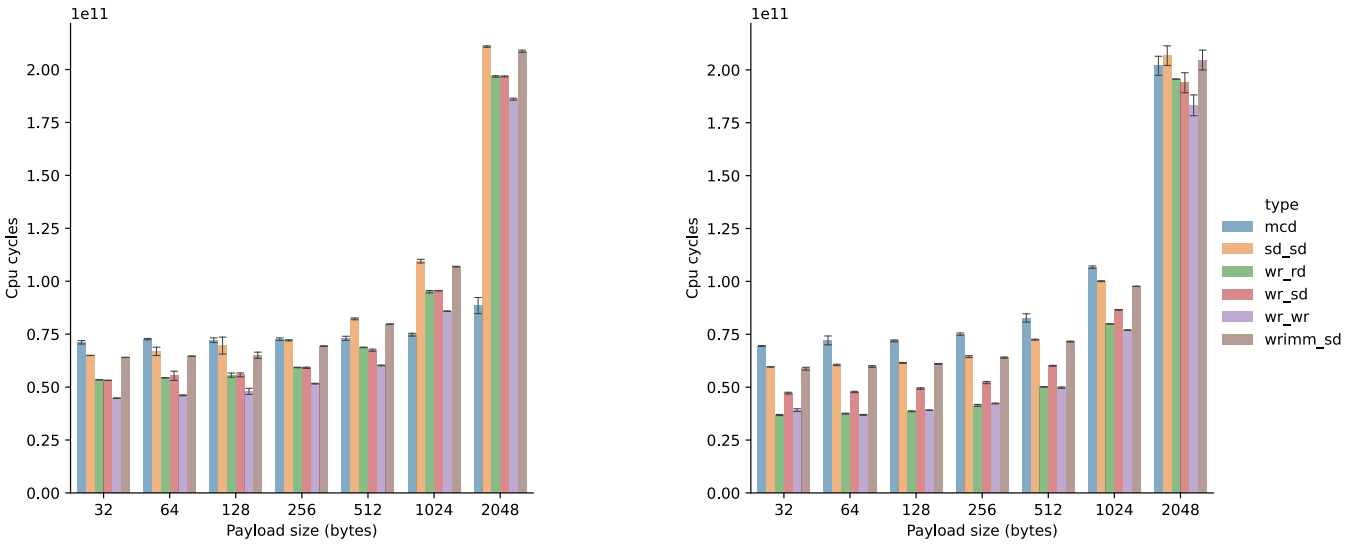
5.3 Latency

The latency of PERK determines the response time, or RTT (round trip time) of a single request. For each request, the time from sending the request until receiving was measured. This excludes any client-side processing like preparing the request or checking the result of the request but does include server-side processing such as inserting or retrieving key-value pairs. Figure 9 shows the average latency over five runs of this experiment. In each run, a single client was connected to a single server thread and sent and received 3,000,000

requests. The SEND/SEND configuration has roughly 2.4 times lower latency, and the WRITE/WRITE about 4.8 times. Besides it being much lower, PERK's latency is much more stable than that of Memcached. This is mostly due to PERK constantly polling memory, and thus never missing any new request due to timeouts, whereas Memcached's server will sleep, i.e. require multiple context switches before being able to process the request. PERK's latency increases only slightly with larger payload sizes while maintaining this same stability. At 2048 bytes, PERK's WRITE/WRITE latency is still >4 times lower than that of Memcached.

5.4 CPU-usage

Because RDMA allows OS-bypassing and avoids various copying mechanisms, the CPU usage should be lower than TCP-based applications. For this experiment, the perf tool was used to measure CPU cycles on both the server-side and client-side. One CPU cycle is the duration of a clock tick of the CPU, during which the CPU is active for a particular process. Figure 10 show the CPU cycles consumed for all configurations and Memcached, both client-side and server-side.



(a) Server-side CPU usage

(b) Client-side CPU usage

Figure 10: CPU usage of PERK for various payload sizes.

In Figure 10a, which shows the server-side CPU cycles consumed, we see a maximum of 35% fewer CPU cycles consumed for the WRITE/WRITE configuration, with overall lower CPU-utilization up to 512 bytes. The SEND/SEND configuration consumes slightly fewer to equal CPU cycles, but still provides much higher throughput for up to 256 bytes. However, this CPU usage increases dramatically at 1024 and 2048 bytes. Figure 10b shows the CPU cycles client-side. PERK's clients and the Memcached client use the exact same methodology up until the moment of sending and receiving, after which the implementation is controlled by Memcached's library and server implementation. Due to this, we see a larger difference in CPU utilization here for all configurations, with a >40% decrease in

CPU cycles consumed for WRITE/WRITE and WRITE/READ configurations. However, we see an equally large increase in CPU cycles for the client-side at 1024 and 2048 bytes.

To better understand the cause of this increase, each component of the request cycles on the server- and client-side were timed to see what these cycles were consumed for. Table 4 shows the time spent for receiving and processing (inserting/looking up) the request, as well as sending the response and remaining cleanup for the next request cycle for the WRITE/WRITE configuration server-side. At 32 bytes, the majority of the time is spent sending the response to the request of the client. At 2048 bytes, the majority swiftly shifts to receiving the request. This means that during this time, the server is constantly polling on request memory while nothing is there, hence wasting CPU. In order to explain this shift, a similar timing was done on the client-side, also for the WRITE/WRITE configuration. The results of this are shown in Table 5. Accordingly, the client-side spends the most time receiving the result in most cases. In the entry for 2048 byte PUT requests (which requires the client to read in a long key-value pair from a file), we see the source of both the increase in CPU-usage client-side and why the server-side has to wait so long before a request is received. The client uses a very inefficient function for reading the request from a file and as a result, forces the server to spin for long periods of time. Although the workload consists only 5% PUT requests, the impact is significant because in the time a single PUT request is loaded the client can load over 1400 GET requests. A simpler input function can improve the performance for larger payload sizes significantly. For example, with a minor change, PERK's WRITE/WRITE configuration can perform 167k ops/sec, rather than 50k ops/sec for 2048 byte payloads. But this is for future implementations to improve further.

	32 bytes				2048 bytes			
	GET	**%**	**PUT**	**%**	**GET**	**%**	**PUT**	**%**
Receive req.	197.8	3.8%	396.7	13.3%	849.3	9.7%	255361.9	98.2%
Handle req.	143.5	2.7%	185.7	6.2%	732.8	8.3%	660.4	0.3%
Send res.	4848.6	92.9%	2376.1	79.9%	7166.2	81.5%	3922.7	1.5%
Cleanup	31.5	0.6%	15.2	0.5%	41.2	0.5%	15.5	<0.1%

Table 4: Server-side request cycle breakdown for each abstract component of the cycle.

	32 bytes				2048 bytes			
	GET	**%**	**PUT**	**%**	**GET**	**%**	**PUT**	**%**
Prep req.	166.8	5.0%	481.1	13.1%	178.6	3.2%	256389.8	97.9%
Send req.	116.9	3.5%	59.1	1.6%	59.3	1.1%	61.4	<0.1%
Receive res.	3028.1	90.6%	3079.6	83.8%	5257.0	95.4%	5307.3	2.0%
Cleanup	30.7	0.9%	53.3	1.5%	17.2	0.3%	33.1	<0.1%

Table 5: Client-side request cycle breakdown for each abstract component of the cycle.

6 Related Work

In this section, various bodies of work pertaining to RDMA-based key-value stores are discussed. This work ranges from the usage of one-sided verbs to reduce efforts server-side, to maximizing throughput by using unreliable transport types.

6.1 A one-sided approach

One-sided verbs have the advantage of completely bypassing the remote CPU. This is generally used to lighten the processing workload server-side, and has shown various successful results.

Stuedi et al. modified Memcached to expose its memory so the client can read it [31]. It identifies several inefficiencies in Memcached such as data copies between kernel and worker threads, context switches for scheduling worker threads, and parsing requests. To address these, they use READs to directly read chunks of the server's data structure (for GETs) and prevent the necessity of worker threads for every request. The addresses of these chunks are initially requested over TCP, and written back by the server to the client using a WRITE. But afterward, the client has the address, and can directly read the chunk from the server's memory. For modifications (SETs), the requests are sent over TCP. With this approach, they created a system that outperforms Memcached by 20% while utilizing CPU more efficiently. Additionally, instead of hardware support, this was done entirely using soft-RDMA (a software stack for RDMA).

Dragojević et al. designed a system for managing objects in remote memory: FaRM [9]. That is, a remote computer can allocate, read, write, and free objects in a shared memory space. This does not just include the memory of a single computer, but rather the memory spaces of all computers connected to this system. FaRM uses READs for accessing remote data and WRITEs for message passing. Unlike general caching system, FaRM optionally writes the data to disk, ensuring persistence. With this system, they implemented a key-value store (using Hopscotch hashing) to achieve a maximum of 167 million lookups per second on a 20-machine cluster. This cluster used 40 Gbps RoCE as interconnect. They also tested FaRM on a larger system with Infiniband interconnects.

Mitchell et al. created Pilaf, one of the earlier systems to use READs to completely bypass server-side CPU usage for GET request [22]. Pilaf does use two-sided verbs (SEND/RECV) for PUT requests, as this simplifies necessary synchronization, which becomes unwieldy due to lock contention on the RNIC itself. This is followed by the server inserting the new pairs. For GET requests, clients READ the remote memory. First, it READs metadata of a special data structure to infer the pair's location and existence. Then, it READs the actual pair from that location (if it exists). To guarantee correctness, the server computes and stores multiple checksums with the key-value pair on insertion. When the client retrieves a pair, it recomputes the checksums and if incorrect, will reREAD the pair. Using this approach, Pilaf reached 1.3 million ops/sec with a single CPU, with low latencies, whilst being more CPU-efficient.

6.2 Defying tradition

Kalia et al. chose an unconventional design for their system [20, 18]. At that time, one-sided verbs were popular for RDMA-based key-value stores. However, they used a

combination of unreliable transports, WRITEs, and SENDs/RECVs to implement a very efficient system. Using UC WRITEs to place requests in server memory, and UD SENDs to return the response to the client. This is much faster for common case scenarios, as Infiniband is rather reliable. For any failures, they use application-level retransmissions. With this novel design, they achieved very high performance: up to 122 million ops/sec. Besides implementing a fast performing key-value store, they provide a number of recommendations for optimizing RDMA performance in a key-value store setting. They systematically show the performance improvements for these optimizations.

7 Conclusion

The evaluation of PERK on a cluster computer has shown promising results. Findings included significant speedup (>400%) and stable, low latency communication (up to 4.6 times faster) with a design that exclusively substitutes the usage of the network. Using RDMA decreased CPU utilization, while at the same time drastically increasing throughput compared to sockets-based Memcached. Only for larger payload sizes (> 1024 bytes) we saw problems regarding CPU usage and throughput. With only RC as transport type and no complicated optimizations, a resource-efficient key-value store can be implemented. Following the evaluation, we can answer the research questions as follows:

RQI. **What performance benefits does RDMA provide over sockets (using TCP)? How do different configurations (combinations of RDMA verbs) affect the performance of applications using it, and is there one 'best' configuration?** By designing a system that can easily switch between various verb configurations, and evaluating this system, we were able to ascertain that the performance benefits of RDMA over TCP are significant. Both in throughput and latency, RDMA vastly outperforms socket-based implementations, showing increases over 400% for configurations using one-sided verbs. Beyond that, it does so while consuming fewer CPU cycles. Although limitations in PERK's request parsing architecture hinder performance for larger payloads, this study shows many performance benefits of using RDMA. Given the evaluation's results, we see some clear winners when it comes to configurations. WRITE/WRITE is the first obvious choice, showing the best performance from every possible dimension. We argue, however, that if PERK's design sees optimizations that allow a sleep-based approach (even at the cost of context switching), that the WRITE/READ configuration would be an excellent choice for reducing processing on the server-side. Even when the client performs multiple READs, the performance is not dissimilar from the WRITE/WRITE configuration's performance.

RQII. **How does one properly evaluate the performance of a distributed system such as a distributed key-value store?** Benchmarking distributed system involves many complexities. Choosing the right micro-benchmarks can provide valuable insights into the performance of said system. For PERK, various commonly chosen metrics were combined in order to show a comprehensive view of performance tradeoffs: throughput, latency, and CPU utilization. Being able to perform these benchmarks on a distributed system such as DAS5 allowed this study to provide meaningful insights. However, even though this study showed various insights into PERK's performance, more details could be extracted by adding more layers, to give for an even more thorough understanding.

Because of the clear trend in performance improvements, it seems only logical for key-value stores in data centers or otherwise distributed environments to incorporate RDMA into the design as soon as possible. Beyond this, many other distributed applications could benefit immensely from the favorable properties of RDMA as well.

7.1 Limitations and future work

Because research time is limited, and gathering a deep understanding of a complex technology takes time, identifying limitations is a process that continually presents itself.

7.1.1 Workloads

Because the workload is very important to properly evaluate the performance of a key-value store, a relatively common one was chosen [5]. However, to better test the robustness of the system, a greater variety of workloads should be tested. For example, a modification-heavy workload (50/50 SET/GET), would have been interesting to test. Also, using a framework such as YCSB for workload generation would have been better, as this produces more accurate real-life workloads for cloud applications [9, 20, 18], but was avoided in the interest of time.

7.1.2 Environmental limitations

As briefly mentioned at the start of this section, the environment in which the application runs is influences the design and implementation of a system. When on the DAS5, you can reserve some set of resources for a time and have these resources available for you completely, creating an interesting image but providing resources that are extremely stable. The goal is for a system such as PERK to run in some data center environment, which generally requires constant sharing of resources between multiple applications. It would therefore be interesting to either add synthetic noise (run other applications), or test this application's performance in an existing data center environment.

7.1.3 Application in dire need of rest

To reduce some of the spinning in PERK, it would have been interesting to implement a type of system that actually polls. That is, sleeps in between it checking memory, this way, it will waste considerably less CPU. Either a system that uses some standard value, e.g. 1 or 2 microseconds. Or a system that once per N requests attempts to determine the average waiting times and dynamically changes sleeping times depending on the demand. Of course, adding this introduces complexity, and by itself requires processing on the CPU, but it would be interesting to explore this.

7.1.4 Many optimizations available

Widening this study with a full overview of all logical RC verb combinations, as well as all primitives available on UC and UD would be very interesting. RDMA networks tend to be reliable, thus making the need for hardware resends extremely rare. This makes it more interesting to use combinations of UC and UD such as in Herd [20, 18]. Although using RC provides programmers with a simpler task, because the application does not need to take care of resends, for very demanding workloads, the extra complexity would most likely be worth it.

Although the hash table implementation in PERK is sufficient for the scope of this project, several options could be explored. In fact, several options could be integrated into the design as options, just like the verbs currently are. This would allow customization

depending on the requirements of the environment. For example, for very short-lived data, this approach may be fine. But for data that requires more consistency and lives for longer periods of time, alternative implementations like Hopscotch hashing could be used [15].

Another interesting idea to explore would be to expose PERKs cuckoo hash table implementation. This has the benefit that when READs are used, a true zero-copy policy can be enforced for GET request. In turn, this will reduce CPU-usage server-side [31, 22, 9]. This does add an extra complication with regards to synchronization, as the clients are unable to find out whether the structure is being written to at a given time. Checksums can be used such that every client can recompute the checksum based on the retrieved pair, but this also causes overhead.

Beyond these three, Kalia et al. (implementers of Herd) [20, 18], included many other minor optimizations. Selective signaling can be used to limit the amount that the application is notified of a verbs completion, which causes internal I/O. Furthermore, batching requests can be very beneficial for throughput, as RDMA implements various optimizations for batched work requests.

Appendices

A Reproducibility

To reproduce results found in this thesis, this section provides various commands for running experiments both on DAS5 and provides simple guidelines for building a new infrastructure in other cluster systems.

A.1 General

The benchmark output generated by PERK itself can be turned on/off by adding/removing definitions to/from the project. This can be done either manually in the source code, or by passing special environment variables during the generation of the build files, allowing automation of the entire benchmarking infrastructure:

- *PERK_BM_LATENCY*, this causes the PERK clients to measure time before sending a request and after receiving a response. Afterward PERK prints the difference in microseconds.
- *PERK_BM_OPS_PER_SEC*, this causes PERK to time before and after the main request-cycle loop, print the total number of operations performed and print the number of operations performed per second.
- *PERK_BM_SERVER_EXIT*, a utility to make the server exit after the last client disconnects (used on the benchmarking infrastructure on DAS5).
- *PERK_OVERRIDE_VALSIZE=<VAL>*, a utility that allows programmatic modification of the maximum value size used by PERK.

For measuring CPU performance you will need to install the perf utility [4].

A.2 Reproducing experiments on the DAS5

Running the experiments on DAS5 has the advantage of the availability of the benchmark script, which relies on DAS5s reservation system: PRUN. For the simplest use, create folders for storing input files and outputting benchmark files:

```
$ mkdir /var/scratch/$USER/input
$ mkdir /var/scratch/$USER/bm
```

Listing 4: Prepare directories for storing input and output files.

Now generate a set of workloads. This can be tweaked at will, the following example generates 16 different 3,000,000-request workloads with a GET/SET ratio of 95:5 and stores it the input folder that was just created:

```
$ python3 ./util/gen_small_workload.py 3000000 0.95 16 /var/scratch/$USER/input
```

Listing 5: Generate 16 3,000,000 request workloads (95% GETs).

Then, to run the benchmarks with those files, for up to 16 clients (1, 2, 4, 8, 16 clients by default) you can simply run the benchmark script:

```
$ ./run_benchmark.sh -i /var/scratch/$USER/input/ -o /var/scratch/$USER/bm/ -n 3000000 -a -r 1
```

Listing 6: Running the automated benchmark script.

After the script has completed, the output folder (/var/scratch/$USER/bm/) should contain a large number of files of the following format:
's_type.cfg.reqcount_psize_getdis.ppn_n.rn.pid', where

- *s* - the side this file was the output from, cl/s for client-side/server-side.
- *type* - the type of benchmark done: bm_scale/bm_lat/bm_cpu for scalability, latency, and cpu-usage.
- *cfg* - configuration used: sd_sd/wr_sd/wrimm_sd/wr_wr/wr_rd/mcd.
- *reqcount* - the number of requests done.
- *psize* - the payload size.
- *getdis* - the get/set distribution (in % GETs).
- *ppn* - the number of processes per node.
- *n* - the number of nodes.
- *rn* - the run number, 'r[n]'.
- *pid* - the process id (not the Linux pid) within the range of 0-N, where N is the maximum number of clients the experiment was run with.

The files are named this way so another python script can automatically read them all and produce plots:

```
$ python3 util/read_benchmarks.py /var/scratch/$USER/bm/
```

Listing 7: Automatically fetch results from all files and generate plots, output tables in txt files.

A.3 Reproducing experiments in other environments

If you do not have access to the DAS5 but still want to reproduce these results, this is possible but will require some extra work. Generating workloads can be done in the same manner as on DAS5, by using Listing 5. But the run_benchmark.sh script cannot be used without PRUN. In order to still be able to reproduce results we will discuss the hypothetical scenario on a system that also uses nodes as compute endpoints, and has some means of executing commands remotely. Say the workload has been generated for 16 clients, with the same properties as the previous section, we now want to benchmark scalability, latency, and CPU usage, in that order. To do so we use the system's local scheduling program called run_on_node, to a specific node can be passed (so the IP is

predictable). The process for benchmarking can easily be automated, but for the purpose of demonstrating it clearly, we present a specific execution of the benchmarks. Here, we show how to benchmark scalability/latency/CPU usage for workloads of 3,000,000 requests (95% GETs), with a resulting total payload size of 32 bytes.

A.3.1 Running scalability benchmark manually

Before running the benchmark, make sure the right global definitions are added to the source code through environment variables, then use run_on_node to execute the server. If you cannot accurately predict the IP-address reliably, you will have to extract it in the command you run, this can be achieved using ifconfig and querying for a particular device.

```
$ # generate build files, set value size to 32−8(keysize)−1(space for null byte)
$ PERK_OVERRIDE_VALSIZE=23 PERK_BM_OPS_PER_SEC=1 PERK_BM_SERVER_EXIT=1 cmake .
$ # build the executables
$ make
$ # run an instance of the server using the WRITE/WRITE configuration on a node with ip address 10.149.0.42 and redirect output to a sensibly named file (the setup we are also expecting the client to run)
$ run_on_node −n 42 ./bin/perk_server −r wr_wr −a 10.149.0.42 −p 20838 > s_bm_scale.wr_wr.3000000_32_95.1.r1.0
$ # then run 1 client
$ run_on_node ./bin/perk_client −r wr_wr −a 10.149.0.42 −p 20838 −i /var/scratch/$USER/input/input_3000000_32_95_0.in −c 3000000 > cl_bm_scale.wr_wr.3000000_32_95.1_1.r1.0
$ # to run multiple clients you can start them in a bash loop, this stores all their outputs in separate files:
$ for i in {0..15}; do (run_on_node ./bin/perk_client −r wr_wr −a 10.149.0.57 −p 20838 −c 3000000 −i /var/scratch/$USER/input/input_3000000_32_95_${i}.in > cl_bm_scale.wr_wr.3000000_32_95.1_16.r1.${i} &); done
```

Listing 8: Running the scalability benchmark.

A.3.2 Running latency benchmark manually

The same applies to the latency benchmark. First the executables must be compiled with latency benchmarks enabled, then you can run and redirect output to files of your choosing.

```
$ # generate build files, set value size to 32−8(keysize)−1(space for null byte)
$ PERK_OVERRIDE_VALSIZE=23 PERK_BM_LATENCY=1 PERK_BM_SERVER_EXIT=1 cmake .
$ # build the executables
$ make
$ # run an instance of the server using the WRITE/WRITE configuration on a node with ip address 10.149.0.42 and redirect output to a sensibly named file (the setup we are also expecting the client to run)
$ run_on_node −n 42 ./bin/perk_server −r wr_wr −a 10.149.0.42 −p 20838 > s_bm_lat.wr_wr.3000000_32_95.1.r1.0
$ # then run 1 client
$ run_on_node ./bin/perk_client −r wr_wr −a 10.149.0.42 −p 20838 −i /var/scratch/$USER/input/input_3000000_32_95_0.in −c 3000000 > cl_bm_lat.wr_wr.3000000_32_95.1_1.r1.0
```

Listing 9: Running the latency benchmark.

A.3.3 Running CPU benchmark manually

For benchmarking CPU-usage, perf was used [4]. The official website has plenty of available examples, for this thesis only the command specified in Listing 10 was used.

```
$ # generate build files, dont add any other benchmarks (can add ops/sec still)
$ PERK_OVERRIDE_VALSIZE=23 PERK_BM_SERVER_EXIT=1 cmake .
$ # build the executables
$ make
$ # run the server on node 42, through perf
$ run_on_node -n 42 perf stat ./bin/perk_server -r wr_wr -a 10.149.0.42 -p 20838 > s_bm_cpu.wr_wr
    .3000000_32_95.1.r1.0
$ # then run 1 client
$ run_on_node ./bin/perk_client -r wr_wr -a 10.149.0.42 -p 20838 -i /var/scratch/$USER/input/
    input_3000000_32_95_0.in -c 3000000 > cl_bm_cpu.wr_wr.3000000_32_95.1_1.r1.0
```

Listing 10: Running the CPU benchmark by using perf.

After that, if the same naming convention was used for the output files, you can again use read_benchmarks.py to generate plots. The folder in which all these output files are stored can be passed, and the read_benchmarks.py will detect and read them.

References

[1] URL: https://www.project-voldemort.com/voldemort/design.html.

[2] URL: https://redis.io/topics/whos-using-redis.

[3] URL: https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[4] URL: https://www.brendangregg.com/perf.html.

[5] Berk Atikoglu et al. "Workload analysis of a large-scale key-value store". In: *ACM SIGMETRICS Performance Evaluation Review* 40.1 (June 2012), pp. 53–64. ISSN: 0163-5999. DOI: 10.1145/2318857.2254766. URL: http://memcached.org/.

[6] H. Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *IEEE Computer* 49.5 (2016), pp. 54–63. DOI: 10.1109/MC.2016.127.

[7] Youmin Chen, Youyou Lu, and Jiwu Shu. "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303968. URL: https://doi.org/10.1145/3302424.3303968.

[8] Martin Dietzfelbinger and Christoph Weidling. "Balanced allocation and dictionaries with tightly packed constant size bins". In: *Theoretical Computer Science* 380.1 (2007). Automata, Languages and Programming, pp. 47–68. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2007.02.054. URL: https://www.sciencedirect.com/science/article/pii/S0304397507001570.

[9] Aleksandar Dragojević et al. "FaRM: Fast Remote Memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87.

[10] "Dynamo". In: *ACM SIGOPS Operating Systems Review* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://dl.acm.org/doi/10.1145/1323293.1294281.

[11] Bin Fan, David G. Andersen, and Michael Kaminsky. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 371–384. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[12] R.R. Hamming. *Art of Doing Science and Engineering: Learning to Learn*. Taylor & Francis, 2003. ISBN: 9780203450710. URL: https://books.google.nl/books?id=49QuCOLIJLUC.

[13] Nathan Hanford et al. "A Survey of End-System Optimizations for High-Speed Networks". In: *ACM Comput. Surv.* 51.3 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3184899. URL: https://doi.org/10.1145/3184899.

[14] Gernot Heiser. 2019. URL: https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html.

[15] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. "Hopscotch Hashing". In: *Distributed Computing*. Ed. by Gadi Taubenfeld. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 350–364. ISBN: 978-3-540-87779-0.

[16] Alexandru Iosup et al. "The AtLarge Vision on the Design of Distributed Systems and Ecosystems". In: *CoRR* abs/1902.05416 (2019). arXiv: 1902.05416. URL: http://arxiv.org/abs/1902.05416.

[17] Robert Y. Al-Jaar. "Book Review: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling by Raj Jain (John Wiley & Sons 1991)". In: *SIGMETRICS Perform. Eval. Rev.* 19.2 (Sept. 1991), pp. 5–11. ISSN: 0163-5999. DOI: 10.1145/122564.1045495. URL: https://doi.org/10.1145/122564.1045495.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 437–450. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 185–201. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-Value Services". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 295–306. ISSN: 0146-4833. DOI: 10.1145/2740070.2626299. URL: https://doi.org/10.1145/2740070.2626299.

[21] Hyeontaek Lim et al. "SILT: A Memory-Efficient, High-Performance Key-Value Store". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 1–13. ISBN: 9781450309776. DOI: 10.1145/2043556.2043558. URL: https://doi.org/10.1145/2043556.2043558.

[22] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 103–114. ISBN: 978-1-931971-01-0. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell.

[23] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 385–398. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[24] John Ousterhout. "Always Measure One Level Deeper". In: *Commun. ACM* 61.7 (June 2018), pp. 74–83. ISSN: 0001-0782. DOI: 10.1145/3213770. URL: https://doi.org/10.1145/3213770.

[25] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *J. Algorithms* 51.2 (May 2004), pp. 122–144. ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.002. URL: https://doi.org/10.1016/j.jalgor.2003.12.002.

[26] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. URL: https://doi.org/10.2753/MIS0742-1222240302.

[27] R. Recio et al. *A Remote Direct Memory Access Protocol Specification*. RFC 5040. RFC Editor, Oct. 2007.

[28] David Reinsel, John Gantz, and John Rydning. *The Digitization of the World From Edge to Core*. Tech. rep. 2018.

[29] *RoCE in the Data Center*. Tech. rep. 2014. URL: https://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf.

[30] Stephen M. Rumble et al. "It's Time for Low Latency". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Napa, California: USENIX Association, 2011, p. 11.

[31] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. "Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 347–353. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi.

[32] Daniel Waddington et al. *A Scalable High-Performance In-Memory Key-Value Cache using a Microkernel-Based Design*. Jan. 2014. DOI: 10.13140/2.1.5084.6086.