Karlsruher Institut für Technologie

# Memory Management for Concurrent RDMA: A Design for a Key-Value Store

Diplomarbeit
von

cand. inform. Benjamin Behringer

an der Fakultät für Informatik

| | |
|---|---|
| Erstgutachter: | Prof. Dr. Frank Bellosa |
| Zweitgutachter: | Prof. Dr. Wolfgang Karl |
| Betreuender Mitarbeiter: | Dipl.-Inform. Marius Hillenbrand |

Bearbeitungszeit: 4. Dezember 2013–3. Juni 2014

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 3. Juni 2014

# Deutsche Zusammenfassung

Neuere Forschung nutzte die Vorteile von *remote direct memory access (RDMA)* um Leseoperationen auf key-value stores für leseintensive Betriebsbelastungen zu beschleunigen. Jedoch zeigen reale Anwendungsszenarien, dass key-value stores auch mit schreibintensiven Betriebsbelastungen konfrontiert sind. Wir präsentieren **Falafel**, ein Design für einen *in-memory key-value store*, welcher gleichzeitiges lock-freies Lesen und Schreiben über RDMA unterstützt. Wir beschreiben unsere Speicherverwaltung, welche es Clients erlaubt gleichzeitig, ohne gegenseitige Beeinflussung, und ohne zusätzliche Kommunikation mit dem Server oder anderen Clients, Lese- und Schreiboperationen auf *regions* im Speicher des Servers über RDMA durchzuführen. Wir verwenden einen serverseitigen *generational semi-space garbage collector* um gelöschte Einträge aus dem Serverspeicher zu entfernen. Wir präsentieren ein Hashtabellendesign basierend auf *open addressing mit linear probing*, welches lock-freie gleichzeitige Operationen der Clients über RDMA erlaubt, und unempfindlich gegenüber Client-Fehlern ist. Wir implementieren einen Falafel-Prototyp welcher 5.2 Mops/s `GET` und 1.9 Mops/s `SET` auf 40 Gbit/s InfiniBand Netzwerkkarten erreicht. Clients können `GET`-Operationen auf den Server mit einer Latenz von 4.5 µs, sowie `SET`-Operationen mit einer Latenz von 6.8 µs ausführen.

# Abstract

Recent work leveraged the advantages of *remote direct memory access (RDMA)* for read operations on key-value stores to improve the performance for read-heavy workloads. However, real-world application scenarios show that key-value stores also have to cope with write-heavy workloads. We present **Falafel**, a design for an *in-memory key-value store* that allows concurrent lock-free read and write operations by clients via RDMA. We show our memory management scheme that allows clients to concurrently read and write via RDMA without interference or communication with other clients or the server to *regions* in the server memory. We use a server-side *generational semi-space garbage collector* to remove deleted entries from the key-value store. We present a hash table design based on *open addressing with linear probing* that allows lock-free concurrent client operations via RDMA and is resilient to client failure. We implement a prototype of Falafel which reaches 5.2 Mops/s `GET`s and 1.9 Mops/s `SET`s on our 40 Gbit/s InfiniBand network cards. A client can perform `GET` operations on the server that take as low as 4.5 µs and `SET` operations that take as low as 6.8 µs.

# Contents

# Chapter 1

# Introduction

In-memory key-value stores are an essential building block in today's distributed services infrastructure. Large companies use them as query cache for databases or generic cache to enhance the performance of their applications [10,54].

Key-value stores such as memcached [21] and redis [62] use traditional socket-based [38] networking that employs *message passing* to communicate with clients. This widely used interface suffers from overhead through message processing [22,39]. Metreveli et. al. show that message processing can take up to 70 % of the overall CPU time for request processing by the key-value store [49]. Already in 1994, Thekkath et. al. introduced an alternative communication model, *remote direct memory access (RDMA)* [68]. RDMA allows a client to access data in the address space of the server process directly, bypassing the operating system and omitting processing through the server process. Thekkath et. al. and others show that the RDMA communication model provides better performance on various network architectures [8,70]

With RDMA-capable networks becoming more widespread, recent work focused on leveraging RDMA to speed up key-value store read operations. These designs make it possible for a client to read key-value pairs directly from the internal data structures in the address space of the server process via RDMA. The server operating system as well as the server process are not involved in the operation, therefore saving processing overhead. Though, write operations to the key-value store still use message-passing. RDMA-enabled key-value stores like *Pilaf* [53], *FaRM* [19], and a modified memcached [66] show that using RDMA can bring substantial performance benefits for read-heavy workloads.

However, key-value stores also have to cope with write-heavy workloads [3,7,54]. In this thesis we present **Falafel**, a design for an in-memory key-value store that allows client read and write operations via RDMA.

We expect to gain the same benefits when using write operations on the key-value store via RDMA as are possible for read operations via RDMA: The client does no longer have to transfer the control flow to the server during a request. It can perform all memory operations on its own. Therefore, it is not limited by the time the server takes to process the request and respond. This also means that the server is free from request processing, lowering CPU utilization on the server. We also expect to lower the memory copy overhead. A client can write data via RDMA directly to the final position in the server address space, in contrast to message passing, where the server moves the data from an intermediate buffer to the final position.

We face several challenges on our way to a key-value store design that allows concurrent RDMA read and write operations: We have to create a memory management scheme that allows clients to concurrently read and write data in the address space of the server process without interference. Conventional key-value stores use a central memory allocator in the server process to manage the available memory. We have to distribute the capabilities of this memory allocator among the clients. We have to take into account that the latency of RDMA operations is higher than the latency of CPU operations. We also have to design a hash table that allows concurrent read and write access from the clients. In case of a failing client, the hash table structure has to stay valid.

In this thesis we make the following contributions:

- We present **Falafel**, a design for a key-value store that allows client read and write operations via RDMA.
- We design a memory management scheme that allows clients to write to the server memory concurrently without interfering with other clients.
- We present a hash table design that allows lock-free concurrent client operations via RDMA.
- We implement a prototype of Falafel and evaluate it on an InfiniBand cluster.

The rest of this thesis is structured as follows: In Chapter 2, we give background information on key-value stores, RDMA, memory management, and hash tables. In Chapter 3 we present our design for the key-value store *Falafel*, and in Chapter 4 we discuss details of the implementation of the prototype of Falafel. We evaluate our prototype and present the results in Chapter 5. In Chapter 6 we list related work and explain how Falafel differs from those. We conclude in Chapter 7 and list directions of future work.

# Chapter 2

# Background

In this chapter we provide background information related to our design. We introduce the concept of a *key-value store* and give usage examples in Section 2.1. Then we provide background information about remote direct memory access (RDMA) in Section 2.2, memory management in Section 2.3, and hash tables in Section 2.4 with regard to their use in key-value stores.

## 2.1 Key-Value Stores

A *key-value store* is a service that offers the functionality of a dictionary over a network to multiple clients. The dictionary stores $(key, value)$ pairs, where each *key* may appear only once in the dictionary. The key-value store interface offers three functions: `SET`, `GET`, and `DELETE`. `SET` takes a key and a value as parameter and stores the $(key, value)$ pair in the key-value store's dictionary. `GET` takes a key as parameter and searches the key-value store for a $(key, value)$ pair with a matching key. If `GET` finds an appropriate key, it returns the corresponding value. Otherwise, it returns an error. `DELETE` also takes a key as parameter and searches the key-value store for a $(key, value)$ pair with a matching key. If `DELETE` finds an appropriate key, it deletes the corresponding $(key, value)$ pair from the dictionary. If it does not find a matching key, it returns an error.

In this work we focus on key-value stores that keep the dictionary in random access memory (RAM). These key-value stores may have *cache semantics* or *store semantics*.

*Memcached* [21] is a key-value store with *cache semantics*. When a client issues a `SET` and there is no more storage space left in the key-value

store, $(key, value)$ pairs get evicted from the dictionary. To decide which $(key, value)$ pairs to evict, Memcached uses a *least recently used (LRU) strategy.*

*Redis* [62] is a key-value store with *store semantics*. $(key, value)$ pairs never get deleted unless a client issues an explicit `DELETE`. When redis runs out of storage space, subsequent `SET`s fail.

Companies such as Facebook use key-value stores extensively to improve the performance of their software [54]. Atikoglu et. al. use real-world data from Facebook's memcached installations to perform a key-value store workload analysis [7]. To be able to evaluate different storage systems under the same workload, Cooper et. al. present a unified benchmark suite called *Yahoo! Cloud Serving Benchmark (YCSB)* [17].

## 2.2 RDMA

In 1994, Thekkath et. al. present RDMA as a new communication model [68]. They emphasize the separation of *data transfer* and *control transfer* in RDMA. In traditional network communication client and server use *message passing* to transfer data as well as control from one node to another. Here, data transfer and control transfer are bound to each other. When a server receives a message, he has to process it and send a reply. The client transfers control and data to the server. In contrast, RDMA allows one-sided operations from a client to a server that do not require server-side processing. Here, only data is transferred from the memory of the client process to the memory of the server process. The server process as well as the server operating system are not involved in the operation and free to perform other work.

The RDMA communication model by Thekkath et. al. features three operations that one node can perform directly on the memory of another node. With RDMA *read* and *write*, a node can read and write directly from the memory of another node. In addition, the model provides an atomic *compare-and-swap (CAS)* operation. The CAS operation takes a pointer as argument to a memory location in the remote memory and two values. It atomically compares the value at the pointer location with the first given value and swaps the value at the pointer location with the second given value, if the first given value and the value at the pointer location match. Otherwise, it returns an error code and the new value at the pointer location.

Thekkath et. al. demonstrate the performance benefits of RDMA compared to traditional networking using an implementation of a distributed file system. They also show that the use of RDMA on the client side can reduce the server load. Balaji et. al. confirm the findings of Thekkath et. al. regarding the benefits of RDMA in a set of different benchmarks [8]. They also present evidence that the advantages of RDMA are independent from the underlying network architecture by evaluating RDMA and traditional networking on different underlying network architectures.

With network hardware that has RDMA capabilities, such as *InfiniBand* [6], *Quadrics* [57], *Myrinet* [13], or software implementations that use already available Ethernet hardware such as *SoftiWARP* [70], RDMA usage has become more widespread.

Several projects leverage RDMA to build or improve network communication libraries. Liu et. al. use RDMA over InfiniBand to speed up the *Message Passing Interface (MPI)* in their implementation *MVAPICH* [30,47]. Other implementations of MPI, such as *MPICH*2 [46] or *OpenMPI* [65], also use RDMA over InfiniBand. Inspired by the communication primitives of the *L4 microkernel* [43], Kehne et. al. present *libRIPC*, a communication library using RDMA [40]. Jose et. al. present the *Unified Communication Runtime (UCR)* and use it to speed up *memcached* [37].

Apart from the use of UCR in memcached, other key-value stores use RDMA to improve performance [19,53,66]. They allow clients to directly read server data structures with RDMA. Though, write operations to the dictionary are still done by the server.

The proposed key-value cache *Nessie* also allows client RDMA write operations on the dictionary [67].

## 2.3 Memory Management

Key-value stores with traditional networking allocate memory for use in the server process. They use *explicit allocators* such as the *slab allocator* in memcached [14]. Explicit allocators keep track of free and occupied memory chunks. A client requests and frees memory explicitly with a call to the allocator. Wilson et. al. provide a survey on explicit memory allocation [75].

In our work we use memory management with *explicit regions* for the clients [25]. This kind of memory management has also been studied under the names *zones* [60], *groups* [34] or *arenas* [28]. Here, clients allocate

memory from a region consecutively and release the complete region and all objects inside when the region is full.

Our server process employs a *generational semi-space garbage collector*. This kind of allocator is a combination of a *semi-space garbage collector* and *generation scavenging*. Clients allocate memory consecutively from a smaller region called *nursery*. When the nursery is full, the clients retrieves a new region and hands the old region to the garbage collector. The garbage collector then collects all live data from the returned nursery and copies it to a larger region called *mature space*. When the mature space is full, the garbage collector collects all live data from it and copies the live data to a similar sized second mature space.

Lieberman and Hewitt designed the first *semi-space garbage collector* [42]. Ungar introduced *generation scavenging* [72]. He designed it mainly to reduce the pauses that the garbage collection imposes on the program execution. Buytaert et. al. study the behavior of a generational semi-space garbage collector and provide advice on when and how to collect [15]. Blackburn et. al. compare the performance of different garbage collection techniques and their generational counterparts [12]. They find generational semi-space garbage collection to be among the most competitive techniques.

There are other kinds of garbage collectors apart from the semi-space garbage collector. Cohen provides a survey on garbage collection methods [16], which has been updated by Wilson et. al. several years later [74].

There are allocators specially designed to work in a multithreaded environment such as *jemalloc* in redis [20]. Other examples of multithreaded allocators include *Hoard* [11] and *Stream Ow* [63]. Michael presents a *scalable lock-free dynamic memory allocator* [52].

Memory management for lock-free data structures is difficult because of the *ABA problem* that we describe in detail in Section 3.5.5 and the reclamation of deleted objects. The reclamation of deleted objects is difficult because in lock-free systems a client does not always know which objects are still in use. Michael gives an overview of the problem and identifies the shortcomings with reclamation of memory of deleted nodes in dynamic lock-free objects. He presents a method for *safe memory reclamation (SMR)* [50] that allows this reclamation. Herlihy et. al. use the same idea as Michael for their lock-free memory management with additions such as the immunity to the *Repeat Offenders Problem* [32].

## 2.4 Hash Tables

Hash tables are one possibility for implementing the dictionary of a key-value store. A hash table is an *array of buckets*. A *hash function* applied to a *key* delivers an *index* into the array, where *values* that correspond to the key are stored. Their advantage over other data structures to implement the dictionary of a key-value store, such as *linked lists* or *trees*, is their superior time complexity for a key lookup of $O(1)$. Knuth provides an introduction and overview on hash tables [41].

Memcached uses a hash table with *closed addressing* and *collision chains* as collision resolution [21]. Such a hash table is an array of list heads. To find a key in the hash table, the server uses a hash function to determine an index in the array and then searches the list beginning at the index position. The hash function used in memcached is the third version of *Jenkins lookup* [36]. Redis [62] also uses closed addressing with collision chains but the *MurmurHash2* [5] hash function.

RDMA offers only a limited set of operations, namely *read*, *write* and *compare-and-swap (CAS)*. As we want the clients to be able to concurrently work on the key-value store internal data structures via RDMA, we have to use a data structure that makes this possible. There are data structures that one can use to build hash tables that employ only the kinds of operations provided by RDMA. Valois presents a design for a linked list using CAS [73]. Harris improves on the design of Valois with the first CAS-based non-blocking linked-list set algorithm [29]. Michael also presents a lock-free hash table design that outperforms the design of Harris' [51]. Gao et. al. also use CAS to build a lock-free dynamic hash table with open addressing [23].

With Read Copy Update (RCU), McKenney et. al. implement a mechanism in the Linux kernel that allows lock-free read-only access to data structures that are modified concurrently [48]. Triplett et. al. use RCU to build a scalable concurrent hash table using a programming scheme called relativistic programming [69]. Apart from the specialized lock-free mechanisms listed above, there are also a number of universal lock-free methodologies [4, 9, 24, 31, 35, 64, 71]. Being universal, they suffer from overhead and are generally slower than specialized algorithms.

CAS is prone to the *ABA problem* as first described in the IBM System/370 Extended Architecture Principles of Operation [1]. There are techniques to avoid the ABA problem like tagging [26], hazard pointers [50], or the Pass the Buck algorithm [32]. Dechev et. al. present a

summary of ABA prevention schemes [18]. To relax the constraints that the size limited CAS imposes, Greenwald proposes a hardware double word CAS (DCAS) [27], which is not available for InfiniBand.

# Chapter 3

# Design

In this chapter we describe our design for the in-memory key-value store **Falafel**. We first explain why we choose an in-memory key-value store as demonstrator for our design in Section 3.1. Second, we give an overview of the design of Falafel in Section 3.2. We then describe the design of the memory management in Section 3.3 and the design of the hash table used in Falafel in Section 3.4. We explain the operations that the client and server perform on the key-value store in the subsequent Sections 3.5 and 3.6.

## 3.1 Application Scenario: In-Memory Key-Value Store

To evaluate our approach for memory management for concurrent *remote direct memory access (RDMA)*, we need an application that will benefit from the high speed and low latency of RDMA, yet requires concurrent access to its memory by multiple clients. Ideally, the application has no overhead to the data transfer and memory management mechanisms so we can study these without interference.

We chose an *in-memory key-value store* as demonstrator application. A key-value store stores $(key, value)$ pairs in an associative array and offers an interface to access the array. The value may be arbitrary data, in contrast to some other forms of data storage like *relational database management systems (RDBMSs)*, which define the structure and type of the data in a schema. The interface of a key-value store must provide operations to `SET`, `GET`, and `DELETE` $(key, value)$ pairs from the store. An

in-memory key-value store stores the data only in RAM. Therefore, the data will not persist when the server process ends.

An in-memory key-value store must feature three mechanisms to provide the functionality as described above: data transfer between server and client, memory management for the data on the server, and data lookup between keys and values. Like key-value stores with similar capabilities such as *memcached* [21] and *redis* [62], our key-value store shall allow concurrent access to the data from multiple clients.

## 3.2 Design Overview

Falafel stores each $(key, value)$ pair together in an own data structure called **container**. Entries in a *central hash table* point to containers of valid $(key, value)$ pairs. Figure 3.1 illustrates the data layout.

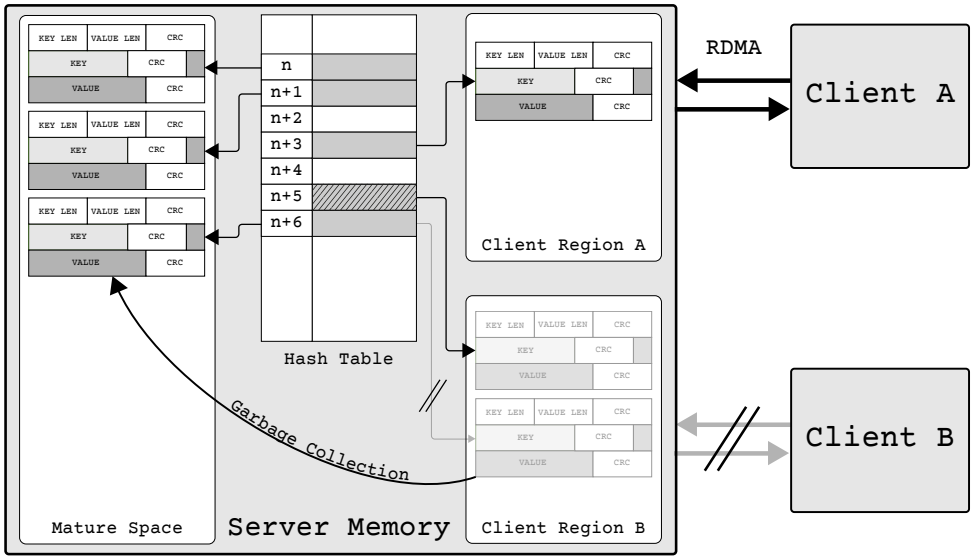



Figure 3.1: Design Overview: server memory with mature space, two client regions and two clients. Client A is connected and may read from the server memory and write to client region A via RDMA. Client B is disconnected. The server garbage collects its client region B, moves the data to the mature space, and updates the hash table entry $n + 6$. The hash table entry $n + 5$ is already deleted and will not be preserved.

A client reads a $(key, value)$ pair by inspecting the hash table whether he can find a pointer to a container. He determines the hash table entry

to inspect using the hash of the key he is looking for as an index. On success, the client reads the container with the $(key, value)$ pair. To write a $(key, value)$ pair to Falafel, a client first writes the container with key and value to the server memory and then atomically sets a pointer to it in the hash table. Again, the client determines the hash table entry to write to using the hash of the key he writes as an index. A client can delete a $(key, value)$ pair by atomically setting a flag in the hash table entry pointing to the container with the corresponding $(key, value)$ pair. Other clients will then consider the $(key, value)$ pair as deleted.

In Falafel, the clients perform all operations to the data and hash table using RDMA. The server memory with hash table and containers is RDMA read and write accessible by the clients. Clients can read concurrently from the hash table and the containers. As a consequence of limiting the client interaction with the server to RDMA operations, a client has no knowledge about the RDMA read and write operations of other clients. This does not matter for read operations. Though, when multiple clients concurrently write to the same location in the key-value store, data corruption may occur.

To keep the data in the key-value store consistent, each client writes containers with keys and values only to a memory region [25] that he can write to exclusively. To make a container visible to other clients, a client has to set a corresponding entry in the hash table. Therefore, clients still compete for write access to the central hash table. To avoid memory corruption, clients are only allowed to modify hash table entries with atomic RDMA compare-and-swap (CAS) operations.

A client acquires a memory region from the server and returns it when it is full or the client disconnects. When a client returns a memory region, there may be still valid entries with pointers to containers in the hash table so the server may not reuse it yet. Instead, the server process performs a garbage collection on the returned region. The server garbage collector searches the hash table for valid entries with pointers to containers in the region that the client returned. When the garbage collector finds such a hash table entry, he copies the corresponding container to a server owned region and alters the hash table entry to point to the new location. This type of garbage collector is called a generational semi-space garbage collector [42, 72]. After the garbage collection the server can reuse the returned region.

## 3.3 Memory Management

When clients write to the same locations in the server address space via RDMA, data corruption may occur. This is possible, as the server memory with hash table and containers is accessible to the clients via RDMA operations. In Falafel, it is the task of the memory management to prevent the clients from writing to overlapping memory locations concurrently.

To achieve a regional separation of the client write operations, the server provides every client with an own memory region. Clients request a region by sending a message to the server. The server then allocates a region of memory from the memory area that is accessible by the clients via RDMA and replies to the client with a message that contains the client region's base address and size. Clients can then write freely and exclusively to the acquired region. The server will give a region only to one client at a time.

During a `SET` operation on the key-value store a client writes a container with a $(key, value)$ pair to its region. The client allocates memory from its region consecutively without reusing memory once allocated. A client cannot know wether a container in his region is still in use without inspecting the hash table. Another client might insert a $(key, value)$ pair with the same key into the hash table, therefore overwriting the hash table entry that points to original container. Instead, the server will garbage collect the region when it is full. Gay and Aiken show that memory management with regions is competitive to explicit allocation and deallocation [25]. Memory management with regions has the additional benefit that the client does not have to communicate with the server or other clients to allocate a chunk of memory from its region. The client and server exchange messages only for the region allocation and deallocation. The region size can be varied through manual configuration, depending on the workload of the key-value store. The larger the region size, the fewer the number of region requests from the client and therefore the overhead for allocation.

From the client's perspective he releases all memory in a region when returning it to the server. For the client this has the semantics of memory management with regions. But there may still be valid entries in the hash table that point to data in the returned region. To save this data one has to garbage collect the returned region. The garbage collection has to inspect all data written to the region and copy the live data to a persistent region. The server performs the garbage collection, as he can access the memory

locally and therefore faster than the client who has to do this via RDMA. We describe the exact mechanism of garbage collection in Section 3.6.1. The server copies the live data in the region to a server-owned region. When this region is full, the server garbage collects it to a new empty server-owned region called *mature space*. This scheme of garbage collection is known as *generational semi-space garbage collection* [42,72].

## 3.4 Hash Table Design

Two of the challenges in building a client-server application over RDMA are the concurrent access through the clients and the high latency of RDMA operations compared to local memory accesses. We address the problem of concurrent access to containers through the memory management that we describe in Section 3.3. But the clients still compete for write access to the hash table entries. We therefore look for a mechanism to provide safe concurrent access to the hash table with a minimal amount of RDMA operations.

For an easier comparison, we first assume that there are no collisions on hash table entries when accessing the hash table from a single client. In this case, a `GET` operation takes at least two RDMA read operations: One read to get the position of the value from the hash table, and another read to get the value itself. A `SET` operation takes at least two RDMA write operations: One operation to write the value to the server memory, and another one to set the corresponding hash table entry. A `DELETE` operation then needs at least one RDMA write to mark the hash table entry as invalid.

When multiple clients access the hash table, they may attempt to write to the same hash table entry when writing a value for the same key. This means that the hash table must allow concurrent write access by multiple clients. This requires some form of mutual exclusion during writes, or clients may concurrently write to the same memory locations, causing data corruption. One possibility to achieve mutual exclusion for writes to the hash table is the usage of locks. Yet, locking and unlocking over RDMA yields several disadvantages: It increases the amount of RDMA operations needed for `SET`, `GET`, and `DELETE` operations and requires a timeout mechanism in case a client does not release the lock. In addition to the normal amount of RDMA operations per `SET`, `GET`, and `DELETE`, the acquire and release operations for the lock itself take at least one round

trip time each. They also can hardly be pipelined, as the client must know the results of the operations and act accordingly. In our hypothetical case, these operations alone increase the time needed for a set operation by at least 100 %, as the two writes of the set operation may be pipelined. For a delete operation, the lock and unlock operation increase the overall time needed by 200 %. Additionally, locking requires the presence of a timeout mechanism, as a client must be able to break the lock of another client that did not release the lock due to, for instance, an unexpected disconnect. This in turn requires timers that are synchronous over the network and precise enough to measure single RDMA round trip times. As of our set of available RDMA operations from Section 2.2, these are not available to us.

Another possibility is to design the hash table to be lock-free. This means that when multiple clients try to perform an operation on the hash table, at least one of them is guaranteed to make progress in a finite number of steps [31]. Using locks, this will not hold true when there is no timeout mechanism and a client that holds a lock disconnects unexpectedly, leaving the lock in an acquired state. We decided to design the hash table with the use of atomic *compare and swap (CAS)* operations. Previous work shows that this is feasible [23, 53, 58].

Because of the *ABA problem* that we describe in Section 3.5.5 we have to design the hash table in a way that an entry can be updated by a single CAS operation. We therefore employ a hash table with *open addressing and linear probing* [41]. In this hash table design the hash table is an array of entries. To insert a value, the client alters a hash table entry at a position he determines using a hash value of the key he wants to insert as an index. If the entry is already filled with the same key, the client overwrites the entry. If the entry is filled with another key, which means there is a hash collision, the client moves to the next array entry and tries to insert the key there. This scheme of *collision resolution* is called *linear probing*. A client retrieves a value by inspecting the hash table at the position the hash of the requested key indicates as before. If the hash table entry is empty, the client is done. If the hash table entry is filled, the client checks whether the key of the entry matches the key he is looking for. If this is the case, the client returns the corresponding value. If the keys do not match, the client moves to the next hash table entry and looks for the key there, similar to the collision resolution when setting a value.

To be able to change a hash table entry with a single CAS, we separate the key from the hash table, that now contains a pointer to the container

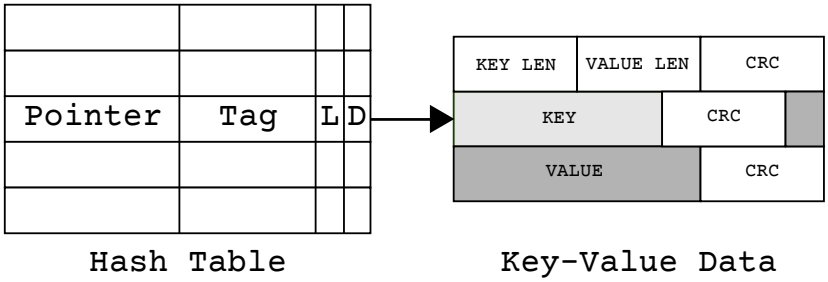



Figure 3.2: A hash table entry with data pointer, tag and Lock/Delete flags pointing to the container with key and value strings.

with the key and value, as the hash table entry would be too large otherwise. Figure 3.2 illustrates a hash table entry as well as the key and value data.

The container with key and value for a hash table entry consists of key, value, header data and cyclic redundancy check (CRC) values. The header contains the length of the key and value, so that a client knows how much memory to read from the server. The CRC values that protect the header, key and value are necessary because a client may read a memory chunk that does no longer contain valid data. He can detect this error through a CRC comparison and restart his operation. We describe the condition under which the error can occur in Section 3.6.1. Mitchell et. al. also use these *self-verifying data structures* in their key-value store *Pilaf* to prevent data corruption. In addition to the pointer to the container, a hash table entry contains a tag to prevent the ABA problem as well as a lock and deleted bit. We describe the use of the tag to prevent the ABA problem in Section 3.5.5. The server uses the lock bit during a garbage collection that we explain in Section 3.6.2. Clients use the deleted bit to mark an entry as invalid. We describe why the bit is necessary and how it is used in Section 3.5.4.

## 3.5 Client Hash Table Operations

The clients may perform three different operations on the key-value store. They can use a `SET` to store a key-value combination, a `GET` to read a value for a given key, and a `DELETE` to remove a key-value combination. Each of these operations uses a `FIND` to locate the hash table entry to perform the operation to, as shown by Knuth [41]. We will describe this basic algorithm

in Section 3.5.1, followed by the descriptions of SET in Section 3.5.2, GET in Section 3.5.3, and finally DELETE in Section 3.5.4.

### 3.5.1 Find

FIND takes a key string as a parameter and locates a matching entry in the hash table in the server memory. Once FIND locates such an entry it returns its value and position in the hash table to the calling function. The entry is either an occupied entry that points to a container with a copy of the given key and the corresponding value, or an empty entry. To locate the entry, FIND hashes the given key to an index of the hash table. FIND then loads the hash table entry with that index. If the entry contains a pointer to a container, it first loads the header of the container and performs a comparison of the calculated CRC of the read data and the read CRC value in the container header. If they do not match, FIND restarts from the beginning. The CRC mismatch indicates that the data at the position of the container has changed. Assuming that this is not the result of a RAM error, the CRC mismatch means that the container that FIND wanted to read has been overwritten. This implies that the region which FIND assumes the container to be in has been garbage collected as described in Section 3.6.1. The garbage collection might have altered the hash table in a way that the hash table entry to the container with the key that FIND is looking for is now located in another position in the hash table. Therefore, FIND has to restart the search from the beginning. When there is no CRC mismatch, FIND loads the key part and its CRC of the container and checks the CRC that protects the key. If it does not match the calculated CRC for the key, FIND restarts with the initially provided values for the same reasons as above. Then FIND compares the key with the given one. If the keys match, FIND is done. Otherwise the entry is part of a collision chain. As collisions are resolved by linear probing, FIND then increments the current index and examines the entry at this position. This continues until FIND either locates an entry with a matching key or an empty entry.

FIND treats hash table entries with the DELETE bit set like hash table entries without DELETE bit set. In contrast, when FIND encounters a hash table entry with the *locked* bit set, it restarts the search at the hash table entry that the hashed key points to. This is because the hash table entries with a locked bit set are subject to a cleanup by the server, and no client may perform operations on it to avoid data corruption. We explain the

hash table cleanup in Section 3.6.2.

### 3.5.2 Set

The SET operation writes a $(key, value)$ pair to the key-value store. It first invokes FIND as described in Section 3.5.1, to find either a hash table entry with a matching key or an empty entry. Regardless of the result, SET then writes the container with key, value, header and CRC values via RDMA to its memory region on the server. Afterwards, it tries to change the hash table entry on the server at the position returned by FIND to point to the location of the container with an RDMA CAS operation. SET increments the tag of the new hash table entry compared to the current one to prevent the ABA problem as described in Section 3.4. The deleted bit of the new hash table entry is not set. If SET writes the new hash table entry to an empty hash table position, it writes a key-value pair with a new key to the key-value store.

If SET overwrites an occupied hash table entry whose deleted bit is not set, SET overwrites another key-value pair with the same key. If SET overwrites an occupied hash table entry whose deleted bit is set, it overwrites a key-value pair with the same key that is already deleted. When the CAS fails, another client or the server modified the hash table entry since FIND has read it. In this case, SET starts again from the beginning, including the FIND. It is not sufficient to just overwrite the now current hash table entry as it might represent another key.

If a client disconnects during a SET, the hash table remains in a valid state. If the client disconnects before he issues the CAS, he may have written data to his region, but no other client will read it as there is no hash table entry for it. If a client disconnects during the CAS, the CAS either succeeds and the hash table entry points to the container that the client has written before, or the CAS does not succeed and the hash table entry remains untouched. In both cases the hash table is in a valid state.

### 3.5.3 Get

The GET operation retrieves the value for a given key from the key-value store. First, GET invokes FIND. If the FIND operation for the given key returns an empty hash table entry or an occupied hash table entry whose deleted bit is set, GET is done and returns. Otherwise, GET loads the value data and its CRC from the container in the server memory that the hash

table entry returned by FIND points to. When the read CRC matches the calculated CRC for the read value, GET returns the value string. If the CRC values do not match, GET restarts for the same reasons as described in Section 3.5.1.

### 3.5.4 Delete

The DELETE operation marks a hash table entry on the server as invalid if the entry represents a $(key, value)$ pair for a given key. To locate such an entry if present, DELETE calls FIND. When find does not return an occupied hash table entry or if it returns a hash table entry whose deleted bit is already set, DELETE is done and returns. If FIND returns an occupied entry whose deleted bit is not set, it tries to swap the hash table entry on the server with a copy of the old one whose deleted bit is set via an RDMA CAS. If the CAS fails, DELETE starts again including the find operation for the same reason as the SET operation that we describe in Section 3.5.2. DELETE cannot erase the hash table entry that it wants to delete by, for instance, overwriting it with zeros. The entry may be part of a collision chain for a different key. When DELETE sets it to zero, the FIND algorithm as described in Section 3.5.1 will terminate upon this entry, discarding all collision chain entries that may lie behind the zeroed entry. This means that DELETE would delete more than the one desired entry from the key-value store while leaving unused but live entries in the hash table.

If a client disconnects during a DELETE, the hash table remains in a valid state. If a client disconnects during the CAS, the CAS either succeeds and the hash table entry is marked as deleted, or the CAS does not succeed and the hash table entry remains untouched. In both cases the hash table is in a valid state.

### 3.5.5 Lock-Freedom and Concurrency

The SET, GET, and DELETE operations are *lock-free* to each other. Lock-freedom means that if multiple clients compete for a resource, one client will always make progress in finite time [31]. Assuming that the underlying RDMA operations will always finish in finite time, the only point in the algorithms where the clients may fall into a loop is the CAS operation at the end of SET and DELETE, where the clients compete for the values of the hash table. GET does not have a CAS operation. For SET and DELETE

we can see that if a client repeats an operation because of a failed CAS, another client or the server must have changed the underlying value in the meantime. A changed hash table entry means another client or the server has finished its CAS and therefore its operation. That one of the clients or the server finished his operation satisfies the condition for lock-freedom as described by Herlihy [31].

When using CAS, one has to deal with the *ABA problem* [1]. The ABA problem can occur when multiple clients perform a CAS operation on the same memory location. Consider client 1 reading the memory location as value A, which he wants to use in a CAS. Then client 2 compare and swaps the same location to B, and in another CAS back to value A. When client 1 now performs its CAS, it will succeed though the value has changed twice in the meantime. In our scenario the hash table entries contain pointers to containers with $(key, value)$ pairs. When a client alters a hash table entry that changed twice since the client read the entry, hey may therefore operate on a hash table entry that points to a container in the same memory location as the original container, but contains a different key. This may happen because the region that holds the original container could be garbage collected in the meantime, and another client could refill the then empty region.

There are techniques to avoid the ABA problem like tagging [26], hazard pointers [50] or the Pass the Buck algorithm [32]. For our design we choose the tagging technique to avoid the ABA problem because in contrast to hazard pointers or the Pass the Buck algorithm, it requires no additional RDMA operations to guard a hash table entry.

When a client reads container data from the server memory, the container data may be corrupted. This may happen when a client reads a pointer to a container from the hash table and then pauses. In the meantime, the region that holds the container gets garbage collected and another client reuses it afterwards. The other client fills it with containers. If the original client then reads from the region, he may read a different container or nonsense data as the pointer he read may not point to the header of a container.

To prevent a client from using corrupted data, the container that holds key and value is *self-verifying*, and therefore protected by three CRC values. Figure 3.2 shows the data structure. The first CRC value guards the header data which consists of key and value length. The second CRC value guards the key, and the third CRC value guards key and value.

As described in Section 3.5.1 and Section 3.5.3, a CRC mismatch in

data read from the server memory results in a restart of the operation that encounters the CRC mismatch. This may lead to a client constantly restarting operations because other clients or the server change the data the client wants to read before he can finish the read. This behavior is similar to `SET` where clients compete for the hash table entries with CAS operations as it also implies that other clients finished their operations in the meantime.

## 3.6 Server Hash Table Operations

While the clients perform the `SET`, `GET`, and `DELETE` operations without invoking the server process, there are two tasks the server has to carry out. He has to reply to region allocation and deallocation requests as described in Section 3.3 and he has to perform the garbage collection of released regions. The garbage collection comprises the actual garbage collection that we describe in Section 3.6.1 and the hash table cleanup that we describe in Section 3.6.2.

### 3.6.1 Garbage Collection

When a client releases a region and returns the ownership back to the server, the server garbage collects all live data from the region. The data in the region may consist of live data or garbage – containers that are no longer referenced through the hash table. Each live container has a hash table entry pointing to it, which is not marked as deleted. When there is no hash table entry with a pointer to a certain container, this container is considered garbage. To collect the live data the server walks the hash table and inspects the pointers in the hash table entries. If the hash table entry points to the region on which the server performs a garbage collection, the server copies the referenced container to the server region. The server has to collect all referenced containers, even when the hash table entry that points to a container carries the deleted flag. If he does not collect such a container, a client that walks a collision chain containing the entry will read an invalid pointer. As the clients that walk a collision chain only inspect the keys in containers, the server only copies the header and key portion of a container if the corresponding hash table entry carries the deleted flag.

In contrast to the clients, the server can employ normal memory operations, whereas the clients would have to use an RDMA operation. After the server finished copying the container he performs a CAS on the hash table entry that pointed to the original container, updates its pointer and increments its tag. When the CAS fails, another client has already altered the hash table entry which means it now points to a container in another region. The container in the region that the server garbage collects has not to be collected any more because it is outdated. The server then continues to walk the hash table until he has inspected all entries.

The garbage collection runs in parallel to answering requests for allocation and deallocation of regions. It does not block the flow of client operations.

The type of garbage collector we employ is called a *generational semi-space garbage collector* [42, 72]. The work the garbage collector has to do on a garbage collection is proportional to the amount of live data in the region to collect, as the garbage collector will only copy the live data of a region to another region. The garbage collector does not check data that is not referenced by the hash table. The garbage collector does not have to operate on once collected data unless the region in which the data lies is full and has to be garbage collected. As Buytaert et. al. argue, to reduce the work that a garbage collector has to do, one can increase the size of the regions and mature space [15]. Larger regions mean that the time between two garbage collection grows, while the amount of live data the garbage collector has to collect stays the same. Though, increasing region or mature space size means that the number of available regions and therefore the number of possible clients shrinks. Therefore, the size of regions and mature space has to be adjusted to the number of clients, the size of available memory, and the particular workload.

### 3.6.2 Hash Table Cleanup

When a client deletes a key he cannot just empty the corresponding hash table entry as the entry may be part of a collision chain and clients would not be able to find keys behind the deleted key in the collision chain afterwards. Hash table entries to deleted and not subsequently overwritten keys may therefore over time pollute the hash table with unusable entries [41]. To clean the hash table from deleted entries our server performs a periodic cleanup. The cleanup starts when the number of deleted entries in the hash table reaches a certain threshold. The server

walks the hash table and looks for deleted entries. When he finds a deleted entry he starts to set the lock bit of the deleted entry and all subsequent entries of the collision chain including the first free one after the collision chain with a CAS. The clients will not alter the hash table entries with the lock bit set, so the server can operate on them exclusively. The server then clears the locked part of the hash table and rehashes all live entries inside and inserts them again into the locked part of the hash table. When the clears the hash table, he increments the tags of all locked hash table entries to avoid the ABA problem. When the server finishes the rehashing, there are no deleted entries left in the locked part of the hash table. Then, the server unlocks the locked hash table entries in reverse order of locking.

# Chapter 4

# Implementation

In this chapter we discuss parts of the implementation of the Falafel design. We implemented Falafel for the InfiniBand network. We first provide a performance evaluation of our InfiniBand hardware in Section 4.1. Then, we list the limitations that InfiniBand introduces and how we worked around them in Section 4.2. In Section 4.3, we describe the optimizations that we added to the Falafel prototype that are specific to our implementation and hardware. We also present a list of features that we'd like to see implemented in hardware in Section 4.4.

## 4.1 InfiniBand Characteristics

To set a context for implementation of the Falafel prototype, we run a series of benchmarks on our InfiniBand setup. We explain our hardware in detail in Section 5.1.

We first test the achievable RDMA *latency* of our setup. Then we measure the achievable *throughput* and *bandwidth* and show how the two are related.

We test the latency of RDMA read and write operations from 4 B to 1024 KiB. We run a server process on one node, and a client process on a different node. The client process performs consecutive RDMA operations on the same position in the address space of the server process. The client waits for every operation to finish before issuing the next one. For every kind of operation and size we perform 1 Mops (1 000 000 operations) and take the median of the measured values. Figure 4.1 shows the test results. An RDMA read operation of 4 B takes 2.8 µs and an RDMA write operation of the same size takes 1.8 µs. When we increase the amount

of transferred data on both operations, the latency increases linearly for both operations up to 6.3 µs for an RDMA read of 4 KiB and 5 µs for an RDMA write of the same size. The gradient of the latency increase for larger transfer sizes lowers for both operation types at 4 KiB transfer size. We believe that this change in gradient is the result of the *maximum transfer unit (MTU)* that is set to 4 KiB. In InfiniBand, the MTU is the maximum payload size a single packet can contain. The gradient for both operation types then stays constant up to the end of our measurement range of 1024 KiB. The measured latency increases faster for RDMA reads than for RDMA writes. At 1024 KiB an RDMA read is 13 µs slower than an RDMA write. The difference at 4 KiB is 1.3 µs. The size RDMA CAS operations is fixed to 4 B. The latency for such an operation is 3 µs.

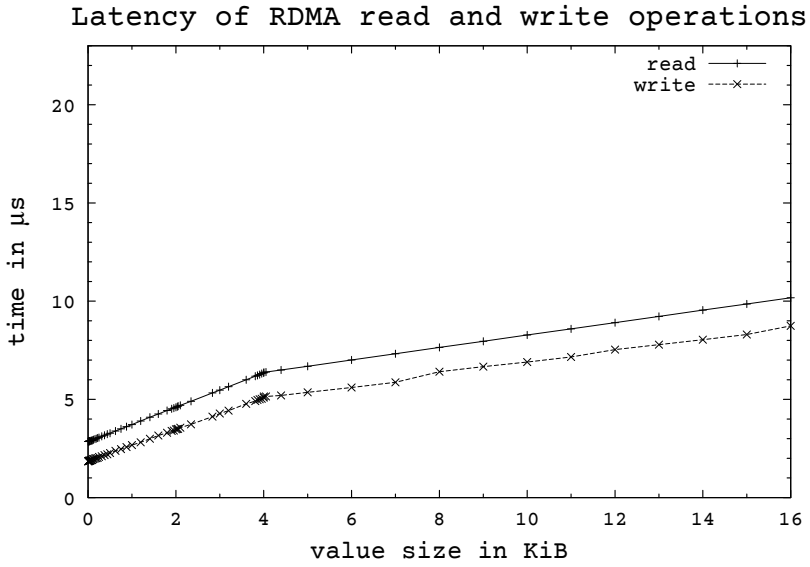



Figure 4.1: RDMA read and write latency between two nodes for 4 B to 16 KiB.

To test the bandwidth and throughput of one node in our InfiniBand setup, we run four server processes on one host node and connect to them with four client processes from one other client node each. We measure the achieved bandwidth and throughput for each client node and coalesce them to the total value. Figure 4.2 shows the results of our benchmark.

For a 4 B value size the InfiniBand NIC on the host can sustain 29 Mops/s on RDMA writes. The bandwidth in this case is 111 MiB/s. For RDMA reads the NIC on the host reaches 15.9 Mops/s for a 4 B value size at a

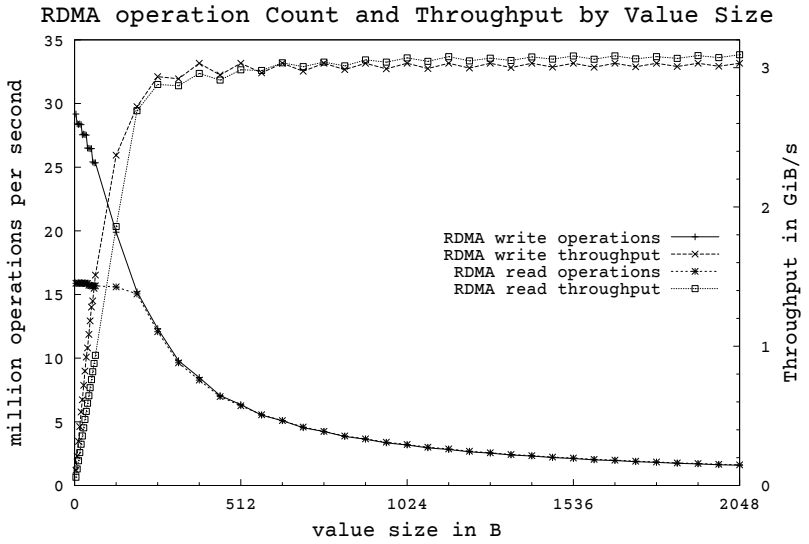



Figure 4.2: RDMA read and write throughput and bandwidth for 4 B to 2 KiB.

bandwidth of 60.6 MiB/s. When the value size increases the operations throughput decreases while at the same time the bandwidth increases. Throughput and bandwidth show properties of *logistic growth*: Their gradient is large close to zero while showing asymptotic behavior for large value sizes. The maximum achieved bandwidth is 3.1 GiB for RDMA reads as well as RDMA writes.

On our InfiniBand NIC, we achieve a maximum of 2.5 Mops/s of compare and swap (CAS) operations.

## 4.2 Limitations

During the implementation of our design we faced two limitations regarding the atomic operations provided by InfiniBand that are induced by the hardware: The size of an atomic CAS is limited to 64 bit, and the *InfiniBand CAS* instruction is not atomic to the *CPU CAS* instruction. In this section we discuss how we worked around these limitations and how they influence different parts of the *Falafel* prototype. We first discuss the effects for the *hash table entry* in Section 4.2.1 and the way we address memory in Section 4.2.2. Then we explain why a client has to send a periodic *heartbeat signal* to the server in Section 4.2.3. Finally, we describe

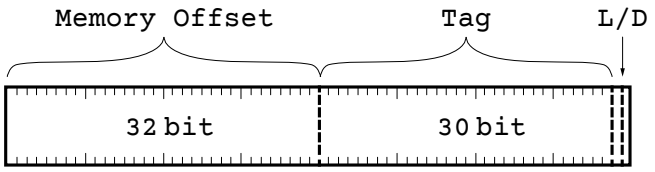



Figure 4.3: A 64-bit hash table entry. The first 32 bits contain the address to the key-value data structure. The following 30 bits contain a tag, the remaining 2 bits form the lock and the deleted bit.

how the server process uses CAS in Section 4.2.4.

### 4.2.1 Hash Table Entry Field Size

InfiniBand supports 64-bit CAS operations. Following our design Section 3.4 we have to fit a memory pointer, a tag, a lock flag and a deleted flag into the 64 bits so that the client can alter a hash table entry with a single CAS. The size of a pointer on x86_64 architecture is already 64 bit. We therefore have to compromise on the size of the fields to fit them all into the available 64 bit. Therefore, we limit the size of the memory pointer to 32 bit and the size of the tag to 30 bit. The lock flag and the deleted flag take one bit each. Figure 4.3 shows the resulting layout of a hash table entry.

### 4.2.2 Addressable Memory

The 32 bit pointer in the hash table entry cannot address an arbitrary memory location in the 64 bit-wide server address space because the pointer size is too small. To be able to address the container data in the server address space, we treat the former memory pointer as a *memory offset* to a memory chunk in the server address space in which all container data and the hash table are located. To make this work, the server allocates a contiguous chunk of memory and keeps all data structures that the clients need to access in this chunk.

All InfiniBand RDMA operations require the absolute addresses of the containers in the server address space. The client therefore calculates the absolute addresses of the containers in the server address space using the memory offset in the hash table entry and a base pointer that points to the beginning of the memory chunk in the server address space. The server

sends a pointer to the beginning of this memory chunk to the client when the client establishes a connection to the server.

Limiting the size of the pointer offset also limits the amount of addressable memory. With a 32 bit offset size we can byte-address 4 GiB of memory. To overcome this limitation, we choose to address larger chunks of bytes with the memory offset instead of single bytes. For chunks of 8 B, the memory offset in our Falafel prototype can address 32 GiB of memory. For chunks of 64 B which is the space overhead of a key-value container the memory offset can address 256 GiB.

### 4.2.3 Client Heartbeat

We need the 30-bit tag in the hash table entry to avoid the ABA problem as we describe in Section 3.5.5. The 30-bit tag can hold $2^{30}$ different values. As we see in Section 5.4 the maximum throughput for `SET` operations is 1.93 Mops/s. Assuming all clients write the same key, the tag would overflow after 536.8 s, opening the possibility for a client to experience the *ABA problem*. We explain the ABA problem in Section 3.5.5.

To avoid the ABA problem we require every client to send a heartbeat message to the server every 500 s that assures that the client is not performing an operation at the moment. As a client always loads the hash table entry he will operate on at the beginning of an operation, he may only operate on a hash table entry that is under 500 s old. Therefore he will not experience a tag overflow to the same value that he read at the beginning of his operation when other clients change the hash table entry in the meantime. So, the ABA problem cannot occur.

The server has a timeout value for each client that resets on every received heartbeat. If a client does not send the heartbeat message in the 500 s window and the timeout on the server fires, the server closes the connection to the client. This ends all open operations on the client, also preventing the ABA problem. The 500 s window fits our hardware. On different platforms with different throughput one must adjust the time window.

### 4.2.4 Compare-And-Swap on the Server

When the server process performs a garbage collection he has to alter the pointers in hash table entries of live data to point to the new locations of the containers that the server copied to the mature space before. The

update of a hash table entry has to be atomic to the clients accessing the hash table over InfiniBand. Otherwise, a client may read partially written data from the hash table. Our server CPUs feature a cmpxchg instruction which makes it possible to perform CAS operations that are atomic on the x86_64 server CPU.

On our combination of InfiniBand NIC, CPU and Mainboard, the InfiniBand CAS instruction is not atomic to the CPU CAS instruction. Therefore the server process has to use the InfiniBand CAS operation to alter hash table entries. The latency of an InfiniBand CAS operation is higher than that for a CPU CAS instruction. To hide the higher latency of the InfiniBand CAS, instead of waiting for the result of every CAS operation, server process issues multiple InfiniBand CAS instructions after another before checking on the result of the first operation. As a downside, the pipelining of CAS operations may lead to more garbage in the mature space. A failed CAS during the garbage collection indicates that a client altered a hash table entry during the garbage collection of this hash table entry and the garbage collector does no longer have to collect the container for this hash table entry. The garbage collector therefore might reuse the space of the container that he needlessly copied to the mature space before he issued the CAS, but that only works when the garbage collector waits for the result of every CAS operation. If we start the collection of another entry and the CAS of this subsequent garbage collection succeeds, the space after the needlessly copied container is occupied. Therefore the garbage collector could only reuse the space of the needlessly copied container instead of an arbitrarily sized space beginning at the start address of the needlessly copied container. The garbage collector would also have to remember the position and length of the containers for which he does not know the result of the CAS yet. We choose the benefit of the pipelined CAS instructions over the ability to easily reuse the space of needlessly copied containers and omit the reuse of the space of needlessly copied containers.

The InfiniBand standard optionally allows for InfiniBand CAS operations that are atomic to CPU CAS operations [6]. Hardware other than ours may support this feature so the garbage collector could use the CPU CAS instruction which is faster than the InfiniBand CAS instruction.

## 4.3 Optimizations

We optimize our implementation of *Falafel* in several ways which we discuss in this section: We describe the use of pipelining for RDMA operations to hide the RDMA latency in Section 4.3.1, the use of special CPU cyclic redundancy check (CRC) instructions in Section 4.3.2 and several kinds of readahead to improve performance in Section 4.3.3. These optimizations are specific to our hardware and may not be beneficial or possible on other hardware.

### 4.3.1 RDMA Pipelining

When RDMA operations do not depend on the results of the previous RDMA operations, we issue all of these RDMA operations in a batch. Thereby we can partially hide the latency of the RDMA operations following the first one. This pipelining approach is limited by the need to collect the results of the RDMA operations in the same order as they were issued, as the InfiniBand specification dictates.

We also eagerly renew client memory regions. A client requests a new region from the server when the filling level of the client region reaches a certain threshold. This way the client does not have to wait for this operation to finish when his current region is full.

### 4.3.2 CRC Hardware Instructions

Each time a client writes or loads a container or part of a container to or from the key-value store he has to calculate the CRC values of header, key and value. During our tests we found that the calculation of the CRC value in software made up a significant share of the client processing time, limiting the client's throughput. We therefore use the CRC32 hardware instruction of our Intel CPU for this calculation, which eliminates the throughput limitation because of the CRC calculation.

### 4.3.3 Readahead

To retrieve the key or value to a hash table entry a client reads the corresponding container that the hash table entry points to. At this stage, the client does not know about the length of key or value, and therefore about the length of the container. Without readahead, the client then first loads

the fixed size header which contains the length of key and value, and then the rest of the container. With readahead, the client reads more than the length of the container speculating that this readahead will be sufficient to load the container, or just the key at the beginning of the container, which is sufficient for some operations like `DELETE`.

When the readahead is sufficient, the client can omit the second RDMA read operation to load the remaining container data from the server memory. Our analysis of RDMA latency and bandwidth in Section 4.1 shows that the additional transfer time for a small increase in transfer size is shorter than the time an additional RDMA operation takes.

When the client looks for a key in the hash table, he first loads a chunk of the hash table from the server memory starting at the position that the hash of the key indicates. Instead of a single hash table entry, the client loads a chunk of entries, which saves subsequent load operations on the following hash table entries if the client encounters a collision chain. If there is a full hash table entry at the hash table index that the hash of the key provides, the client inspects the subsequent hash table entries before loading the first container. If the hash table entries form a collision chain, the client not only issues an RDMA load for the container that corresponds to the first hash table entry, but for containers that correspond to subsequent hash table entries, too. This way the client does not have to wait a full RDMA load round trip time until he can inspect a subsequent key.

We base this *pre-loading of containers* on the average number of probes required until the client finds a matching key as described by Knuth [41]. He predicts the average number of probes to find a matching key as:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

Here, $\alpha$ is the load factor, a value ranging from 0 to 1 indicating how much of the available hash table entries are full. This prediction assumes that the key can be found in the hash table. We optimistically assume that this is the case for every key the client is looking for. As the client can only load integral numbers of keys, we round up $C_n$ to:

$$N_n = \left\lceil \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \right\rceil$$

## 4.4 Proposed Hardware Features

In this section we list the features that would either improve the implementation of *Falafel* or enable us to choose a better approach.

The compromises we have to make to fit the hash table entry fields into the 64-bits CAS size of InfiniBand could be resolved with a *double-word* 128 bit *CAS (DCAS).* This would make room for a complete pointer rather than just an offset. We would also be able to fit in a tag large enough so that we could omit the heartbeat mechanism. If the two words of the DCAS could be *addressed separately*, it would be easily possible to use other hash table designs like closed addressing with collision chains that has better performance characteristics on higher load factors. Greenwald presents a linked list algorithm depending on DCAS [27].

Atomic InfiniBand instructions that are also atomic to the CPU and other InfiniBand NICs in one server would allow to incorporate the host CPU more into the data processing by using the CAS instruction of the host CPU and increase the bandwith and throughput to other nodes by adding more InfiniBand NICs. An InfiniBand NIC that is capable to saturate the network for small packet sizes would also increase the throughput, as small packet sizes are common for our usage scenario.

# Chapter 5

# Evaluation

To explore the performance characteristics of our design, we evaluate the *Falafel prototype* on an InfiniBand cluster. We specify our experimental setup in Section 5.1. We present evidence for variations in the InfiniBand RDMA operation latencies in Section 5.2 and show that they are important for our evaluation context. In Section 5.3, we examine whether Falafel conforms to the theoretical costs per operation for our hash table design. We perform a set of microbenchmarks on the Falafel prototype. In Section 5.4, we present our findings regarding the achievable throughput of the Falafel prototype. The benchmarks in Section 5.5 are focused on latency.

## 5.1 Experimental Setup

We run our tests on an *InfiniBand cluster* with up to 6 nodes. Five of the nodes are equipped with an Intel Xeon E3−1230 CPU, one with a Intel Xeon E3−1220CPU. All of them have access to 16 GiB of RAM and a Mellanox ConnectX-3 InfiniBand network interface card (NIC) (MCX353A−QCBT Rev. A2 with firmware 2.10.2280). The InfiniBand NICs are connected through a single Mellanox 8-port InfiniBand switch (MIS5022Q−1BFR Rev. A5). The maximum transfer unit (MTU) of the InfiniBand NICs is set to 4096 B. The NICs are connected via PCIe 2 to a Supermicro X9SCM−F motherboard. The nodes run 64-bit CentOS 6.5 with Linux kernel 2.6.32−431.5.1.el6.x86_64 [2]. On each node, we deactivated the dynamic in-kernel CPU frequency scaling so that the nodes with Intel Xeon E3−1230 CPUs constantly run at a CPU frequency of 3.2 GHz and the node with Intel Xeon E3−1220 CPUs at a CPU frequency of 3.1 GHz, respectively. We also deactivated Intel Turbo Boost on all nodes, which

would allow the CPUs to run above their specified maximum frequency depending on power, current and temperature limits. Hyperthreading, which provides two hardware threads per physical core is enabled on all nodes with an Intel Xeon E3−1230 CPU, while the Intel Xeon E3−1220 CPU is not capable of Hyperthreading.

On each test run we use one of the nodes to run the Falafel server process and the remaining nodes to run the required number of client processes. We start at most one process per hardware thread. During our tests, we allow no other processes to run on the nodes apart from the Falafel processes and the system processes. In all tests we configure the Falafel server to have access to a memory chunk of 2 GiB and set the hash table size to $2^{20}$ entries. The initial mature space size is 256 MiB. All clients run with adaptive pre-lookup that we explain in Section 4.3.3.

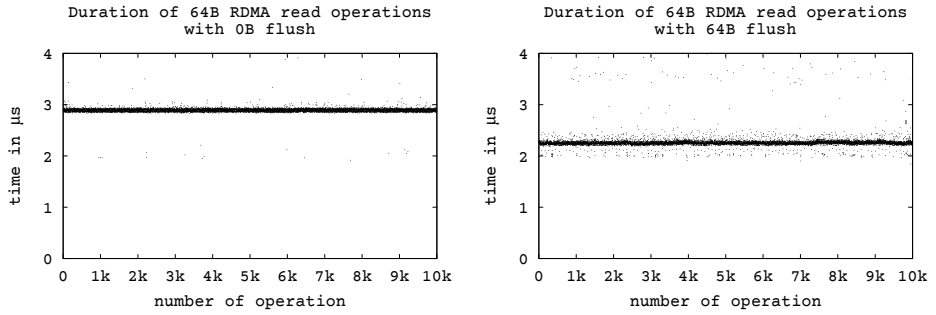
Throughout the tests we focus on `SET` and `GET` operations. `DELETE` operations are the same as `SET` operations without the RDMA write for the new container. We will see in Section 5.5 that we can approximate a `DELETE` operation with a `SET` operation with a very small key and value, because we can hide the latency of this RDMA write through pipelining.

## 5.2 InfiniBand Noise

During our tests we experienced several different *latency variations* in the InfiniBand RDMA operations. In this section we want to give two examples of the latency variations that we observed and show how to provoke them with a series of RDMA, host, and messaging operations. We can not explain the source of these latency variations yet. We assume that the variations are the result of NIC-internal caching and communication overhead between NIC and CPU. We consider explaining the source of the observed latency variations as out of scope for this thesis. These results are relevant to our evaluation as combinations of RDMA, messaging, and host operations are common during the Falafel operations. We use the RDMA read operations as an example, as the majority of the RDMA operations in the Falafel operations are RDMA reads.

We use the same benchmark as in Section 4.1: A client performs 10 000 consecutive 64 B RDMA read operations on the same memory location in the address space of a server process on a different node. Figure 5.1a shows the raw results of the test run.

We see that the latency for a 64 B RDMA read is around 2.9 µs. Also,




(a) Consecutive RDMA read operations.

(b) With 64 B memset between two RDMA read operations.

Figure 5.1: Raw latency results of 64 B RDMA read benchmark. The benchmark comprises 10 000 RDMA reads to the same memory location.

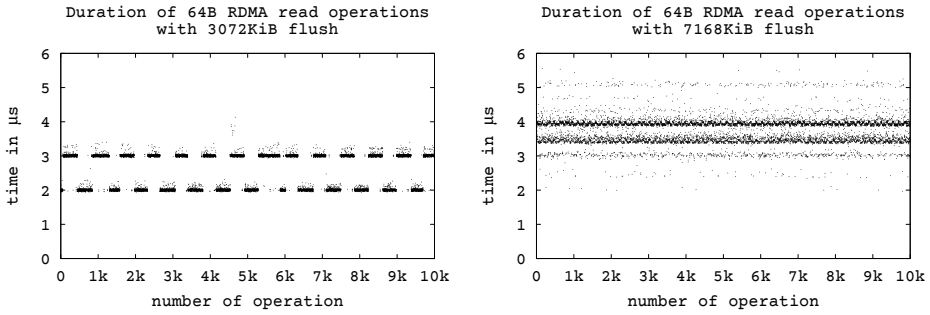
the variance between the measure points is low with only a few outliers. This is the result that we expect after our measurements in Section 4.1.

Now we modify the benchmark, so that the client performs a message exchange with the server between two RDMA operations. Before the server sends a reply to the client, he performs a 64 B *memset* to a memory location that the client does not read from and is not in the same memory page as the location the client reads from. Figure 5.1b shows the raw results for the test run of the modified benchmark.

We see that due to the additional InfiniBand and host operations the latency of the RDMA read operations decreased to around 2.3 µs. Again, the measurements are very uniform but more operations deviate from the median than in the unmodified test run.

We extend the benchmark again, so that the client now performs an RDMA read and an RDMA write operation in addition to the message exchange with the server and the memset on the server side between two RDMA read operations. We call the combined operation of RDMA read, write and message exchange with memset on the server a *flush*. The client performs the RDMA read and write operations of the flush on a different location in the server virtual address space than the RDMA read operation which latency we measure. The RDMA read and write, as well as the memset of the flush are of the same size. The RDMA read of which we measure the latency is of the same size as before, 64 B. Figure 5.2 shows the raw results for two different sizes of intermediate operations.

For a flush size of 3 MiB we observe two bands of latencies in the plot




(a) With 3 MiB flush between two RDMA read operations.

(b) With 7 MiB flush between two RDMA read operations.

Figure 5.2: Raw latency results of 64 B RDMA read benchmark. The benchmark comprises 10 000 RDMA reads to the same memory location, with a flush comprising an RDMA read, an RDMA write and a memset with variable size between the 64 B RDMA read operations.

of the raw results. We show this in Figure 5.2a. The latency of the RDMA read operation alternates in intervals of around 500 consecutive operations between 2 µs and 3 µs. The latencies in the lower band are lower than our results from the previous test with a 64 B memset and the consecutive RDMA reads without intermediate operations. The latencies in the upper band are higher than the latencies in our test with consecutive RDMA reads.

When we increase the flush size to 7 MiB, we see a shift in the latencies. The majority of measured latencies lies in two continuous bands at around 3.5 µs and 4 µs. There are two additional, less pronounced bands at 3 µs and 5.1 µs. The measured latencies in the two major bands are higher than the measured latencies in the previous tests.

We conclude that there are variations in the latencies of RDMA operations and that they appear under circumstances that are similar to a run of the Falafel prototype. We expect them to influence the results of our evaluation benchmarks.

## 5.3 Lookup Cost

Knuth shows that a hash table with open addressing and linear probing as used in Falafel has distinct performance characteristics [41]. In this section we compare the performance of the *Falafel prototype* to the characteristics

given by Knuth.

The number of lookup operations for a search in the hash table determines how long a `SET`, `GET` or `DELETE` will take, not including the time it takes to transfer the value for `SET` or `GET`. Knuth predicts the average number of required lookups in the hash table in case of a successful search as:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

Here, $\alpha$ is the load factor, a value ranging from 0 to 1 indicating how much of the available hash table entries are full. The number of required lookups in the hash table in case of an unsuccessful search is given as:

$$C'_n \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$$

We experimentally confirm that our client is in line with the number of lookups per operation as predicted above, depending on the load factor of the hash table. We fill the key-value store with a set of key-value pairs until the hash table reaches a certain load factor and then perform `GET` operations on the hash table, recording the number of lookups that each `GET` operation takes. To measure the number of lookups in case of a successful search, we perform the `GET` operations on the same set of keys that we filled the hash table with. To measure the number of lookups in case of an unsuccessful search, we perform the `GET` operations on a different set of keys than the one we filled the hash table with. For this test, we deactivate the adaptive pre-lookup on the client.

Following Knuth, *we count load operations on keys as lookup*. Therefore, load operations on $(key, value)$ containers count as lookup, not the initial load of a hash table chunk. As a client can determine whether a hash table entry is empty by testing if the pointer in the entry points to a container or not, a client does not have to perform a load on a container in this case. The operations that Knuth describes do not make this distinction, so we have to modify Knuth's $C'_n$ by an offset of one:

$$C''_n \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) - 1$$

For successful lookups we run the test for load factors from 0.01 to 0.91 in 0.01 increments. For unsuccessful lookups we run tests with load factor

from 0.01 to 0.73 in 0.01 increments. In both cases, the maximum load factor equals a predicted average lookup count of 6, which we choose as upper limit to keep the test duration manageable. Figure 5.3 shows the average number of lookups in comparison to the ideal values as given by Knuth. The data points are average values over 10 different key sets.

For both successful and unsuccessful lookups, we calculate the coefficient of determination [44]. The coefficient of determination indicates how much of the observed data is explained by the given equations $C_n$ and $C_n''$. The measured data fits well to the predicted values. The coefficient of determination is 0.9999 for the successful lookups which means that we can explain 99.99 % of the measured data through the prediction. For unsuccessful lookups the coefficient of determination is 0.9999

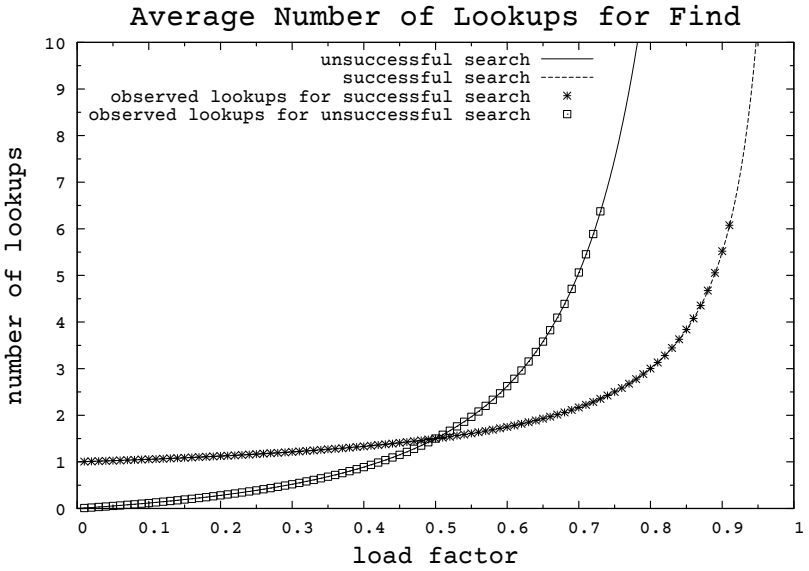



Figure 5.3: Average number of observed lookups for a successful and unsuccessful *find* in comparison to the ideal values as predicted by Knuth.

As stated above, Figure 5.3 shows the results for the average number of lookups depending on the hash table utilization. Though, lookups may consist of several RDMA operations. Figure 5.4 shows the average number of RDMA operations from the same test as above depending on the hash table utilization. Due to the initial load of a hash table chunk, we assume that the average number of RDMA operations will increase by one compared to the average number of lookups. The equation for successful lookups is then:

$$C_n''' \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) + 1$$

While the number of required RDMA operations for an unsuccessful search should be described by $C_n'$.

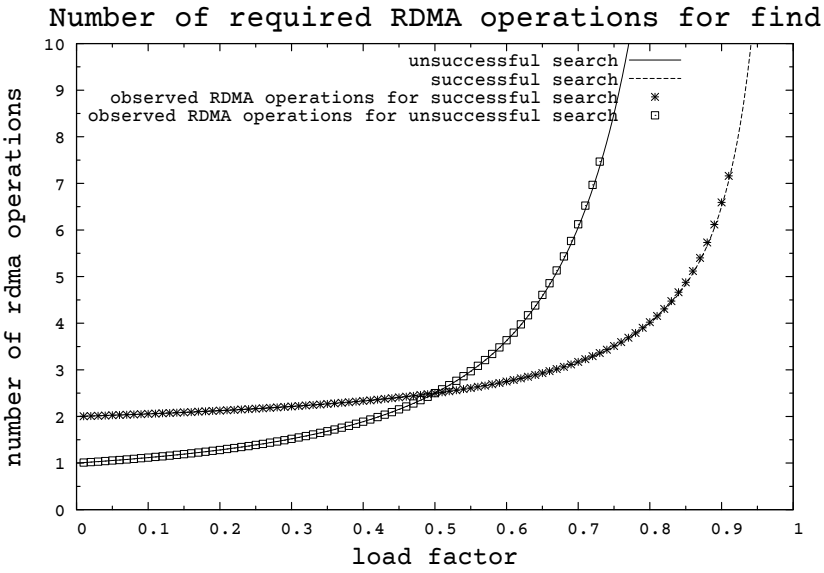



Figure 5.4: Average number of observed RDMA operations for a successful and unsuccessful *find* in comparison to the ideal values as predicted by Knuth.

Again we calculated the coefficient of determination to see how well the measured data fits the predicted values. For successful lookups the coefficient of determination is 0.9996 and for unsuccessful lookups it is 0.9998 , which means the equations $C_n'$ and $C_n'''$ describe the behavior of the Falafel prototype very well.

We see that our Falafel prototype matches the theoretical performance characteristics as given by Knuth for a key-value store with a hash table that uses open addressing and linear probing.

## 5.4 Throughput

We evaluate the achievable number of operations, the throughput of our Falafel prototype. We describe the benchmark that we use in Section 5.4.1. In Section 5.4.2 we present our findings for the throughput of a single

client, and for multiple clients in Section 5.4.3. In Section 5.4.4 we discuss the influence of the hash table load on throughput.

### 5.4.1 Throughput Benchmark

In our benchmark, one or more clients perform `SET` and `GET` operations on the server. All clients decide randomly which operation to choose but use the same percentage of `SET` and `GET` operations. Each client operates on its own working set of keys. The clients load the keys before the test run. Each key is 14 B in size, each value is 24 B in size. The resulting container is 80 B in size. We choose this small key-value size because the workload analysis of Atikoglu et. al. indicates that small keys and values make up a significant share of the workload of key-value stores [7]. We set the readahead to 100 B, sufficient to load a complete container. The clients loop over the keys of their respective working set to decide which key they will perform a `SET` or `GET` on next. The total amount of keys is 10 % of the available hash table entries, keeping the load factor at 0.1. With more clients, the working set of each client gets smaller. Clients perform the `SET` and `GET` operations sequentially. Each client loops over the keys in his working set, approximating a linear distribution of keys.

### 5.4.2 Throughput for a Single Client

We measure the achievable throughput for one client with our Falafel prototype. We expect to get two reference points, one for only `SET`s, and one for only `GET`s. As the client performs all operations consecutively, the remaining measure points that mix `SET`s and `GET`s should represent the *weighted average* between these two reference points. Figure 5.5 shows the measurements. For each percentage of `SET` operations, the client performs 1 000 000 million operations (1 Mops).

For a workload with only `SET` operations, the client reaches 98 kops/s. The throughput reaches 147 kops/s for a workload with only `GET` operations. This is the result of the shorter `GET` operations in comparison to `SET` operations as we show in Section 5.5.

We test whether the remaining values form the weighted average. Through the number of achieved operations, we can calculate that a single `SET` took 10.2 µs, while a single `GET` took 6.8 µs. For a factor $g$ that gives the fraction of `GET` operations, the weighted average $A_S$ would be:

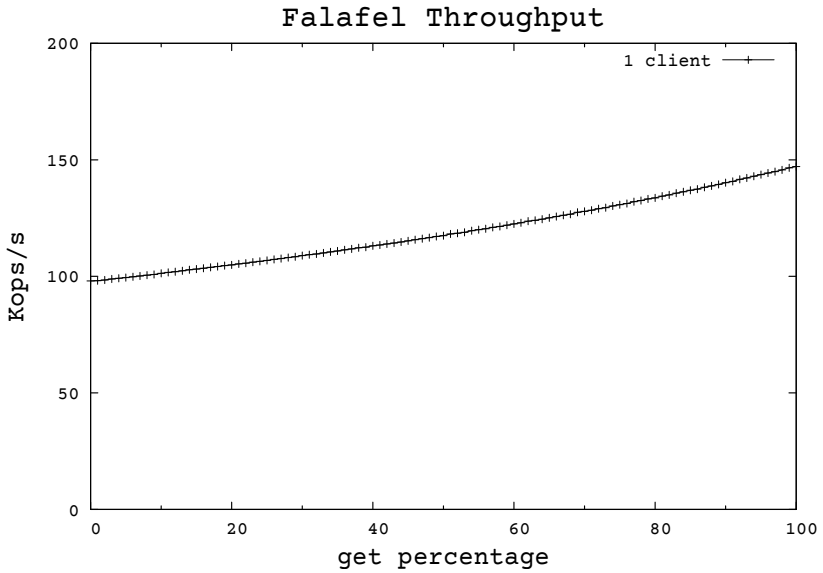



Figure 5.5: Throughput for a single client operating on its working set. The workload consists of reads and writes, with the read percentage on the x-axis. The hash table load is 10 %, the readahead is set to 100 B. All containers are less than 100 B in size.

$$A_S = \frac{1\,000\,000\,\mu s}{g * 6.8\,\mu s + (1-g) * 10.2\,\mu s}$$

Again we calculate the coefficient of determination, to see how much of the measured data we can explain with $A_S$. The coefficient of determination in this case is 0.9999 which means we can explain 99.99 % of the measured data with $A_S$.

We see, that the throughput for a single client is well predictable with a weighed average. It depends solely on the latencies of `SET` and `GET`.

### 5.4.3 Throughput for Multiple Clients

We run our benchmark of Section 5.4.1 for up to 40 clients. Considering our results for a single client in Section 5.4.2, we expect the throughput to increase linearly, adding the number of operations per second of the single client test for every new client until we reach a bottleneck. For a pure `SET` workload and our hash table load factor of 0.1, we expect a `SET` to comprise an average of 2.06 RDMA operations per lookup, an RDMA write to write the new container and an RDMA CAS operation to change

the hash table entry. The highest RDMA CAS throughput for one server node is 2.5 Mops/s as we show in Section 4.1. The limit for other RDMA operations is higher but a client has to perform a RDMA CAS at the end of every `SET`. We therefore expect that the maximum number of achievable `SET`s is below 2.5 Mops/s.

For a pure `GET` workload and the same hash table load factor, we expect `GET` to comprise an average of 2.06 RDMA operations per lookup. This is also the final number of RDMA operations per `GET`, as the readahead is sufficient to load a complete container. The highest RDMA read throughput for one server node in our tests of Section 4.1 is 15.6 Mops/s for a comparable data size. We therefore expect that the maximum number of achievable `GET`s is below 7.26 Mops/s.

Figure 5.6 shows our measurements. We show the operation count per second of 5 workloads in relation to the number of clients.

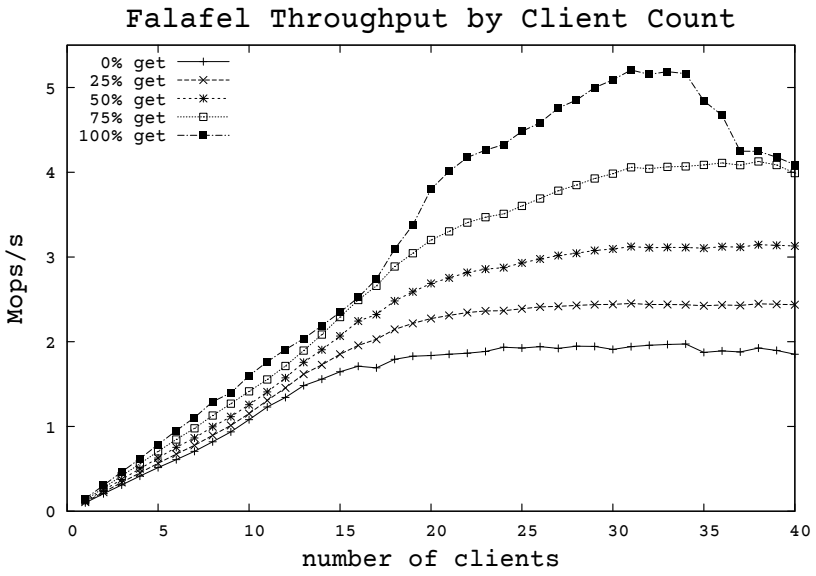



Figure 5.6: Throughput for different workloads by client count, measured over 1 Mops. For every workload and client count the hash table load factor is fixed at 0.1. We use five client nodes and distribute the client processes equally among them. We use a uniform key distribution for `SET` and `GET`.

As expected, the throughput of all workloads increases linearly up to a certain point when we increase the number of clients. Beyond that point there is only a slight variation in throughput with additional clients.

The lower the maximum throughput, the less clients we need to reach it. For a workload with only `SET` operations the throughput reaches the plateau of about 1.9 Mops/s with 20 clients. We see that the throughput for the composed `SET` operation is 25 % lower than the maximum CAS throughput.

For a workload with 25 % `GET` operations we need 25 clients to reach the plateau of about 2.4 Mops/s. For 50 % `GET` operations we need 28 clients for 3 Mops/s and for 75 % `GET` operations we need 30 clients to reach the plateau at around 4 Mops/s. When the operation count for a workload reaches its plateau through the increase of the client count and we add additional clients, the per-client-throughput decreases.

When we run a workload of only `GET` operations the throughput continuously increases until it reaches its peak at 5.2 Mops/s with 31 clients. This maximum throughput of 5.2 Mops/s is 28 % below the theoretical limit of 7.26 Mops/s that we calculated above.

Between 34 and 37 clients the throughput for the pure `GET` workload drops to 4.2 Mops/s and even further to 4 Mops/s with 40 clients. From 35 to 40 overall clients we run 7 to 8 clients per node. Each client uses several threads, in particular one thread that performs the RDMA operations and polls for their results. To make sure that the decrease in throughput is a network limitation and not the result of some other resource limitation on the client nodes, we run the same test as above on a single node for 1 to 8 clients. The server process is on a different node as before. Figure 5.7 shows the results of this test. We observe that the increase in operation count gets lower when we add new clients. Though, for every new client the overall number of operations per second increases. Therefore, the non-network hardware limitations on a single client are not the reason for the decrease in overall throughput that we observe for more than 35 clients.

Our observations show that the Falafel prototype is able to handle the throughput of multiple clients. The throughput is limited by the capabilities of the InfiniBand NIC in the server node, not by the other hardware capabilities of server or clients.

### 5.4.4 Throughput and Hash Table Load

In Section 5.3 we discussed the increasing cost per operation on an increasing load factor. We show that the cost per operation increases drastically for higher load factors. In this section we explore the implications of this

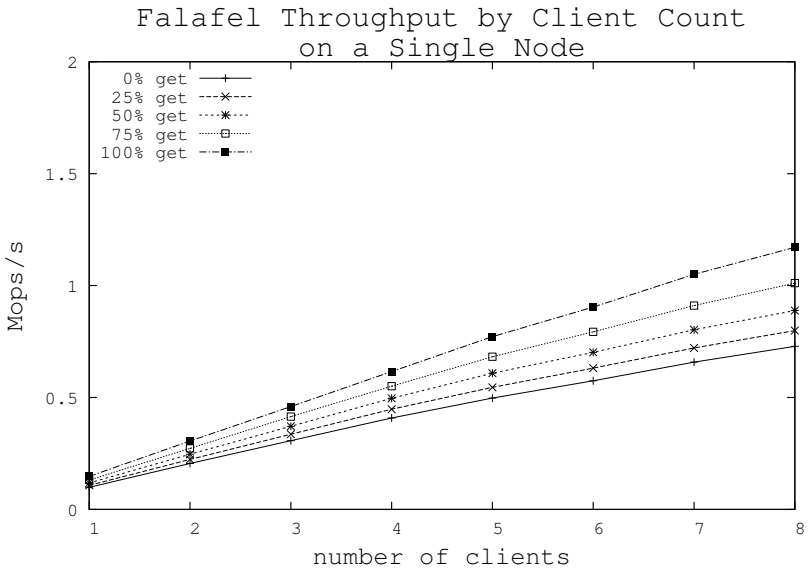



Figure 5.7: Throughput for different client counts and workloads from a single client node, measured over 1 Mops. For every workload and client count the hash table load factor is 0.1. We use a uniform key distribution for `SET` and `GET`.

behavior on throughput.

We first examine the throughput implications for a single client. We use the same benchmark as in Section 5.4.2 but vary the load factor between runs. We run the benchmark for a load factor of 0.001 and 0.05 to 0.9 in 0.05 increments. Figure 5.8 shows our measurements for pure `SET` and pure `GET` workloads.

For both workloads, we see a degradation of the throughput with an increasing load factor. This is what we expect due to the increasing operation cost on an increasing load factor as we discuss in Section 5.3. Also, the throughput difference between the two workloads becomes smaller with an increasing load factor. We can explain this with the search proportion of `SET`, `GET`, and `DELETE` operations, which is the same for all three operations. The difference in throughput comes from the RDMA operations that are different for `SET`, `GET`, and `DELETE`. For higher load factors, the clients spend most of the time of an operation finding a suitable hash table entry for the key they are looking for. Therefore, the latency difference of the other RDMA operations that `SET`, `GET`, and `DELETE` perform become insignificant and the throughput of the `SET` and

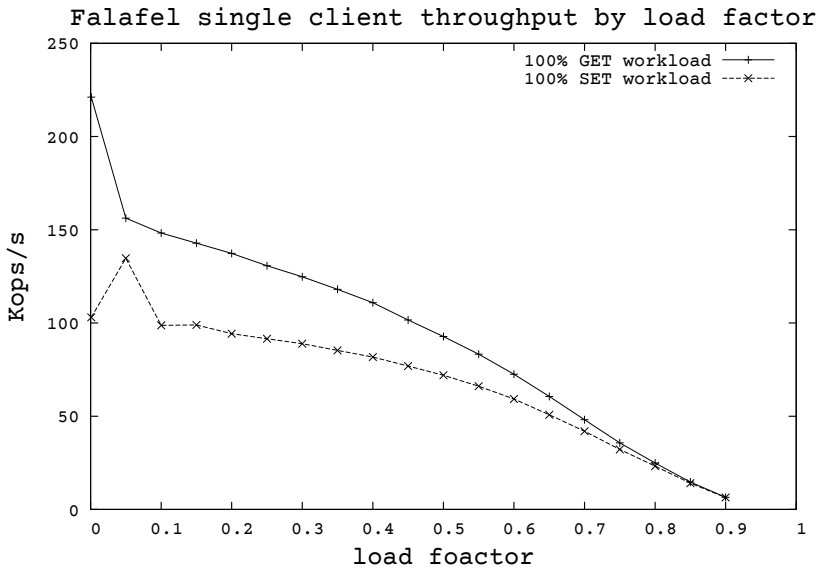



Figure 5.8: Throughput for pure SET and pure GET workloads for a single client depending on the load factor of the hash table. For each measure point the client performs 100 Mops. We use a uniform key distribution for SET and GET.

GET workloads converge.

We perform the same test with 15, 31, and 40 clients. For 31 clients we achieved the highest throughput in Section 5.4.3, while for 40 clients we could observe a performance degradation through the high number of clients. The test with 15 clients showed no abnormality. We present our results in Figure 5.9.

We see that all workloads and client counts are affected by the increasing load factor, similar to the results of Figure 5.8. The pure SET workload is less affected than the pure GET workload. We assume that though the NICs are are bound to a maximum number of RDMA CAS operations, they still can process additional RDMA read operations. The latency of the RDMA reads is then hidden through the RDMA CAS until the NIC is unable to process further RDMA reads on an increasing load factor and the throughput of the pure SET workload and the pure GET workload converge.

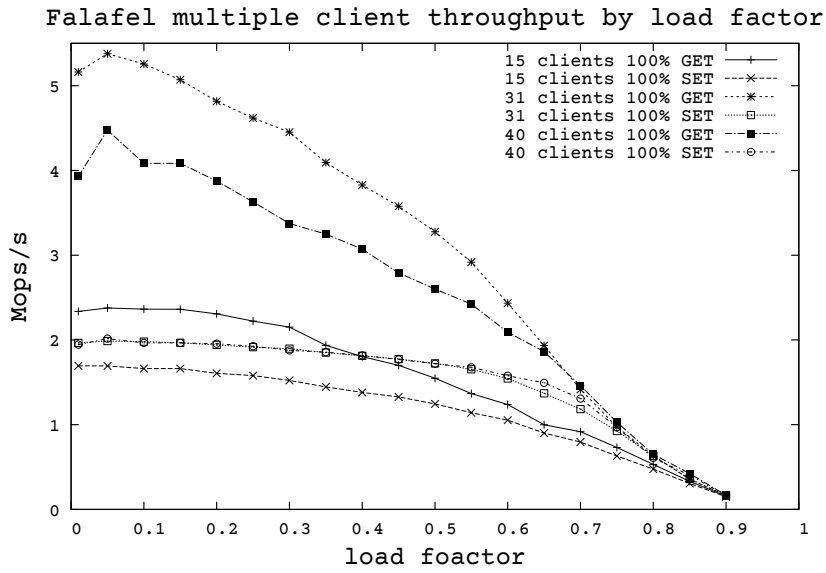



Figure 5.9: Throughput for pure SET and pure GET workloads for a multiple clients depending on the load factor of the hash table. For each measure point every client performs 10 Mops. We use a uniform key distribution for SET and GET.

## 5.5 Latency

We evaluate the latency for SET and GET operations for our Falafel prototype. We first measure the latency of individual operations and present our findings in Section 5.5.1. In Section 5.5.2 we evaluate the latency of SET and GET operations with multiple concurrent clients. In Section 5.5.3 we present our findings regarding the behavior of the Falafel prototype for various hash table loads.

### 5.5.1 Latency of Individual Operations

We measure the time it takes a single client to perform SET and GET operations for different value sizes.

The client performs the respective operations on the key-value store for three different load factors. The first one resembles an empty hash table, where the client finds an empty hash table entry for the requested key. The second one resembles a near-empty hash table with the client's working set fully loaded. Here, the client can find the requested key in the hash table on the first lookup. The third load factor resembles a hash

table with 80 % utilization and the client's working set fully loaded. He can find the requested key in the hash table and is likely to run into a collision chain that requires multiple lookups.

We simulate the hash table load factor. Instead of filling the hash table for each of the test runs of the second and third case we simulate the hash table state. We manually insert a collision chain into the hash table and instruct the client to start the search for an entry to set, get or delete, at the beginning of the collision chain. In Section 5.3 we examine the number of required lookups per hash table utilization and provide evidence that our test setup with the manually inserted collision chain correctly models the behavior of Falafel with an actually filled hash table.

We measure the latency for different value sizes. The size of the key that the client is looking for is 12 B, the key size of the other entries is 10 B. The readahead is set to 512 B. For each level of utilization and value size, we performed 10 000 runs. Figure 5.10 shows the medians of the measured data for the respective values.

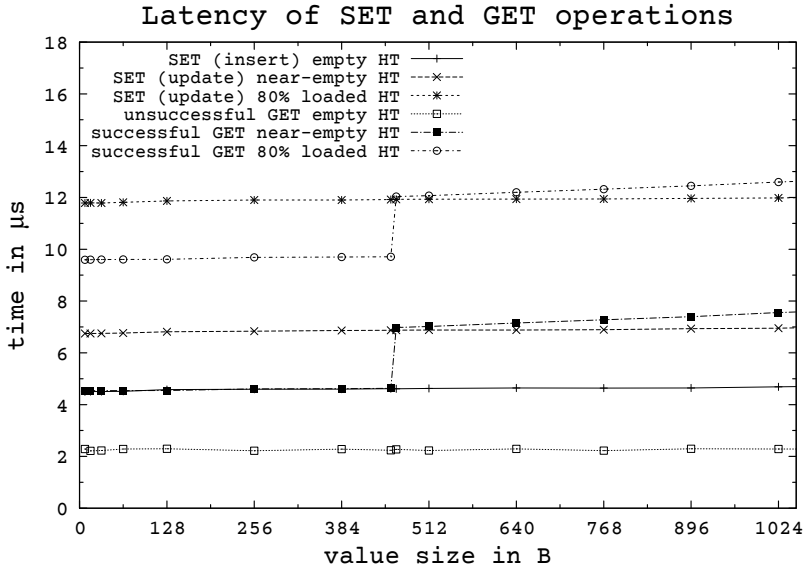



Figure 5.10: Latency for complete `SET` and `GET` operations on Falafel between two InfiniBand nodes for an empty, near-empty and an 80 % full hash table. The readahead is set to 512 B. The steep increase in latency for get operations around 464 B shows that the readahead is not sufficient from there on.

We see that for all given data sizes the latency for an unsuccessful `GET` on an empty hash table remains constant at around 2.3 µs. The reason for

this behavior is the way the client looks up the requested key in the hash table. When a client begins a `SET`, `GET`, or `DELETE`, he first loads a chunk of the hash table. In case of an unsuccessful `GET` on an empty hash table, the client does not find a valid entry in the hash table and returns from `GET` without loading any other data. As `GET` in this case does not transfer a variable size of data, the latency remains constant for all value sizes. A client may encounter an empty hash table entry for all load factors, yet the probability for this event decreases as the load factor increases.

The latency of an unsuccessful `GET` that encounters an empty hash table entry is the baseline latency for all operations. No operation can be faster. In most cases operations will perform additional RDMA operations that increase the overall latency. A `SET` that finds an empty hash table entry at the position of the key that it wants to write a value for will perform an additional write and compare and swap (CAS) operation. With the additional operations, the latency for `SET` that inserts a new key into the hash table increases to about 4.5 µs for small values of about 8 B to 1024 B. If it encounters a full hash table entry at the position it wants to write to, `SET` must read the key of the entry from the container the entry points to, to test if it matches the key it wants to write a value for. Therefore, the latency for `SET` that updates an already existing key increases to about 6.8 µs for small values. If `SET` encounters a collision chain and has to check all keys along it until finding an appropriate entry to overwrite, the latency increases again. For a hash table utilization of 80 % and a `SET` that updates a key, Knuth predicts an average number of 3 lookups in the hash table until `SET` finds a suitable entry [41]. The additional lookups increase the latency for `SET` to about 11.8 µs for small values.

When `GET` finds a full hash table entry in the hash table at the position it is looking for a key, it loads the container for that entry and all containers for subsequent full entries until it either finds the key it is looking for or reaches the end of the collision chain. In contrast to `SET` which has to write a container for the new value to the client region and swap the hash table entry, `GET` is done when it loaded the container for the key it is looking for. Therefore, as can be seen in Figure 5.10, `GET` operations have lower latency than `SET` operations for values up to 446 B. Reading from a near-empty hash table takes around 4.5 µs, which is 2.3 µs faster than a `SET` that overwrites a key in the same hash table. The difference stays the same for a hash table utilization of 80 %, where the `GET` takes 9.5 µs.

When the 512 B readahead is not effective anymore to load the complete container of the requested key, the latency for `GET` increases by the amount

of an RDMA read round trip. This happens at a value size of 464 B where the size of the value plus header and CRC values in the container add up to 520 B. From the point on the readahead is not effective anymore, the latency of the additional RDMA read to load the complete container adds directly to the overall latency of find. In contrast, the pipelining of RDMA operations that is possible for SET partially hides the additional write transfer cost for the larger value. Yet, from a value size of around 2 KiB the RDMA write takes too long to be hidden. Its latency then adds up to the overall latency of SET. Figure 5.11 shows this effect.

The gradient for SET and GET decreases at a value size of 4 KiB. This decrease is the result of the change in the latency increase of single RDMA operations that we show in Section 4.1.

SET latency surpasses GET latency in our benchmark at around 15 KiB. When GET returns a value to the calling process, it passes a pointer to the library-internal RDMA buffer instead of creating a new buffer. This saves an unnecessary copy of the value. In contrast, SET first copies a value that the calling process passes to the library-internal RDMA buffer. This copy operation yields additional overhead, resulting in a larger increase in latency for SET in comparison to GET, despite the larger latency increase in RDMA reads compared to RDMA writes as we show in Section 4.1.

From 15 KiB on, SET and GET latencies increase linearly up to a value size of 1024 KiB, which is the largest value size that we tested.

Using the same test setup and hash table utilization levels as for SET and GET above we evaluate the latency of DELETE operations. Figure 5.12 shows the result for value sizes of 8 B to 1024 B. We see that the latency does not depend on the value size. In contrast to SET or GET, a DELETE operation doesn't transfer a variable size of data provided the readahead is sufficient to load the key in case DELETE has to check the key of a hash table entry. In our test setup the readahead is sufficient to load the key of all containers. Therefore the latency of a DELETE operation only depends on the time it takes DELETE to find the appropriate hash table entry.

For successful DELETE operations where DELETE finds a matching key to delete, we see that a DELETE operation has approximately the same latency as a SET for a small value size for the same load factor. They are similar, because a DELETE operation comprises the same operations as a SET operation without the RDMA write operations that SET needs to write the new container data to the server memory. For small value sizes, the latency of the RDMA write is hidden through pipelining as we can see in the tests above. Therefore we can approximate the latency of successful

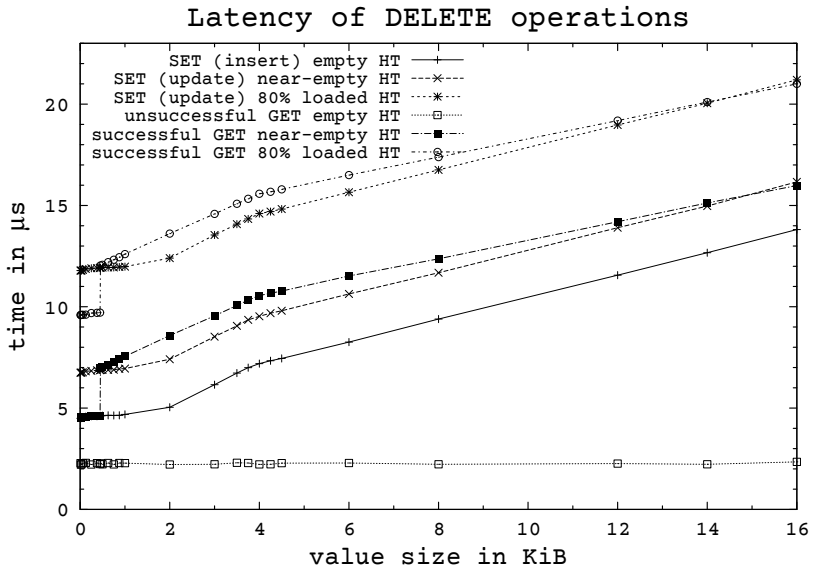



Figure 5.11: Latency for complete SET and GET operations on Falafel between two InfiniBand nodes for value sizes between 8 B and 16 KiB. Between 464 B and 15 KiB the GET latency is higher than the SET latency.

DELETE operations with a SET operation with a small value size for the same level of hash table utilization.

For unsuccessful DELETE operations where DELETE does not find a matching key to delete, the DELETE operation has the same latency as an unsuccessful GET operation. GET and DELETE perform the same RDMA operations in this case. They walk the hash table to find a matching key but do not find one and return. Therefore we can approximate unsuccessful DELETE operations with unsuccessful GET operations.

### 5.5.2 Latency of Concurrent Operations

We evaluate the latency of SET and GET operations when running Falafel with multiple clients. We use the same test setup as in Section 5.4. Figure 5.13 shows the results of this benchmark.

We see that for up to 10 clients the latency of SET and GET is nearly the same. After a certain number of clients, the latency for all workloads increases. As we can see in Figure 5.9, the number of clients from which on the latency increases for each workload corresponds with the number of clients at which the aggregate throughput gained with each client

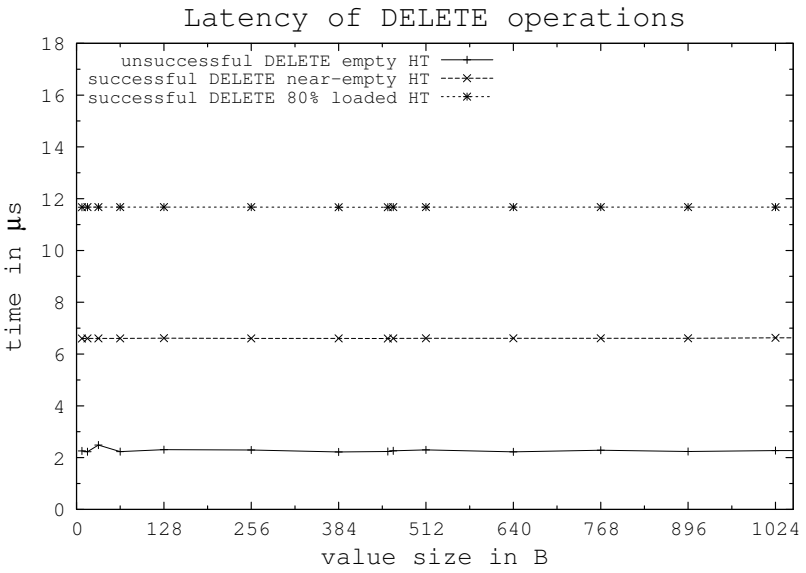



Figure 5.12: Latency for complete DELETE operations on Falafel between two InfiniBand nodes for a near-empty and an 80 % full hash table. The readahead is set to 512 B.

decreases. The server NIC is unable to process more RDMA operations, and therefore each operation takes longer.

Like for the point where the throughput reaches its peak in Figure 5.9, the point for when the latency increases is different among the workloads. The higher the percentage of GETs in a workload, the later the latency starts to increase, and the lower is the increase through the higher client number. This also corresponds to our throughput measurements, where workloads with higher GET portion reach a higher throughput.

Figure 5.14a shows the latency of SET operations for different numbers of clients depending on the percentage of GET operations. For 1 to 15 clients SET latency lies between 8 µs to 10 µs. In this range of number of clients we observe several types of client behavior that we cannot fully explain. We see outliers in the measured data for one client though the measure points are the medians of the measured data. In our tests, these outliers are not reproducible and are not bound to a specific number of clients or percentage of get operations. They also appear when we increase the number of runs from 1 000 000 to 10 000 000 or more. Also, the latency for one client is higher than for 5, 10, and 15 clients and increases for 10 and 15 clients with higher percentages of GET operations. We attribute

these observations to the InfiniBand noise that we describe in Section 5.2.

For each run with 20 to 40 clients we see that the latency is highest for a workload with 100 % SET operations. With an increase in GET operation count in the workload, SET latency decreases. This behavior is bound to the number of operations that we examine in Section 5.4. For a workload with 100 % SET operations 20 clients already reach the maximum throughput of 1.9 Mops/s. Additional clients increase the average latency of SET operations for all clients.

The latency for GET operations shows the same pattern as the latency for SET operations. We show this similarity in Figure 5.14b. The latency for GET operations is lower though, which is in line with our observations of Section 5.5.1.

For the test run with 40 clients we observe that we reach the lowest latency with a workload with 74 % GET operations. From there on the latency increases again. For the same number of clients we observe a decrease in throughput as we show in Section 5.4.3. In Figure 5.13 we see that the decrease in latency for workloads with 99 % and 100 % GET percentage becomes more pronounced when we add more clients from 34 clients onwards.

Again we make sure this is a network limitation and not the result of some other resource limitation on the nodes. As in Section 5.4.3, we run the same test as above on a single node for 1 to 8 clients. The server process is on a different node as before. Figure 5.15 shows the results of this test. On a single node, we observe no increase in latency for SET or GET operations through the number of clients. Therefore, the non-InfiniBand hardware limitations on a single client are not the reason for the increase in overall latency that we observe for more than 34 clients.

### 5.5.3 Latency and Hash Table Load

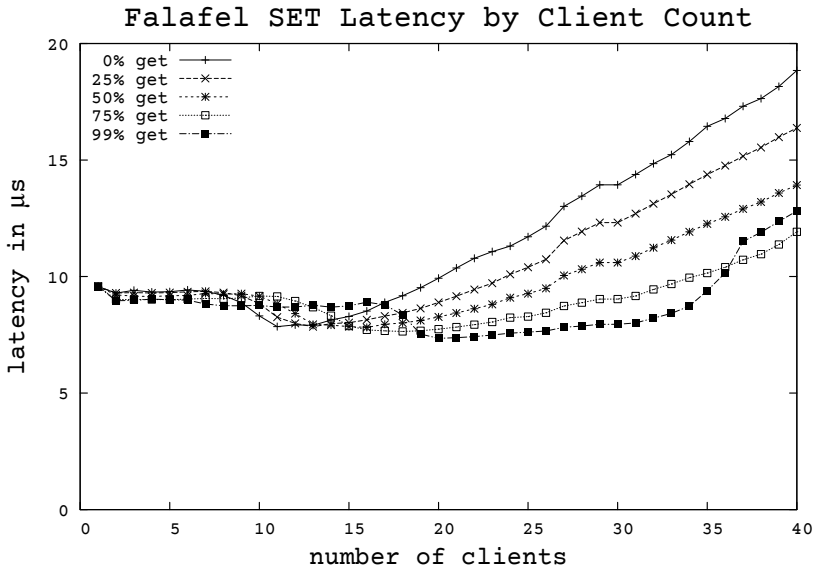
As for throughput in Section 5.4.4, we discuss the implications of higher load factors that we describe in Section 5.3 for the latency of SET and GET in this Section. We use the same test setup as in Section 5.4.1 and vary the load factor. We run the benchmark for a load factor of 0.001 and 0.05 to 0.9 in 0.05 increments. Figure 5.16 shows our measurements for a pure SET and a pure GET workload.

The measurements combine the results of Section 5.3 and Section 5.5.1. We see the characteristic steep increase in latency from a load factor of 0.7 onwards for both workloads that resembles the higher number of lookups.

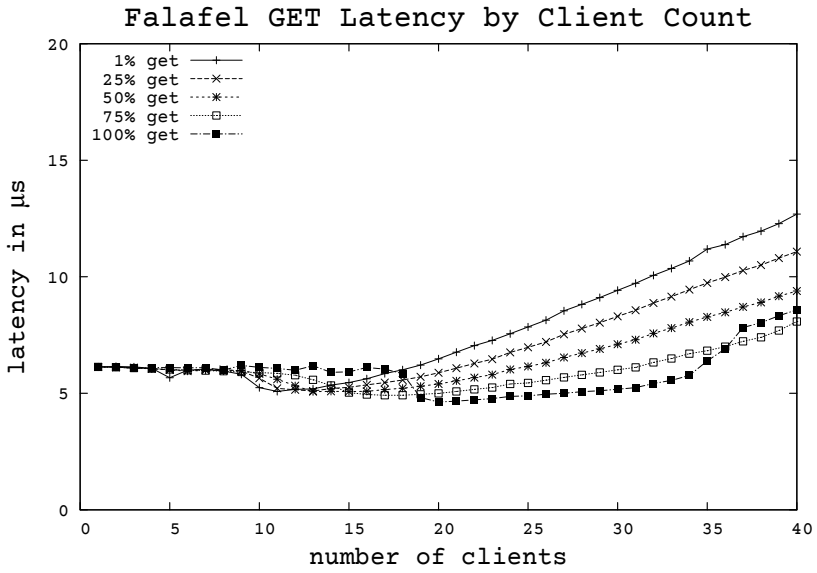
The latency for the pure SET workload is higher than the latency for the pure GET workload as the latency of the single SET operation is higher than the latency for the single GET operation under the same circumstances, as we show in Section 5.5.1.

We perform the same test with 15, 31, and 40 clients. For 31 clients we achieved the highest throughput in Section 5.4.3, while for 40 clients we observed a performance degradation through the high number of clients. The test with 15 clients showed no abnormality. We present our results in Figure 5.17.

The SET and SET latencies for multiple clients behave like the corresponding latencies for a single client. Up to a load factor of 0.6 there is only a slight increase in latency, followed by an exponential increase for higher load factors. Similar to our results in Section 5.5.2, the latencies are higher for runs with more clients.

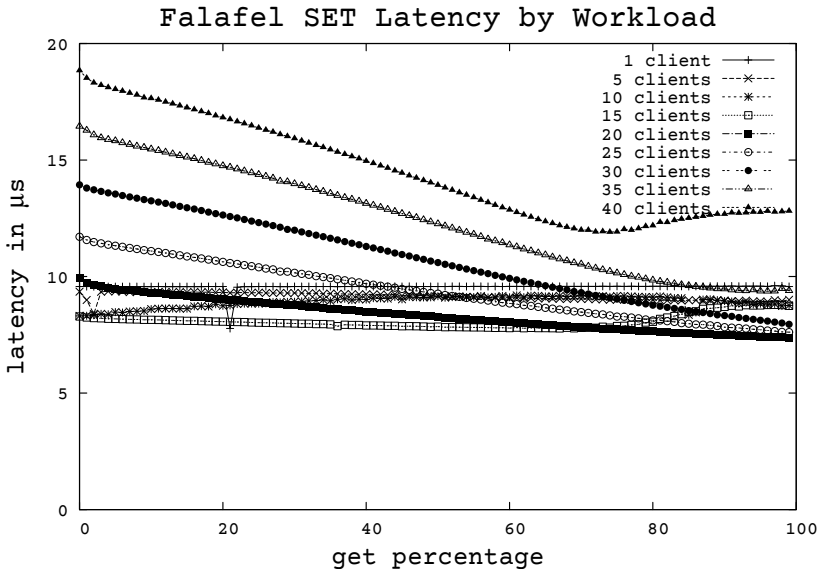



(a) Latency of SET operations by client count.

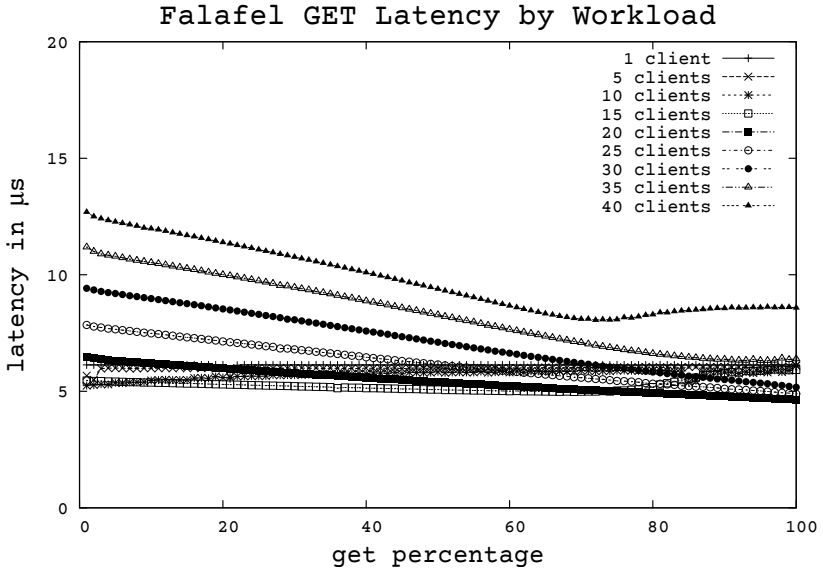



(b) Latency of GET operations by client count.

Figure 5.13: SET and GET latency by client count for different workloads. Measure points are medians of 1 000 000 operations. For every workload and client count the hash table load factor is 0.1 We use five client nodes and distribute the clients equally among them. We use a uniform key distribution for SET and GET.

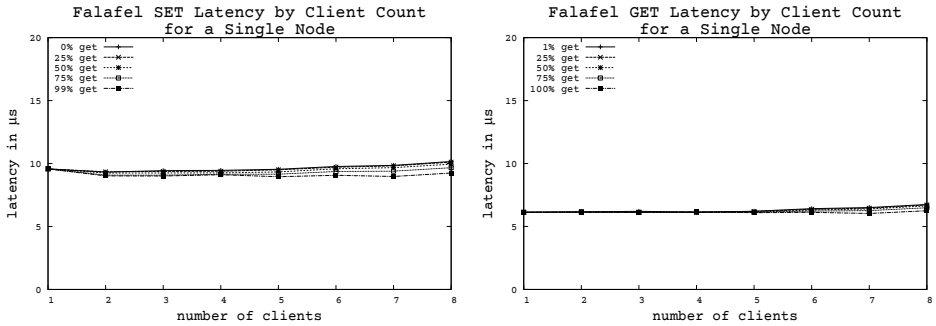



(a) Latency of SET operations by workload.




(b) Latency of GET operations by workload.

Figure 5.14: SET and GET latency by workload with different numbers of clients. Measure points are medians of 1 000 000 operations. For every workload and client count the hash table load factor is 0.1. We use five client nodes and distribute the clients equally among them. We use a uniform key distribution for SET and GET.




(a) Latency of `SET` operations by client count for a single client node.

(b) Latency of `GET` operations by client count for a single client node.

Figure 5.15: `SET` and `GET` latency by client count for different workloads for a single client node. Measure points are medians of 1 000 000 operations. For every workload and client count the hash table load factor is 0.1 We use a uniform key distribution for `SET` and `GET`.

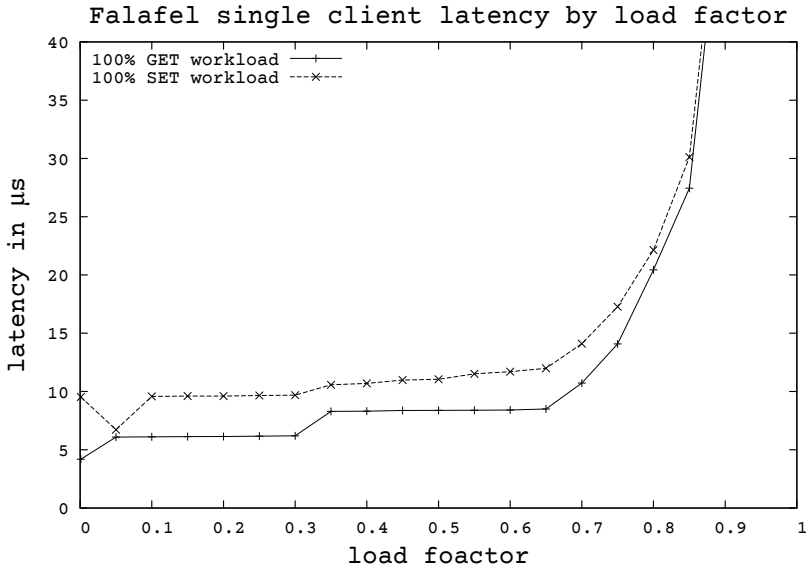



Figure 5.16: Latency for `SET` and `GET` workloads depending on the hash table load factor. The measure points are medians of the latencies of 100 000 000 operations.

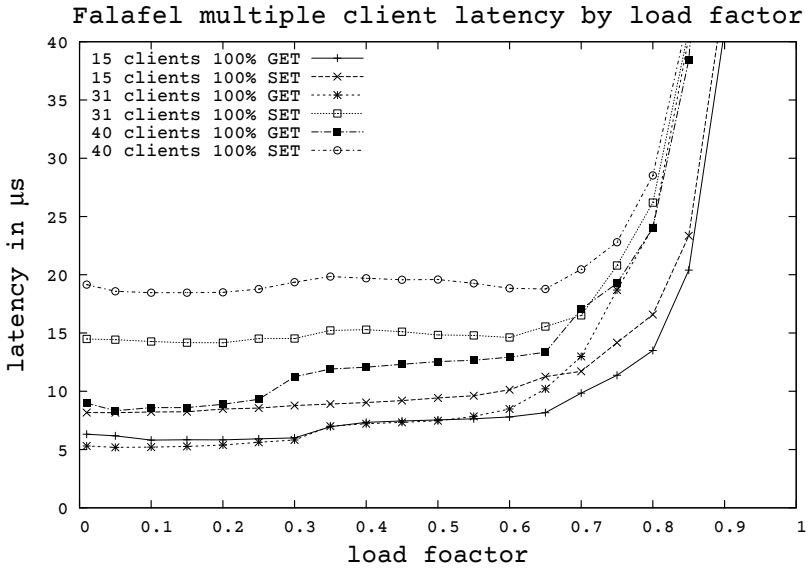



Figure 5.17: Latency for `SET` and `GET` workloads depending on the hash table load factor. The measure points are medians of the latencies of 100 000 000 operations.

# Chapter 6

# Related Work

In this chapter, we discuss key-value stores that influenced the design of *Falafel* and how Falafel differs from previous work in terms of memory management, hash table design, and network access. We divide the related work into two subgroups: In Section 6.1 we list the key-value stores that use classical networking. In Section 6.2 we review the key-value stores that use remote direct memory access (RDMA) [68] on their relation to our work.

## 6.1 Key-Value Stores with Message-Based Networking

Classical key-value stores such as *memcached* [21] and *redis* [62] offer their functionality over the socket interface [38] to clients. Here, the server processes requests that he receives from the clients and sends an appropriate answer. The client has no access to the server memory. For every request, the client has to pass control and data flow to the server. Both memcached and redis use a central memory allocator and a central hash table with closed addressing and collision chains. Despite these differences to Falafel, these key-value stores share the basic interface of a network-accessible dictionary with our work.

Rumble et. al. create a log-structured memory management for their *RAMCloud* storage system [55,61]. The log-structured approach for memory management that is influenced by log-structured file systems [59] is similar to the way that clients write data to the server memory in Falafel. Like in Falafel, the RAMCloud memory is divided into regions and the

log structured memory management uses a cleanup mechanism to clean regions from garbage. Though, in RAMCloud the server process performs the read and write operations to the memory as RAMCloud relies on message passing instead of RDMA. Also, the cleanup mechanism is dedicated to the replication mechanism of RAMCloud and compacts regions after cleaning instead of reusing them.

Rumble et. al. modified memcached and replaced the builtin slab allocator [14] with RAMCloud's log and cleaner. They show that RAMCloud's memory management can improve the performance and memory utilization of the key-value store memcached. Apart from the now similar memory management, memcached still bears the same differences to Falafel as discussed above.

Hyeontaek et. al. present a scalable in-memory key-value store called *MICA* [45]. They still rely on message passing between server and clients but use direct network interface card (NIC) access to bypass the operating system and direct the request packets to distinct CPU cores. Instead of using a central hash table like Falafel, MICA shards data in partitions between CPUs so that these can access the data in parallel and therefore avoids concurrent access within each partition. Similar to Falafel and RAMCloud, MICA can use a log structured memory management approach where it stores items in a circular log. In contrast to Falafel and RAMCloud, this log structured memory management method has cache semantics where keys may be evicted arbitrarily without explicit request through a least recently used (LRU) strategy. MICA is optimized for small key-value data sizes. Hyeontaek et. al. consider a key size of 128 B and a value size of 1024 B as large. Larger sizes lead to older data being evicted earlier from the circular log. In contrast, Falafel imposes no special penalty for larger key-value sizes.

## 6.2 Key-Value Stores using RDMA

Instead of message passing, some key-value stores use RDMA [68] for the clients as medium of data access.

Stuedi et. al. modify memcached to use the RDMA capabilities of SoftiWARP [70] over Ethernet for `GET` operations [66]. Clients can read data from the server's memory with one-sided RDMA operations. Though, this kind of operation is not always possible and the client might have to fall back to a message based request, therefore passing the control flow

to the server process. This is not necessary in Falafel. With the modified memcached, the client still has to perform `SET` operations on the key-value store via message passing. The server processes the message and inserts the new data into the hash table.

*Pilaf* by Mitchell et. al. is a key-value cache that also features RDMA read operations for the client [53]. In contrast to the modified memcached by Stuedi et. al., Pilaf is specifically designed for client RDMA read operations. Similar to Falafel, clients can perform lock-free reads from the hash table and data structures directly without involving the server process. For write operations, however, Pilaf relies on message exchange and a server-side central memory management. Pilaf uses cuckoo hashing [56] to limit the worst case number of necessary RDMA lookups in the hash table.

Dragojević et. al. present a main memory distributed computing platform called *FaRM* that also incorporates a key-value store interface [19]. They employ RDMA read and write operations to speed up the key-value store operations. Like in Falafel, `GET` operations from the server are lock-free and are done with one sided RDMA read operations. In contrast to our work the FaRM server manages the data structures in the server memory, while clients use RDMA to transfer data during write operations to an intermediate buffer first. For the FaRM key-value store interface, Dragojević et. al. use a newly designed hash algorithm that combines *hopscotch hashing* [33] with chaining and associativity.

Szepesi et. al. propose a key-value cache named *Nessie* that shall make client RDMA writes possible [67]. With cuckoo hashing Szepesi et. al. try to employ the same hash table design as Pilaf. Apart from the different hash table design, they also intend to organize the server memory differently to our design: Instead of memory regions with garbage collection, clients get a number of fixed-size storage slots. Clients notify each other about overwritten entries, adding communication overhead to the `SET` and `DELETE` operations. Because of the complexity of cuckoo hashing paired with CAS-based hash table operations, the hash table in Nessie may stay in an erroneous state when a client fails during a write operation. In contrast to our hash table design which shows a continuous exponential decrease in performance on an increasing load factor, cuckoo hashing as used in Nessie only requires constant time for most operations performed on it. With increasing load factor there is a growing chance that an insert operation cannot succeed without server assistance.

# Chapter 7

# Conclusion and Future Work

In this thesis we presented **Falafel**, a design for an in-memory key-value store that allows concurrent remote direct memory access (RDMA) reads and writes by clients. Inspired by previous work such as Pilaf [53] and FaRM [19] which leverage RDMA read operations to make *lock-free `GET` operations* on the key-value store possible, we use RDMA write operations to enable *lock-free `SET` and `DELETE` operations* on the key-value store. This development is driven by the need for faster `SET` and `DELETE` operations in addition to fast `GET` operations on a key-value store [3,7,54].

To make concurrent `SET` operations possible, we designed a memory management that provides the client with *explicit regions* to write to, and that uses a *generational semi-space garbage collector* on the server side to save the data written by clients when their region is full. Our memory management scheme allows the client to write exclusively to a region of memory in the server address space, while being able to read the data written by other clients from their respective regions.

We use a *central hash table with open addressing and linear probing* that the clients modify with RDMA compare-and-swap (CAS) operations. The client `SET`, `GET`, and `DELETE` operations on the key-value store are lock-free and resilient to client failure.

We implemented the Falafel design in a prototype and added hardware-specific optimizations. We evaluated the Falafel prototype in a set of benchmarks and showed that in our hardware environment the prototype is able to handle up to 5.2 Mops/s `GET`s and 1.9 Mops/s `SET`s, in both cases limited by the InfiniBand NIC on the server node. A client can perform `GET` operations on the server that take as low as 4.5 µs and `SET` operations that take as low as 6.8 µs. Throughput and latency of our Falafel prototype

show an exponential decline on higher load factors of the hash table, as is typical for a hash table design based on open addressing and linear probing.

In future work we will explore possibilities to reduce the impact of higher load factors, such as *hinting* which would speed up the lookup process in the hash table, and more intelligent *readahead* that saves transfer time by adjusting to the size of the key-value data. We also want to investigate whether the RAMCloud [61] memory can improve memory utilization in our client. We want to evaluate our Falafel prototype against other key-value stores such as Pilaf [53] and FaRM [19], and see how well our approach scales multiple sharded servers. As our design shifts work from the server to the clients, we want to perform a power-efficiency comparison against conventional key-value stores.

# Bibliography


[1] IBM system/370 extended architecture principles of operation. (SA22-7085-0), March 1983.

[2] CentOS, 2014. http://vault.centos.org/6.5/.

[3] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, page 123–132, New York, NY, USA, 2012. ACM. http://doi.acm.org/10.1145/2124295.2124312.

[4] James H. Anderson and Mark Moir. Universal constructions for large objects. In *Distributed Algorithms*, page 168–182. Springer, 1995. http://link.springer.com/chapter/10.1007/BFb0022146.

[5] Austin Appleby. Murmurhash 2.0. 2009.

[6] InfiniBand Trade Association. InfiniBand architecture specification: Release 1.2.1. November 2007.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. ACM. http://doi.acm.org/10.1145/2254756.2254766.

[8] Pavan Balaji, Hemal V. Shah, and Dhabaleswar K. Panda. Sockets vs RDMA interface over 10-gigabit networks: an in-depth analysis of the memory traffic bottleneck. In *IEEE Cluster 2004 RAIT Workshop (RDMA Applications, Implementations, and Technologies)*, 2004. https://nowlab.cse.ohio-state.edu/publications/tech-reports/2004/balaji-rait04-10gige-tr.pdf.

[9] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, page 261–270, New York, NY, USA, 1993. ACM. http://doi.acm.org/10.1145/165231.165265.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, July 2013. http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024.

[11] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 117–128, New York, NY, USA, 2000. ACM. http://doi.acm.org/10.1145/378993.379232.

[12] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 25–36, New York, NY, USA, 2004. ACM. http://doi.acm.org/10.1145/1005686.1005693.

[13] Nanette J. Boden, Alan E. Kulawik, Charles L. Seitz, Danny Cohen, Robert E. Felderman, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995. http://www.computer.org/csdl/mags/mi/1995/01/m1029.pdf.

[14] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, page 6–6, Berkeley, CA, USA, 1994. USENIX Association. http://dl.acm.org/citation.cfm?id=1267257.1267263.

[15] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Garbage collection hints. In *High Performance Embedded Architectures and Compilers*, page 233–248. Springer, 2005. http://link.springer.com/chapter/10.1007/11587514_16.

[16] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Survey (CSUR) 13.3 (1981)*, 13(3):341–367, September 1981. http://doi.acm.org/10.1145/356850.356854.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. ACM. http://doi.acm.org/10.1145/1807128.1807152.

[18] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 185–192, May 2010.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX, April 2014. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-dragojevic.pdf.

[20] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*. Citeseer, 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.87&rep=rep1&type=pdf.

[21] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5–, August 2004. http://dl.acm.org/citation.cfm?id=1012889.1012894.

[22] A.P. Foong, T.R. Huff, H.H. Hum, J.P. Patwardhan, and G.J. Regnier. TCP performance re-visited. In *2003 IEEE International Symposium on Performance Analysis of Systems and Software, 2003. ISPASS*, pages 70–79, 2003.

[23] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 18(1):21–42, July 2005. http://link.springer.com/article/10.1007/s00446-004-0115-2.

[24] H. Gao and W. H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205(2):225–241, February 2007. http://www.sciencedirect.com/science/article/pii/S0890540106001234.

[25] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 313–323, New York, NY, USA, 1998. ACM. http://doi.acm.org/10.1145/277650.277748.

[26] David Gifford and Alfred Spector. Case study: IBM's system/360-370 architecture. 30(4):291–307. http://doi.acm.org/10.1145/32232.32233.

[27] Michael Greenwald. Non-blocking synchronization and system design. Technical report, Stanford University, Stanford, CA, USA, 1999.

[28] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, 1990. http://onlinelibrary.wiley.com/doi/10.1002/spe.4380200104/abstract.

[29] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, page 300–314. Springer, 2001. http://link.springer.com/chapter/10.1007/3-540-45414-4_21.

[30] Rolf Hempel. The MPI standard for message passing. In Wolfgang Gentzsch and Uwe Harms, editors, *High-Performance Computing and Networking*, number 797 in Lecture Notes in Computer Science, pages 247–252. Springer Berlin Heidelberg, January 1994. http://link.springer.com/chapter/10.1007/3-540-57981-8_126.

[31] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993. http://doi.acm.org/10.1145/161468.161469.

[32] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. 23(2):146–196. http://doi.acm.org/10.1145/1062247.1062249.

[33] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Distributed Computing*, page 350–364. Springer, 2008. http://link.springer.com/chapter/10.1007/978-3-540-87779-0_24.

[34] Yuuji Ichisugi and Akinori Yonezawa. Distributed garbage collection using group reference counting. In *Software Science and Engineering: Selected Papers from the Kyoto Symposia*, volume 31, page 212. World Scientific, 1991. http://books.google.de/books?hl=de&lr=&id=pjf0ENmYEIAC&oi=fnd&pg=PA212&ots=6bkR0AOxPZ&sig=9Ka4LyYfs4Z3X8ZPJt3Tn02LHcs.

[35] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 151–160, New York, NY, USA, 1994. ACM. http://doi.acm.org/10.1145/197917.198079.

[36] B. Jenkins. A new hash function for hash table lookup. *Dr. Dobb's Journal*, 1997.

[37] J. Jose, H. Subramoni, Miao Luo, Minjia Zhang, Jian Huang, M. Wasiur Rahman, N.S. Islam, Xiangyong Ouyang, Hao Wang, S. Sur, and D.K. Panda. Memcached design on high performance RDMA capable interconnects. In *2011 International Conference on Parallel Processing (ICPP)*, pages 743–752, 2011.

[38] William Joy, Robert Fabry, Samuel Leffler, M. Kirk McKusick, and Michael Karels. Berkeley software architecture manual 4.3 BSD edition. *UNIX Programmer's Supplementary Documents*, 1, 1986. http://dandelion-patch.mit.edu/afs/athena/astaff/project/docsourc/OldFiles/doc/unix.manual.progsupp1/06.sysman/sysman.PS.

[39] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in TCP/IP. *SIGCOMM Comput. Commun. Rev.*, 23(4):259–268, 1993. http://doi.acm.org/10.1145/167954.166262.

[40] Jens Kehne, Marius Hillenbrand, Jan Stoess, and Frank Bellosa. Lightweight remote communication for high-performance cloud networks. In *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pages 143–147, 2012.

[41] D. E. Knuth. *The Art of Computer Programming. Sorting and Searching, vol. III*. Addison-Wesley, Reading, 1973.

[42] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. http://doi.acm.org/10.1145/358141.358147.

[43] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996. http://dl.acm.org/citation.cfm?id=234473.

[44] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. http://books.google.de/books?hl=de&lr=&id=R8RLniX5DNQC&oi=fnd&pg=PR11&dq=measuring+computer+performance+&ots=irFr_wyquy&sig=PlgDfeqaA2PtNarNHIvXw6f_JNg.

[45] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-lim.pdf.

[46] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K. Panda, David Ashton, Darius Buntinas, William Gropp, and Brian Toonen. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16. IEEE, 2004. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1302922.

[47] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, page 295–304, New York, NY, USA, 2003. ACM. http://doi.acm.org/10.1145/782814.782855.

[48] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. 2001. http://www.ittc.ku.edu/~niehaus/classes/750-s07/documents/rcu-ols-2002.pdf.

[49] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: a cache-partitioned hash table. In *Proceedings of the 17th*

*ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 319–320, New York, NY, USA, 2012. ACM. http://doi.acm.org/10.1145/2145816.2145874.

[50] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, page 21–30. ACM. http://doi.acm.org/10.1145/571825.571829.

[51] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. ACM. http://doi.acm.org/10.1145/564870.564881.

[52] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 35–46, New York, NY, USA, 2004. ACM. http://doi.acm.org/10.1145/996841.996848.

[53] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, page 103–114. USENIX Association. http://dl.acm.org/citation.cfm?id=2535461.2535475.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, and Paul Saab. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, page 385–398. USENIX Association, 2013. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf?utm_content=buffer6e057&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer.

[55] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*,

43(4):92–105, January 2010. http://doi.acm.org/10.1145/1713254.1713276.

[56] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. http://www.sciencedirect.com/science/article/pii/S0196677403001925.

[57] F. Petrini, Wu-chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network (QsNet): high-performance clustering technology. In *Hot Interconnects 9, 2001.*, pages 125–130, 2001.

[58] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In Pierre Fraigniaud, editor, *Distributed Computing*, number 3724 in Lecture Notes in Computer Science, pages 108–121. Springer Berlin Heidelberg, January 2005. http://link.springer.com/chapter/10.1007/11561927_10.

[59] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. http://doi.acm.org/10.1145/146941.146943.

[60] Douglas T. Ross. The AED free storage package. *Commun. ACM*, 10(8):481–492, August 1967. http://doi.acm.org/10.1145/363534.363546.

[61] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, page 1–16. USENIX Association. http://dl.acm.org/citation.cfm?id=2591305.2591307.

[62] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2013. http://redis.io.

[63] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, page 84–94, New York, NY, USA, 2006. ACM. http://doi.acm.org/10.1145/1133956.1133968.

[64] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, page 204–213, New York, NY, USA, 1995. ACM. http://doi.acm.org/10.1145/224964.224987.

[65] Galen M. Shipman, Timothy S. Woodall, Richard L. Graham, Arthur B. Maccabe, and Patrick G. Bridges. Infiniband scalability in open MPI. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10–pp. IEEE, 2006. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1639335.

[66] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost memcached. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, page 31–31, Berkeley, CA, USA, 2012. USENIX Association. http://dl.acm.org/citation.cfm?id=2342821.2342852.

[67] Tyler Szepesi, Bernard Wong, Ben Cassell, and Tim Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads. April 2014. http://research.microsoft.com/en-us/events/wrsc2014/.

[68] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating data and control transfer in distributed operating systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 2–11. ACM. http://doi.acm.org/10.1145/195473.195481.

[69] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev.*, 44(3):102–109, August 2010. http://doi.acm.org/10.1145/1842733.1842750.

[70] Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A case for RDMA in clouds: Turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, page 17:1–17:5, New York, NY, USA, 2011. ACM. http://doi.acm.org/10.1145/2103799.2103820.

[71] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms

nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, page 212–222, New York, NY, USA, 1992. ACM. http://doi.acm.org/10.1145/137097.137873.

[72] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, page 157–167, New York, NY, USA, 1984. ACM. http://doi.acm.org/10.1145/800020.808261.

[73] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 214–222, New York, NY, USA, 1995. ACM. http://doi.acm.org/10.1145/224964.224988.

[74] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42. Springer Berlin Heidelberg, January 1992. http://link.springer.com/chapter/10.1007/BFb0017182.

[75] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, page 1–116. Springer, 1995. http://link.springer.com/chapter/10.1007/3-540-60368-9_19.