Doctoral Thesis

Building Distributed Storage with Specialized Hardware

Author(s):
Istvan, Zsolt

Publication Date:
2018-06

Permanent Link:
https://doi.org/10.3929/ethz-b-000266096

Rights / License:
In Copyright - Non-Commercial Use Permitted
Building Distributed Storage with Specialized Hardware

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZÜRICH

(Dr. sc. ETH Zürich)

presented by

ZSOLT ISTVÁN

Master of Science ETH in Computer Science, ETH Zürich

born on 08.09.1988

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zürich), examiner
Prof. Dr. Torsten Hoefler (ETH Zürich), co-examiner
Prof. Dr. Timothy Roscoe (ETH Zürich), co-examiner
Dr. Ken Eguro (MSR Redmond), co-examiner

2017
Abstract

In an effort to keep up with increasing data sizes, applications in the datacenter are scaled out to large number of machines. Even though this allows them to tackle complex problems, data movement bottlenecks of various types appear, limiting overall performance and scalability. Pushing computation closer to the data reduces bottlenecks, and in this work we explore this idea in the context of distributed key-value stores.

Efficient compute is important in the datacenter, especially at scale. Driven by the stagnation of single-threaded CPU performance, specialized hardware and hybrid architectures are emerging and could hold the answer for more efficient compute and data management. In this work we use Field Programmable Gate Arrays (FPGA) to break traditional trade-offs and limitations, and to explore design scenarios that were previously infeasible.

This dissertation focuses on distributed storage, a building block in scale out applications. It explores how such a service can benefit from specialized hardware nodes. We focus in particular on providing complex near-data computation with the goal of reducing the data movement bottleneck between application layers. Furthermore, this work addresses a shortcoming in the design of most distributed storage nodes, namely the mismatch between computational power and network/storage media bandwidth.

The mismatch is present because, if regular server machines are used, there is plenty of processing power to implement various filtering and processing operations, but the overall architecture is over-provisioned compared to the network. In contrast, if specialized hardware nodes are used (e.g. network-attached flash) the internal and external bandwidths are better matched, but these nodes will not be able to carry out complex processing near the data without slowing data access down. Our solution, Caribou, proposes a balanced design point: small-footprint hardware nodes that, even though offer high throughput and low latency, are also flexible to adapt to different workloads and processing types without being over-provisioned.

The work presented in this dissertation is not a one-off effort: it provides an extensible and
modular architecture for storage nodes that can be used as a platform for implementing near-data processing ideas for various application domains. The lessons are be applicable for different storage media or networking technologies as well.
Zusammenfassung

Als Reaktion auf die immerzu steigenden Datenmengen werden Applikationen in Datenzentren auf eine grosse Anzahl von Maschinen verteilt. Obwohl dadurch komplexe Probleme gelöst werden können, tauchen jedoch auch Datentransferengpässe, die die Gesamtleistung und Skalierbarkeit limitieren, auf. Durch das Platzieren von Rechenressourcen nahe bei den Daten, können diese Engpässe reduziert werden. In dieser Studie erforschen wir diese Idee im Zusammenhang mit verteilten Key-Value-Speichern.

Effiziente und vor allem skalierbare Verarbeitung von Daten ist wichtig in Datenzentren. Als Antwort auf die Leistungsstagnation in Bezug auf einzel-thread Ausführung, kommen spezialisierte Hardware und hybride Rechnerarchitekturen auf, welche effizientere Datenverarbeitung und -verwaltung versprechen. In dieser Arbeit benutzen wir Field Programmable Gate Arrays (FPGA) um herkömmliche Kompromisse und Limitierungen zu umgehen und um bisher nicht realisierbare Designs zu erkunden.

ein ausgeglichenes Design vor: Hardwareknoten mit kleinem Profil, die nicht nur hohen Durchsatz und tiefe Latenz bieten, sondern auch flexible sind und sich an verschiedene Funktionen und Datenoperationen anpassen können.

Das System, dass in dieser Dissertation präsentiert wird ist nicht eine einmalige Anstrengung: Es zeichnet sich durch eine erweiterbare und modulare Architektur für Speicherknoten aus, die als Plattform zur Umsetzung von weiteren Ideen im Bereich der datennahen Datenverarbeitung in verschiedenen Domänen dienen kann.
Acknowledgments

First of all I would like to thank Gustavo for being a great advisor, guiding me through research with an expert hand. Ken, Mothy and Torsten, thank you for accepting to be on my committee and for providing valuable feedback on how to improve the dissertation. Marko, Michaela, Jens, I am grateful for your help in shaping me as a young researcher.

This work wouldn’t have been possible without our many collaborations with David.

People of the Systems Group, I am also indebted to you. I would like to especially thank Louis, Pratanu, Kaan, Muhsen, Claude, Johannes, Ingo, Ankit, Ce, Jana, Darko, Gerd, Besmira, Cagri, Tudor, Alex, Simonetta, Jena and Nadia.

Last but not least, Nico, family and friends: your unconditional love and support made the last five years worthwhile.
We have to continually be jumping off cliffs and developing our wings on the way down.

Kurt Vonnegut

*If This Isn’t Nice, What Is?: Advice for the Young*
# Contents

1 Introduction 1  
   1.1 Motivation and Problem Statement ........................................... 1  
   1.2 Overview of the Proposed System ............................................... 4  
   1.3 Contributions ................................................................. 6  
   1.4 Dissertation Outline .......................................................... 8  

2 Data Management 9  
   2.1 Background .............................................................................. 10  
      2.1.1 Short Primer of FPGAs ....................................................... 10  
      2.1.2 Introduction to Key-value Stores ......................................... 12  
   2.2 Design Overview ....................................................................... 14  
      2.2.1 Architectural Layers .......................................................... 14  
      2.2.2 Caribou Variants ............................................................... 16  
      2.2.3 Anatomy of a Hardware Hash Table ..................................... 17  
      2.2.4 Hash Function ................................................................. 19  
   2.3 Caribou$_{MC}$ Pipeline ............................................................. 22  
      2.3.1 ASCII Protocol Processing ................................................... 22  
      2.3.2 Lookup table ..................................................................... 24  
   2.4 Caribou$_{CH}$ Pipeline ............................................................... 28  
      2.4.1 Protocol Description .......................................................... 28  
      2.4.2 Cuckoo Hash Table ............................................................ 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3</td>
<td>Slab-based Memory Allocator</td>
<td>33</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Processing</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>Related Work</td>
<td>38</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Key-value stores in FPGAs</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Software optimizations and specialized software systems</td>
<td>41</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary and Discussion</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Distributed Consensus and Replication</td>
<td>45</td>
</tr>
<tr>
<td>3.1</td>
<td>Background</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Hardware Implementation</td>
<td>48</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Networking</td>
<td>50</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Dataflow Architecture</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3</td>
<td>State Machine</td>
<td>55</td>
</tr>
<tr>
<td>3.2.4</td>
<td>State Table</td>
<td>56</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Quorums and Committing</td>
<td>57</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Maintaining a Log</td>
<td>57</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Synchronization</td>
<td>59</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Bootstrapping and Leader Election</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>Related Work</td>
<td>60</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Coordination in Large Systems</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Speeding up Consensus</td>
<td>61</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Quicker and Specialized Networking</td>
<td>63</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary and Discussion</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>Near Data Processing</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Use-case Examples</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>Regular Expression Matching</td>
<td>68</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Design Overview</td>
<td>68</td>
</tr>
</tbody>
</table>
4.2.2 Inside a Single Unit ............................................. 71
4.2.3 Increased Throughput through Parallelism .................. 75
4.3 Conditionals ...................................................... 78
  4.3.1 Design Overview ............................................. 78
  4.3.2 Internal Architecture ....................................... 80
  4.3.3 Throughput Scaling .......................................... 82
4.4 Decompression .................................................... 83
  4.4.1 Design Overview and Algorithm ............................ 83
  4.4.2 Internal Architecture ....................................... 84
  4.4.3 Increased Throughput through Parallelism ................. 85
4.5 Related Work .................................................... 87
  4.5.1 Regular Expressions on FPGAs .............................. 87
  4.5.2 Near-data processing ....................................... 88
4.6 Summary and Discussion ......................................... 89

5 Evaluation .......................................................... 91
  5.1 Experimental Setup ............................................. 92
    5.1.1 Evaluation Board and Deployment ......................... 92
    5.1.2 Caribou Variants .......................................... 95
    5.1.3 Software Clients .......................................... 97
  5.2 Resource Consumption ......................................... 99
  5.3 Get-Put Workloads ............................................. 101
    5.3.1 Baselines .................................................. 101
    5.3.2 Response Times .......................................... 106
    5.3.3 Mixed workloads .......................................... 110
    5.3.4 Scalability ................................................. 117
  5.4 Scan and Processing ........................................... 121
    5.4.1 Pipeline Limits ........................................... 121
Chapter 1

Introduction

1.1 Motivation and Problem Statement

The need for specialized hardware. We are facing a significant challenge in the datacenter: the amount of data that needs to be stored and processed grows faster than the performance of general purpose processors. For decades the performance of CPUs has been increasing conforming to Moore’s law, but today this trend has leveled off. Figure 1.1 shows how the expected data growth is exponential, whereas the single-threaded performance of traditional CPUs has been stagnating for more than 10 years. The figure also hints that in order to change the status quo we need to look beyond traditional processor designs in the future. This is because even though it is possible to add more transistors to a CPU, using these to create additional cores is unlikely to benefit applications (the figure illustrates that even if 95% of the application’s execution can be parallelized perfectly, speedup on multi-cores over a single thread is capped below 20x). As an alternative to adding conventional cores, specialized processing elements can be used to extend the capabilities of CPUs. Thanks to specialization, these handle computation more efficiently and help narrow the gap between data growth and compute capacity. The work in this dissertation focuses on computation on stand-alone specialized nodes, but is part of a larger effort of exploring the use of heterogeneous platforms and re-imagining the nodes in the modern datacenter.

The data movement bottleneck. Most distributed applications separate compute and storage resources from each other into their own layers, managing and provisioning the two independently [11, 79] (Figure 1.2). This way they make better use of resources and can scale, for instance, the number of processors working on a task without adding the
Figure 1.1: There is an increasing gap between data growth and CPU performance increase. This leads to changes in the way we design computers and in how we use distributed systems for data processing.

An equivalent number of hard drives, which would be underutilized or wasted altogether. This separation can be implemented either using logical specialization of nodes [111, 72, 90], in that storage is still provided by regular server machines, or physical specialization [41, 67, 45] where storage nodes comprise of memory or flash directly attached to the network.

Processing can be of many different types, such as map-reduce jobs, distributed databases, machine learning pipelines, etc., but in most applications, storage nodes have similar interfaces and provide similar functionality. Typically, data is organized into key-value pairs allowing quick random lookups. In many use-cases it is important to allow retrieving multiple key-value pairs as a sequence (ordered or belonging to a set) in a single operation. In common deployments, data is replicated across multiple nodes to avoid data loss in the presence failures.

The adoption of scale-out applications and decoupling of compute and storage resources has introduced, however, new inefficiencies related to data movement. Communication and data access that used to take place inside the same physical host is now spread over, potentially, tens of machines in the datacenter. The scalability of data processing applications can suffer both from the bandwidth and latency bottlenecks related to data movement. These can, however, be addressed by moving enough of the compute logic into
1.1. Motivation and Problem Statement

Figure 1.2: Many data processing applications are designed with decoupled resources in mind. By disaggregating resources inside a rack, more flexibility is available in provisioning different workloads.

the storage layer that allows filtering or aggregating the data.

Even though the idea of pushing operators into the storage has been explored over the years both in the context of main memory (active memory) [85, 40, 115], and of disk drives (active disk) [3, 31, 96, 111, 113, 55], distributed storage opens up new challenges. This dissertation applies near-data processing to distributed storage through a co-design approach. As a result, management and processing logic inside the nodes match by design the bandwidth of the network and storage medium.

Redesigning distributed storage nodes. While increasing performance is important goal, with larger and larger distributed systems energy efficiency cannot be neglected either. Datacenters are composed of thousands of nodes, and energy consumption is becoming a limiting factor in large deployments. Overall, it is important that we design nodes that are efficient at the task they perform. This dissertation applies this design goal to distributed storage and explores the following question: How can we implement a balanced point in the design space of distributed storage (Figure 1.3(b)) if we move away from traditional processors and use specialized hardware instead?

In this context, balanced means that data management operations should be performed at the highest rate at which requests can be received over a specific network protocol and processing operations should be carried out such that they are always bound by the bandwidth of the storage, without having to over-provision the compute elements. To avoid repeating the process for different storage media and network types, this work shows how to include several “knobs” in the design that allow for increasing or decreasing the throughput of in-storage filtering performed.
Today, most solutions have un-balanced designs. On the one hand, if regular server machines are used to implement storage functionality (e.g. in Oracle Exadata [111]), there are plenty of traditional CPU cores to execute complex computation and filtering on the data, but these will be over-provisioned compared to the network or storage bandwidth (Figure 1.3(a)). When strictly considering data movement in and out of the servers, both the software stack and the hardware architecture of these nodes will introduce inefficiencies in packet-processing. On the other hand, when building distributed storage starting with the storage medium first (Figure 1.3(c)), for instance as in network-attached drives such as Seagate’s Kinetic HDD, or disaggregated flash [57, 110], compute resources are often under-provisioned. Even though these devices are equipped with small multi-core processors, most of the cycles are dedicated to the actual data access and management, leaving only a fraction available for processing. Recent work by Gunjae et al. [68] explores in depth the behavior one can expect from smart flash devices conforming to this design. They conclude that the frequency and cores of these processors would have to be dramatically increased to achieve end-to-end speedups of even 1.5-2x on database workloads. A risk that appears when using these kind of devices is that by pushing down operators that are too complex, the overall data retrieval rate could suffer, in effect slowing down the entire system.

![Figure 1.3: Caribou represents a balanced design point in which in-storage compute matches the network and storage medium bandwidth.](image)

1.2 Overview of the Proposed System

We use field programmable gate arrays (FPGAs) to create Caribou\(^1\), a prototype of our ideas for distributed storage with near-data processing in a small footprint device. By replacing traditional processors with specialized hardware, we are free to explore different

\(^1\)Project website at https://www.systems.ethz.ch/fpga/caribou
design decisions and evaluate how well different aspects work when moving away from CPU-based processing.

Caribou has been designed as a distributed system and its nodes can replicate individual write operations on their own for fault-tolerance. This functionality is not present in most network-attached flash solutions, but is a usual requirement of any large-scale software key-value store. The ability to perform this in hardware also shows that it is feasible to build complex distributed systems with FPGAs.

Processing near the data is provided as a set of pre-defined filtering operations that can be chosen at run-time and re-parameterized with each request. The offload capabilities of Caribou are reconciled with high throughput access thanks to its internal pipelined design. Latencies are kept low thanks to the low-overhead communication between modules and a design that guarantees constant-time handling for most operations. Networking, data management and data processing are all steps in a large pipeline that can take advantage of the massive parallelism of FPGAs. While related work has already explored each of these aspects separately, in Caribou we show how all three can be combined on a stand-alone FPGA.

Since the adoption of specialized hardware solutions is often hindered by the need of redesigning the systems around it, we made Caribou easy to integrate in the datacenter by using a general key-value interface exposed over traditional sockets (10Gbps TCP/IP). This allows clients written in different languages and running on different operating systems to access it.

In terms of energy efficiency, Caribou benefits from using specialized hardware (in a microserver footprint). Most of the general-purpose components present in regular server machines are removed and what is left are the essentials for providing a distributed storage service. This saves on power consumption while offering high throughput. Caribou’s energy efficiency (operations/s/Watt) is on a par with highly optimized but bare-bones software approaches [75], but the FPGA has an order of magnitude smaller energy footprint than a CPU. The exact amount of power saving per node depends on the type of storage medium, capacity and type of CPU replaced, but as we show in Section 5.9, in cases where the power budget for storage is less than that of computational resources, a specialized solution will be beneficial.

For storage we rely on DRAM, as a proxy for emerging non-volatile memory technologies and NVMe flash. However, the throughput of the computing elements inside Caribou are
not limited to a particular choice of storage and can be adjusted to slower or faster targets as well. As later explained in more detail, we included several “knobs” in the design to adjust the rate at which circuits can process data, and this allows us to balance the computational resources inside Caribou with the storage bandwidth. The ability to scale compute up or down is not fundamentally different from how software behaves, but in software this usually happens through the addition of cores and there are clear overheads when extending a single-threaded application to run on more than a single core, or single socket. On the one hand, communication latencies between threads will grow, and on the other hand, the application’s design might need to be changed to take advantage of the parallelism. Scaling up/down a hardware-based application on a single node is a more efficient process.

The work presented in this dissertation, however, faces an inherent challenge of all FPGA-based designs: there is a strict tradeoff in terms of maximum supported computation complexity and desired throughput. The reason for this is that all code needs to be placed on the device, even if it is not active all the time, and the chip area will limit the degree of parallelism or complexity of operations supported (conversely, in a software system there is no penalty for seldom executed code paths). In Caribou we address this tradeoff by investing more resources in common-case behavior of the nodes than in fallback/exceptional behavior (e.g. favoring reads over writes, throughput with no faults in replication over that of recovery).

1.3 Contributions

Caribou is a specific implementation of a distributed key-value store, but the ideas and design methodology behind it have relevance for future systems as well. We investigated what is needed to move a design beyond a proof-of-concept phase so that it is usable in real-world applications. We showed that it is possible to build a specialized hardware device that 1) offers software-like interfaces and features, 2) does so over traditional networking, 3) has high throughput and low latency for most workloads and 4) has a design that uses network and storage bandwidths efficiently.

In terms of extending the state of the art, Caribou makes the following contributions:

- Building distributed systems with FPGAs is an emerging topic, and aspects such as replication done entirely in hardware, or managing storage without the help of
1.3. Contributions

A host computer is missing from most related work to date. Caribou shows that it is feasible to provide these features on an FPGA and, in the case of the replication protocol, even offer better performance than software using commodity networks.

- In contrast to previous FPGA implementations that often make strong assumptions about key and value sizes, data management in Caribou favors functionality over raw performance, while still ensuring that data management logic matches the network bandwidth. Due to this approach we can provide multiple access methods to the clients and implement features such as memory allocation, that ensures that the FPGA can manage a wide range of value sizes and large amounts of storage. This makes the solution portable to different devices and different storage media.

- In terms of processing, Caribou, on the one hand, follows typical FPGA design techniques to architect data-parallel circuits that are easily scaled up or down to match the storage bandwidth. On the other hand, it departs from hard-coded designs and shows that it can be acceptable to use flexible circuits that, even though require more resources than fully hard-coded ones, are parameterizable at run-time. This makes it possible for many different workloads to benefit from the compute capabilities deployed on chip. The size of FPGAs is still steadily increasing, therefore being frugal with logic resources at the expense of flexibility is not a good design decision for use-cases such as ours.

Related Publications  This dissertation is based mainly on work that has also been presented as part of the following three publications:

- **Consensus in a Box: Inexpensive Coordination in Hardware.** by Zs. Istvan, D. Sidler, G. Alonso, M. Vukolic. in NSDI’16 – This work explores building a consensus algorithm in hardware, and provides a simple key-value store as a back-end. The system acts as a drop-in replacement for memcached. It also explores opportunities related to using a dedicated backbone interconnect between nodes in addition to the regular cluster network.

- **Runtime Parameterizable Regular Expression Operators for Databases.** by Zs. Istvan, D. Sidler, G. Alonso. in FCCM’16 – One of the filters used in Caribou implements regular expressions and this module has also been evaluated in the context of hybrid database engines and database acceleration. This publication covered
Chapter 1. Introduction

the internals of the regular expression matching unit, which was later integrated into Caribou [52] and DoppioDB [100].

• Caribou: Intelligent Distributed Storage. by Zs. Istvan, D. Sidler, G. Alonso. in VLDB’17 – We built a complete system by redesigning the hash table to offer lossless storage (as opposed to the memcached use-case), a memory allocator, and secondary indexes for scanning through values directly inside the storage node. This work also explores the performance of a multi-node setup and various processing scenarios.

1.4 Dissertation Outline

The dissertation is organized as follows:

In Chapter 2, we focus on how data, in the form of key-value pairs, is managed inside Caribou, looking at data structures and algorithms used.

In Chapter 3, we present how atomic broadcast, and replication on top of it, can be implemented in specialized hardware.

In Chapter 4, the different types of processing implemented in Caribou are presented: decompression, conditional matching and regular expressions.

The system is evaluated through experiments, both as a whole, and each component separately, in Chapter 5.

Chapter 6.1 summarizes the dissertation. The future opportunities that open up thanks to Caribou are discussed in Chapter 6.2.
Chapter 2

Data Management

The main purpose of distributed storage is to manage data and expose it over the network to clients. For this reason, we begin this dissertation by looking in depth into the data management aspects of *Caribou*, by explaining how clients can access the data stored on *Caribou* nodes and by exploring the trade-off of different access protocols.

*Caribou* implements a storage model that organizes data into key-value pairs. One of the main challenges is that the underlying random access data-structure on the FPGA has to be able to guarantee network-bound throughput in order to achieve an overall balanced design. Furthermore, since *Caribou* nodes have to manage gigabytes of storage, we show how memory allocation can be implemented in hardware such that it provides flexibility to support different workloads on the same device. Finally, the data management logic is extended with a secondary access method: in addition to random single-key lookups, clients have the ability of retrieving a large number of key-value pairs with a “scan” operation that leverages the metadata of the memory allocator.

Overall, what this chapter demonstrates is that it is possible to build stand-alone storage nodes with FPGAs without requiring assistance from host machines. Based on the lessons learned while building *Caribou*, we present two different hash table designs that offer different trade-offs. However, both designs achieve high throughput, matching the bandwidth of the network, and can be easily adapted to different network speeds and hardware platforms.

The chapter is organized as follows: first a background on FPGAs and key-value stores is presented, then a high level overview of *Caribou* and its building blocks follows. The largest part of the chapter is devoted to the two hash table and pipeline variants inside
Chapter 2. Data Management

Caribou. Finally, related work is discussed.

2.1 Background

2.1.1 Short Primer of FPGAs

FPGAs are hardware chips that can be reprogrammed and, once programmed, deliver similar properties as application specific integrated circuits (ASICs) but have lower clock rates. For network-facing applications they are usually clocked in the 150-300MHz range.

Internally they are a collection of small look-up tables (LUTs), on-chip memory (BRAM) and specialized digital signal processing units (DSPs), which can be configured and interconnected in such a way to implement any hardware logic on top (Figure 2.1). All elements of the FPGA execute in parallel and are tightly connected, therefore, data processing pipelines can be implemented without communication or data transfer overhead between stages. For a more detailed background on FPGAs and examples of their use in data-processing applications, please refer, e.g., to the book by Teubner & Woods [106] on the topic.
2.1. Background

There are two types of parallelism applications take advantage of in FPGAs. First, pipeline parallelism means that complex functionality can be executed in steps without reducing throughput. This type of parallelism is relatively hard to exploit in software, but it is possible by building software pipelines, or using the different cores of a multi-core machine to execute different stages. One important benefit of the FPGA is that the communication between pipeline stages can be done directly, by “wiring them up”, or, in case some buffering is needed, using low-latency on-chip FIFO buffers. The second type of parallelism that is often exploited on FPGAs is data-parallel execution of the same logic. This is similar to SIMD (single instruction multiple data) processing in CPUs, but it can also implement a SPMD (single program multiple data) paradigm if the operations are more complex. What makes FPGAs very interesting prototyping tools is that these two types of parallelism can be combined even inside a single application module to provide both complex processing and scalable throughput.

In this dissertation we exploit both types of parallelism because internally, Caribou is implemented as a large pipeline, with each stage consisting of another pipeline or a network of pipelines. Most data processing units (hashing, regular expression matching, etc.), on the other hand, rely on data parallel execution to increase throughput to the desired levels. Thanks to the tight integration between memories and logic on the FPGA, it is possible to distribute work items into multiple FIFOs or collect the results with virtually no overhead.

One aspect of current FPGAs that limits to some extent the usable real estate is that both the network endpoints and the DDR memory controller are implemented in the logic fabric. This means that, instead of residing outside the FPGA, they consume resources on the device. In future systems, at least the memory controller is expected to become “hardened” which will free up resources for the applications. The Xilinx MPSoC device line\textsuperscript{1} is, for instance, a step in this direction.

Overall, the energy footprint of an FPGA-based application is proportional to its “size”, i.e., the number of programmable elements it uses, and the operating frequency of the device. For larger (but not the largest) chips, such as the Xilinx Virtex7 XC7VX690T, that we used in this dissertation, an energy consumption in the range of 10-20W is expected for network-facing applications. This number is an order of magnitude less than that of server-grade CPUs.

\textsuperscript{1}https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html
2.1.2 Introduction to Key-value Stores

In contrast to relational databases that store data organized as records inside tables that have a strict schema, key-value stores (KVSs) handle records based on their identifier (key) and store them as they are, without a schema (often as an opaque value). It is the task of the application in most cases to define what the internal structure of a record is. Due to the simplicity of the key-value model, KVSs typically provide low latencies, high throughput and good scalability.

Even though key value stores have been in use for a long time, it is with the appearance of the cloud and the emergence of scale-out applications that KVSs proliferated. It is at the same time that NoSQL (“not only SQL”) databases started appearing, most of them being key-value stores. To name just a few: MongoDB\(^2\), Redis\(^3\), Cassandra\(^4\) and Zookeeper\(^5\). Their use ranges from caching relational query results, through meta-data storage, to replacing relational databases as a main data store (Figure 2.2).

![Diagram of key-value stores in the datacenter, especially in web-facing applications.](Image)

Figure 2.2: Key-value stores are used in many different settings in the datacenter, especially in web-facing applications.

One differentiating factor among key-values stores is the choice of data structure that holds key-value pairs. Usually this is either a hash table [38], a tree [43], or a log-structured merge tree (LSM Tree) (e.g., in LevelDB\(^6\)). In this work we focus on read-heavy workloads, which favors hash tables as underlying data structures because they offer good random access properties. In case the workloads were more write-heavy, a log-structured setup would have been the sensible choice.

---

\(^2\)https://www.mongodb.com/
\(^3\)https://redis.io/
\(^4\)http://cassandra.apache.org
\(^5\)https://zookeeper.apache.org/
\(^6\)http://leveldb.org/
2.1. Background

Figure 2.3: Internally, keys can be organized either as a flat structure or as a hierarchy.

As for choice of data storage medium, while some systems use disks to persist the data (e.g., Zookeeper, Couchbase, Redis), many solutions such as Memcached [38], Redis, Ramcloud hold data only in memory. Even from those which persist data on disk, by default most of their operations will be performed on in-memory cached data. While persisting to disk ensures durability, it can also be ensured for main-memory key-value stores through the means of replication. In Caribou we rely on this method to make sure that there is no data loss when failures appear, but in the future it would also be possible to persist the replication log (like a journal) to a disk on the side.

Ordering (or indexing) the data is done in some cases to speed up secondary access paths. For instance, if keys were to represent dates, being able to retrieve a whole range of the data instead of asking for keys belonging to that range one by one will lead to application speedup. While maintaining the data structures that allow scan or range-based access are expensive, in some use-cases they are necessary. One such example is if the KVS is used as storage for relational data for a distributed database running on top [70, 90].

Client Interface. Even though they all implement different communication protocols, most key-value stores expose the following abstract interface for managing the data:

- Insert(K, V) – This operation inserts a key-value pair, if the key does not exist yet.
- Set(K, V) – This operation overwrites the value belonging to the specified key. Depending on the implementation of the KVS, Insert and Set can also be merged into a single operation.
- Get(K) – Retrieves the value belonging to a key.
Chapter 2. Data Management

• **Delete(K)** – Deletes a key-value pair. Depending on the KVS implementation and its use-case, key-value pairs can be deleted if, for instance, storage is running low (e.g., in Memcached).

In this dissertation we will also discuss the following operations:

• **STC(K,V, version)** – The “store conditional” operation compares the provided version number of the key with the one in the KVS and only sets the value if these are equal. The number is incremented after after set operation, and the STC operation only succeeds if there has been no writes to the value since the last read. This operation helps implement atomic read-modify-write operations on a single key.

• **Scan(Collection)** – This operation retrieves all key-value pairs belonging to a collection. The collection could be implicit (in which case all data is returned) or explicit in which case only a specific subset of keys qualify.

• **ScanCondition(Collection, Expression)** – Same as above, but apply a filtering expression on the values before sending them to the clients. Those values which do not fulfill the condition are dropped.

• **GetCondition(K, Expression)** – Performs a get operation but the value is checked against the provided expression. If it does not match, it is not sent to the client.

### 2.2 Design Overview

#### 2.2.1 Architectural Layers

Figure 2.4 shows a high level view of the components that make up Caribou and abstracts away the idiosyncrasies of our prototyping platform. Logically, functionality is composed of four layers: networking, replication, data management and processing. Even though, as later shown, this representation abstracts away a lot of implementation details and how the different modules are connected to each other, it is nonetheless useful for describing the high level functioning of Caribou.

The *Network processing* layer is responsible for communicating with clients and with other *Caribou* nodes. It can implement multiple communication protocols (e.g., TCP/IP, UDP,
2.2. Design Overview

A Caribou node is composed of five main building blocks that can be modified independent of each other.

or bare Ethernet) and has been designed explicitly for 10Gbps networking. Inside this layer there is no knowledge about the application-level protocol and from its perspective, data transmitted in network packets is opaque. Hence, even though from an integration perspective, there is no overhead between this layer and the lower ones, they are logically independent and decoupled.

Client-facing traffic could, in principle, run on either UDP or TCP. Because the former introduces less overhead in hardware and is overall cheaper, it is often the go-to choice for FPGA-based key-value stores [14, 15, 103]. TCP, however, ensures reliable ordered communication between clients and storage, and more importantly, between storage nodes when replicating data. With the emergence of high session-count TCP/IP stacks for FPGAs, it is now possible to use the same socket-based abstractions of the network as we are used to do in software. As a result, in this work we mainly focus on TCP.

The Replication layer takes care of disseminating changes performed at the leader node to all other participants. In key-value terms this translates to replicating operations that alter the storage contents, e.g., writes, deletes. Chapter 3 provides an in-depth view of this layer, but conceptually, while in the process of replication, messages are trapped inside this layer and are only passed further down once a consensus among nodes has been reached. While this layer needs to partially parse the requests to understand what constitutes a write, it is mostly agnostic to the rest of the request and treats it as a BLOB for the
purpose of replication.

The *Hash table* constitutes the part of *Caribou* that manages keys and organizes the metadata necessary for accessing values. This layer is custom-designed for the particular key-value store protocol (API) in mind and will parse all messages into an internal format. The choice of the specific instance of hash-table, or other key-management data structures, are driven by the high level requirements. The data structures chosen will differ if, for instance, the goal is to minimize latency, or to maximize throughput for write-intensive workloads. In this Chapter we present two design options, one for low-latency caching scenarios, and one for general purpose storage that needs to handle efficiently both random access and scan-based access to the data.

*Value management* refers to logic that is needed to allocate and deallocate memory for values and to write them to storage. This layer is both physically and logically coupled with the Hash Table layer. In case *Caribou* would be extended to hybrid memory (e.g., smaller capacity but faster storage for small objects, and slower but very high capacity for large objects), this layer would be the only one that would have to be modified. This ensures that *Caribou* is extensible in the future.

*Processing* happens on the path that retrieves data, before sending it to clients. In Chapter 4 we provide an in-depth view of the different operations implemented, but at a high level, this layer is designed as a stream-processing unit. The filtering operations see a steady stream of values, regardless whether these are read as a result of many individual random access operations, or as part of a full scan on the storage.

### 2.2.2 Caribou Variants

*Caribou* is a project that has evolved over several iterations but has its origins in an effort to provide low-latency, energy efficient, replacement for memcached. However, while exploring this idea, it has become clear that today’s storage requires complex filtering options and mechanisms such as replication for fault tolerance or high availability are indispensable in the datacenter. As a result, the more recent version of *Caribou* targets a general-purpose use-case, not just caching, and focuses on offering all these additional services without slowing data retrieval down.

In this dissertation both variants of *Caribou* are covered, with slightly more emphasis on the more recent one, *Caribou_{CH}*, because this offers more functionality than *Caribou_{MC}*. The feature listing of the versions is as follows:
2.2. Design Overview

**Caribou\textsubscript{MC}** This version uses the ASCII protocol of memcached. Internally, the key-value store utilizes a lossy hash table design with a fixed memory allocation strategy to minimize latencies both for reads and writes. This setup has either a single TCP interface, or one TCP and up to three dedicated point-to-point ones to explore the benefits and potential drawbacks of a dedicated backbone interconnect for replication traffic.

**Caribou\textsubscript{CH}** This version uses a more streamlined binary protocol and a Cuckoo hash table that offers lossless data storage, at the cost of lower write-only performance. It also has proper memory allocation based on slabs and it maintains bitmap indexes to help with scans. It communicates with clients and other Caribou\textsubscript{CH} nodes via a single TCP interface.

![Diagram of Caribou variants](image)

*Figure 2.5: The two Caribou variants presented in this dissertation.*

2.2.3 Anatomy of a Hardware Hash Table

*Caribou* implements a key-value store based on a hash table. In line with software related work, e.g. [5], data is organized in the following way: keys reside in the hash table and have pointers to the memory area where values are stored (Figure 2.6). While keys can be of variable size, they range in the 4-16 B range, with values spanning a wider spectrum. The benefit of separating key and value management is that different methods can be used for each. Furthermore, in a hardware-based design, the hash table and value access can be treated as different stages of a large pipeline.
Chapter 2. Data Management

Figure 2.6: Many key-value stores store keys and values separately for better flexibility in management.

Unless a hash table is built exclusively with on-chip BRAM memory (which would limit its size to a few megabytes), it needs to be designed around off-chip DDR memory access. If the goal is to implement a hash table that is not bottlenecked by memory access latency, it has to be pipelined. Through pipelining the memory access latency can be hidden and the performance of the hash table will only be limited by the random access rate of the memory or its bandwidth. Regardless of desired throughput and choice of hash function, hardware hash tables can be seen as composed of four steps (Figure 2.7): 1) hashing the key to determine its location in the table, 2) issuing a read command to this location, after the data has been brought in from memory 3) determining how to modify it, and 4) writing back the modified data, if necessary. Related work, such as the group-by aggregation in IBEX [113] or FPGA-based key-value stores by Xilinx Labs [15], and various hash table designs [108, 74, 50] follow a similar high level architecture.

Figure 2.7: All pipelined hash tables in hardware are composed of four main steps. The choice of, e.g., hash function or collision resolution strategy results in a specific type of hash table.

One challenge of building such a pipeline is the presence of read-after-write hazards in memory. Since multiple keys can reside in the pipeline at the same time, it is possible that the changes written by the ones leaving the write-back step are not yet be visible in memory when others issue read commands. To ensure that data is not corrupted through such a hazard, the hash table design must use some method of avoiding it. This can
be achieved either by delaying keys that would read memory locations currently under modification, or by “back-propagating” changes in the pipeline.

As part of this dissertation I worked on two different hash table designs for FPGAs, which employ different data hazard avoidance techniques. The first, that will be referred to as the “caching” hash table (Caribou\textsubscript{MC}), has been designed to handle workloads where the key-value store is not the primary holder of the data and focuses on reducing response times as much as possible. In an effort to save resources, it avoids data hazards by delaying writes to the same memory address (it does not reorder operations, however). Reads are allowed in without delay even if the underlying data is being modified, because they will not see corrupted memory lines. In the worst case they will read the previous state of the entry, and, in caching scenarios, this might be preferred over increasing the latency of reads. From a client’s perspective, who sends sets and gets to the same key asynchronously, this behavior might lead to an outcome as if the requests would have been reordered. If this is not acceptable, the hazard avoidance logic could be made more restrictive, but this would potentially reduce throughput in the presence of writes.

We remove this tradeoff in the hazard avoidance logic of Caribou\textsubscript{CH}. That solution ensures that no keys are delayed. This is achieved by writing memory updates both to the memory and a writeback cache at the same time. This cache works like a circular buffer, that holds the modifications to memory addresses that are being written. If after receiving data from memory the same address is also present in the writeback buffer, that version is taken and the one read from memory is discarded. Since in Caribou\textsubscript{CH} maximum key sizes are smaller than in Caribou\textsubscript{MC}, in the hash table each “bucket” maps to a single memory line. This makes caching and writing updates easier.

Both of the above mechanisms will be presented in more detail in the coming sections describing Caribou\textsubscript{MC} and Caribou\textsubscript{CH}.

### 2.2.4 Hash Function

There is a rich body of related work in FPGA-based hashing, ranging from cryptographic hash functions such as SHA and its variants [95], to lightweight hashing schemes, such as simple tabulation or murmur [61]. For Caribou we formulated the following requirements: 1) the hash function needs to produce high quality, but not necessarily cryptographic, output so that hash collisions are avoided, and 2) it works on variable sized input without
a large initialization overhead. Given these conditions we implemented a multiplication-based function, shown in Listing 2.1, similar to a Knuth’s hash function.

The hash function consumes two bytes every two cycles, which translates to a throughput of 1B/cycle. Since the circuit is clocked at 156.25 MHz, a single hash function module can reach only 156 MB/s. To meet the line-rate requirements we rely on the parallelism of the FPGA and create multiple “hashing cores” that process the input in a data parallel manner, in the order of arrival (Figure 2.8). Even though the network rate is the same for all Caribou variants, this translates to different maximum input rate in terms of keys depending on the protocol used. The hash module can be deployed with a variable number of parallel engines, and over-provisioning it does not really bring benefits. If the keys are fixed size, or small, the latency saved by having multiple units work in parallel is negligible. For instance, in cases such as CaribouCH, assuming a value size of at least 16 Bs, even a single unit is enough to meet demand. In the case of CaribouMC, where keys can be larger than 100 B, adding multiple units will significantly reduce latencies inside the pipeline.

An earlier version of this hash function has been presented in [50]. In that work we explored the quality of hashed values and compared the hash function to simpler, CRC-based one, and a much more complex function, used in memcached. The results show that this multiplication-based function represents a good midway points in terms of robustness and resource consumption. Please refer to [49] and [50] for more detail.

Listing 2.1: The hash function used din Caribou is based on multiplication and addition. For the hardware implementation 32bit multiplications are replaced with two 16bit ones.

```c
uint32 MultHashFunc(uint8* key, int len)
{
    uint32 mconst = 47303; // large prime number
    uint16*wkey = (uint16*)key; // processing happens in 16bit steps
    uint32 hashvalue = 0;

    for (int i = 0; i < len/2; i++, wkey++) {
        hashvalue = mconst * hashvalue + *wkey;
    }

    // In hardware each iteration is computed in 2 cycles to avoid 32bit multiplication:
    // 1: hashvalue_int = mconst * ((uint16)hashvalue) + *wkey;
    // 2: hashvalue = (1 << 16) * (mconst * (hashvalue>>16) ) + (uint16)hashvalue_int;

    if (len%2==1) {
        uint8* leftover = (uint8*)wkey;
        hashvalue = mconst * hashvalue + (uint32)*leftover;
        // The above multiplication is broken up the same way in two cycles.
    }
}
```
2.2. Design Overview

```python
hashvalue = RotateRight(mconst * hashvalue, 4);
return hashvalue;
```

Figure 2.8: Multiple hash units can work in a data-parallel way to reach the desired bandwidth or to reduce latencies when processing long keys.
2.3 *Caribou*$_{MC}$ Pipeline

This version of the hash table is mostly inspired by my previous work done as part of my Master’s thesis [49]. The key-value store pipeline has been ported from the Maxeler Workstation\(^7\) platform to the Xilinx VC709 evaluation boards. This entailed rewriting most of the logic interfacing the memory and designing the logic handling the values from scratch. Since a large part of the design of this pipeline is prior work, it will be presented only shortly, covering the main design decisions so that it can be contrasted with the Cuckoo hash table later.

As seen in Figure 2.4, replication is performed by a module that is independent of the hash table. It handles writes requests to the key-value store as BLOBs and is mostly agnostic to request contents. This has the benefit that the hash table implementation, and even the protocol, can be changed without modifying the replication control logic. Replication is presented in more detail Chapter 3.

![Figure 2.9: Building blocks of *Caribou*$_{MC}$](image)

2.3.1 ASCII Protocol Processing

For this hash table we chose to support the protocol of memcached which is based on an ASCII encoding. This means that request are sent as strings. For instance, below is a set request that sets the value belonging to key “somekey” to a 9 B value “somevalue”. The tuple never expires (expiration=0) and the flags stored with the key are “1234”:

```
set somekey 0 1234 9 somevalue
```

While this command format is very easy to parse by humans, it introduces several challenges for a hardware-based design. First, the length of the key is implicit and depends on

\(^7\)https://www.maxeler.com/products/desktop/
the position of the second whitespace. Second, it is not possible to determine where is the end of a request in a data stream unless the request has been fully parsed (other protocols that encode key and value lengths explicitly would allow this). Finally, the conversion from ASCII-based decimal numbers to a binary representation requires relatively complex logic. This is further detailed in the Resource Consumption section 5.2.

On the flip side, however, being able to handle requests in memcached’s format means that all benchmarking tools and libraries targeting memcached can be seamlessly used with the FPGA-based key-value store as well. This allows the FPGA to become a drop-in replacement for memcached.

![Figure 2.10: ASCII commands are parsed in a pipeline.](image)

Requests arrive at the memcached pipeline stripped of all networking related data, and tagged with the socket ID that identifies them on the return path in the networking module. Each request is parsed in the decode step to determine the type of operation that needs to be performed, and to extract the key, value, and flags provided in the request. For this task we used a pipeline that converts from ASCII to an internal representation (transforming among other things the length of the value provided in ASCII form to a 32bit binary number). Any malformed requests are dropped at this point.

Figure 2.10 shows how commands enter the parsing module on a single input, and are broken up into three different outputs: key, value, and metadata. The content of the

---

8This module is based on the one we used in the project my Master’s thesis was part of [14].
metadata will determine whether a command has a value attached to it or not. For this reason, the module reading from the parser will have to dequeue data in the right order. Internally, each possible field of the memcached request is decoded in a separate pipeline stage. Data that has been already “used” by a previous module can be ignored in subsequent steps. When a request reaches the value decoding step, it has been fully parsed and no more output is necessary.

On the other end of the Caribou_{MC} pipeline there is an ASCII Writer module that formats the results of operations, e.g. the status of a set or delete, or the value retrieved in a get, into a valid memcached response. This data is then sent to the network module for transmission. The steps employed for constructing a response packet are similar to parsing, but somewhat simpler because all necessary metadata is given (e.g. length of key and value).

The processing pipeline preserves the socket ID of each incoming request as part of the meta-data, so that the response can be sent back over the same socket to the client.

### 2.3.2 Lookup table

The hash table that we implemented relies heavily on parallelism as a way of avoiding loops and branching in logic. It is a common design decision in hardware hash tables to trade-off higher bandwidth and memory utilization in exchange for constant time or mostly constant time answers [30, 51, 8].

![Diagram of lookup table](image)

Figure 2.11: Buckets are preallocated in hardware with a fixed number of slots. Collisions beyond these slots are not resolved.

We implemented a hash table that solves collisions by chaining, and chains of 4 entries are pre-allocated for each hash table address. These entries are laid out in memory such that
they can be read in parallel. As depicted in Figure 2.11, each entry consists of a header and the actual key, whereby the header contains the length of the key, flags and a pointer to the value. For each incoming key it is enough to read from the hash table proportional to the key length. Since the length of the key is encoded in the header in memory, it is not possible to falsely match a lookup key that is a prefix of the key stored in memory. They have to be equal in length as well.

We limit the size of keys to 112B, which is more than enough for most common workloads [6]. We arrived at this number because when combined with the header this yields 512B blocks for each 4-bucket address on our 64B wide memory interface (Figure 2.12). Entries span multiple memory lines, with the header of each entry residing on the first line. Then, each consecutive memory line contains 16 bytes of each key.

Hash collisions on inserts are solved in parallel inside the same bucket for up to 4 locations. If all locations are already occupied, the oldest of the keys will be evicted and replaced by the input one. This results in a lossy behavior that favors more recently added keys.

Figure 2.12: Each bucket has four slots and each slot spans at most 8 memory lines.
Chapter 2. Data Management

The main target of hash table in Caribou_{MC} were caching workloads, in which cases the storage is not assumed to be lossless in the first place (e.g. memcached does not guarantee not to evict items even if there is space in memory).

Nevertheless, as I studied it in detail in my Master’s thesis, on a previous iteration of the hash table design, the percentage of such “overwrites” remains negligible as long as the hash table is not utilized at a high load factor (<50%) \[49\]. This is in line with the results of related work, e.g., \[30\], and it must be noted that since values are expected to be at least an order of magnitude larger than keys \[6\], trading memory utilization inside the hash table (not for the whole memory) in exchange for predictable response times is a reasonable trade-off.

Clearly, the above described assumptions about the workload and the expected behavior of the key-value store are not applicable to all use-cases. To address more general use-cases, in Caribou_{CH} we explore how collisions can still be solved without slowing down the pipeline, but this requires a different hash table design altogether.

**Hazard Avoidance** In order to avoid read-after-write conflicts in this pipeline we used a small data structure inside the Read module to hold all memory addresses that have outstanding write operations in the pipeline. Addresses are removed from this small data structure when writes have finished in memory. Keys whose address is present in this filter data structure are stalled.

The advantage of this design is that it requires very little resources on-chip (<1000 slices and no BRAMs) and as long as the workload is not write-heavy stalls will be rarely experienced. A detailed exploration of this effect is to be found in \[49\], but at a high level in case the workload reaches a write percentage higher than 10% and comprises of a set of hot keys smaller than 500, line-rate processing can not be achieved anymore.

**Support for Store Conditional** In some workloads it is beneficial to offer atomic operations in storage that avoid repeated round-trips to clients. One such operation could be, for instance, an increment or decrement operation on the value, but this would require the key-value store to associate types or even a schema with the value. The type of operation we added to the hash table ensures that the state of the key that is being modified has not changed compared to its last read version by the client.
As per memcached’s protocol, there is a field called “flags” associated with each key-value pair, that could be used, for instance, to encode the data type of the value, or other metadata. This is optional, and clients can decide how to use the field. This flag is opaque to the server, and upon a get operation it returns both the value and the flag to the client. We implemented the store conditional (stc) operation by re-purposing the flag field of entries to represent the virtual version of the tuple. Then, the client constructs the stc operation as a set, but inserts the old version into the request where normally the expiration time would be, and the new version where the flags would be. If the version of the tuple has changed in the meantime, the operation will fail and the client has to retry it.

The following illustrates the syntax of a stc operation:

\[ \text{stc} \ <\text{key}> \ <\text{version-nr}> \ <\text{new-version-nr}> \ <\text{value-length}> \ <\text{value}> \]

Since in the hash table checking the version number requires neither additional compute, nor additional memory accesses, and the command structure is the same, this operation behaves exactly as a set from all perspectives. The only difference is that while a regular set will never fail in the hash table (an old tuple will be removed if the bucket is full to make place), the stc operation can return failure if the versions have changed, or the key has been deleted.

**Memory Management**  The work in my Master’s thesis required a host machine to be present for the FPGA to be able to allocate and de-allocate memory. In that setup pointer lists were communicated through PCIe between a helper thread in software, and the FPGA that handled all other operations. In the case of the stand-alone nodes in Caribou\textsubscript{MC}, on the other hand, memory management had to be carried out on the device. The chosen method was a naive one, dividing the memory into equal chunks and assigning each entry of the hash table to one memory chunk. The mapping is determined through simple pointer arithmetic. While this design introduces no overhead in the processing it has two drawbacks in terms of memory utilization.

First, the overall memory usage is directly proportional to the fill rate of the hash table, hence it will never reach 100%. Second, the maximum value size is a function of the total capacity of the hash table \( \text{max size} = \text{value} - \text{area size} / |\text{hashtable}| \), and the part of the chunk not used by the value is wasted. While these properties were acceptable for a proof-of-concept system, if one would attempt to use the same hash table for general
purpose storage they lead to poor memory utilization. The next section will show how we implemented memory allocation inside \textit{Caribou}_{CH} to overcome this limitation.

### 2.4 \textit{Caribou}_{CH} Pipeline

![Diagram of CaribouCH node](image)

Figure 2.13: High level view of a \textit{Caribou}_{CH} node.

#### 2.4.1 Protocol Description

In this version of \textit{Caribou}, our goal was to provide a general purpose service, and an associated protocol that would allow later extensions easily. For this reason we moved away from the ASCII-based protocol in \textit{Caribou}_{MC}, and used a simpler, binary one. This is easier to create and to parse as well. Each message sent to \textit{Caribou}_{CH} starts with a fixed size header (16 B) that encodes both details related to the key-value store and the replication protocol. Then, an optional key field and value field follow. In our prototype we handle up to 1KB values, but the protocol allows for larger value sizes as well.

The protocol has not been designed to be the absolute smallest overhead possible but instead we created it with flexibility and convenience in mind. Especially for adding new operation types, this binary format is preferred in contrast to an ASCII type of protocol where a new parser would need to be written for each new command keyword. The detailed structure of the protocol is described in Appendix 6.2.

#### 2.4.2 Cuckoo Hash Table

**Algorithm** Cuckoo hashing, first described by Pagh et al. [88] is a scheme that instead of chaining items, uses reorganizations of the hash table to solve hash collisions. In a
cuckoo hash table every item has two possible locations, determined by two different hash functions (shown as the two colorful arrows in Figure 2.14). When a key is looked up, it has to be in either one of its two locations, otherwise it is not in the table. An insert succeeds if one of the two locations are still free, or already contains the key. If upon an insert both locations are occupied, a randomly chosen key in one of these locations is “kicked out”. The kicked out key will be reinserted using the same two-location method, and the process will continue recursively until all keys have found a spot. With high fill rates and pathological cases it is possible that the kickouts form a cycle that can not be solved. The solution to this problem, as proposed by the original algorithm, is to rehash the whole hash table with two new hash functions. We defer the implementation of this functionality for future work and mitigate the chance of a cycle happening by allocating memory for the hash table such that even when the value store is full, it is not utilized beyond 50%.\footnote{Based on the original Cuckoo hashing paper \cite{CuckooHashing}, when loading the hash table to less than 50\% reorganizations should not be needed}

The way collisions are handled in cuckoo hash tables ensures that reads take constant time and writes take amortized constant time. This property is desired for a hardware-based implementation as well, however, handling kick-outs introduces a challenge. In software this can be done in a loop, and thanks to the high clock frequency of CPUs, without significantly slowing processing down. In hardware, on the other hand, iterative logic would introduce long stalls into the pipeline. As a solution, in our cuckoo hash implementation kick-outs are handled in parallel with the regular request processing.

Figure 2.14: In a Cuckoo hash table every item has two locations, determined by two different hash functions. If both of them are taken, one of the two occupying keys is moved to its second location. This continues until every key has a free slot.
**Hardware Layout**  Figure 2.15 shows the block diagram of the hardware cuckoo hash table. Incoming keys are hashed and the hash value is used to request the corresponding hash table bucket from the memory. We modified the Hash module to compute two different hash functions on the key. This was achieved by creating two copies of the function in Section 2.2.4 with different magic numbers.

In order to reduce reorganizations on collisions, each location in the hash table is a “bucket” with three slots. The size of the bucket is a function of memory line width (64 B on our prototype platform) and the maximum key size chosen (16 B). Additional metadata is stored for each key, containing the length of its corresponding value and a pointer to the data area of the memory. Larger keys could be stored only at the expense of “bucket size” because the hash table requires every location in the hash table to map a single DRAM memory line.

When inserting a new key, if all three slots at both its memory locations are taken, one is selected at random and re-queued on the connection on top for a re-insert. In parallel, it is also saved in the on-chip data structure called “Kicked Keys” to be able to serve gets or updates targeting this key while it is reinserted. Note the presence of a second Hash module on the reinsert path. This is needed because for kicked out keys the other memory location needs to be computed. It is not stored in memory because this would lead to wider meta-data and, as a result, less slots per memory line. The re-computation of the hash value adds several cycles of latency to the feedback loop, but has not detrimental effect on the rate at which keys can be “recirculated”.

The additional background traffic for failed sets will require additional bandwidth to solve. The is estimated to be at most two additional hash table locations (2x64 B) for highly loaded cases [88], but since CaribouCH targets read-heavy workloads, the overall bandwidth requirement will increase more modestly and will be dominated by the read operations. These will either incur two read costs (if both locations are read at the same time) or between one and two depending on the fill rate (if the locations of the key are read iteratively). To give a concrete example, for the former case, the DRAM bandwidth required for the hash table for a 90%-10% read-write workload at 10 Mio. operations/s would be 1.22 GB/s.

The FPGA has no caches in front of DRAM, so each read is served directly from memory. Read requests are pipelined to hide the memory latency. Incoming keys are queued at the decide module and consumed when the corresponding data from the memory arrives. Multiple memory requests can be in flight, and although there is no need for locking in the
software sense, read-after-write hazards might still occur. That is, the modifications that a previous key has written to the hash table might arrive to the DRAM after the current key’s table entry has already been fetched. To prevent this, the \textit{decide} module implements a small write-through cache that stores recent writes with their address ("Write History"), much like the write buffers present in CPUs.

\textbf{Hazard Avoidance}  
Read-after-write hazards are avoided by keeping all pending writes in the “Write history” module. Internally this is composed of two BRAM-based structures that implement circular buffers. One holds the list of addresses under modification and the other ones holds the memory line contents. Items are added to the list when the write command is issued to the DRAM. They are removed from the list when the data is certain to be present in memory. In our case, due to the infrastructure around the DRAM, this information is not actually readily available, so instead we add a fixed delay based on micro-benchmarks.

Because we designed the distributed storage with read-heavy workloads in mind, we took the design decision to make finding an address in the “written” list execute as a loop, iterating over the non-empty positions in the list. This means that it is possible to perform between 156 Mio. and 4.8 Mio. lookups in it, assuming a depth of 32 entries. Also, the access bandwidth to this data structure is independent of the DRAM. The search in this list adds latency that is proportional to the length of the list. A design alternative that would be needed for higher worst-case throughput, would be to make the list behave as a content-addressable-memory (CAM). This option however requires significant resources and becomes difficult to place and route such that the circuit meets timing (we have
explored this limitation in previous work [49]).

**Reader module**  The Reader module issues read commands to the DRAM memory based on the hash values computed by the hash function. It has two inputs, one for requests received over the main pipeline and one for kicked keys, or “recirculated keys.” The arbitration between the two channels is done in a round-robin manner.

Each request will have two associated hash addresses, and the reader module will use these to request data from memory. While some operations, such as an insert, must inspect both locations before deciding how to proceed, in the case of read operations (GETs) it is possible to break the access up into two steps. First issuing a read to the first location of the key in the table, and only if it is not found there, then requesting the second address. This optimization saves significant memory access bandwidth in the average case, and this bandwidth can be used to implement other, more complex features in the FPGA, for instance, new secondary indexes, or different memory allocation schemes.

The above optimization comes, however, with a change in overall behavior. If all operations except for GETs are performed in one go, whereas a GET might be re-queued for the second address, they might become out of order with respect to other operations. Of course, if clients issue their requests one-at-a-time and synchronously, this effect will not be observed. As an alternative, if clients dedicate additional logic to deal with the situation, this is acceptable. For this reason, in most-use-cases, and unless DRAM random access bandwidth becomes as issue, using this optimization might not pay off. In *CaribouCH* it is possible to use either mode.

**Compare & Decide (Writer module)**  We have decided to implement the Compare & Decide module as a small state machine instead of a deep pipeline. For each input it will first iterate through the write history and the kicked keys in parallel to determine if these hold more up-to-date information than the DRAM memory. For read-mostly workloads the contents of both data structure should be empty, or almost empty, which allows keys to pass through this module at the rate of one every 3-4 cycles. If there are more updates in the workload as many additional cycles are needed as there are entries in these data structures. Inserts or updates, however, contain both a key and a value and therefore take longer to arrive over the network and line-rate can be met even with less operations handled per second. Furthermore, when increasing the write percentage,
the memory allocation logic would become the bottleneck sooner than the hash table’s writeback memory.

In the case of inserts and deletes, this module will interface with the memory allocation unit to fetch or return pointers. The memory allocation unit has a set of prepared pointers available for allocations and the overhead of this operation is low. As an optimization, it would also be possible to “prefetch” pointers for writes already in the read or hashing phase. The memory allocation logic is presented in detail in the next subsection.

Properties of Hash Table Modules The most important properties of each module of the hash table are summed up in Table 2.1. These properties are maximum throughput, latency and required on-chip memory. As the table shows, the current design is better suited for read-heavy workloads, otherwise the memory allocation becomes the bottleneck.

<table>
<thead>
<tr>
<th>Module</th>
<th>Processing Rate</th>
<th>Latency</th>
<th>BRAM requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashing</td>
<td>1.25GB/s</td>
<td>1 cycle / Byte</td>
<td>Proportional to throughput (queues for hashing cores)</td>
</tr>
<tr>
<td>Issue read</td>
<td>39-52M reads/s</td>
<td>3-4 cycles / key</td>
<td>None</td>
</tr>
<tr>
<td>Decide</td>
<td>At most 15M/s</td>
<td>Proportional to size of writeback buffer</td>
<td>Proportional to memory latency (buffer depth)</td>
</tr>
<tr>
<td>Memory allocator</td>
<td>At most 2M/s allocations/deallocation</td>
<td>Up to 2x memory access latency (hidden)</td>
<td>Proportional to allocation classes</td>
</tr>
</tbody>
</table>

2.4.3 Slab-based Memory Allocator

Choice of algorithm We use a slab-based memory allocator in Caribou. It allocates and deallocates memory in constant time and its performance does not degrade with higher fill rates. Background processing is avoided through combining housekeeping tasks with allocation/deallocation requests (e.g. reclaiming large chunks of freed memory). The memory allocator can handle up to 2 million operations per second, currently bottlenecked only by the DRAM access latency.
An alternative decision would have been to opt for log-structured storage and its corresponding allocation scheme. In software, log-based structures have the advantage of simple append-only writes and updates, the drawback is that they require periodic garbage collection [5]. This is not only because of deletes, but also updates that do not happen in place. In software this is less of an issue because CPUs run at high clock rates and are very efficient at executing iterative, loop-based code. In hardware, iterative code would run too slow, so we opted instead for a method that introduces no “background noise” in the system during normal execution and requires no lengthy garbage collection operations, thereby making response times more predictable.

Figure 2.16: Slab based memory allocator. Metadata is stored in form of bitmaps and pointers are stored in in-memory queues.

Management of Pages and Blocks We tailored the allocation logic to predefined classes of value sizes (this is done similarly in memcached and other related work [66, 103]). The current circuit supports 4 allocation classes, but the modules are parameterizable and can support more. The size classes can be dynamically chosen as multiples of 64B, which is the smallest allocation unit supported (called a “block” in Figure 2.16). This is because our DRAM bus width is 64B. An empty memory page is 32 KB and initially it can hold any size values. Once a page is used to allocate data of, e.g., 128 B, it becomes associated with this size class and the leftover space has to be allocated in the same increments. The pointer lists shown in Figure 2.16 encode free space as pointers to pages, extended with an associated class (or none if the page is empty). When a page is freed completely, it can be used to allocate any size data again. The way we keep track of the allocated and free or reclaimable space is through bitmaps that represent each block with one bit, as seen in Figure 2.16.
On an FPGA, accessing DRAM has more than an order of magnitude higher latency than accessing the on-chip BRAMs, therefore we keep the head and tail of each free-page list in on-chip memory and periodically flush/grab a batch of entries from main memory. Since there is only one component reading and writing these lists, storing parts of them in BRAM does not lead to inconsistencies. Because the BRAM buffers can hold thousands of entries per list, the cost of additional memory accesses is amortized over many operations, leading to near constant cost for allocation.

**Bitmaps and Scanning** Two bitmaps are used to keep track of allocated memory ("written") and memory that has been freed but it is not contained yet in a page descriptor as usable space, i.e., it is "reclaimable". The "written" bitmap is updated on every insert and delete operation, and is used in scan operations. The scan reads the bitmaps representing the whole memory and for each bit set to 1, it issues a read command to memory to fetch that address. Since in this module the actual size of values is not known, we rely on meta-data stored with the values to process them in the reading modules.

Deletions are handled by returning pointers and updating the "reclaimable" bitmap indicating that it has been freed but not yet reclaimed. A memory page is reclaimed and put in the "free list" if either a) all data on it has been freed (fully free page), or b) in case the system is low on free memory, if it has a large enough empty space to allow for further allocations (partially free page). Every time the bitmap is updated, the system will attempt to reclaim memory. This is done by inspecting the amount of contiguous "free bits" in the bitmap representing the page. If either condition a) or b) is fulfilled, the bits representing the reclaimed memory are flipped and the newly created page descriptor is appended to the corresponding free list.

For relational database use-cases, it is beneficial if the storage nodes can differentiate between data belonging to one table, a set of tables, and others. This way, the cost of scan operations, for instance, are not influenced by the size of other tables. The allocator module has support for tablespaces but, in our experiments, by default a single tablespace is used given that the prototype platform has less DRAM than a real system would. A tablescape is defined as a set of keys that belong to the same logical collection and is encoded in the high order bits of the key. They are implemented by extending the "written" bitmap to contain more than one bit per entry, signifying to which tablescape the data belongs to. This does not affect insert/delete performance because these operations access the bitmap at 64 B granularity. As for scan performance, larger bitmaps naturally
Table 2.2: The overhead for engine data structures is small.

<table>
<thead>
<tr>
<th>Type</th>
<th>MBs</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space for data</td>
<td>3840</td>
<td>93%</td>
</tr>
<tr>
<td>“Reclaimable” bitmap</td>
<td>8</td>
<td>0.2%</td>
</tr>
<tr>
<td>“Written bitmap”</td>
<td>8</td>
<td>0.2%</td>
</tr>
<tr>
<td>Pointer lists (1+4 classes)</td>
<td>25</td>
<td>0.6%</td>
</tr>
<tr>
<td>Hash table (1Mio. @ 75%)</td>
<td>21.3</td>
<td>0.5%</td>
</tr>
<tr>
<td>Hash table (12Mio. @ 75%)</td>
<td>256</td>
<td>6.2%</td>
</tr>
<tr>
<td>Total overhead for engine</td>
<td>297</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

Table 2.2 depicts how much memory each management data structure uses, the largest portion being taken up by the hash table. We use Xilinx VC709 Evaluation boards to build Caribou equipped with two 4 GB DDR3 SODIMMs. We configured the the hash table and the data area (value storage) to use one of the SODIMMs, while the TCP buffers, bitmaps, pointers and replication log overflow reside on the other one. This partitioning is a limitation on our current evaluation platform and should not be needed on future systems. In our evaluation setup the management data structures use less than 8% of the memory space, and even if the expected size of values would be halved, effectively doubling the number of required hash table entries, the management overhead would still amount to less than one fifth of the memory.

2.4.4 Processing

Processing inside Caribou_{CH} is covered in depth in Chapter 4. Below is only a summary of how this functionality fits in with the rest of the pipeline.

In Caribou_{CH} we have implemented various processing elements to filter data as it is retrieved from main memory. Since the key-value store sees values as BLOBs, it provides two filters that work on unstructured data. First, conditional filters compare parameters provided by the clients to a specific offset in the value. Second, a regular expression
matcher can be used to identify different byte patterns in the value. In addition to filtering, we also show how data transformation (decompression) can be carried out on the values as they are retrieved from memory.

As seen in Figure 2.17, processing has been added to CaribouCH such that values read by the key-value store stream through the different processing units before being sent to the client. The values are tagged with a bit which signifies whether they should be dropped or not. In case they are dropped, they are discarded at the end of the pipeline, and in case not, they are sent to the client.

Even though filtering directly inside the storage will always reduce the processing cost on the client side and is in essence “free”, in our prototype the bandwidth inside the
processing pipeline is higher than in the network, as this operates at a 8-times wider bus-width (512 bit vs. 64 bit) which makes in-storage filtering even more attractive. This is because without filtering data retrieval would be likely bottlenecked by the network, whereas with filtering this will not be the case (given a low selectivity workload).

Parameters for the processing elements are sent either as part of a Conditional GET or Conditional SCAN operation, where the value would be in a SET request. The client needs to know what types of processing elements the Caribou node has, because they need to format the request accordingly. As an example, if there are three processing elements in the pipeline each with a differently sized parameter, the request will concatenate the parameters in the same order as the processing elements are present on the FPGA. In our prototype we hard-code the configuration in the client (we deployed Caribou\textsubscript{CH} only with a handful of different setups), but in the future it should be possible to ask for the node to list its processing capabilities and parameter sizes.

The three types of processing modules are presented in detail in Chapter 4 that focuses on near-data computation.

### 2.5 Related Work

#### 2.5.1 Key-value stores in FPGAs

Even though CAMs have been implemented with FPGAs for a long time, one of the first projects to implement a key-value store with a logic footprint independent of its capacity was the project by Chalamalasetti et al. [20], is a memcached appliance for 1Gbps networks whose design resembles software approaches of building hash tables, in that collisions are solved by iteration, and the processing logic is not fully pipelined. As a consequence, this design does not fully utilize the potential of specialized hardware chip, but as the earliest work in this topic, it does bring good arguments for specialized servers.

Figure 2.18 shows the throughput achieved by the system in [20] on a 1Gbps network, as a function of value size for get requests. The blue line represents the maximum possible throughput over the same network. As mentioned above, the design of this related work is not fully pipelined, and as a result the per-request overhead negatively impacts performance for small packets. In the work presented in this dissertation our goal was to meet network line-rate for most operation mixes, and relied on pipelining to achieve this goal.
The first fully pipelined 10Gbps key-value store design for an FPGA-based key-value store was the project I contributed to during my Master’s studies [14]. That key-value store was also designed to be compatible with memcached’s protocol and had a lossy hash table design, similar to the one in CaribouMC. The solution was, however, not standalone, and required a host to be present for memory allocation purposes (using queues over PCIe to allocate/deallocate memory on the FPGA).

Over the years there were two follow-up publications from Xilinx Labs on the initial design. Karras et al. [63] explored how a key-value store can be built using High Level Synthesis (HLS). The authors reported performance numbers similar to the hand-crafter version, but expressed in much less lines of code, and development was reportedly easier. The work in [15] sketched how the 10Gbps key-value store pipeline developed at Xilinx Labs could be extended to support 80Gbps traffic. The underlying storage medium chosen were SATA SSDs, and to overcome the large overhead of 4KB page accesses, the authors proposed a hybrid storage scheme, storing small values in DRAM, and large ones in flash. There is little detail on the type of memory allocator they used in that work, but the slab-based allocator inside CaribouCH could be used to offer seamless allocations over different storage media by limiting the memory in DRAM to small slab types.

The design by Tanaka et al. [103] also uses a hybrid DRAM+Flash setup, but in that system the memory is used a cache in front of the flash drives. The hash table uses a
collision resolution technique based on linear probing, which will lead to variable response
times, and will require table reorganizations on delete, otherwise reads will have to touch
multiple addresses in the table.

BlueCache [116] is a flash-based key-value store for caching. The nodes are connected to
each other through a special purpose dedicated network and to host machines via PCIe.
Clients issue requests by going through the host machines. The performance of a single
node reaches up to 4 M operations/s demonstrating that flash-based FPGA designs can
achieve high performance. In contrast to Caribou, BlueCache nodes are not stand-alone
and offer no built-in replication or processing. The nodes can, however, load balance keys
and fetch data from remote FPGAs over the dedicated backbone network. This delivers
excellent latency and eliminates virtually all packet loss.

Lim et al. [76] extensively benchmark memcached with different hardware setups to explain
the sources of its inefficiency on common CPUs, which mostly springs from its low data
and instruction-locality, rendering the large multilevel caches useless. As a solution they
propose offloading the read path (get requests) to a smart network card, implemented with
an FPGA, that can more efficiently execute the “optimistic” path, and keep the CPU as
a fallback. The system shows promising results in terms of performance, it is however
designed for 1Gbps networks only. Fukuda et al. [39] also follow a similar reasoning
but they focus on minimizing latency with the smart network interface card (an FPGA
with its own DRAM memory); the benefit on peak performance is not discussed in detail
though. These works represent an alternative design point to Caribou, in which only parts
of the application are implemented in hardware, or rather, there is a CPU for fallback.
Performance can be boosted by optimizing for the average case, although integration in a
regular host could also mean a significant loss of energy efficiency – an interesting trade-off
to consider nonetheless.

The KVS from Algologic [78] is a great example of a truly specialized key-value store
implementation in hardware. It has been designed for high-frequency trading and related
use-cases and it offers excellent latency over Ethernet packets (<1μs). This is achieved by
fixing key and value sizes, and using on-chip BRAM memory for the hash table instead of
off-chip DRAM. Even though this decision limits the maximum capacity, it offers minimal
latency. While in Caribou it was one of our goals to provide low latency access to data, we
also want to provide, at the same time, a flexible general purpose service. This requires
using TCP/IP networking and storing all data in large off-chip memory.
2.5.2 Software optimizations and specialized software systems

There is a growing body of work on optimizing memcached-like systems for multi-cores and also on reducing the overhead of networking. MemC3 [37] optimizes memcached with algorithmic and data structure improvements. The original hash table used in memcached is replaced with a cuckoo-hash [88] based structure and this significantly reduces locking conflicts. As a result, MemC3 can reach three times higher throughput than the standard implementation (almost 4.5M requests/s). These performance numbers for MemC3 are in the same ballpark as results reported in two Intel whitepapers [112, 48] that also increase performance by reducing locking effects in the hash table.

To reach even higher throughput in software it is necessary to “specialize” servers to the key-value store workload. For instance, recent work from Solarflare [102] achieves more than 9M lookup requests/s by fine-tuning the network card and the underlying system to this application. MICA [75, 73] is another such example. The key-value store is built from the ground up, using Intel’s DPDK [46] library. DPDK allows applications to interact directly with the network without going through the OS, reducing latencies and removing the overhead of the operating system’s networking stack. The results of this work are very promising: when using a stripped down stateless protocol similar to UDP the complete system demonstrates 120 million requests per second over four 40Gbps NICs in a single machine. It is important to note however that in MICA skewed workloads experience slowdowns due to the partitioned nature of the data structures and to reach such a high throughput lossy data structures have to be used. Additionally, a significant part of the server’s logic (for instance computing the hash function on keys) is offloaded to clients.

We consider Caribou to be an interesting alternative to software because, as will be shown in Chapter 5, it can reach a similar performance per Watt while providing general-purpose data structures and processing capabilities to the clients.

There is a trend in using RDMA for applications that were traditionally built around a client-server architecture [86, 33, 91] because it reduces latencies drastically. In FARM [33], the authors implement a distributed key-value storage designed for large, scale-out workloads. Replication and offloading is built into the nodes, and per node mixed-workload throughput surpasses 10MRPS. These systems are however not able to fully saturate the underlying network (Infiniband QDR and 40GbE) for small request sizes and this illustrates that in software, even when running on fast networks and using RDMA, complexity will often limit performance. Furthermore, adding in-storage scans or offloading function-
ality, such as the regular expression matching in CaribouCH, would also be more difficult. At the other end of the spectrum, network-attached storage devices, such as the Open Kinetic\textsuperscript{10} drives by Seagate implement a flexible and rich key-value store interface over TCP using small embedded processors. While they offer no offloading capabilities and these small CPUs provide only modest performance, and for use-cases where one wants to implement large object stores in a small energy footprint, these devices are a compelling option. The ideas in Caribou could be combined with Kinetic Drives to provide high performance network-attached drives with rich functionality and offloading capabilities.

2.6 Summary and Discussion

In this section we presented the design of two key-value store pipelines on FPGAs, with focus on the hash table data structure that ensures that random-access operations can be carried out at high rate. One of the main principles behind building a hash table that stores data in off-chip DRAM is that in order to achieve high throughput, the latency of the memory needs to be hidden (we want the circuit to be bandwidth not latency-bound). This is done by pipelining all memory access operations and the underlying data structure has to be chosen such that it avoids iterative accesses for the same key-value store request. As we shown in Caribou, this pipelined design introduces so called “data hazards” because multiple reads/changes to the same memory location could be in flight at any time. On the spectrum of possible solutions, we explored two options: stalling the pipeline whenever a data hazard is detected, or building an on-chip write-back cache that can be used to “materialize” in-flight changes and this way avoid stalling. The former method requires minimal additional resources, and in case the hash table’s throughput is over-provisioned when compared to the network, the occasional stalls will not be visible to the clients. The latter requires more resources but guarantees a lower bound on the throughput of the hash table. This option is a better choice for cases when the hash table performance is not over-provisioned.

The memory allocation inside Caribou has been designed based on a common software algorithm and its main goal was to show that managing large amounts of storage are feasible with FPGAs without having to use naive methods. In the current design the memory allocation logic produces free pointers and claims storage off the critical path.

\textsuperscript{10}https://www.openkinetic.org/
This means that unless the workload is skewed for some data size and, at the same time, has temporal locality on sets, increasing the latency of the medium used for storing bitmaps and pointer lists will not have a visible effect. Therefore the current design is expected to work well not only in DRAM but also when using NVM storage. The current memory allocator could even be used in a setting where storage spans both DRAM and flash (e.g. in the experimental system by Blott et al. [15]), by mapping the different slab types to the address space belonging to different storage media.

*Caribou* offers two access methods to the data: random access and scan-based access. The scan we implemented is based on a simple bitmap index that keeps track of which memory locations have valid data, to skip over unallocated space, but it would be possible to implement more complex structures as well. Range-based data retrieval could be for instance useful in database workloads. This opens up the research question on tree-type data structures in hardware. While there are some examples in the FPGA related work on implementing B-Trees in hardware, the use-case in *Caribou* requires these to be built and maintained entirely in hardware, which is a novel challenge.

Finally, in this chapter we also contrasted two protocol options for the key-value store, one binary and a more verbose ASCII based one. The latter is compatible with memcached, and allows *Caribou* to be used as a drop-in replacement for this service, but it also requires more resources than the binary protocol because it needs for instance to translate numbers from a decimal representation to binary. What these two options show, however, is that the protocol used to communicate with *Caribou* can be easily changed or adjusted to match the needs of a particular use-case.
Chapter 3

Distributed Consensus and Replication

Distributed storage deployed in the datacenter will inevitably encounter failures, either of the nodes themselves or of the network infrastructure. To ensure that no data is lost upon failures, in most software solutions the nodes replicate data among themselves. In the case of Caribou, we implement a primary/backup scheme inspired by state of the art software, where the node holding the primary copy of the data (the leader) handles all writes to the dataset and disseminates the changes to the backup nodes (the followers). In order to provide this functionality, we had to solve a more general problem present in distributed systems: distributed consensus. The FPGA-based version provides the same behavior as software but executes consensus rounds with lower latency and higher throughput than its counterpart.

Apart from showing that it is possible to implement a consensus protocol between all-hardware devices, which in turn allowed us to extend Caribou with proper replication functionality, this module could be used in various other settings as well. Examples include smart switches and middleboxes that could use the consensus module to ensure, for instance, that rule changes are propagated consistently. We discuss these and other potential use-cases in Section 6.2.

The main take-aways of the consensus protocol’s design and implementation on an FPGA can be summarized as follows:

- Choosing a protocol that allows pipelining translates to higher performance on the FPGA because matches its execution model well.
• Even though the FPGA can implement all aspects of the protocol, including recovery, when using Verilog/VHDL as the programming language, the common case processing is much easier to express than the corner-cases. In our prototype these situations are handled by code that is “catch-all” for simplicity. This shows that there is potential for future work either using higher level languages to express the logic on the FPGA, or combining specialized hardware and small CPU cores for flexibility.

• FPGA-based nodes experience very little jitter in their operation, which will keep the distributed algorithm in its common execution case most of the time.

3.1 Background

One of the general challenges of building distributed systems comes from the fact that failures and asynchrony make it difficult to maintain an accurate view of the global state in each node. As a result, methods of coordination are used, such as Atomic Broadcast [24]. Atomic Broadcast is responsible for disseminating changes to the distributed system state (e.g., writes to a database) in a way that all participants see the same updates and in the same order at any point in time (i.e., a total order). This enables the distributed system to have consistent state, even in the face of failures or network partitions. In this work we explore leader-based approaches to broadcast, which means that all updates to the global state have to be initiated at a single node, which will then execute the bulk of the broadcast protocol.

Atomic Broadcast and Distributed Consensus solve very similar tasks [29], and in many systems broadcast is implemented on top of a consensus algorithm, such as Paxos [69]. For this reason, in this dissertation I will use the two terms interchangeably.

There are many distributed systems that require some form of coordination for achieving their core services, and since implementing distributed consensus [65, 69] correctly is far from trivial [84], reusable libraries and solutions such as Zookeeper have emerged. Zookeeper is a “turnkey” application that provides distributed synchronization, stores configuration data, and implements naming services for large distributed systems. It achieves fault tolerance and high availability through replication.

At the core of Zookeeper is an atomic broadcast protocol (ZAB) [58]. As Figure 3.1 illustrates, the atomic broadcast is used to ensure the consistency of modifications to
key-value store backing Zookeeper. ZAB is roughly equivalent to running Paxos [69], but is significantly easier to understand because it makes more assumptions about the communication channels, namely that they are lossless and guarantee ordered message delivery. For this reason, Zookeeper in principle requires TCP.

In the following we provide a brief description of the ZAB protocol focusing on aspects relevant for this work, but for a more complete overview, please consult the original publication by Junqueira et al. [58]. The atomic broadcast protocol of Zookeeper is driven by a leader, who is the only node that can initiate a modification of the global state. These modifications are results of “write” operation issued by a client (this can mean a set/update/delete in the key-value store sense). The leader proceeds by preparing so called “proposals”, that incorporate the desired changes in a packet, to be sent to other participants. Figure 3.2 shows the main steps of the protocol in a 3 node setup, with the first step being the proposal sending. Once the followers receive proposals, they will acknowledge the receipt of these proposals. This tells the leader that the participants are not faulty, and can be reached over the network. Once a majority of followers has sent an acknowledgment back, the leader can decide to actually perform the state update and to instruct participants to do so as well. This is done by issuing a “commit” message for each follower that acknowledged to apply the changes. For nodes that lag behind, or experiences a network failure, or a complete failure, the protocol includes steps for bringing them “up to date” with the state that the majority of nodes agreed on.

In Zookeeper’s specific implementation committed messages are persisted to a a disk in addition to being applied to the key-value store, but depending on the nature of the data stored in the service, writing the log to memory might be enough. The main role of the
disk is durability, and keeping the log in memory will only lead to data loss if all nodes fail at the same time.

In Zookeeper terminology, the total order of messages is defined using monotonically increasing sequence numbers: the “Zxid,” incremented every time a new proposal is sent, and the “epoch” counter, which increases with each leader election round.

![Zookeeper’s Atomic Broadcast (ZAB) Protocol](image)

Figure 3.2: Zookeeper’s Atomic Broadcast (ZAB) Protocol requires a leader to propose changes to followers.

Zookeeper can run with two levels of consistency: strong [42] and relaxed (a form of prefix consistency [105]). In the strong case, when a client reads from a follower node, it will be forced to consult the leader whether it is up to date (using a `sync` operation), and if not, to fetch any outstanding messages. In the more relaxed case (no explicit synchronization on read) the node might return stale data. In the common case, however, its state mirrors the global state. Applications using Zookeeper often opt for relaxed consistency to increase read throughput. In this work we implement the relaxed model as well, but as will be shown in the Evaluation, the latency of consensus rounds in Caribou is in the order of 10µs which means that nodes will never lag behind significantly (unless they are in a network partition).

### 3.2 Hardware Implementation

Similarly to Zookeeper, in our system there are two types of requests: local ones and replicated ones. Local requests are those that a node serves from its local data store bypassing the atomic broadcast logic completely. Gets, for instance, fall in this category. Those that change the state need to be replicated to all nodes. Replicated requests are “trapped” inside the atomic broadcast module until the protocol reaches a decision and
3.2. Hardware Implementation

Figure 3.3: Read and write requests take different paths: reads go directly to the hash table, while writes are first replicated.

only then are sent to the data store, which will process them and return the responses to the client. For reaching consensus the atomic broadcast module will send and receive multiple messages from other nodes. As for terminology, we use read, get operation, and local request interchangeably. The same applies to the use of write, set and replicated request.

The distinction between types of requests is made by means of a 16B header inserted in front of messages destined to the key-value store in CaribouMC, and merged with the KVS header in CaribouCH. This header is used to decide in the network unit whether a request is to be replicated, that is, whether to pass the data on to the atomic broadcast unit, or directly to the key-value store. The networking layer treats the encapsulated key-value store request as binary data and uses the information in the header to find out its length without parsing it. The same header is used for communication between different node’s atomic broadcast units (encoding different message types of the protocol in the operation code field), and often have no encapsulated payload.

One of our design goals was to ensure that the atomic broadcast module is usable with other key-value store implementations on the FPGA, and as a result it treats the data to be replicated as opaque binary data. We combined it already with two different hash table implementations (CaribouMC and CaribouCH) in main memory. It could be, however, combined with other types of storage as well.
Header Structure  The structure of ZAB headers for replication messages is shown in Table 3.1. They contain fields that apply both to requests with payloads and messages without payloads. While there is opportunity to compact this header, in our current prototype we use some parts of it to encode node-local (non-replicated) operations that are handled directly by the key-value store. For instance, the higher two bytes of the length field are used for this if the opcode is zero. Since the payload length is expressed as the number of 64-B words, even two bytes are sufficient for the length to cover our use-case (in Caribou we support at most 1KB values in the current implementation).

Control messages, such as ones adding a node or removing a node, are also formatted to have 16-B length. The first 8-B word of the command is used according to Table 3.1, while the second 8-B word is specific to the type of operation. For instance, an “add node” control message, that is issued by a coordinator to all nodes in a cluster will encode IP address and port in the second 8-B word of the command. While it would be possible to add the operation-specific data as a third 8-B word to the command, leaving zxid and epoch intact, only replication-related operations actually need these fields. This would mean wasted space for all other types of messages.

Table 3.1: Structure of the ZAB request header in Caribou

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[23:16]</td>
<td>Sender Node ID</td>
<td>[95:64]</td>
<td>Zxid (req. sequence no.)</td>
</tr>
</tbody>
</table>

3.2.1 Networking

As Figure 3.4 shows, the network module connects to the Ethernet Network Interface provided by the FPGA vendor that handles Layer 1 and 2 (including MAC) processing before handing the packets to the IP Handler module. This validates IP checksums before forwarding packets to their corresponding protocol handlers. The other modules seen in the top part of Figure 3.4 are used to provide additional functionality that is needed when integrating the FPGA in a real network: ARP, DHCP and ICMP.

For some deployments of Caribou, point-to-point connections are also enabled between FPGAs, in addition to the main network connection. The module that takes care of
validating these packets is labeled “App. specific” in Figure 3.4. The application specific protocol is running directly on top of Ethernet and it does not require network routing.

**TCP-based Connections**  Zookeeper’s Atomic Broadcast requires reliable channels between participants and this requirement can be fulfilled with TCP/IP. It is a stateful network protocol that provides the abstraction of a reliable, ordered, communication stream (socket) between machines. Implementing a fully functional TCP endpoint in hardware has been long considered unfeasible because of its complex control logic, but Sidler et al. showed that it is possible to handle thousands of connections at 10Gbps line-rate in hardware [98]. The main benefit of using hardware for implementing a network stack is that it allows for building true data flow pipelines that decouple send and receive paths completely. This means that operations happening on one, for instance, retransmissions taking place on the TX path, do not impact the other negatively. Furthermore, networking and application are tightly connected, so there is no overhead in communication between the two. Packets are delivered via FIFOs that are built with low latency on-chip BRAM.

In Caribou we use a modified version of this TCP/IP stack\(^1\) [99]. The modifications are driven by the properties of modern datacenters and the idiosyncrasies of client-server workloads. Furthermore, they are also driven by the fact that the DRAM on our evaluation boards (and indeed many FPGA evaluation boards) is slower than in regular servers, and multiplexing it between different modules on the FPGA can be limiting performance-wise.

As a result, optimizations that reduce the effective DRAM bandwidth usage of the TCP stack are needed. The stack we use in Caribou has an asymmetric setup in which the

\(^{1}\text{https://github.com/fpgasystems/fpga-network-stack}\)
receive path operates with small on-chip buffers (BRAMs) and only the send path uses 
DRAM for buffering packets. In the original work, the TCP/IP stack would use buffers 
in DRAM by first writing data in, and then immediately reading it out to send it over 
the network. In our updated version the data is sent to over the network and written to 
DRAM buffers in parallel.

The reasoning behind the asymmetric design this is two-fold: on one hand, the FPGA 
is fast enough to keep up with close to line-rate throughput in most cases, so requiring 
substantial buffer space for incoming packets should be rare. Most often the input buffer 
is used to re-construct segmented messages, and therefore the size of the receive buffer in 
BRAM needs to be adapted to the requirements of the rest of the design, e.g., maximum 
request size to the key-value store. In the case that an input buffer gets full, we can use 
TCP’s mechanisms to wait for the retransmission of “dropped” packets. On the sending 
path, on the other hand, we need to make sure that no created packets are ever lost, as 
recreating them is not an option. Also, when an FPGA talks to another FPGA one of the 
two needs to have enough buffer-space to make sure no data is lost, meaning that the send 
buffers (TX path) must reside in DRAM. Currently we allocate 64KB buffers per session 
in DRAM, which we found enough for most purposes. In case backpressure builds up in 
the system all the way to the RX buffers, the implementation will drop requests before 
the buffers.

As already described, an additional optimization that we performed was to modify the 
way the sending logic uses the buffers in DRAM. In the original implementation, data to 
be sent would be written to DRAM, and then read again to be sent over the network. 
This adds latency and doubles the DRAM usage in the general case where packets do not 
need to be retransmitted. In the version used in Caribou, the data is written to DRAM 
and sent over the network at the same time. This cuts the memory bandwidth usage by 
half, and reduces the latency as well, without sacrificing safety or functionality [99].

**Point-to-point Connections** In addition to the regular, switched, fabric we have also 
developed a solution for connecting nodes to each other on dedicated links, while remaining 
in-line with the reliability requirements (ordered delivery, retransmission on error). 
For this, we used the additional network interfaces of our boards to connect each board 
with every other one, as shown in Figure 3.6. The FPGA boards we use have four net-
work connectors, which means that the largest number of nodes we can connect with the 
backbone is also four. This, however, does not mean that the cluster size is also limited
to four. The FPGAs can communicate with other nodes both over regular TCP sockets, or the dedicated links, which are treated as special, always-connected sockets.

We built a lightweight protocol on top of Ethernet packets that uses the ZAB headers directly and encodes only a sequence number and the length of the packet from the network’s perspective. Since connections are point-to-point and the other node’s identity is explicitly defined there is no need for addressing in the header. The sequence numbers are inserted into requests where normally the magic number is in the header (so the sequence number is actually increased with each logical request, not with each packet). If data was dropped due to insufficient buffer space on the receiving end, or because of corruption on the wire, the receiver can request retransmissions. Since the links are point-to-point, congestion control is not necessary beyond signaling backpressure (achieved through special control packets).

The design of the buffers follows the same principle as in the TCP module: the sending side maintains a single buffer (64KB) per link from which it can resend packets if necessary, and the receiving side only reserves buffer space for request reassembly. Since the latency between FPGAs is in the order of microseconds, this buffer is enough to cover a time window of $50\,\mu$s on a 10Gbps Ethernet link, more than enough capacity for our purposes.

To send and receive data over this protocol the application uses special session numbers when communicating with the network stack (see example in Figure 3.5). This will associate packets with the block labeled “Application-specific” in Figure 3.4 instead of the TCP.

At the moment our design only accommodates a limited number of nodes connected together with this protocol because there is no routing between interfaces. The Catapult platform [94] includes for instance a 2D torus interconnect where FPGAs route packets over the dedicated network, while using a very similar application-specific protocol. We plan to eventually evaluate our system at larger scale using such an interconnect.

### 3.2.2 Dataflow Architecture

The big advantage of implementing the atomic broadcast protocol in hardware is that the control and data “planes” can be separated. This means that the two main modules in the atomic broadcast logic, the Control State Machine and the Log/Data Manager (Figure 3.7) can on the one hand work in parallel and latencies can more easily be hidden, and on the
Figure 3.5: From the perspective of the key-value store and replication logic, specialized backbone connections are not different from TCP sockets.

Figure 3.6: The specialized point-to-point connections act as a backbone interconnect between nodes. These links only carry replication-related traffic.

On the other hand, they can be modified without affecting each other. This also provides the degree of freedom required when porting the logic to new platforms (logging to BRAM, DRAM, even to SSD can be done without changing the controller).

Figure 3.7 shows two more modules, one to split the incoming requests into command word and payload (Splitter) and one to recombine commands with payloads for proposals for instance (Recombine). Headers (command words) are extracted from requests by the Split module and reformatted into a single 128-bit wide word. Data is aggregated into 512-bit words to match the memory interface width. When the control state machine (controller)
issues a command (header) that has no encapsulated data, such as the acknowledgment or commit messages of the ZAB protocol, this goes directly through the Recombine module. If, on the other hand, there is encapsulated data to be sent then the controller will request the data in parallel to creating the header. The Recombine unit can then take the header and append the data to it. The system is pipelined, so it is possible to have multiple outstanding requests waiting for data from the log/memory.

### 3.2.3 State Machine

The state machine that controls the atomic broadcast is an adaptation of the original ZAB protocol [82]. We have kept the broadcast behavior the same, but simplified the leader election and synchronization steps based on assumptions we can make about the hardware. Figure 3.8 show the “super states” in which the state machine can be (each such oval in the figure hides several states of its own). In normal operation mode, the controller waits
in an idle state either for the arrival of a command or in special cases the firing of a timer.
Then, depending on the current role of the node and the contents of the command it will
transition into another state.

Table 3.2 shows an overview of how many clock cycles (at the 156.25MHz network clock)
each state of the state machine takes through a simple formula for understanding how this
cost is affected by the number of nodes in a setup (nodeCnt). Additionally, we included
the time it takes to seek in the command log as the parameter cmdLogLatency². Currently,
the state machine itself never becomes the bottleneck because the system is limited by
10Gbps networking. Looking ahead, if we wanted to scale our system up to 40Gbps
networking, this component could be clocked up to 300MHz (independently from the rest
of the pipeline) to have enough performance to handle the increased message rate. The
rest of the logic inside the atomic broadcast module handles the payloads only, and these
paths could be made wider for four times higher throughput.

<table>
<thead>
<tr>
<th>Operation/State</th>
<th>Cost (clock cycles)</th>
<th>Max for: 3 nodes</th>
<th>Max for: 7 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁ Create and send proposal</td>
<td>2+Cnodes×2</td>
<td>19.5 M/s</td>
<td>9.75 M/s</td>
</tr>
<tr>
<td>F₁ Recv. proposal and send acknowledgment</td>
<td>2</td>
<td>78 M/s</td>
<td>78 M/s</td>
</tr>
<tr>
<td>L₂ Recv. acknowledgment and check majority</td>
<td>2+Cnodes+LcLog</td>
<td>17.3 M/s</td>
<td>13 M/s</td>
</tr>
<tr>
<td>F₂ Commit</td>
<td>1+LcLog</td>
<td>39 M/s</td>
<td>39 M/s</td>
</tr>
<tr>
<td>L₃ Commit</td>
<td>3+Cnodes</td>
<td>26 M/s</td>
<td>15.6 M/s</td>
</tr>
<tr>
<td>Consensus round (leader)</td>
<td>L₁ + L₂ + L₃</td>
<td>7.1 M/s</td>
<td>4.1 M/s</td>
</tr>
<tr>
<td>Consensus round (follower)</td>
<td>F₁ + F₂</td>
<td>26 M/s</td>
<td>26 M/s</td>
</tr>
</tbody>
</table>

Table 3.2: Cost of ZAB operations and the theoretical maximum consensus rounds per
second when using a single 10GbE link per node.

### 3.2.4 State Table

On each node there is a local table that holds relevant meta-data on the other nodes in
the cluster. This table describes the state of the cluster as seen locally, and is updated
whenever nodes are added or removed from the cluster. Each entry in the table corresponds
to a participant and consists of information associated with Zookeeper atomic broadcast
(Zxid, epoch, last acknowledged Zxid, etc.), a handle to the session opened to the node,

²In our current design looking up the pointer to an earlier payload takes 3 cycles in most cases.
3.2. Hardware Implementation

and a timestamp of the node’s last seen activity. Since the mapping from session number
to network protocol or even network interface is made in the networking module, the
controller is independent of the network details and works the same for messages received
over any protocol or connection. The table resides in BRAM and its maximum size can
be adjusted at compile time.

3.2.5 Quorums and Committing

Zookeeper atomic broadcast allows the pipelining of requests, so when the leader’s con-
troller receives a client request that needs to be replicated it will send out the proposal
and mark its Zxid as the highest that has already been sent but not acknowledged or
committed. When an acknowledgment is received from another node, the leader’s con-
troller will test if a quorum (majority) has been reached up to that particular Zxid. This
is done by iterating through the active nodes in the state table: if enough have already
acknowledged, the leader’s controller will send out commit messages to all nodes that al-
ready acknowledged the proposal. Then the leader will instruct the log unit to mark the
operation as successful and to return the payload so that the application can process the
original request. On the follower the receipt of a commit message will result in the same
operations of accessing the log and preparing the data for the application. In case a node
sends its acknowledgment after the operation has already been committed, the leader will
issue a commit to that node as a response.

Our implementation of tunable consistency makes it possible to choose the quorum-
checking function at runtime. To be more specific, one can change between either waiting
for a majority or waiting for all nodes to respond. The latter behavior could be useful
in cases when failures of nodes are assumed to be transient, but updates have to happen
absolutely at the same time on all nodes (like changing a policy on a set of middleboxes).

3.2.6 Maintaining a Log

After the payload is separated from the command it is handed to the Log and Data Man-
ger (bottom half of Figure 3.7). The payload is added to an append-only log. When
an operation commits, this event is also added to the log (a physically different location,
reserved for command words) with a pointer to the payload’s location. In case a synchro-
nization has to happen, or the node needs to read its log for any other reason, the access
starts with the part of the log-space that contains the commands. Since each entry is of fixed size, it is trivial to search a command with a given Zxid. Log compaction is carried out in an implicit manner. Only a fixed number of entries are stored in the log and if a synchronization message arrives asking for an earlier consensus round, for instance from a node joining the consensus group, the entire state of the key-value store is copied over (bulk synchronization). Depending on the types of failures experienced most often in the cluster, keeping a relatively short log that suffices for catching up followers experiencing short-lived network failures might be a more efficient use of memory. Furthermore, we expect FPGAs to fail fully if they do, so in case of a hardware failure and subsequent recovery, a full copy of the state would be required.

In the key-value store use-case latency was important and for this reason, we have designed the log predominantly with BRAM storage in mind. The most recent entries are written to BRAM, which is spilled to DRAM memory in large increments, if at all. Of course if the payloads are large only a small number will fit into the first part of the log, this is however not really an issue because in the common case each payload is written once to the log when received and then read out in a window of 10-20\mu s. Depending on the rate at which write requests are received, it is very likely that the whole “working set” of pending requests fit in on-chip memory.

For other use-cases, in which the application consuming the replicated requests cannot be easily “cloned”, it might be necessary to persist the entire log or a very large portion of it. For this reason, larger capacity off-chip storage could be used, e.g. an SSD. While we have not implemented writing the log to flash, we have measured the performance of a SATA III drive\textsuperscript{3} connected to our key-value store pipeline using the Groundhog SATA driver\textsuperscript{4}. The bulk read and write bandwidth of a single device is in the range of 5Gbps, enough to handle a high rate of replicated writes. Random access to the same device is, however, much slower, without native command queuing and 4 KB accesses it barely reaches 90k operations per second. This shows that even in this scenario, the recent portion of the log would need to be cached either in DRAM or BRAM to maintain high throughput.

\textsuperscript{3}OCZ Vertex 4 SATA III, 256GB
\textsuperscript{4}Groundhog
3.2.7 Synchronization

When a node fails, or its network experienced outages for a while it will need to recover the lost messages from the leader. These are sync messages in the ZAB protocol. In our implementation, when a follower detects that it is behind the leader it will issue a sync message. The leader will stream the missing operations from the log to the follower. These messages will be sent with an opcode that will trigger their immediate commit on the other end. In the current design the leader performs this operation in a blocking manner, where it will not accept new input while sending this data. It is conceivable to perform this task on the side, but for simplicity we implemented it this way for this prototype design.

If for some case the follower would be too far apart from the leader, and syncing the whole log would take longer than copying the whole data structure in the key-value store, or the version is not contained anymore in the log, there is the option of state transfer at bulk: copying the whole hash table over and then sending only the part of the log that has been added to the table since the copy has begun. In the case of Caribou\(_{CH}\), in addition to the hash table and values also the memory allocation meta-data is copied over, practically cloning the entire state of the node.

3.2.8 Bootstrapping and Leader Election

When starting up our hardware-based distributed system for the first time, each node is assigned a unique ID that is increased as nodes are added. If a node joins after the initial setup, it gets the next available ID and all other nodes are notified. If a node fails and then recovers, it keeps its ID. These identifiers are determined by an outside coordinator. Even though it is true that ZAB itself could be used to assign IDs to newcomers and to discover the state of the cluster, we defer this to future work.

The only part of the ZAB protocol that we replaced with our own solution is leader election. Our implementation aims to reduce complexity in hardware, so we opted for solution that requires less messages between participants in the common case. We implement a round robin leader election that will try and elect the node with the next larger ID after the old leader as the new leader. If this node is not reachable, the search continues by the same rule.

Nodes enter “leader election mode” when a node suspects the leader is down. This happens if the leader is unreachable (a timeout large enough has passed). The node will broadcast
its decision to abandon the old leader and at the same time stating its move to the next ID as leader. By the time this message is received in other nodes, they have already likely detected that the leader is unreachable, otherwise, they abandon the current leader and enter election mode. Once the prospective leader receives such a “vote” it will wait until it receives a majority vote (including itself), or an other leader is proposed. Once the next leader has a majority of the votes, it can start sending regular replication traffic to followers. Since a leader failure is likely to result in inconsistent state across nodes, our leader election round finishes with a synchronization step, as presented in the ZAB specification.

Even though the leader election should not take more than an atomic broadcast round, in some cases it could still introduce significant waiting in the system, and replicated writes can not be carried out in this period. Therefore, write requests arriving from clients during leader election have to wait until all steps are finalized. Unless running in a strong replication mode, the reads can still be services locally, even in the period of uncertainty. This increases the availability of the system in the face of failures.

3.3 Related Work

3.3.1 Coordination in Large Systems

Paxos [65, 69] is a family of protocols for reaching consensus among a set of distributed processes that may experience failures of different kinds, including ones in the communication channels (failure, reordering, multiple transmission, etc.). While Paxos is proven to be correct, it is relatively complex and difficult to implement, which has led to alternatives like Raft [84], ZAB [43, 82] or chain replication [109]. There is also work on adapting consensus protocols for systems that span multiple physical datacenters [77, 81, 23], and while they address difficult challenges, these are not the same problems faced in a single data-center and tight clusters.

Paxos and related protocols are often packaged as coordination services when exposed to large systems. Zookeeper [43] is one such coordination service. It is a complex multi-threaded application and since its aim is to be as universal as possible, it does not optimize for either the network or the processor. Related work [91, 32] and our benchmarks show that its performance is capped around sixty thousand consensus rounds per second and
that its response time is at least an order of magnitude larger than the FPGA (300-400\(\mu\)s using ram disks). Etcd [1], a system similar to Zookeeper, written in Go and using Raft [84] at its core has lower throughput than Zookeeper. This is partially due to using the HTTP protocol for all communication (both consensus and client requests) which introduces additional overhead.

Many systems (including e.g., the Hadoop ecosystem) are based on open source coordination services such as Zookeeper and Etcd, or proprietary ones (e.g., the Chubby [17] lock server). All of them can benefit from a faster consensus mechanisms. As an illustration, Hybris [32] is a federated data storage system that combines different cloud storage services into a reliable multi-cloud system. It relies on Zookeeper to keep metadata consistent. This means that most operations performed in Hybris directly depend on the speed at which Zookeeper can answer requests.

### 3.3.2 Speeding up Consensus

Recently, there has been a high interest in speeding up consensus using modern networking hardware or remote memory access. For instance DARE [91] is a system for state machine replication built on top of a protocol similar to Raft and optimized for one-sided RDMA. Their 5 node setup demonstrates very low consensus latency of <15\(\mu\)s and handles 0.5-0.75 million consensus rounds per second. These numbers are similar to our results measured on the leader for 3 nodes (3-10\(\mu\)s) and, not surprisingly, lower than those measured on the unoptimized software clients. While this system certainly proves that it is possible to achieve low latency consensus over Infiniband networks and explores the interesting idea of consensus protocols built on top of RDMA, our hardware-based design achieves higher throughput already on commodity 10 GbE and TCP/IP.

FaRM [35] is a distributed main-memory key value store with strong consensus for replication and designed for remote memory access over 40Gbps Ethernet and Infiniband. It explores design trade-offs and optimizations for one-sided memory operations and it demonstrates very high scalability and also high throughput for mixed workloads (up to 10M requests/s per node). FaRM uses a replication factor of three for most experiments and our hardware solution performs comparably both in terms of key-value store performance (the hardware hash table reaches 10 Gbps line-rate for most workloads [50]) and also in terms of consensus rounds per second, even though the FPGA version is running on a slower network.
NetPaxos [26] is a prototype implementation of Paxos at the network level. It consists of a set of OpenFlow extensions implementing Paxos on SDN switches; it also offers an alternative, optimistic protocol which can be implemented without changes to the OpenFlow API that relies on the network for message ordering in low traffic situations. Best case scenarios for NetPaxos exhibit two orders of magnitude higher latency than our system, FARM, or DARE. It can also sustain much lower throughput (60 k requests/s). The authors point out that actual implementations will have additional overheads. This seems to indicate that it is not enough to push consensus into the network but it is also necessary to optimize the network and focus on latency to achieve good performance. In more recent work [25] the same authors explore and extend P4 to implement Paxos in switches. While P4 enables implementing complex functionality in network devices, the high level of abstraction it provides might make it difficult to implement the kind of protocol optimizations we describe in this work and that are necessary to achieve performance comparable to that of conventional systems running over Infiniband.

Similar to the previously mentioned work, Speculative Paxos[93] suggests to push certain functionality into the network, e.g., message ordering. The design relies on specific datacenter characteristics, such as the structured topology, high reliability and extensibility of the network through SDN. Thereby, it could execute requests speculatively and synchronization between replicas only has to occur periodically. Simulations of the proposed design show that with increasing number of out-of-order messages the throughput starts to decrease quickly, since the protocol and application have to rollback transactions.

AllConcur [92] implements a leaderless atomic broadcast protocol. Its main goal is to provide high rate coordination in a distributed system, as opposed to the goal of reliable replication in Caribou. While leader-based replication has benefits in terms of update serialization and assumptions about consensus round termination, it has the drawback that the leader’s network bandwidth will eventually become a bottleneck with a large number of nodes (see Section 5.3.4 for detailed discussion). AllConcur, on the other hand, is able to spread load across participating nodes and can scale to hundreds of nodes, which would be unfeasible with Caribou’s leader based approach. In terms of performance, AllConcur uses either Infiniband verbs (in which case its atomic broadcast latency is similar to Caribou), or TCP (in which case it has latencies an order of magnitude higher than the FPGA). Furthermore, because in regular servers reaching the network interface introduces a higher overhead than in a specialized hardware solution, high performance can be achieved only when using batching of requests.
3.3.3 Quicker and Specialized Networking

One of the big challenges for software applications facing the network is that a significant time is spent in the OS layers of the network stack [89, 54] and on multi-core architectures response times can increase as a result of context switching and memory copies from the NIC to the right CPU core. As a result, there are multiple frameworks for user-space networking [46, 54], and on the other end of the spectrum, operating systems [10, 89] that aim to speed up networking by separating scheduling and management tasks. The use of RDMA [34, 83] is also becoming common to alleviate current bottlenecks, but there are many (legacy) systems that rely on the guarantees provided by TCP, such as congestion control, in-order delivery and reliable transmission. Although some functionality of the network stack is offloaded to the NIC, processing TCP packets still consumes significant compute resources at the expense of the applications. Hardware systems, as we present in this work, are implementing network processing as a dataflow pipeline and thereby can provide very high performance combined with the robustness and features of TCP.

A good example of what can be achieved with user-space networking is MICA [75], a key-value store built from the ground up using Intel’s DPDK [46] library. The results of this work are very promising: when using a minimalistic stateless protocol the complete system demonstrates over 70 million requests per second over more than 66Gbps network bandwidth (using a total of 8 network interfaces and 16 cores). It is important to note however that in MICA and similar systems skewed workloads will experience slowdowns due to the partitioned nature of the data structures. Additionally, a significant part of the server’s logic (for instance computing the hash function on keys, or load balancing) is offloaded to clients. Our aim with the hardware solution on the other hand was to offer high throughput, low latency while relying on simple clients and commodity networks.

3.4 Summary and Discussion

In this chapter we described the design of the replication module in Caribou. Our main goal was to provide fault tolerance for a distributed key-value store in hardware with minimal performance overhead. The replication functionality is built on top of a common atomic broadcast protocol, that is part of Zookeeper [43]. This protocol allows pipelining the sending of proposals and waiting for protocol-level acknowledgments, and therefore maps
well to an FPGA-based implementation. Furthermore, the logic driving the protocol can be expressed as a state machine, simplifying the implementation effort in hardware. Our implementation separates control and data “plains” from each other, allowing them to be scaled separately. This is an important property for ensuring that the consensus logic can be ported to other devices and faster networks.

In the process of implementing this functionality in hardware, we also explored a design point not covered in related work. Most related work that aims to speed up distributed consensus (or an equivalent operation) does so either by using traditional networks and algorithmic changes [109, 84], or specialized networks (RDMA over Infiniband) and protocols designed from scratch for these networks, e.g., [91]. The former methods are usually bottlenecked by the networking overhead in most commodity operating systems. In contrast, the latter methods achieve excellent performance, but are constrained to special environments. Caribou takes a different approach: it relies on a well-established algorithm and runs on commodity networking, but exchanges the NIC+CPU combination for a single specialized hardware device. As we will show in Chapter 5, the consensus protocol of Caribou is competitive with RDMA-based software solutions both in terms of throughput and latency. As a result, this work opens up new directions of research in offloading consensus operations to network cards or switches.

The network stack of CaribouMC supports, in addition to regular TCP/IP sockets, dedicated point-to-point connections between FPGAs. Through these we could also explore how the hardware consensus logic would benefit from lower overhead networking. As we will show in the Evaluation, the dedicated links cut latencies between nodes dramatically, and speed up consensus rounds by more than 2x. This means that even though Caribou does not require specialized networking to reach good performance, it can benefit from its presence.
Chapter 4

Near Data Processing

We added in-storage processing capabilities to Caribou to be able to filter data out before it is sent to the clients, thus reducing the data movement bottleneck between application layers. When pushing down computation to storage it is important that it never slows data retrieval down, even if none of the values can be filtered out.

This chapter describes two different filtering options built into Caribou: comparison-based and regular expression-based filtering. These two can be combined with an optional decompression module, to enable processing even on compressed data. It is possible to include other types of processing as well, but the ones mentioned above illustrate already different design trade-offs. The modules have been designed such that their performance can be adjusted (by increasing their parallelism) to different target levels, corresponding to existing and emerging storage media.

Beyond the actual processing capabilities, the main research contribution of this section is showing that it is possible to retain flexibility in a specialized hardware solution through the use of runtime parameterizable processing units. That is, instead of creating a circuit for a specific filtering expression, we create one for a “class” of expressions. This of course results in larger circuits but it allows the clients to pick filtering expressions and to parameterize the circuits accordingly at run-time.

4.1 Use-case Examples

In Figure 4.1 we illustrate the benefit of in-storage processing with a query that aims to determine the aggregate spending by customers whose name contains “John”, broken
down by region. As shown in the figure, if we move all data to the client for filtering, network will be the bottleneck. If we use in-storage processing to offload the filtering and perform the group-by aggregation on the client, the bottleneck can be reduced or removed altogether.

```
SELECT region, SUM(spent) FROM customer
WHERE age>13 AND name LIKE "%John%"
GROUP BY region
```

![Diagram](image)

**Figure 4.1:** Example of offloading filtering for an SQL-style query to the storage layer.

*Caribou* allows clients to store any type of information as the value associated with a key, but for the in-storage processing part of this work, we assume that the data will be based on relational tuples. We envision two ways that these could be mapped to a key-value store (Figure 4.2):

- Documents – It is possible to express the tuple as a JSON document for instance, explicitly encoding the value of all columns (attributes) in the original table.

- Binary tuple – This representation is closer to how tuples are stored in traditional database engines. In this case the schema of the tuple is implicit and the clients have the knowledge of which column spans which bytes. Variable length columns are assumed to be at the end of the value. The length of each value is encoded in its first two bytes.
In this chapter we will describe a regular expression-based filtering unit that is suitable to work on both formats and a condition-based filtering unit that is suitable to be used with the second, binary, format. We also present a module that decompresses data before the filtering steps, this could be used for both types of data.

In this work we use DRAM for storage as it allows us to explore near-data processing ideas for upcoming memory technologies without being limited by either random access performance or bandwidth of the underlying medium. Furthermore, storage is already moving away from strict block-based interfaces and allows more flexible access methods [13]. For instance, Intel Optane SSDs\(^1\) can be used as a persistent extension of RAM, blurring the boundaries between main-memory and persistent storage. Our processing modules are designed such that their nominal throughput can be adjusted to any desired level in the 1-10 GB/s range, and as a result, should be useful even outside the context of Caribou.

### Relational table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Employer</th>
<th>Hired-on</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John S.</td>
<td>32</td>
<td>Some Inc.</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Adam W.</td>
<td>50</td>
<td>Some Inc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Mark X.</td>
<td>21</td>
<td>Other Inc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Document store:

```
{ somekey, "{name: John Smith, age: 12, employer: SomeInc., hired-on: 01-01-2013}" }  
```

### Tuple store:

```
{ somekey, | value_length | value_length | age   | hired-day | hired-month | hired-year | name-length | emp-length | name | name | name | name | emp | emp |
        | 1 byte       |              |       |           |             |            |             |           |      |      |      |      |     |     |
```  

Figure 4.2: Two examples of mapping relational tuples to a key-value stores.

---

4.2 Regular Expression Matching

4.2.1 Design Overview

The main reason we chose to implement regular expression matching is that it is a versatile tool for expressing different conditions, both in text data and in binary data as well. For values that contain strings, they can be used to implement the equivalent of database LIKE operators, or basic JSON or XML parsing, to identify some specific subfield. In binary data it can be used to check, for instance, if a part of the value belongs to a set or not.

Regular expressions are a way of defining a search pattern in a string [4], or a pattern for replacement. In our case we focus on searching and support a subset of features of POSIX Regular Expressions\(^2\). In particular, we include single characters, ranges, sets, choice, repetitions and wild cards. Due to the fact that expressions are mapped to a circuit on the FPGA, there is an upper bound on expression complexity that can be supported.

Our work has been inspired by related work such as [118] and [107], with the difference that in our system the regular expressions must be quickly modifiable on the FPGA. Most of the related work uses regular expressions in the area of network intrusion detection, where the goal is to compile a large number of rules into a single large expression that can be matched in a data stream arriving at high rate. Since the rule set changes very little, the circuit can be fixed at compile time with no need to change it at run-time. In our use-case, however, each incoming request might require a different regular expression to be executed.

In software regular expressions are usually translated to deterministic finite state automaton (DFA [4]). The advantage of DFAs is that, by construction, only a single state can be active at any time and, determined by the next parsed symbol, the choice of next state is also deterministic. In software this means that with each input symbol, only the transitions of the current state have to be considered (usually a small number), which helps with cache locality and reduces the compute overhead. The main drawback of using DFAs, however, is that removing non-determinism from the pattern can require in essence listing all possible states, which leads to a so called “state explosion”, leading to very large DFAs.

\(^2\)https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions
4.2. Regular Expression Matching

While non-deterministic finite state automaton (NFA) solve the state explosion problem, because any number of states can be active at the same time, taking transitions can become very compute intensive in software. Furthermore, locality is also lost because different parts of the NFA might need to be looked at for the same input. For this reason, NFAs are less often used in software, but are a good match for hardware. This is because in FPGAs parallel execution comes for free, but space for storing the automaton is more scarce.

In Figure 4.3 we show how the following regular expression maps to an NFA:

\[ \text{Name: } [\ \text{t}]^* \text{John.}.* \text{Smith} \]

Figure 4.3: Example NFA matching a regular expression. Transitions between states are taken when encountering specific characters in the input.

It is clear that in these types of expressions, the number of total characters is much larger than the actual states required to express the matching logic. This particular expression can be represented with “tokens” as AB+C.*D. Figure 4.4 shows the tokens highlighted on the original NFA.

Figure 4.4: Example NFA matching a regular expression. Characters that are used to match a linear sequence of states have been highlighted as “Tokens.”
It is possible to map the above presented NFA to a circuit on the FPGA, but if translated as-is, it would only be able to match this particular expression. Instead, to make it able to match any expression that fits in some number of NFA states, it is possible to create a fully connected state graph on the FPGA, where edges are enabled or disabled at runtime, and their transition conditions are also specified at runtime. Figure 4.5 shows how such a circuit would look like (only showing the edges leaving State 0). It is clear that such a solution will quickly grow too large on the FPGA, because each connection requires hardware resources.

To provide flexibility and at the same time to overcome the resource consumption issue, in our design we decouple character matching from the NFA state transition logic and allow a sequence of characters to be matched as a single token. The token matchers are always sequential inside, therefore require less resources for a given number of characters than fully-connected states for the same number of characters. Ranges (e.g., [0-9]) are just special characters, and can also be part of a token. When creating the circuit we can modify two parameters: a) the maximum number of characters it can identify and b) the maximum number of states that the NFA can contain. The actual number to use at runtime and the characters to associate with each token, as well as the state transitions are all runtime parameters.

To derive the parameters, the software clients rely on a small C++ program to parse the expression. In this program we extract character sequences and replace them with tokens, then set up the states of the NFA based on these tokens.
4.2. Regular Expression Matching

4.2.2 Inside a Single Unit

Overview. The core of our implementation is the regular expression matcher module that works on the input one character at a time and is built from three parts: Input Manager, Tokenizer (containing the character matchers) and State Blocks (the NFA states). Figure 4.6 provides an overview of an example matcher with up to 4 tokens and 4 states (we chose these sizes for illustration purposes only).

Inside the Tokenizer there are several Character matchers. These are simple units that have a small register inside, that holds the character to compare against. They output a signal towards the state graph if the input character matches the one in the register. They output the same signal, with one cycle delay, to the next character matcher. This is used by the next character matcher when implementing a sequence (token). A Token is a sequence of character matchers where only the last one’s “activated” output signal is taken into account.

To map character matcher (token) activations to state transitions, we use a series of bitmaps that define which character matcher’s output should act as an activation signal to which state (see Figure 4.7 on how the output of the character matchers is translated to activation signals for states). Similarly, the states have a bitmap that defines which state transitions to which other state. The previously mentioned problem with the size of the fully connected graph is clear here: the bitmaps grow quadratically with the number of states, but only linearly with the number of character matchers.

Figure 4.6: High level view of a single Regular Expression matching unit.
Chapter 4. Near Data Processing

Figure 4.7: Output from character matchers is represented as a vector and AND-ed with the sensitivity list of each state. If the result is not zero, the state is activated.

In Figure 4.8 we show how the different bitmaps map to a single configuration vector. Clients have to communicate this vector to the regular expression matcher to parameterize it at runtime.

Figure 4.8: Mapping the bitmaps used for configuring each part of the regular expression matcher to a single configuration vector.

**Input Manager.** As seen in Figure 4.6, the Input Manager is responsible for managing runtime parametrization. Once the configuration register has been loaded, it will enable
4.2. Regular Expression Matching

the configuration mode of all character matchers in the Tokenizer and will set up state transitions.

The Input Manager unit, in addition to the configuration vector, takes care of converting from 512 bit input words coming from the processing data bus to single characters, and feeding these to the Tokenizer. It also counts how many characters have been processed and, whenever a match happens, it will output the character location of that match. Additionally, if a match happens, it will skip the remainder of the string and fast-forward the input to the beginning of the next string.

![Figure 4.9: Inside a character matcher.](image)

**Tokenizer.** The Tokenizer unit is built using a series of Character Matchers. These matchers can be assembled either to match a sequence of characters or a range. To match a range, two character matchers are fused, one comparing the lower bound and the other one the upper bound. Corresponding to this, matchers can be of two types: $L$ and $H$, as shown in Figure 4.9. When matching a single character they behave the same, but when matching in “range” mode, the $L$ type evaluates the lower bound (larger or equal) and the $H$ type the upper bound (smaller or equal comparison).

They are deployed in an alternating sequence (in Figure 4.6, blocks 0 and 2 are type $L$, and 1 and 3 are $H$). This is so that they can be either used one by one as character matchers (simple equality) or as a pair to implement range comparisons. In the Tokenizer a sequence of character matchers is mapped to a Token by ignoring the *active* output for all but the last character in the sequence – so in some sense the Tokens only exist in the way the circuit is parametrized at runtime. The Tokenizer block has a delay of one clock cycle and can consume an input character every cycle. The output is a bit-vector concatenated from each character matcher’s output (left-hand side of Figure 4.7).
Chapter 4. Near Data Processing

The Tokenizer is parametrized using the following data ($C$ denotes the number of character matchers and $S$ denotes the number of states in the expression matcher):

- **CHAR_CONTENT**: $C \times 8$ bits, defining the character to match against for each character matcher block.
- **IS_CHAINED**: $C$ bits, indicating if the character is part of a sequence, therefore only matches if the preceding character also matched in the previous cycle. The first character in a sequence will have this not set.
- **IS_RANGE**: $C/2$ bits, indicating if a pair of matchers is used for comparing the input to a range.

Each character matcher runs the following logic:

```plaintext
char_match[t] := (CHAR_CONTENT[t]==in_char || (IS_HTYPE[t] & IS_RANGE[t/2] &&
char_ssmaller[t-1]==1 && CHAR_CONTENT[t]>in_char))
char_ssmaller[t] := (CHAR_CONTENT[t]<=in_char)
token_active[t] := (char_match[t] && (!ISCHAINED[t] || was_active[t-1]))
```

The states are represented as very simple logic blocks that in essence have a single bit of state (whether they are active or not). The output of the Tokenizer is received every cycle as a bit-vector (**TOKEN_ACTIVE**) and the states are updated all at once, synchronously. To perform this update the following parametrization data is used:

- **STATE_TRIGS**: $S \times C$ bits, a vector for each state that encodes which tokens (i.e., character matcher “active” output) the state is sensitive to.
- **STATE_DELAY**: $S \times \log_2(C)$ bits, the length of the token the state is sensitive to. If there are multiple tokens of different length that should trigger the same state, the software will create additional states.
- **STATE_INEDGE**: $S \times S$ bits, a vector encoding which states can transition to state <i>i</i> (incoming edge-based encoding).
- **STATE_STICKY**: $S$ bits, if a state is sticky then it has a .* edge onto itself. This is an optimization to save states for expressions with wild cards between sequences.
These vectors are used in the following way to determine at the next cycle what states will be active:

| may_activate[s] := (TOKEN_ACTIVE & STATE_TRIGS[s] != 0) || STATE_STICKY==1 || STATE_TRIGS[s]==0 |
| has_active_inedge[s] := (STATE_ACTIVE & STATE_INEDGE[s] != 0) || STATE_INEDGE[s]==0 |
| will_activate[s] := may_activate[s] && has_active_inedge[s] |

The states also have a small shift register inside them which delays the incoming activation signals using a shift register as many cycles as the sequence takes to be detected for the given state. This is configured using the STATE_DELAY vector.

The regular expression is considered to be a match if the last state becomes active. The configuration software always maps the accepting state to the last state. If there are multiple accepting states, these have to be merged together so that they can be mapped to the implicit accepting state of the circuit.

It is clear that, in the current setup, one will always be able to create a regular expression that does not fit in the available space. It would be possible to extend this architecture with a configuration register that can hold multiple configuration words and on partial match of the regular expression the circuit can re-parameterize itself with the next word. This way, long regular expressions can be tiled into multiple shorter ones and still be executed in hardware. Since the cost of reloading the configuration is only 2 clock cycles, the processing could be resumed quickly. It is clear, however, that in this case the design will potentially waste logic in the average case in the expectation of a very complex regular expression.

### 4.2.3 Increased Throughput through Parallelism

The throughput of a circuit on an FPGA is directly proportional to the clock frequency used. The clock determines the rate at which logical functions are re-evaluated. The designer of a circuit has complete control over the clock, and can choose any frequency up to 400-500MHz on most evaluation boards. The choice is limited, however, by the complexity of combinatorial logic that needs to be evaluated between two rising clock edges (for more detail, see Chapter 6 of [59]). The more complex the logic, the lower maximum frequency will the circuit be able to run at. It is possible to define multiple
clock-regions on an FPGA, which means that parts of the circuit can execute at a higher
clock than, for instance, the parts that interface the network.

While increasing clock frequencies leads to immediate gains in compute performance, this
effect is limited to a factor of two or three improvement (if we compare against a 156MHz
network clock). In the case of the regex module, if we want a single unit to be able to
consume input at 10Gbps, we would need to clock it above 1GHz. This is unrealistic on
an FPGA and as a result we have to look at other methods for increasing the throughput
of this module without redesigning it.

We use data parallelism to increase the throughput of this module. That is, we deploy
multiple units in parallel and distribute work equally to all of them. As shown in Algo-
rithm 4.1 and 4.2, this is achieved using a round-robin algorithm both on the input side
and on the output side. On the output side each result is constant size, but the input will
have strings of variable length. Therefore, the String Router has to observe the last signal
associated with values.

Algorithm 4.1: String Router in the Regular Expression Matcher Module

```
1 IDX ← 0
2 for each input WORD, LAST
3 begin
4   InputFifo[IDX].push WORD
5   if LAST = true
6     IDX ← (IDX + 1) mod N
7 end
```

Algorithm 4.2: Output Merger of the Regular Expression Matcher Module

```
1 IDX ← 0
2 while true
3 begin
4   if ResultFifo[IDX] not empty
5     output ← ResultFifo[IDX].pop
6     IDX ← (IDX + 1) mod N
7 end
```

In order to save resources, we use a faster clock to drive the regular expression units
than for the main pipeline. This way the same throughput levels can be reached with
less parallel units (requiring less resources overall). As shown in Figure 4.10, the regular
expression module, with the parallel matchers inside, has two input clocks. One is the
4.2. Regular Expression Matching

Figure 4.10: Regex throughput is increased with a data-parallel architecture. Note the clock domain crossing through the input and output FIFOs.

network clock, and this drives the writing side of the input FIFOs and the reading side of the output FIFOs, interfacing with the modules before and after the regular expression matcher module. The second clock drives the actual matching logic inside the module. This can be changed independently from the network clock, and we experimented with it up to 343 MHz (i.e., a multiplier of 2.2 on the network clock). Clocking the circuit beyond this rate leads to timing violations that are increasingly challenging to resolve.

The clock domain crossing happens over the individual input/output FIFOs of each regular expression matching unit because, would we have chosen to cross it before the data enters the module, the whole distribution logic and round-robin assignment would have had happened in the faster clock domain. Even though the combinatorial logic is simple enough to meet timing, placing and routing would become more difficult with shorter maximum signal paths.

Table 4.1 shows how the throughput of the regular expression unit scales with the number of parallel matcher cores and the clock rate chosen. These values are nominal maximum
throughput, assuming that the pattern is not found in the input (otherwise the matchers can discard data immediately after a match, which would result in a much higher bandwidth). In our experiments we saw that typically more than 90% of the nominal throughput can be achieved for any input.

In the Evaluation section we explore in more detail the exact configurations in which we deployed the regular expression module. In most cases we run it at double the network clock with 16 parallel units (16 @ 312MHz), that translates to a maximum nominal throughput of 4.9GB/s. As for the number of characters and states, we configured the matchers most of the time with characters in the range of 10-20 and states between 4 and 8. In a real-world deployment, the exact choice would be driven by the expected complexity and type of expressions executed on the data.

Table 4.1: Maximum throughput of the regular expression matchers as a function of parallelism.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Tput. at 156 MHz</th>
<th>Tput. at 312 MHz</th>
<th>Tput. at 343 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1248 MB/s</td>
<td>2496 MB/s</td>
<td>2746 MB/s</td>
</tr>
<tr>
<td>16</td>
<td>2496 MB/s</td>
<td>4992 MB/s</td>
<td>5491 MB/s</td>
</tr>
<tr>
<td>24</td>
<td>3744 MB/s</td>
<td>7512 MB/s</td>
<td>8263 MB/s</td>
</tr>
<tr>
<td>32</td>
<td>4992 MB/s</td>
<td>9984 MB/s</td>
<td>10982* MB/s</td>
</tr>
</tbody>
</table>

Note: items marked with * are beyond the maximum rate of the data bus that brings data from DDR3 memory at the rate of 64B x 156.25MHz.

4.3 Conditionals

4.3.1 Design Overview

For this filtering unit we assume, as earlier explained in Section 4.1, a mapping from relational tuples to byte-arrays in the storage. Tuples can contain a combination of fixed and variable size columns, with the fixed size columns usually stored at the beginning of the record. The regular expression matcher can be used to filter out string-based columns, and to be able to filter based on the fixed-size ones, we created so called “condition matchers” that compare a specific offset in the value to a constant.

This enables clients to push down the filtering expression such as the following one to the storage: \texttt{where col2=123 and col3<4 and col4='abcd'}. All three parts can be
4.3. Conditionals

handled by our condition matchers, but it is up to the client to map the name of a column
(attribute of the tuple) to a specific byte-offset in the value.

A single conditional matcher compares a 32 bit word at a specific offset of the value to a
constant. The comparison can be done using different functions. We have implemented
==, !=, < and >, but this could be easily extended. Each matcher is parametrized at
runtime. This takes one clock cycle and does not degrade performance, even if every
lookup uses predicates. The parameter word encodes the predicate for each comparator
unit as a triplet: 1) a byte offset in the value representing the start of the column, 2) a
comparison function, and 3) a 32 bit value that is evaluated against the value at the given
offset with the given function. For the previous example, for instance, col2 might map to
byte 0-3 in the value, = is function number 0 and the constant 123 is provided as-is.

To be able to execute filters that contain more than a single comparison, we instantiate
multiple matchers. They are assembled into a pipeline, as shown in Figure 4.11. This
allows data to be simply streamed through these steps and be tagged in the process with
a match-bit. The match-bit will determine whether a value is to be sent to the client,
or to be dropped inside the storage node. In our current implementation we only allow
the logical “and” operation between different conditions, but this could be extended to
arbitrary combinations using the same lookup table technique we used in IBEX [113].

Figure 4.11: Conditional matchers compare a given offset in the value to an input constant.
By chaining multiple of them, more complex conditions can be expressed.
Figure 4.12: Conditional matchers are pipelined and contain several parameterizable elements (offset, function to compare with and constant to compare to).

### 4.3.2 Internal Architecture

Figure 4.12 illustrates the internal of each conditional matcher. Data flows through from left to right in 512 bit words (this is the memory word width). This circuit is designed to process values at one word/cycle rate, and it requires a single cycle pause between values.

In addition to the valid/ready signals, that have been omitted from the figure for simplicity, the data is tagged with a last and drop signal as well. The last word of the value is marked by the last flag, and in case drop is also enabled with the last flag, it means that the value has not met one of the conditions and will not be sent to the client.

Even though comparing for equality is not expensive in hardware, less-than and more-than comparisons require significantly more circuitry. Since a conditional matcher needs to be able to compare its constant to any offset in the value, two implementation options are possible: one could replicate the constant and the comparison circuit to perform a parallel comparison starting at each byte in the input in parallel, and then discard all but one result, or have a single comparison circuit and “bring” the desired subset of bytes to it. We chose the second option, and use a multiplexer to select the part of the value that is of interest.
The lower bits of the offset parameter are used to drive a multiplexer, as can be seen in the upper part of Figure 4.12. The result of this comparison is written into a register. On the output side, the last signal is used to read out this register and to output the drop signal. We illustrate the behavior in Figure 4.13, where it can be seen that all output values have the “last” signal set on their last data word, with some of them also having a “drop” signal set in the same cycle. This module re-computes the last signal for each value for which it is not provided with the input (see the values without an asterisk in the figure). This functionality is necessary because values coming directly from memory will not have the last bit set\(^3\).

The last signal is generated using the length of the values that is encoded in the first two bytes, and therefore it is possible to simply count down from the beginning of a value to find its last word. The same logic is used to enable the storage of the comparison result, when the word in processing is the one that contains the desired offset.

Figure 4.13: The condition matcher receives values that can span multiple cycles and can optionally be already tagged with a “last” signal corresponding to the last word of a value (see values with * in the figure). Otherwise, the module creates the last signals based on the length stored within the value. In this example, only Value2 does not match the condition. Ready signals are omitted for simplicity.

\(^3\)It will be actually set, but it represents the last word belonging to a memory request. For single-key lookups this will correspond to the end of the value. In case of scans, on the other hand, a single read command might retrieve many values at the same time. As a result, we can’t rely on the last bit being set for all values coming from memory. In case the Decompression unit is instantiated, this sits in front of the other processing units and outputs data that is correctly tagged with last bits.
4.3.3 Throughput Scaling

In our current design the conditional matchers run at the network clock (156.25 MHz) and use a 64 B wide data bus. One of the expensive parts in their architecture is the registering of wide data words and the multiplexing from every possible byte-offset to the comparators. It would be possible to save resources by doubling the clock frequency of this module and halving its internal data lines to 32 B (the overall throughput would stay constant).

Table 4.2 illustrates what is the throughput of this module with increasing value sizes, as the one cycle overhead between values becomes less and less significant. It also shows how its behavior would change with the faster clock / narrower bus modification. While at first thought the throughput should stay the same, the one cycle overhead is reduced in absolute terms thanks to the faster clock and as a result the achievable throughputs are higher.

Table 4.2: Measured and projected throughput of the conditional matchers for different value sizes.

<table>
<thead>
<tr>
<th>Value size [B]</th>
<th>Tput. at 156 MHz</th>
<th>Tput. at 313 MHz and 32 B data width</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4992 MB/s</td>
<td>6677 MB/s</td>
</tr>
<tr>
<td>128</td>
<td>6656 MB/s</td>
<td>8012 MB/s</td>
</tr>
<tr>
<td>256</td>
<td>7987 MB/s</td>
<td>8903 MB/s</td>
</tr>
<tr>
<td>512</td>
<td>8874 MB/s</td>
<td>9426 MB/s</td>
</tr>
<tr>
<td>1024</td>
<td>9397 MB/s</td>
<td>9712 MB/s</td>
</tr>
</tbody>
</table>
4.4 Decompression

4.4.1 Design Overview and Algorithm

Another form of processing in storage that we explore is decompression. Since in many modern systems compression is used as a way of fitting more data into the available storage, or to optimize data movement across layers we implement a decompression module to make it possible to evaluate selections even if the data is stored in a compressed format in memory.

We implement LZ77 [120] as proof of concept, because it is a basic building block for more complex decompression solutions and works well for text data. Since the decoding works at the granularity of bytes, this module in hardware can produce one byte of output per clock cycle.

This algorithm relies on a sliding window that keeps track of the decompressed data. The compressed data is a combination of “literal” bytes, that are transcribed to the output as is, and pointer/length pairs that instruct the copying of a number of bytes from the sliding window starting from a specific offset (counting backwards from the current position). The distinction between the two is made by a flag bit that prefixes all words.

As an example, if the data to be compressed is:

ETH Zurich Uni Zurich

It could be compressed into the following sequence:

'E', 'T', 'H', ' ', 'Z', 'u', 'r', 'i',
'c', 'h', ' ', 'U', 'n', 'i', [offset:10 length:6]

In this particular example, each literal byte requires an additional bit for storage and the copy operation has to be encoded as multiple bytes. In our deployment we used 12 bits for the offset and 4 for the length, which means that the compressed string would be 18 bytes long, compared to the 21 bytes in the original string.

With the sliding window approach it is possible to express data copy operations that extend into the “future”, because the state of the window is updated with each character that is output. For instance, if we want to compress the following phone number: 00417777778 it could be expressed as:

'0', '0', '4', '1', '7', [offset:1 length:6], '8'
This works because even though when we start copying from the end of the sliding window there is only a single character to be read, as we copy it to the output, it gets added to the sliding window as well, allowing us to copy the second character, and so on.

To compress values for our benchmarks we used a Python-based implementation of the LZ77 compression algorithm\(^4\). One detail specific to our implementation is that, because the decompressed value needs to encode the true value length in its first two bytes, we compress the value together with these two bytes. As a result, the first two bytes in the compressed value do not correspond to the uncompressed length, but to the compressed one.

### 4.4.2 Internal Architecture

![Decompression unit diagram](image)

Figure 4.14: Decompression units are built with a state machine controlling the input and decoding commands, and a BRAM for storing the sliding window data.

Figure 4.14 shows the internal workings of the decompression unit. Data is received on the left and the commands are decoded one by one by shifting the input word repeatedly. One of the challenges in implementing this unit has been the variable length shifting required by the logic (9 bits for literal values, 17 bits for copy commands) which at 512 bit word length leads to fairly complex shifting logic. The output of the module is reassembled into 512 bit words using a simple shifter, that always shifts by 1 byte.

\(^4\)https://github.com/manassra/LZ77-Compressor
The state machine that controls the decompression will decide whether the input is a literal byte, and if yes, it will write it to the sliding window and to the output shift register at the same time. If it is a copy command, it will issue the read address to the memory implementing the sliding window, and change the multiplexer at the write-side of the memory to the output word. This is required because the sliding window needs to be immediately updated with any output word as the algorithm allows for copying data that has not been written yet. Herein lies the second challenge of the decompression module, because the memory has a one cycle latency on reads. In the current implementation, copy operations introduce a one cycle delay which is required to read the BRAM. We will show in the Evaluation that this reduces throughput by a significant factor only if the data is heavily compressed and it is composed of copy words predominantly. With the LZ77 algorithm this, however, will likely not happen unless the data is along sequence of repeated words. As a result, for the current version of the system we chose to introduce one cycle delay in the decoding instead of opting for a more complex logic on the input side.

Currently, this design has difficulty meeting timing when clocked beyond 250MHz and this mostly is due to the combinatorial paths around the decoding and shifting of the input words. In the future this could be improved, however, by relaxing the requirement that the module is able to consume an input command (be it literal or copy) every cycle. This way the shifting could be done at more regular sizes and this would help with timing. This way, however, handling a copy word would still require more than one cycle, but the overhead would be moved from the BRAM access part to the input handling part, and the impact of the extra cycle would also decrease the faster we clock this module.

### 4.4.3 Increased Throughput through Parallelism

The unit produces at most one byte per cycle on the output and therefore it needs to be replicated in a data-parallel fashion for meeting the bandwidth requirements. This is done similarly to the regular expression matching units, with a round-robin logic distributing input and reading out decompressed words from the output FIFOs. This architecture, shown in Figure 4.15, is very similar to the one inside the regular expression matching module.

The input distribution is performed as described in Algorithm 4.1, but in case of the output (see below in Algorithm 4.3), the transition from one decompression unit’s output-FIFO
to the next one is only allowed when the end of the value has been reached. Otherwise different words from different values could be mixed together.

Algorithm 4.3: Output Merger of the Decompression Module

```
1 IDX ← 0
2 while true
3 begin
4 if ResultFifo[IDX] not empty
5 begin
6 output ← ResultFifo[IDX].peek
7 if ResultFifo[IDX].pop is last
8 begin
9 IDX ← (IDX + 1) mod N
10 last ← true
11 else
12 last ← false
13 end
14 end
15 end
```

Figure 4.15: Multiple parallel decompression units are used to reach the desired throughput level.

In contrast to the regular expression matcher cores, where the 1B/cycle throughput is guaranteed (it is a lower bound), a decompression core’s throughput is dependent on the
composition of the input (how many copy-words). The 1B/cycle output of the decompression cores is actually an upper bound, but as we further explore in Section 5.8, on average a throughput close to it can be expected.

4.5 Related Work

4.5.1 Regular Expressions on FPGAs

A large body of related work uses regular expression matching on FPGAs for network intrusion or event detection, e.g., [97, 118, 12, 114]. These are mostly designed for a pre-defined line-rate of 10 to 40 Gbps and their goal is to compare many concurrent patterns to the input stream. In Caribou, and database-related use-cases in general, a single expression needs to be evaluated on a table or collection of data at-a-time after which the circuit needs to be adapted to the next expression. This rules out designs that require reprogramming.

Considerable work has been devoted to increasing the throughput of regular expression matchers on FPGAs, and one way of doing this is by accepting more than one character per cycle. Examples of such work are: [118, 16, 22, 117]. One approach for multi-character processing is building an NFA with multi-character transitions. This leads to an exponential increase in the number of states in the NFA. For that reason further techniques have to be applied to compact the state space and keep the resource consumption at a feasible level. In our work we decided to achieve higher throughput through data parallelism by deploying multiple regex matchers.

There are different ways that other projects implemented runtime parametrization or re-loading of regular expressions. For instance, Eguro et. al. [36] propose dynamic re-configuration to trade off performance for capacity. Thereby virtualizing the fabric and time-multiplexing the computation of multiple complex patterns on the same chip. Other works employ very similar methods to our solution in terms of runtime parameterization by loading registers (or BRAMs) with character and/or state transition data at runtime. In [107] the authors use a Tokenizer that is similar to what we have used, but in their domain the NFA is fixed and only the transition parameters are configurable. In the work of Kaneta et. al. [60] the states transitions can be changed at will, but the characters are processed separately (no compaction of sequences) and the resulting NFAs can not capture some of the features of regular expressions (for instance the | operator for two alternate
possibilities, e.g. (a|b). The authors of [104] implement a similarly flexible system, but they use a DFA-based representation, which only allows a state transition every 2 cycles while our NFA can consume a character every cycle.

4.5.2 Near-data processing

Data movement between layers can be reduced by pushing operators down to storage. This is already done in shared-disk architectures that rely on traditional servers to provide storage supporting query offloading, e.g. in Oracle Exadata [111]. Exadata Storage Servers are capable of complex near-disk processing to filter out tuples early before sending data to the main Oracle engines. As explained before, however, such server-based designs suffer from inefficiencies both in terms of software stack, and in terms of hardware architecture. IBM Netezza [44] uses a similar appliance-based design, but its storage nodes actually contain FPGAs for performing parts of the filtering.

In an effort to reduce the inefficiencies that occur when using server machines to provide storage, there are several proposals for offloading directly to specialized disk or flash controllers [28, 31, 55, 56, 113].

Ibex [113] is one such example, that uses an FPGA. It is a plug-in storage engine for MySQL that uses an FPGA to implement a SATA-II controller for an SSD. It supports offloading of projection, selection and group-by aggregation operations inside the storage engine. IBEX was designed, however, for SATA-II speeds and communicates with the database engine using a 1Gbps link. Nevertheless, the group-by aggregation operator is an interesting candidate for Caribou as well.

Examples such as the SmartSSD work from University of Wisconsin [31] explores how the small ARM processor inside an SSD can be used to execute arbitrary code to push down parts of an application to the storage. In that work, however, the limited compute capability of the small processor was an important limiting factor. More recent work by Samsung [55] demonstrates better performance (by deploying a small pattern matcher IP module beside the ARM Cortex cores) and also a way of compiling application snippets for offloading purposes.

Compared to these two approaches, we explore an alternative design point: smart distributed storage with replication that exposes a high level interface such as those found in shared disk DBMSs and provides performance characteristics on a par with CPUs, but
is implemented with energy efficient specialized hardware such as that found in “active” SSDs.

In the SLICE/CORFU work [7, 110] the authors build a replicated log based on flash memory and specialized hardware connected to a 1Gbps network. This work is targeting an ASIC and implements a Token Ring which connects multiple task-specific processor cores that can perform computation on the data as it is retrieved. In contrast, in our design we expressed the logic as a series of pipelines with well-defined data flows. This makes reasoning about throughput easier. Nonetheless, these works show that ASICs are more efficient than FPGAs, and in case there is enough commonality in workloads, it would be interesting to consider hybrid architectures, mixing general-purpose and specialized cores.

4.6 Summary and Discussion

In this Chapter we presented the design of three different processing units in Caribou. We selected those with database-like workloads in mind, and assuming data that is either stored in a binary format, or is text-based. Furthermore, since storage capacity is often a limiting factor in large distributed systems, we designed the processing elements such that they can be used even if the data in storage has been compressed by the clients. The decompression unit is the first stage in the processing pipeline and can decompress data without reducing the throughput of the rest of the pipeline.

One of the modules we provide for filtering is a simple condition-evaluator that compares a specified offset in the value to a four-byte parameter. This operation can be carried out at high rate, and even in software it would not be compute-bound. Offloading it to storage is beneficial, however, because it reduces the traffic on the network link. In contrast, the regular expression matching module illustrates that when using specialized hardware, it is possible to turn an algorithm that is compute-bound in software, to one that is bandwidth-bound in hardware. This re-design of data processing steps is key in achieving higher efficiency in emerging compute-intensive workloads.

The different modules also showcase two ways that the parallelism of the FPGA can be harnessed: either to provide higher throughput for “processing cores” by implementing a data-parallel block, or to increase the complexity of the expressions that can be evaluated by implementing deeper pipelines.
Chapter 4. Near Data Processing

The main take-away of this chapter is that it is not enough to build circuits that execute the filtering with high bandwidth, it is also important to ensure that they are flexible enough so that potentially each data access could specify its filtering conditions. We achieved this by designing the conditional filters and the regular expression units such that their behavior is driven by a “register file” that can be loaded at run-time. In the future it would be also beneficial to combine runtime parameterization with partial reconfiguration. This would mean that Caribou nodes could have a large library of processing building blocks, each of them flexible in their own way. Even though not all of these would fit on the FPGA at the same time, several of them could be deployed together, on demand, to create a processing pipeline with functionality that matches a specific workload the most.
Chapter 5

Evaluation

This chapter evaluates Caribou both as a whole and as independent modules. The results show that, on the one hand, Caribou offers low latency high throughput data management operations, and on the other hand, it can perform complex but flexible processing on the data. More specifically, Caribou is shown to successfully fulfill the following requirements:

- The ability to communicate with hundreds of software clients – this makes deployment in the datacenter possible.

- Low latency random-access reads and writes, with various value sizes – this shows that Caribou does not introduce a large overhead in accessing the underlying storage media.

- Network-limited throughput both for random access and in-storage scan use-cases – if storage node makes efficient use of the network bandwidth, provisioning storage resources for the application becomes easier.

- Predictable, non-prohibitive, cost of replication to other nodes – and high availability can be ensured.

- Near data processing that is independent of filter complexity and runs at predictable rate – operation push-down to the storage will not have a negative impact on performance, even if no items can be filtered out.
Chapter 5. Evaluation

5.1 Experimental Setup

5.1.1 Evaluation Board and Deployment

Caribou has been designed for and evaluated on Xilinx VC709 evaluation boards\(^1\) (Figure 5.1). These boards are targeted at network-facing applications and are equipped with a relatively large Virtex7 VX690T FPGA. They also have 4+4 GB of DDR3 memory mounted on them. The four SFP+ cages can be independently configured or disabled. In this project we have not used the PCIe connectors. The FMC connector can be used to provide, for instance, SATA connectivity. SATA-III unfortunately, is too slow for our 10Gbps target bandwidth, but in the future the same extension connectors could be used to attach NVMe flash or similar high-throughput durable storage to Caribou.

![Figure 5.1: Overview of the Xilinx VC709 evaluation board. We used the boards stand-alone, powered through a separate power supply not PCIe. (Photo credit: Xilinx)](image)

Our FPGA nodes were deployed stand-alone, without being attached to server machines. However, to ensure that they do not pose a risk inside our cluster-room, we have hollowed out a desktop PC case (removing everything except for the power source) and mounted the VC709 boards in this case (Figure 5.2). They are all powered through the power supply of

\(^1\)https://www.xilinx.com/products/boards-and-kits
the case, and cooling is helped by the fans on the case (the regular operating temperature of the FPGAs is in the 60C-75C range). The FPGAs have been running in our cluster room for more than 2 years and in this time the only failure we experienced was a cooler fan on one of the FPGAs.

We used an additional PC deployed in the vicinity for programming the FPGAs. For this purpose we used the included USB-Jtag cables.

Figure 5.2: Our evaluation boards were installed in a hollowed out desktop machine for deployment in the cluster room (here only 4 FPGAs are pictured). The boards are powered through the power supply of the case.

Even though the TCP/IP stack inside Caribou is able to request an IP address over DHCP, to reduce extra traffic or FPGA-induced chatter over our 10Gbps network, we have used a hand-assigned IP scheme, controlled by the switches on the device. With these, we could control both the subnet and the IP address of each node. All FPGAs were connected to the same 10Gbps switch, but this is not a requirement, they could be spread over multiple racks and switches as well.
Figure 5.3: Abstract view of *Caribou* nodes. In reality all communication happens over a 10Gbps switch.

As depicted in Figure 5.3, by default, we use one node as the leader (master copy) and the others are replicas. Measurements are performed either with the leader only, or by replicating to two more ($R3$) or four more ($R5$) other nodes.

In the beginning, we considered using NetFPGA-10Gs as load generators because they can easily saturate the network for any sized package and are able to measure latency very precisely. We have, however, quickly discovered that traffic generated by a hardware device is too regular and “clean” when compared to real-world clients running in software. For this reason, we decided to conduct all experiments using software clients and measure most response times either in software or directly on the evaluation boards. To generate load, we use up to twelve servers, with dual Xeon E5-2630 v3 CPUs (8 physical cores running at 2.4 GHz) and an Intel 82599ES 10Gbps network adapter per machine, with the standard “ixgbe” drivers. All machines are connected to the same 10Gbps switch as *Caribou*. 
5.1. Experimental Setup

5.1.2 Caribou Variants

We run experiments on Caribou using the two different versions of the key-value store described in earlier chapters. The replication logic is the same for both of them. As a baseline, we use a bare-bones key-value store without replication, that is based on the work in my Master’s thesis. See below the notation and explanation on what variant offers what functionality:

- *Caribou\_MC*: As already presented, this KVS variant uses a memcached-compatible ASCII protocol without processing capabilities. The hash table is potentially lossy. Both the replication module and the KVS uses TCP/IP by default, but the ability to connect FPGAs to each other via the dedicated point-to-point links is there.

- *Caribou\_CH*: This variant is the most feature rich. It offers both random access and scan-based access to the data, ensured by a cuckoo hash table and secondary bitmap data structures. It also includes the processing pipeline that used to filter out values before sent to clients. Its replication module and networking stack is identical to *Caribou\_MC*, but it uses a single TCP interface for communication with both clients and peers.

- *KVS\_Light*: To show that the added features, in terms of networking, replication and memory management do not fundamentally slow the hardware implementations down, we will use as the baseline the key-value store implementation from my master’s thesis, ported to the VC709 boards. This uses a lossy hash table (an earlier version of the one in *Caribou\_MC*), memcached’s ASCII protocol and simple UDP-based networking. Thanks to its simplicity, this system shows what are the lowest latency and highest throughput numbers achievable over 10Gbps Ethernet (and how both *Caribou\_MC* and *Caribou\_CH* are very close to these).

<table>
<thead>
<tr>
<th>Operation types</th>
<th>KVS_Light</th>
<th>Caribou_MC</th>
<th>Caribou_CH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random access read/write</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Scan access &amp; Filtering</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Replication</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 5.1: We benchmark different types of operations with different versions of the system.
Chapter 5. Evaluation

In Table 5.1 we show which variant of the system is used to measure what kinds of operations. Then, in tables 5.2, 5.3 and 5.4 explain how the different variants relate to each other (beyond the differences in hash table design and value management). Overall, the biggest difference is in network packet overhead between $KVS_{Light}$, that uses UDP, and the other variants, that use TCP. Even this difference, however, becomes negligible for packets larger than 128 Bs. For replication-only experiments, due to the decoupling of key-value store and replication logic, $Caribou_{CH}$ and $Caribou_{MC}$ produce virtually identical results.

<table>
<thead>
<tr>
<th></th>
<th>$KVS_{Light}$</th>
<th>$Caribou_{MC}$</th>
<th>$Caribou_{CH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network headers</td>
<td>IP + UDP (8 B)</td>
<td>IP + TCP (20 B)</td>
<td>IP + TCP (20 B)</td>
</tr>
<tr>
<td>Protocol: Writes (req./resp.)</td>
<td>&gt;16 B / 6 B</td>
<td>&gt;16 B / 8 B</td>
<td>16 B / 16 B</td>
</tr>
<tr>
<td>Protocol: Reads (req./resp.)</td>
<td>6 B / &gt;24 B</td>
<td>6 B / &gt;24 B</td>
<td>16 B / 16 B</td>
</tr>
</tbody>
</table>

Table 5.2: Overheads important when comparing throughput of node-local operations on the different variants.

<table>
<thead>
<tr>
<th></th>
<th>$KVS_{Light}$</th>
<th>$Caribou_{MC}$</th>
<th>$Caribou_{CH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client type</td>
<td>memaslap</td>
<td>memaslap</td>
<td>Go-based</td>
</tr>
<tr>
<td>TCP TX latency</td>
<td>N/A</td>
<td>1 DRAM access</td>
<td>Cut-through</td>
</tr>
<tr>
<td>ASCII parsing in HW</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Hash table</td>
<td>Lossy</td>
<td>Lossy</td>
<td>Cuckoo</td>
</tr>
</tbody>
</table>

Table 5.3: Implementation details important when comparing the response time of node-local operations of the different variants

<table>
<thead>
<tr>
<th></th>
<th>$KVS_{Light}$</th>
<th>$Caribou_{MC}$</th>
<th>$Caribou_{CH}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication type</td>
<td>N/A</td>
<td>ZAB</td>
<td>ZAB</td>
</tr>
<tr>
<td>TCP TX latency</td>
<td>N/A</td>
<td>1 DRAM access</td>
<td>Cut-through</td>
</tr>
</tbody>
</table>

Table 5.4: Implementation details important when comparing the throughput and response time of replicated operations of the different variants.
5.1.3 Software Clients

Modified memaslap

For load generation with Caribou$_{MC}$ and $KVS_{Light}$ we used memaslap, which is part of the C/C++ client library for the memcached server\(^2\). We have modified the code to include replication request headers as necessary in front of the requests.

Each client machine is able to generate up to 600 k+ requests/s (the exact figure depends on the transport protocol and request sizes). The benchmarking setup in [112] shows similar per-client capacity: they achieve 800 k+ requests/s for machines with twice as many cores as we have in our servers.

In most of the cases we performed the measurements with memaslap configured to run on 8 threads (spread over multiple processes) and simulating at least 64 connections per client machine, using the ASCII protocol over UDP or TCP. We set the -o flag to 1, which means that after building the initial working set, all other write operations are updates to existing key-value pairs. The access pattern of memaslap is uniformly distributed.

For detailed response time analysis we ran memaslap with the detailed statistics gathering option enabled, so that it would provide not only average response times, but also the distributions. We extended the code path that measures response times to provide a full histogram output in the range of 0-512 $\mu$s, instead of reporting response times in steps of powers of two. Additionally, we added a custom flag to the tool to use the internally measured latencies of the FPGA instead of system time when reporting results.

Binary protocol clients (Go-based)

This client started out as a workaround for the relatively poor extensibility of the libmemcached-based client when it came to implementing an additional protocol to talk to Caribou$_{CH}$.

In the meantime it grew to the main benchmarking client we use.

The client is written in Go, based on an open-source memcached client \(^3\). We have replaced the protocol with our custom one, but kept the overall structure of the library. We added several “knobs” to the client so that it supports the following operational modes:

\(^2\)http://libmemcached.org/libMemcached.html
\(^3\)https://github.com/bradfitz/gomemcache
Chapter 5. Evaluation

- Throughput-oriented mode – will use many parallel sockets and several virtual clients multiplexed on the sockets. Responses are parsed for correctness, but the values are not collected for later presentation. In this mode, it is possible to enable “soft batching” of requests, that is, sending several requests before starting to wait for answers (bulk-synchrony). On the server side each request is seen individually, but we save a lot of Linux-level overhead by flushing the TCP sockets less often, and reading larger chunks of data when parsing the answer.

- Latency measurement mode – Each request is timed with a high resolution timer and a histogram is updated with response times. This mode ideally has only a single client on a single socket\(^4\) to avoid interactions.

- Scan-mode – Data is scanned with a request that encodes the filter parameters. In this mode the client uses a small helper program to compile the regular expression into a configuration vector, and also the conditional filters.

- Conditional-read mode – Similar to the scan mode, but issues conditional get requests. Best combined with latency measurement mode.

Software for managing the replication

The replication group and leader are configured using a small Java helper program. In addition to the configuration, this also implements the same replication protocol and message format as the FPGA, so it can be used to launch dummy nodes for testing purposes.

Real-world dataset

We tested the in-storage processing capabilities of Caribou\(_{CH}\) with a dataset containing stock information. This dataset is from Kaggle\(^5\) and we loaded it in the following way in the key-value store:

The key is a four byte ticker symbol, and a four byte timestamp identifying the day. The value is a concatenation of the date in a year-month-day format and the five properties

---

\(^4\)Not a single core, because due to how the Go application manages network sockets and uses threads, for lowest latencies it needs to be given at least two cores. In contrast, the libmemcached-based client could be run on a single core for minimal latency.

\(^5\)https://www.kaggle.com/dgawlik/nyse contains historical data from the New York Stock Exchange
(open, low, close, high, volume). All these columns are 4 byte numbers. We added a variable length column to the end of the data which contains the ticker symbol, full company name and address as a string. We padded values to 64 and 128 byte length, to coincide with our allocation sizes. Data is not compressed.

The original dataset (S=1) has 800k entries that occupy 128B on average each, leading to a total of 100MB in the key-value store. We have artificially scaled the dataset by repeatedly inserting the same data “offset” by a decade every time (e.g. S=10 is 1GB in size).

The clients that populate the dataset and issue scans are written in Go and they parse the results of a scan, as they arrive, on a single thread. As will be shown later in this section, this single thread quickly becomes a bottleneck for high selectivities. However, for low selectivity, the filtering performance can be measured without slowdown from the clients.

### 5.2 Resource Consumption

On an FPGA the application logic is translated to gates and memory blocks on the device, regardless which parts of it are more commonly used. For this reason, resource consumption is an important metric on FPGAs because it shows what percentage of the chip area is required to implement each individual functionality or module of the application.

In the case of Caribou, most designs do not saturate the chip area completely, which means that further processing capabilities could still be added. Otherwise, in a production-setting, the same logic could be deployed on a smaller FPGA without reducing its performance.

In Table 5.5 we list the real estate requirements of each module in our system, divided into “logic” and “memory” categories. The former is a measure for how many logic slices (a subdivision of the FPGA area) the module uses, while the second one expresses how many blocks of 36Kb on-chip memory it requires for its operation. These are mostly used for buffers and various caching structures. Unfortunately, the resource consumption numbers are poorly comparable across vendors and even across chip families. They can also be heavily influenced, for instance, by the placing and routing process that, in an effort to make the circuit meet timing can replicate parts of the circuit. Nevertheless, these figures provide a relatively reliable metric for comparison between modules.
As depicted in Table 5.5, all modules have fairly modest requirements on the prototype FPGA, and there is space left for implementing additional functionality. The memory controller is one of the largest modules because it is responsible both for accessing memory and acting as an arbiter for many concurrent access channels. The memory controller exposes two physical channels that have to be multiplexed between multiple logical ones (e.g., for the TCP send buffers, hash table, values, etc.). Since the memory channels are 512 b wide proper AXI-Stream channels, the infrastructure with AXI Datamovers and AXI Switches requires significant logic and BRAM resources. Newer FPGA devices have hardened memory controllers that are faster and would not require real estate on the programmable fabric.

The hash tables used in this work are BRAM-heavy due to the various buffers between modules, used to hide the memory access latency, or to act as data buffers for avoiding read-after-write hazards. Somewhat surprisingly, the memory allocation logic is actually comparable in size to the hash table. This is likely because there are multiple operations inside the memory allocator that work at the bit-level on full 512 b words. Furthermore, active pointers are cached on the chip instead of DRAM, which results in higher BRAM requirements.

The string matcher unit has high real estate requirements because it is a combination of multiple parallel units, and the BRAM memories are used as FIFOs for distributing and collecting work. In our current design, even though processing is performed at 1 byte/cycle...
in the “cores”, the input data lines and FIFOs match the memory data width, i.e., 64 B. Other modules, such as the hash table or the memory allocator, have more complex logic than the previously mentioned ones but, since they have narrower internal buses (typically less than 32 B), they consume less resources overall.

Comparison-based filtering is one of the least resource intensive modules because it requires only a few memory-word width registers and combinatorial logic to carry out the comparisons. In our standard deployment we use 4 of these, but as seen in the table, the aggregate resource consumption increases linearly with their number.

The decompression units have a footprint comparable to the comparison-based filters because internally they require at least two large shift-registers and one BRAM to parse the input and compose the output word. Just as with the Regex module, the aggregate resource consumption of the Decompression step is a function of the number of parallel “cores” used.

The size of the networking module is also significant, but it is one of the most complex modules inside Caribou, given the different actions the TCP stack needs to take depending on lost packets, retransmissions, etc. This module has been written in HLS, which means that it is somewhat harder to directly compare its resource usage to the other modules, that have been written in Verilog or VHDL – but at least based on our experience, its final size is in-line with its complexity.

5.3 Get-Put Workloads

5.3.1 Baselines

<table>
<thead>
<tr>
<th>Purpose of the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>In this subsection we establish what behavior we can expect from a hardware-based key-value store. We use response time and throughput numbers from $KVS_{Light}$ to show that hardware-based system have network-bound performance, in contrast to common software solutions. In terms of response time, hardware offers a flat behavior. Since $KVS_{Light}$ has very few features compared to the other two variants, we measure how the added functionality impacts the latency of the core of the key-value store.</td>
</tr>
</tbody>
</table>
Chapter 5. Evaluation

Throughput and Response Time Expectations

Most key-value stores built in hardware follow a design similar to the one we presented in [14], and rely on a large pipeline connecting input and output. If the computations along these pipeline and the memory accesses are sized such as to keep up with the network line-rate, the key-value store can achieve full network-bound throughput. Since most FPGA-based works rely on UDP networking instead of TCP, this even further reduces the overheads in the system.

Figure 5.4 illustrates the maximum throughput of $KVS_{Light}$, and as an extension most similar systems, using UDP packets on a 10Gbps network link. The x-axis represent how the workload goes from read-only (requests to retrieve data), to write-mostly (requests to store data). The size of a read request is very similar to that of a write response, and the other way around, which means that throughput lines are symmetrical. The dashed lines represent the maximum achievable theoretical throughput, and it is only close to the 50% case that the system cannot keep up. The reason for this is that at this point it becomes bottlenecked on its internal data lines connecting “get” and “set” data movers. In case one would want to squeeze out the last bits of throughput for these points, the single pipeline design would need to be modified to have two separate paths.

![Figure 5.4: Throughput of $KVS_{Light}$ for different value sizes as function of read-write mix (dashed lines = 10GbE maximum)](image)

Since $KVS_{Light}$ implements the same protocol as Memcached, we can compare the throughput/response time trade-off of the hardware key-value store and software (Figure 5.5). Memcached is not the fastest key-value store available, but it provides a general purpose service over commodity networks, and latency-wise it is representative of its class.
5.3. Get-Put Workloads

We benchmarked memcached in a four-thread setup, both running on TCP and UDP networking, and our measurements show a trend similar to other recent studies [21]: memcached’s median and 99th percentile increase significantly with load, especially once 75% of the maximum load is reached. Hardware, on the other hand, has the ability to provide flat response time even close to saturation. The behavior of the hardware pipeline here is the best case achievable on the FPGA.

For $KVS_{Light}$ we plot the response time both as measured on the clients and as measured internally (first byte in, last byte out). As can be seen in the figure, the overhead of software and network transmission in our cluster dwarfs the internal cost of handling requests, which is in the single digit $\mu$s range.

![Figure 5.5: Median response time of $KVS_{Light}$ with increasing load for 90% read workload (light lines = 99th percentiles).](image)

(a) 16B key, 128B value  
(b) 16B key, 512B value

**Result summary**

It is possible to design hardware-based key-value stores that saturate the network for any workload and have response times that behave flat even with increasing load. For a simple key-value store implementation, the software client overhead dominates the response time. As a result, for some parts of the evaluation, e.g., when looking at replication, we will report response times as measured on the leader node.
Chapter 5. Evaluation

Internal Latencies for More Complex Logic

Using $KVS_{Light}$, we can establish how much time is spent inside different modules of the design. Table 5.6 show latencies as a function of request size for the networking modules and the key-value store. Using UDP for communication adds only a marginal overhead on top of the Ethernet core. Having these numbers, we can quantify the overhead of additional features and different data structure designs used in Caribou.

Not surprisingly, the TCP/IP requires more cycles to manage sessions and move the data than the stateless UDP protocol, but even so its latencies are low.

Access to DRAM takes typically 40-60 cycles at 156.25 MHz, and this is the same in all versions of Caribou.

<table>
<thead>
<tr>
<th>Module</th>
<th>32B Request</th>
<th>512B Request</th>
<th>1KB Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet loopback</td>
<td>0.6µs</td>
<td>1.4µs</td>
<td>2.2µs</td>
</tr>
<tr>
<td>UDP loopback</td>
<td>0.7µs</td>
<td>1.5µs</td>
<td>2.3µs</td>
</tr>
<tr>
<td>TCP loopback [98]</td>
<td>2.1µs</td>
<td>4.2µs</td>
<td>6.3µs</td>
</tr>
<tr>
<td>DRAM read</td>
<td></td>
<td>0.2-0.3µs</td>
<td></td>
</tr>
<tr>
<td>Ping from client</td>
<td>15µs</td>
<td>35µs</td>
<td>–</td>
</tr>
<tr>
<td>$KVS_{Light}$ hash table + value access</td>
<td>1.2µs</td>
<td>1.7µs</td>
<td>2.3µs</td>
</tr>
</tbody>
</table>

Table 5.6: Latencies of different “infrastructure blocks” in $KVS_{Light}$ and Caribou$_{MC}$. All values, except the last row, are independent of our implementation.

The time it takes to push a request through the key-value store mostly determined by the cost of hashing, parsing and hash table access. In $KVS_{Light}$ this cost is between 1.2-2.3µs. As shown in Table 5.7, the latencies of Caribou$_{MC}$ are in a similar range. This is not surprising because the two systems are based on the same underlying hash table design and their protocol parsing is logically equivalent (the newer implementation has more error-handling logic and some new features, but fundamentally the two systems are equivalent). Its latency is more sensitive to the key-size than the value-size, because the hashing unit adds latency relative to the key-length, but overall the numbers are low. The overhead of the protocol parsing logic on the receive side is at least 60 cycles (384 ns running at 156.25 MHz).

For Caribou$_{CH}$, the range of key sizes is different because its hash table supports at most 16B keys. We show in Table 5.8 how the internal latencies of the Cuckoo hash table evolve
with larger value sizes. Even though for sets and inserts memory allocation also has to be carried out, this is done in parallel to the main hash table logic and therefore its latency is mostly hidden from the clients. Furthermore, Caribou_{CH} offers overall lower latencies than the other two variants, even though those have no memory allocation logic, because in Caribou_{CH} parsing the requests is much cheaper than in Caribou_{MC} and KVS_{Light}, which use an ASCII-based protocol.

<table>
<thead>
<tr>
<th>Key / Value Size</th>
<th>64B</th>
<th>128B</th>
<th>512B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get 8B Key</td>
<td>1.3µs</td>
<td>1.3µs</td>
<td>1.7µs</td>
</tr>
<tr>
<td>Get 16B Key</td>
<td>1.3µs</td>
<td>1.3µs</td>
<td>1.7µs</td>
</tr>
<tr>
<td>Get 32B Key</td>
<td>1.4µs</td>
<td>1.4µs</td>
<td>1.8µs</td>
</tr>
<tr>
<td>Get 64B Key</td>
<td>1.7µs</td>
<td>1.7µs</td>
<td>2.1µs</td>
</tr>
<tr>
<td>Set 8B Key</td>
<td>1.2µs</td>
<td>1.2µs</td>
<td>1.2µs</td>
</tr>
<tr>
<td>Set 16B Key</td>
<td>1.2µs</td>
<td>1.2µs</td>
<td>1.2µs</td>
</tr>
<tr>
<td>Set 32B Key</td>
<td>1.3µs</td>
<td>1.3µs</td>
<td>1.3µs</td>
</tr>
<tr>
<td>Set 64B Key</td>
<td>1.5µs</td>
<td>1.5µs</td>
<td>1.5µs</td>
</tr>
</tbody>
</table>

Table 5.7: Internal latencies of the Caribou_{MC} hash table and value access.

<table>
<thead>
<tr>
<th>Key / Value Size</th>
<th>64B</th>
<th>128B</th>
<th>512B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get 8B Key</td>
<td>1.1µs</td>
<td>1.2µs</td>
<td>1.6µs</td>
</tr>
<tr>
<td>Get 16B Key</td>
<td>1.1µs</td>
<td>1.2µs</td>
<td>1.6µs</td>
</tr>
<tr>
<td>Set 8B Key</td>
<td>1.0µs</td>
<td>1.0µs</td>
<td>1.1µs</td>
</tr>
<tr>
<td>Set 16B Key</td>
<td>1.0µs</td>
<td>1.0µs</td>
<td>1.1µs</td>
</tr>
<tr>
<td>Insert 8B Key</td>
<td>1.1µs</td>
<td>1.1µs</td>
<td>1.2µs</td>
</tr>
<tr>
<td>Insert 16B Key</td>
<td>1.1µs</td>
<td>1.1µs</td>
<td>1.2µs</td>
</tr>
</tbody>
</table>

Table 5.8: Internal latencies of the Caribou_{CH} hash table and value access.

**Result summary**

Even with added features, the internal latency of Caribou_{CH} and Caribou_{MC} are very similar to those of KVS_{Light}, used as baseline.
5.3.2 Response Times

**Purpose of the experiments**

When *Caribou*$_{CH}$ and *Caribou*$_{MC}$ are used as the storage layer for a distributed application it is important that they provide data with low latency and that these latencies do not increase prohibitively with value size, operation type or load on the system. We demonstrate this behavior through experiments. Overall, our goal is not to study the exact breakdown of response times (module-specific costs will be discussed in later sections), but rather, to show that behavior is predictable for the client.

**Basic operations in *Caribou*$_{CH}$**

As a first baseline we establish the response times for various operations performed on *Caribou*$_{CH}$. We measure them by running a single client issuing requests and reporting the median of the time taken per operation (Figures 5.6 and 5.7). The measurements taken at the client contain not only the hardware cost of the operations but the overhead of traversing the software stack and NIC in the client. Since we use the standard linux TCP stack, this cost is actually comparable to the time spent in the server nodes.

We approximate the minimum overhead introduced by the operating system and NIC in the client machine using a ping-flood measurement. The measurements showed a ping time between the client machines and *Caribou*$_{CH}$ to be 17-27µs depending on the value size in the range we present in Figure 5.6. This overhead is marked as a black line. Of course, TCP/IP and packet decoding adds additional overheads that are harder to measure, but included in the graphs, together with the FPGA-side TCP stack overhead and the actual processing. This can be seen shaded in red in the figure.

Figures 5.6 and 5.7 show the response time of non-replicated and replicated writes to the key-value store. The response time increases with value size, mostly due to higher transmission times and more time spent on decode in the clients. Overall, response time stays under 50µs, and is predictable, even with replication. Replication introduces an overhead between 6µs and 10µs as seen from the client. The difference between replicating to two (*R3*) or four other nodes (*R5*) is not visible from the client because proposal sending is pipelined and reaching majority is quickly achieved.
5.3. Get-Put Workloads

![Graph](image)

Figure 5.6: Insert response time increases with the value size (key=8B). Replication adds constant overhead (Caribou\textsubscript{CH}).

Read operations (Figure 5.7) follow a trend similar to writes, resulting in a predictable behavior. Since in our implementation reads can be directed at any node, but the followers will not actively synchronize with the leader on a read (i.e. eventual consistency), the read times are the same regardless of the type of node.

Decompression, when enabled, adds as many cycles latency as the uncompressed value size (e.g. 1.6\textmu s for 256 B) but when compared to the total response time, these overheads are acceptable. This effect is discussed in more detail in Section 5.8.

![Graph](image)

Figure 5.7: Reads and Updates become more costly with large values (key=8B). Replication overhead is the same as for Inserts (Caribou\textsubscript{CH}).
Latency trend with increasing load  To show the trend of response times as a function of throughput, we ran the system with a single threaded client machine to measure the response time and the rest of the machines as load generators. The issued operations are 50% reads and 50% updates. Figure 5.8 shows that both median and 99th percentile response times are stable up until reaching maximum throughput. If we put more load on the system the input link to the FPGA would get congested leading to dropped packets. This in turn would trigger TCP retransmissions, lowering the achievable throughput. Compared to the latency reported in Figure 5.5, we can see the same effect as before, latencies increased by a few microseconds mainly due to the choice of network protocol. In addition, the \( KVS_{\text{Light}} \) numbers stay flat to the end because congested UDP links drop packets whereas TCP will retransmit lost packets.

![Figure 5.8: Median and 99th percentile response times are stable even with high load (workload: 50% read 50% update, 64 B value)](image)

Comparing different versions  If we compare the write response times of \( \text{Caribou}_C\text{H} \) to the other two variants (Figure 5.9) we can see the following trends: First of all, a local write is the cheapest in \( KVS_{\text{Light}} \), because it uses UDP instead of TCP. Local writes in \( \text{Caribou}_C\text{H} \) and \( \text{Caribou}_\text{MC} \) are virtually the same, even though they have different hash table designs and the clients are different (\( \text{Caribou}_\text{MC} \)'s response time is measured with \text{libmemcached}, whereas \( \text{Caribou}_C\text{H} \)'s is done using the Go client). This illustrates that a large part of the measured time is actually spent in the client machines in the OS level.

There is a larger difference when it comes to replicated writes, as shown in Figure 5.10,
Caribou_{CH} is significantly faster than Caribou_{MC}. The reason is likely not the difference in software clients, but instead it is the fact that Caribou_{CH}'s TCP stack incorporates several optimizations explained in [99] (not part of this dissertation). As opposed to the TCP stack variant we used in the Caribou_{MC} work [98], the optimized one has a cut-through design for TX buffers in that data is sent out over the network and written to DRAM buffers in parallel. This way the access latency is hidden, memory bandwidth is saved, but in case a retransmission is needed, the data is available. The net result of this change is that the TCP stack’s latency is halved.

![Figure 5.9](image1)

Figure 5.9: Write response times in the different Caribou variants are very similar, the main differentiating factor being UDP vs. TCP networking.

![Figure 5.10](image2)

Figure 5.10: Replicated write response times differ somewhat between Caribou variants due to Caribou_{CH}'s TCP optimizations.
Chapter 5. Evaluation

### Result summary

Response times in CaribouCH and CaribouMC are consistently low (<50µs) and do not increase much under load either. The majority of time is spent in the clients, not in the FPGA servers. Finally, the TCP optimizations we included in CaribouCH improve replicated writes by several µs.

#### 5.3.3 Mixed workloads

**Throughput vs. Value size**

<table>
<thead>
<tr>
<th>Purpose of the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>We measure the throughput of CaribouCH with different operation mixes (writes and reads) and different value sizes, with the goal of showing that its performance is close to line-rate. This is important, because as a storage device CaribouCH should be able to saturate the output network link regardless the types of operations arriving.</td>
</tr>
</tbody>
</table>

To test the throughput of CaribouCH under a more realistic mix of operations [6] we start experiments with a pre-populated store and issue reads and updates from the clients. Figure 5.11 shows that performance stays close to the maximum read-only throughput achievable over 10Gbps Ethernet and TCP/IP (marked with a black line on the graph). For small response packets and no replication, the system does not achieve the theoretical maximum but it is close to 90% of it. As the value size increases beyond 32 B, the theoretical maximum can be achieved and the system becomes network bound. With more updates, the average size of a response packet decreases, which in turn increases the achievable throughput over the network link.

Performance with replication for this 10% write workload is as high as 70% of the non-replicated one (but somewhat lower than the expected maximum of 75%). The reason for the lower throughput is the overhead introduced by both the replication logic and the TCP/IP stack has to issue and receive additional messages for each replication round[6]. From the perspective of replicas, the overhead of replicated writes is very close to that of

[6]The amount of data to be sent out from the leader for a write request increases dramatically from 16 B to 16 + CountFollowers * (16 + 16 + SizeKey + SizeValue) Bytes.
local writes. Overall, the performance of Caribou\textsubscript{CH} is close to be network bound for reads and updates.

![Figure 5.11: Caribou\textsubscript{CH}: Throughput is mostly limited by the network for read-heavy operation mixes.](image)

As opposed to Caribou\textsubscript{MC} and KVS\textsubscript{Light}, in Caribou\textsubscript{CH} inserts and deletes do more work than reads and updates because they also modify the bitmaps and pointers in memory. Even though their response times are low, they cannot be executed at the same rate as read requests. The maximum rate at which inserts can be performed in our current implementation is capped at 2M Ops./s by memory access latency. Regardless of value size, the “written” bitmap updating operations incur a full memory access latency. While this can be optimized in principle, it requires additional data structures for temporal caching of bitmaps (similar to the write-back cache in the hash table). We defer such an optimization to future work as the current maximum rate is fast enough for most workloads. Delete operations access two bitmaps but these are pipelined so only a single memory access latency is incurred. Therefore the delete performance is equal to insert performance on average.

In order to compare the behavior of Caribou\textsubscript{CH} with the baseline behavior of KVS\textsubscript{Light} over the complete range of get-set mixes, we ran an experiment with 128 B values on both systems. Figure 5.12 shows how the throughput of both systems changes (the one for the baseline is the same as in Figure 5.4). What the numbers show is that for the read-mostly workloads their behavior is virtually the same (modulo the protocol overhead differences), and even though Caribou\textsubscript{CH} is slower for write-heavy workloads, its performance does not degrade catastrophically.
Chapter 5. Evaluation

Figure 5.12: Comparison of Caribou\textsubscript{CH} and KVS\textsubscript{Light} for 128 B values and different operation mixes shows similar behavior in read-mostly mixes. Caribou\textsubscript{CH} has a higher overhead in the write-heavy part of the graph.

Result summary

We showed that the behavior of Caribou\textsubscript{CH} is predictable with different workloads and value sizes, and that its throughput is close to that of KVS\textsubscript{Light} in most cases. Full line-rate performance is achieved with the exception of small (<64B) packets and very write-heavy workloads. In contrast to KVS\textsubscript{Light} where inserts, deletes and updates were equivalent operations, in Caribou\textsubscript{CH} inserts and deletes are slower than updates. They are capped at 2M operations/s. This should, however, be enough for read-mostly workloads.

Replication Throughput

Purpose of the experiments

We demonstrate that not only regular operation, but replication traffic as well can reach high throughput levels. We furthermore show that by adding specialized point-to-point connections, throughput goes up due to the additional bandwidth available for the node.

For these experiments related to replication we choose Caribou\textsubscript{MC} as the default system because its replication module is identical to the one in Caribou\textsubscript{CH} but its insert rate is not bottlenecked the same way as Caribou\textsubscript{CH}. It is important to point out however, that
5.3. Get-Put Workloads

since the replication module is decoupled from the particular hash table implementation, the numbers apply for both systems. Furthermore, in Caribou\textsubscript{MC} replication traffic can be sent also over dedicated links in parallel to the TCP traffic on the main interface. This enables us to explore how the additional bandwidth available for the nodes translates to increased throughput.

If we look at replication-only performance of Caribou\textsubscript{MC} nodes in Figure 5.13, we can see that with three nodes using direct links vs. one single TCP link has a clear benefit. The increase in performance is not proportional to the extra bandwidth (30Gbps total vs. 10Gbps in the TCP case) because the internal pipeline of Caribou\textsubscript{MC} has been designed for 10Gbps line-rate. So even though data can be sent out in parallel on the two dedicated links, the logic that prepares the values will have to do so iteratively, at a rate less than 30Gbps. Nevertheless, this experiment already shows the benefit of having a backbone interconnect.

For reference, we have included in the same graph the numbers for two software systems. Even when the software solutions log to a ram-disk, their performance is almost two orders of magnitude less than that of the hardware. This resembles the comparison between memcached and KVS\textsubscript{Light} from Figure 5.5, in that the software systems behave fundamentally different from the hardware system\textsuperscript{7}.

![Figure 5.13: Consensus rounds (writes operations) per second as a function of request size measured on the leader.](image)

When comparing Caribou\textsubscript{MC} with Caribou\textsubscript{CH} we expect to see the same behavior because the replication logic is slower than the hash table in both systems (for most workloads).

\textsuperscript{7}There are software solutions that reach the throughput of the hardware nodes, but these rely on specialized networking not on TCP/IP. We will contrast the different solutions in Section 5.10.
Figure 5.14 shows throughput on the leader with 2 or 3 followers as a function of value size. We see similar trends in behavior and overlapping throughput for larger value sizes. For smaller value sizes (< 64 B) Caribou's performance is lower than CaribouMC, but this is expected. The reason is that the two handle data sent to the network that bypasses the key-value store logic (proposals to followers) slightly differently. In CaribouMC there is a small re-ordering module in place that will batch requests per socket ID, so that the overhead of the TCP protocol is reduced. Since this re-ordering could artificially increase latencies for larger values, it is mostly in use for small packets and as the figure illustrates it is effective in that range, and when network bandwidth is utilized to a higher rate (R4).

![Figure 5.14: Replication-only behavior follows a similar trend in CaribouMC and CaribouCH.]

We can also explore the benefit of the dedicated point-to-point links as part of a mixed workload experiment. We compare in Figure 5.15 the throughput of CaribouMC using dedicated links and that of CaribouCH using a single TCP link. We run a read-heavy workload with 10% replicated writes. We can see that CaribouMC, when replicating using the dedicated links performs as well or slightly better than CaribouCH in a non-replicated case.

When using replication in CaribouCH, on the other hand, the main network interface is shared between client-facing and cross-node traffic, reducing the “goodput” significantly. This is a result, on the one hand of the transmission overhead of the replicated messages, and on the other hand, because answers are small, the TCP connections are also used sub-optimally (if we only transmit 16B, the TCP headers are actually a significant overhead).
5.3. Get-Put Workloads

One solution in the future would be to reorder messages inside the TCP stack’s transmit path and try to batch together small packets going to the same socket.

![Graph showing throughput vs value size]

Figure 5.15: Mixed workload throughput measured on leader with replication over different network interfaces (\textit{Caribou}_{MC}: TCP+Dedicated, \textit{Caribou}_{CH}: TCP)

**Result summary**

Throughput of replication reaches almost 2.5M operations/s, for three nodes without batching, which makes Caribou a very competitive system even when compared to state-of-the-art software. Adding more nodes, or larger messages have predictable effect on the throughput.

From the experiments it is clear that there are two improvements that could be done in the future: First, using a small amount of response batching in the TCP stack to avoid sending very small messages in quick succession. Second, if the design is to have dedicated links, some parts of the internal logic would need to be scaled up to take full advantage of the additional bandwidth.

**No Impact of Skew**

**Purpose of the experiments**

We showed that the system can handle mixed workloads well, and in this experiment we investigate what effect skew has on throughput. Since the hardware key-value store is a single pipeline and we implemented memory hazard avoidance in way that does not stall the pipeline, we expect to see no effect at all.
Chapter 5. Evaluation

The hardware pipeline handles locking and concurrency differently from software. In software multi-threaded access to data structures can lead to contention, especially in the presence of hot-spots. Conversely, in hardware there is a single logical execution thread that is not impacted by access skew. To illustrate this point, we ran a 90% read 10% update experiment on Caribou$_{CH}$ where an increasingly small percentage of a 128 thousand large key-set is accessed. Figure 5.16 shows that even when this hot set is as small as 128 keys, the performance stays virtually identical to the uniform access pattern case (the throughput variation is <1% over the points in the figure).

![Figure 5.16: Skew has no negative impact on throughput due to the single processing pipeline (10% updates, 64 B values)](image)

**Result summary**

Due to the logical “single thread” of execution, the hardware pipeline is not impacted by access skew. This makes estimating workload performance easier because we can abstract from skew.
5.3.4 Scalability

Throughput

Purpose of the experiments

We built Caribou with scale-out operation in mind. In this subsection we explore how its throughput changes with increasing cluster size and replicated write percentage. Having implemented a leader-based replication protocol, the expectation is that with high percentage of writes throughput will be bottlenecked at the leader.

In our current setup we are constrained by the number of load generators and FPGAs, but given our measurements on the 3 and 5 node setup, it is possible to extrapolate for larger deployments of Caribou. Clients can read from all nodes and as a result, the projected read-only throughput increases linearly with the number of nodes. Writes have to go through the master copy (leader in consensus terms), which can be a limiting factor for write-heavy workloads. For the write-only workloads with 64 B values, the leader can sustain 2.2M requests/s for a group size of 3 and 1.1M requests/s for a group size of 5. Currently our system implements eventual consistency, but in our case, unless there is a failure, nodes are at most microseconds apart. If strong consistency is required by the application logic, either all reads on followers will incur at least one additional round-trip to the leader, or a transaction-manager [64] should be used. This would be external to Caribou, and would keep a periodically refreshed view on the latest replication sequence number of each node, and how these sequence numbers map to application-level transactions. Then read operations can be load-balanced on many nodes depending on what application-level transactions they need to see.

Figures 5.17 and 5.18 project the aggregated throughput of CaribouCH nodes for larger deployments than what we could measure in our lab for two different value sizes (64 and 128 B). For high replicated write percentages the behavior of the system is not bottlenecked only by network bandwidth but also by the ops/s limitation of the TCP module in conjunction with the replication module. The most important reason for the decline in performance for more writes is the bottleneck on the leader’s output bandwidth and is typical for systems replicating from a single master copy [109]. In larger deployments it could be beneficial to partition the data among multiple smaller groups of CaribouCH nodes, instead of treating all nodes as belonging to one logical unit. This way both high performance and fault tolerance could be achieved, even in the presence of write-intensive
workloads.

![Figure 5.17: Projected scalability of Caribou with replication to all nodes and different workloads (64 B values). Aggregate throughput is considered, in contrast to the leader-only approach of the previous sections.](image1)

![Figure 5.18: Projected scalability of Caribou with replication to all nodes and different workloads (128 B values). Aggregate throughput is considered, in contrast to the leader-only approach of the previous sections.](image2)
5.3. Get-Put Workloads

Result summary

Caribou’s measured and projected behavior is in line with what one would expect from a leader-based replication scheme. For low write percentages, the system should scale to at least 7 nodes. For higher write percentages it will be bottlenecked on the leader’s network bandwidth and sharding, or a similar technique, will be needed to create smaller replication groups.

Latency with Increasing Load

Purpose of the experiments

Assuming a read-mostly workload, we want to explore what clients could expect from a scale-out deployment of Caribou. We measure throughput and latency with increasing clients and increasing cluster size, in an effort to show that behavior is predictable even under high aggregate load.

In the following we evaluate how predictable the latencies of Caribou nodes are in larger setups. We use 3, 4 and 5 Caribou nodes and generate load using 11 client machines. The 12th machine is used to measure latencies. In order to increase the load each client can generate, we enabled asynchronous request sending on the client side, issuing up to 10 get requests back to back. They are not batched, however, each request has an individual header and the FPGA treats them as separate commands.

Writes are sent to the leader at a fixed rate of 300 kOps/s and in addition to this we increase the read load on the cluster. We use the dataset from Section 5.4.3 with an average value size of 128 B. This setup is useful to evaluate how the system would scale out for reads. We chose the write rate such that it represents 1% of the workload at high saturation. This ensures that the leader is not the bottleneck and ensures that the changes in replication latency are only driven by the increasing read request load on the nodes.

Figure 5.19 shows the response time as a function of aggregate throughput in the cluster. The graph contains measurements taken with 3, 4 and 5 Caribou nodes. The median response time of gets stays flat and the three sudden increases in 99th and 99.9th percentile correspond to the points where the smaller clusters get saturated.
Chapter 5. Evaluation

Figure 5.19: Response time of operations with 3, 4, and 5 nodes as a function of aggregate throughput. Read response times are on top and replicated write response times are on bottom. \( \textit{Caribou}_{CH} \), constant write rate of 300kOps/s, value size = 128B).

The behavior of \( \textit{Caribou}_{CH} \) is predictable for replicated writes as well. Figure 5.19 shows that, even though they are not as flat as gets, even under load the follower nodes do not introduce much jitter or delay in the consensus traffic. It is interesting to note that in case the followers are serving read requests, their acknowledgments will, as expected, be sent with a delay, but this is around 5\( \mu \text{s} \) even for high load. Overall, this experiment shows expected behavior because in our setup the replication traffic shares part of the pipeline with the read traffic and there is no priority given to either. In the future, depending on the use-case, it could be beneficial to assign priorities to different types of operations.
Overall, this experiment shows that the same aggregate throughput can be achieved in different setups (3, 4 or 5 nodes), but thanks to the flat response time behavior of the cluster, get response times are almost indistinguishable. Replicated sets are more dependent on the load level of participants, but they increase predictably. Overall, this shows that provisioning with hardware nodes is straightforward and given a workload, the expected throughput of a larger cluster can be determined easily.

## 5.4 Scan and Processing

### 5.4.1 Pipeline Limits

Caribou offers scan-based access to the data in addition to random access. In the following experiments we explore at what rate the data can be filtered, and to what extent does the selectivity of a filter influence the rate at which clients receive their data.

**Purpose of the experiments**

Scans and conditional reads are very similar in Caribou, the main difference being the amount of data passed through the selection operators per request. For a conditional read operation the throughput is limited not by the selection performance but by the network. The request has to encode both the key to lookup and the parameters for the selection computation. The size of the configuration parameters for a conditional matcher is 6 bytes, for the regular expression matcher it depends on the number of characters/states supported (but, e.g., for a 16 character / 8 state setup 48 B are needed to represent the configuration). Not surprisingly, for small values the system could become bottlenecked on the input bandwidth. The output becomes a bottleneck if selectivity is high. For low selectivity the “non-match” responses are smaller than the requests (16 B) making the input the main limiting element. Figure 5.20 depicts the maximum selection rate inside the processing pipeline, and the best achievable throughput both on the input and on the output side for increasing value sizes. Network is the main limiting factor in this scenario, with processing elements being clearly over-provisioned. The lines for the string matcher...
units depict the worst-case throughput and might actually increase given early matches in the strings (for more detail, see Section 5.6.3).

Figure 5.20: When combining single-value reads and selection, network is the limiting factor. Internal bandwidth is clearly over-provisioned

The selection operators are best utilized for predicate evaluation when the client initiates a scan query. This way a large number of values can be streamed through the processing units with minimal overhead. The result of a scan query only contains the matching values, other values are discarded from the result set. As a consequence, for low selectivity queries the outgoing network link is less likely to become the bottleneck. Of course, if many values fulfill the predicate, the network link will become the bottleneck. Figure 5.21 depicts the rate at which the data area can be scanned given the selectivity and different value sizes (we used a fully synthetic dataset where we could control the selectivity of a scan precisely). The trend in the graph is expected because, regardless of the choice of selection types, the processing pipeline of the FPGA is bound by the performance of the regular expression matcher (it can be deployed using various parallelism levels, in this experiment it is 16). The fixed bandwidth translates to lower tuple/s rate with larger values. For low selectivity this limits the throughput, while for high selectivity the network and the client’s capacity to ingest data dominates.
5.4. Scan and Processing

![Graph showing scan rate vs. selectivity]

Figure 5.21: Scan performance is limited by regex selection (16 units @ 312MHz) for low selectivity, and by the network and client for high selectivity

**Bitmap overhead:** The overhead for scanning the bitmaps which determine what memory lines to retrieve amounts to 400\(\mu s\) for each GB of value storage. This should not impact performance unless the table space is very sparsely populated. To counter these cases, in the future these bitmaps could be augmented with pointers to skip over non-allocated areas.

**Result summary**

We show that filtering is best combined with scanning (highest bandwidth requirement), but it can also be added to regular read operations (which lead to a lower overall bandwidth requirement).

The investigation reveals a slight mismatch between the behavior of different filtering units, but also highlights the flexibility of the hardware, in that parallelism can be used to increase the throughput of a module.

### 5.4.2 Matching Compute to the Storage

**Purpose of the experiments**

In this subsection we explore how the processing pipeline could be scaled to different target bandwidths. If this can be achieved, it makes Caribou applicable in a wide range of use-cases.
Chapter 5. Evaluation

When designing Caribou, we used the on-chip DRAM both as a possible storage medium, and as a placeholder for emerging memory technologies. These are expected to strike a balance between DRAM and NAND flash [119] in every aspect. While today’s latencies in flash storage are relatively low (10\(\mu\)s range), their IOPS, even for the high end devices, are less than 1 million/second. Assuming that multiple flash drives are attached to the same FPGA this number can be increased, but the true potential of Caribou will be exploitable when storage can be accessed at lower than 4KB page granularity and IOPS in the order of millions. In Table 5.9 we show a few example scenarios based on today’s devices of how the processing pipeline would need to be scaled to match the bandwidth of the storage for scanning operations. The table shows, that once we want to process at a rate higher than 6GB/s, the condition matching unit will become a bottleneck and it needs to be replicated (or clocked up to twice the speed).

Table 5.9: When designing Caribou, we had emerging memory technologies in mind, that offer similar throughput to NAND flash devices but with higher IOPS and smaller page sizes.

<table>
<thead>
<tr>
<th>Target Tput.</th>
<th>Target storage</th>
<th>Regex Cores x Freq.</th>
<th>Comparison Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1GB/s</td>
<td>2x SATA-III (a)</td>
<td>8 x 156Mhz</td>
<td>1</td>
</tr>
<tr>
<td>3.1GB/s</td>
<td>1x NVMe (c)</td>
<td>11 x 312MHz</td>
<td>1</td>
</tr>
<tr>
<td>3.8GB/s</td>
<td>2x SAS (b)</td>
<td>15 x 312MHz</td>
<td>1</td>
</tr>
<tr>
<td>6.2GB/s</td>
<td>2x NVMe (c)</td>
<td>22 x 312MHz</td>
<td>1 (except 2 for &lt;64B values)</td>
</tr>
<tr>
<td>9.3GB/s</td>
<td>3x NVMe (c)</td>
<td>32 x 312MHz</td>
<td>2</td>
</tr>
</tbody>
</table>

Example devices: (a) Intel SSD DC S3710 Series; (b) Toshiba PX04s SAS SSD; (c) Toshiba PX04P NVMe SSD.

Result summary

A compute bandwidth of almost 10 GB/s can be achieved with Caribou relying on the parallelism of the FPGA to scale circuits to multiple parallel cores. Clocking plays a very important role as well. Processing can also be scaled down to slower storage medium, so that the circuit is not over-provisioned.
5.4.3 Combined Filters on Real-world Data

<table>
<thead>
<tr>
<th>Purpose of the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before focusing in more detail on each individual processing module in the next sections, we want to understand whether the limits described before apply in the “real world”. We use the Stock Market dataset and run “queries” with different complexity and selectivity to determine how the behavior of the system changes. The expectation is stable, predictable behavior, that always delivers a good worst-case performance.</td>
</tr>
</tbody>
</table>

Playing with Parallelism and Clock Frequencies Figure 5.22 shows that scanning the data set incurs fixed cost overhead and even though the internal scan rate is the same for any data size, the maximum throughput can not be reached (from the perspective of the client) for very small datasets. In this experiment we use low selectivity filter combination\(^8\) (0.1%) so that the throughput can be measured without being bottlenecked on the network. The graph shows that as soon as the data set size is larger than 100MBs (S=1) the scan rate quickly converges to its maximum.

Figure 5.22 also shows that in specialized hardware solutions there are multiple degrees of freedom when it comes to building a circuit with a specific throughput in mind. The figure shows that starting from the default throughput level of 4.5GB/s achieved by 16 matchers running at 312MHz, it can be either increased by adding more matchers (20) or increasing the clock rate (343MHz). Table 5.10 shows the resource consumption of each of these points, illustrating that, for instance 5GB/s throughput can be achieved with different trade-offs in terms of chip space and clock frequency.

\(^8\)F1 – Conditions: \textit{year} \geq 2010, \textit{day} \leq 24, \textit{month} \geq 2, \textit{month} \leq 7, \text{Regex}='\text{Netflix}'
Figure 5.22: When scanning the data, there is a fixed overhead in going through the bitmaps and encoding the request and decoding the answer in the client. The larger the data sizes, the closer we get to the theoretical maximum of the filtering modules. The selectivity of the filters in this experiment is 0.1%.

Table 5.10: Resource consumption of the different regular expression matcher deployments, and their maximum measured throughput.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Slices</th>
<th>BRAMs</th>
<th>Throughput (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 units @ 312MHz</td>
<td>7610</td>
<td>120</td>
<td>4610</td>
</tr>
<tr>
<td>18 units @ 312MHz</td>
<td>8744</td>
<td>135</td>
<td>5140</td>
</tr>
<tr>
<td>16 units @ 343MHz</td>
<td>7654</td>
<td>120</td>
<td>5050</td>
</tr>
<tr>
<td>20 units @ 312MHz</td>
<td>10038</td>
<td>150</td>
<td>5690</td>
</tr>
</tbody>
</table>

Selectivity and its Effect on Network Transmission  Figure 5.23 shows how the scan rate, as measured on the client degrades with increasing high selectivity\(^9\). The network bandwidth is less than the internal bandwidth of the scan pipeline, so the behavior is expected, but the maximum achieved bandwidth is much less than 10Gbps. The reason for this is that the client parses the response headers on a single thread, and it will receive one per matching tuple. Since the TCP stack that we use has only 64KB buffers, in case the software can not keep up with the data arriving, it will backpressure through the

\(^9\)The filter setup is based on the previous one, increasing first the permissiveness of the dates, then the regular expression
TCP flow control, leading to stalling of the scanning pipeline. Thus, every time the TCP window gets full additional penalties are paid in processing.

Figure 5.23: Effect of increasing selectivity on the scan rate for the stock market dataset (and its scaled-up version that is 10 times larger). A single-threaded client measures the runtime including the parsing of response headers.

**Regular Expression Artifact** When experimenting with real-world data an interesting artifact of the regular expression matcher module can be measured. Internally each matcher works on the data 1 byte at a time, but because the length of the value is known, once a match is found, the regular expression matchers discard their input at 64 bytes per cycle. As a result, in case they are used to match on an empty string, or a common character in the data, their processing rate will be higher than the worst-case figures shown so far. Figure 5.24 shows the scan rate as measured on the client for $S=1$ and a 0% selectivity filter (year==1500) and with different regular expressions. From left to right the regular expression matches less and less values and as a result has to process a larger and larger percentage of the input at 1 byte/cycle speed. As the figure shows, the complexity of the expression plays no role, all four patterns on the right yield the same performance.
Figure 5.24: The maximum scan rate can be somewhat increased in case the regular expression matcher finds a match very early in the string and can discard the rest of the input immediately. Otherwise, throughput is the previously reported maximum regardless of complexity.

**Adding Decompression**  We measured the throughput of the CaribouCH filtering pipeline during scans with increasing data sizes and decompression turned on. The compressed data was based on the stock dataset and therefore the values arriving at the conditional matchers and regular expression matchers were the same with or without decompression. Figure 5.25 shows the scan rate, as measured at the client, with a very low selectivity query (returning a single tuple). The decompression module introduces a negligible overhead, due to the cycles added by copy words (Section 4.4), but overall the performance of the system follows the same trend.
Figure 5.25: Scan performance of Caribou, with and without decompression. Selectivity is 0% and the average value size is 128B.

**Result summary**

We tested Caribou with a real-world dataset in a setup where nominal processing rate is 5 GB/s and demonstrate that more than 90% of this can be achieved when factoring in client overhead, sparse datasets and overheads associated with small values.

We explored the design trade-offs in the vicinity of this performance point and showed how flexibly we can deploy more or less processing cores. Caribou was also tested using decompression, and the system delivers the same scan throughput as without it.
5.5 Focus on Replication

5.5.1 Cost of Consensus

<table>
<thead>
<tr>
<th>Purpose of the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed consensus (that we use to provide replication) is one of the more complex operations performed by Caribou, and we explore in this subsection its cost in more detail in a three-node setup. Our goal is to show that the FPGAs provide low latency and predictable turnaround times for consensus messages.</td>
</tr>
</tbody>
</table>

**Single Client** Systems such as Zookeeper require hundreds of microseconds to perform a consensus round [91, 32] even without writing data to disk. This is a significant overhead that will affect the larger system immediately, and here we explain and quantify the benefits of using hardware for this task. Instead of testing the atomic broadcast module in isolation, we measure it as part of CaribouMC, using memaslap on a single thread sending one million consecutive write requests to the key-value store that need to be replicated. We chose very small request size (16 B key and 16 B value) to ensure that measurements are not influenced by the key-value store and stress mostly the atomic broadcast module. To determine the cost of consensus, we measured latencies on the leader FPGA. It starts when CaribouMC receives the first byte of the replication request, and it stops when the final response has been passed on to the networking module (it excludes the networking overhead to clients and software overhead). The two timestamps are sent back to the client as part of the response.

Figure 5.26 depicts the probability distribution of a consensus round as measured on the leader both when using TCP and direct connections to communicate with followers. Clearly, the application-specific network protocol has advantages over TCP, reducing latencies by a factor of 3, but the latter is more general and needs no extra infrastructure. Figure 5.27 shows that the latency of consensus rounds increase only linearly with the request size, and even for 1KB requests stay below 16µs on TCP. To put the hardware numbers in perspective we include measurements of Libpaxos3 [2] and the Raft implementation used in Etcd [1]. We instrumented the code of both to measure the latency of consensus directly on the leader and deployed them on three nodes in our cluster. Unsurprisingly the software solutions show more than an order of magnitude difference in
average latency, and have significantly higher 99th percentiles even for this experiment where the system handles one request at a time.

![CDF of Latency measured on leader (us)](image1)

**Figure 5.26: Latency of consensus in Caribou vs. Related work**

![Consensus round latency on leader (3 nodes, TCP)](image2)

**Figure 5.27: Consensus round latency on leader (3 nodes, TCP)**

**Strong vs. Eventual Consistency** The leader can be configured at runtime to consider a message replicated either when a majority of nodes acknowledged or all of them. The second variant leads to a system of much stronger consistency, but might reduce availability significantly. We measured the effect of the two strategies on consensus latency, and found that even when the system is under load waiting for an additional node does not increase latencies significantly. This is depicted in Figure 5.28 both for TCP and direct, “2/3” being the first strategy committing when at least two nodes agree and “3/3” the second strategy when all of them have to agree before responding to the client.
Chapter 5. Evaluation

Figure 5.28: Latency of consensus on leader when waiting for majority or all nodes, on TCP and direct connections

**Under Load** Previously we determined the lower bound of the consensus round. In this experiment we show that even with load the system’s response times do not degrade catastrophically. Figure 5.29 shows that the FPGA system fares well under increasing load of replicated requests. As later shown in the experiments, with small payloads (<48 B) the system can reach up to 2.4 million consensus rounds per second over TCP and almost 4 million over direct connections before hitting bottlenecks. In our graph the latencies do not increase to infinity because we increased throughput only until the point where the pipeline of the consensus module and the input buffers for TCP could handle load without filling all buffers, and the clients did not need to retransmit messages. Since we measured latency at the leader, these retransmitted messages would lead to false measurements from the leader’s perspective.

Figure 5.29: Load vs. Latency on leader
5.5. Focus on Replication

Result summary

In this subsection we showed that the cost of consensus in hardware is $10\mu s$ (between three nodes, using TCP/IP) over commodity networking. This is an order of magnitude less than traditional software solutions, and as later will be shown, Caribou_{MC} is competitive even with RDMA-based software solution.

When under heavy load, the latencies increase predictably. Close to the saturation point both medians and 99th percentiles grow less than $2\mu s$ for the dedicated links, and less than $10\mu s$ for TCP-based communication.

5.5.2 Leader Election and Recovery

Purpose of the experiments

While minimizing the cost of leader election and recovery was not the main goal of this work, we want to show that a) both can be carried out on the FPGA, and b) recovery happens reasonably quickly in order to keep the cluster available even after failures.

To exercise leader election and recovery we simulate a node failure of the leader, which results in a leader election round without significant state transfer since the FPGAs do not drift much apart under standard operation. Hence leader election over TCP/IP takes approximately as long as a consensus round ($10\mu s$ in average), not counting the initial timeout of the followers ($50\mu s$) Figure 5.30 depicts the experiment: we generate write-only load from several clients for a three node FPGA cluster communicating over TCP and at the 56 s mark the leader node fails and a new leader is elected. To make client transition possible we modified memaslap and added a timeout of 100 ms before trying another node (the clients retry in the same round robin order in which the FPGAs try to elect leaders). The graph indicates that the dip in performance is due to the 100 ms inactivity of clients, since leader election takes orders of magnitude less.

Synchronization of state between nodes happens for instance when a new node joins the cluster. In Figure 5.30 we shows the previously failed node recovering after 2 minutes and prompting the new leader to synchronize. Since at this point the log has been compacted, the leader will bulk transfer the application state that consists of the hash table and value area, occupying 256MB and 2GB, respectively. During synchronization the leader will not
Chapter 5. Evaluation

handle new write requests to keep the state stable, hence the clients will repeatedly time
out and resume normal operation only once the leader has finished the state transfer. The
results show that, as expected, this step takes between 2-3 seconds, the time necessary to
send the state over 10Gbps network plus clients resuming.

This experiment allows us to make two observations. First, leader election can be per-
formed very quickly in hardware because detecting failed nodes happens in shorter time
frames than in software (i.e., in order of tens of µs). Hence, leader-change decisions can be
taken quickly thanks to low round-trip times among nodes. Second, the cost of performing
bulk transfers shows that in future work it will be important to optimize this operation.
The hardware could benefit from the approach used by related work, such as DARE [91],
where the followers synchronize the newly added node. This leads to smaller performance
penalty incurred by state transfer at the cost of a slightly more complex synchronization
phase.

![Figure 5.30: Leader election triggered while client issue an all-write workload, nodes con-
nected via TCP links](image)

**Figure 5.30:** Leader election triggered while client issue an all-write workload, nodes con-
nected via TCP links

**Result summary**

We showed that leader election happens in a similar time as a consensus round,
thanks to the round-robin optimization we implemented for picking the next leader.
Transferring state to a node that has recovered or that has newly joined will most
likely incur a full copy of the key-value state, but at least this can be performed at
full network bandwidth.
5.6 Focus on Regex

The regular expression matchers developed in Caribou can be used in different use-cases and have been successfully deployed in a hybrid database engine as well [53, 100], in addition to the smart storage use-case. In this section we look at performance and characteristics of this module, and compare it to state-of-the-art software regular expression matching libraries.

Table 5.11: Patterns used for evaluation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Complexity</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁ 'P,O, Box'</td>
<td>low</td>
<td>DB</td>
</tr>
<tr>
<td>P₂ 'Next.*Day.*Shipping'</td>
<td>medium</td>
<td>DB</td>
</tr>
<tr>
<td>P₃ 'a(REQIMG</td>
<td>RVWCFG)b'</td>
<td>medium</td>
</tr>
<tr>
<td>P₄ 'Max-dotdot[ \n][0-9]{3,'}</td>
<td>medium</td>
<td>Snort</td>
</tr>
<tr>
<td>P₅ '(P,O, Box</td>
<td>PB).*{87[0-9]{4}}'</td>
<td>high</td>
</tr>
<tr>
<td>P₆ 'SITE[t,r,n,v,f]+NEWER'</td>
<td>high</td>
<td>Snort</td>
</tr>
</tbody>
</table>

Table 5.11 shows the regular expressions used in this section. We use three patterns that resemble queries in the TPC-W\(^{10}\) benchmark. Additionally, as a reference to related work, three representative patterns from the Snort\(^{11}\) community rule set were also added. As seen in the table, we classify the expressions into three complexity classes, based on how many regular expression features they use. P₁ and P₂ can be expressed in databases using LIKE, but the others require the more general REGEXP\_LIKE\(^{12}\). Only P₁ can be answered with an index lookup.

5.6.1 Compressed NFAs

The compaction of “sequential states” into a single token is one of the main ideas behind our regular expression matcher. In the following experiment we evaluate how much we gain in terms of state-reduction for a set of realistic regular expressions.

---

\(^{10}\)www.tpc.org/tpcw/

\(^{11}\)www.snort.org/downloads/

\(^{12}\)https://docs.oracle.com/cd/B12037_01/server.101/b10759/conditions018.htm
Chapter 5. Evaluation

As a first experiment, we looked at the proposed set of regular expressions and investigated what is the benefit of sequence compaction on the final NFA complexity. Figure 5.31 shows how many NFA states are required to implement each pattern, if one character corresponds to one state. It also shows the equivalent DFA, and then the compressed NFA states, and the number of characters needed in total to evaluate the pattern (the number of tokens is smaller than the characters). Interestingly, the DFAs only suffer from “state explosion” for certain patterns. Namely $P_2$ and $P_5$, both contain an unlimited wild card repetition ‘.*’. Further, we conclude from this experiment that the extraction of sequences is a useful optimization for these kinds of expressions as it reduces the number of states significantly (NFA Compr.).

![Figure 5.31: Number of NFA vs. DFA states for each pattern and the benefit of compressing NFAs through character extraction](image)

**Result summary**

As expected, regular expressions which contain words are well compressible. For a small subset of the expressions the so called state explosion problem of DFAs was also visible, showing the need for using NFAs.
5.6. Focus on RegEx

5.6.2 Performance Comparison to Software

<table>
<thead>
<tr>
<th>Purpose of the experiments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even though our design point is an FPGA, it is important to understand how well a CPU could be used to perform the same filtering tasks as we do in hardware. We will compare the regular expression matcher module to state-of-the-art software to understand how many CPU cores would be needed for an equivalent performance.</td>
</tr>
</tbody>
</table>

As a baseline comparison for string matching, we chose two high performance software regular expression engines: Intel Hyperscan and Google RE2, and a widely adopted commercial database (for legal reasons we will call this system DBx). They run on a 10-core Intel Xeon E5-2680 v2 with 96GBs of main memory. The software regular expression engines are operating on a data structure that contains a list of pointers that point to the strings organized in a heap. DBx uses a row-based format to store data so to minimize the overhead of record parsing or reconstruction, we added the strings to a relation with a single column of type VARCHAR with fixed length. The hardware traverses the strings as part of a scan operation, using the bitmap data structure as a secondary index to retrieve values from memory.

For our evaluation we used natural text in the form of address strings. Each string contains names, street, city, and area code, generated at random and concatenated into a single string. The length of the strings was chosen explicitly for different experiments, and they were either shortened or concatenated together to have a predefined fixed length of 64 B. For each regular expression P1-P6 we inserted a fixed amount of “hits” into the strings uniformly at random (20% unless otherwise stated), but for the sake of the experiment in hardware we dropped all values before sending them on the network. The actual contents of the strings are only relevant to ensure that character frequencies follow a distribution similar to real-life data.

Figure 5.32 illustrates how the increasing complexity of the regular expressions impacts the software solutions (DBx uses LIKE for $P_1$ and $P_2$, and REGEXP_LIKE for the rest). The FPGA can consume input at a constant rate, regardless the regular expression complexity, and stays at the expected throughput based on its parallelism level (16 cores in this case). The software libraries are single threaded and run on a single core, but even with perfect linear scaling for 10 cores they will be slower than the FPGA for all but the simplest pattern. This result is important because it shows that if one wants to push processing into
the storage layer using software there is a trade-off between complexity of the computation and the achievable throughput of the system because the cores also have to take care of networking and data management, not just filtering. In comparison, as long as the logic fits on the FPGA, processing can be added in a complexity-agnostic way to the pipeline.

Figure 5.32: Throughput as function of pattern complexity for 64 B values.

**Result summary**

For the more complex regular expressions even a full CPU socket would have difficulty keeping up with the hardware. Furthermore, the FPGA can perform both data management tasks and processing in a large pipeline, whereas software would need to dedicate some cores to the former as well.

### 5.6.3 Early Discard of Data

**Purpose of the experiments**

So far we reported the throughput of the regular expression matcher for the “worst case”, that is, when there is either no match, or it happens towards the end of the input. If this not the case, the module can actually process data faster than its nominal throughput. In this subsection we quantify this effect.

The throughput presented above applies to the regular expression matcher if it has to actually process most of the bytes in the input. If matches happen always in the beginning
5.6. Focus on Regex

Figure 5.33: The regex module’s throughput is 1 byte/cycle in the worst case (when the hit appears at the end of the value, or not at all). In other cases, it can reach much higher throughput.

If enough of the input, the rest of the input will be discarded faster than 1B/cycle. The FIFOs leading to the matching modules are 64B wide and can be drained at 64B/cycle if a match is already found. The input distributor feeds the modules actually at a lower rate than that because it introduces one cycle delay between values (i.e., 64Bs require two cycles, 128Bs three, and so on). In the current design every value streams through the regular expression matcher and matching will be performed even if it is already known that the value is to be dropped. For large values it would be useful to use this information to immediately drop the value inside the regular expression matcher, making the scan rate for those cases higher than that of the regex.

Figure 5.33 shows how the behavior of the regular expression matchers changes depending on how much of the input string it has to consume before a match is found. The results for latency increasing linearly are not surprising because the string is searched one byte at a time. Throughput however increases dramatically for immediate matches (e.g. if the expression is .*) or if the pattern is at the beginning of the input. In this case the module can discard the remaining input at the rate of 64B/s (the width of the data line coming from memory). Not present in this graph, is the maximum rate at which the input can be fed into the regular expression matcher module from the conditional matchers, which, for
Chapter 5. Evaluation

128B values is 42B/cycle. Throughout the dissertation, when we discuss the throughput of the regular expression matching module, we use the worst case value because this is guaranteed to be achieved regardless of the expression and value contents.

**Result summary**
The throughput of the regular expression matcher increases dramatically when a matching pattern is situated at the beginning of the input string.

5.6.4 Resource Consumption

In Table 5.12 we show how many resources each individual building block needs, and also the space required for different configurations. It is clear that a large part of the resources is consumed by the input and output FIFOs and the “glue” logic holding the individual character matchers and states together. The amount of this logic is fairly stable however across different configurations. More importantly, the resource consumption increases linearly with number of characters and quadratically with states because they form a fully connected graph. While these numbers could be reduced by hard-coding the patterns in that case the accelerator would no longer be suitable for runtime parametrization and lose its appeal for our use-case.

Table 5.12: Resource consumption of basic building blocks (Virtex7, 312MHz).

<table>
<thead>
<tr>
<th>Component</th>
<th>Slice LUTs</th>
<th>Slice Reg.</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character matcher (L)</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Character matcher (H)</td>
<td>27</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Single state</td>
<td>12</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Final state</td>
<td>40</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>In+out FIFOs</td>
<td>120</td>
<td>290</td>
<td>7.5</td>
</tr>
<tr>
<td>Single core: S=4, C=12</td>
<td>500</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>Single core: S=4, C=20</td>
<td>700</td>
<td>1200</td>
<td>0</td>
</tr>
<tr>
<td>Single core: S=8, C=12</td>
<td>610</td>
<td>1160</td>
<td>0</td>
</tr>
<tr>
<td>Single core: S=8, C=20</td>
<td>900</td>
<td>1400</td>
<td>0</td>
</tr>
</tbody>
</table>
5.7 Focus on Conditionals

### Purpose of the experiments

We are interested in showing the effect of implementation overheads on this module, and demonstrate how it can match the target processing rate even for small values. We look at its resource consumption and show that it is comparable to the other modules, and permits adding more pipeline stages in hardware.

### 5.7.1 Throughput

We measured the throughput of the conditional matcher units using synthetic data and a processing pipeline without decompression and regular expression matchers. The condition has been chosen such that the selectivity of the operator is 0%, minimizing the response size.

Figure 5.34 shows that the scan rate increases slightly with larger values because the conditional matcher introduces one cycle of stall in the pipeline for each value. For 64B values this translates to every second cycle being lost, but as the value size increases, this overhead matters less and less. At 1KB values, the circuit is bottlenecked on DRAM access.

In this scenario the bottleneck is likely not the DRAM module itself (that should be able to provide more bandwidth running at 800MHz), but the AXI data movers and switches connected to the DRAM channel. These provide a 156MHz access interface, and they likely can not reach 100% utilization all the time. Compounded with the 6% overhead introduced by the condition matchers, this results in a maximum utilization of 90% of the available bandwidth at 8.8GB/s.

#### Result summary

The conditional matchers can provide up to almost 9GB/s throughput for large values, but for smaller ones, they introduce an overhead. In the future, if the goal is to reach beyond 9GB/s even for small values, the conditional matcher pipeline will have to be duplicated on the chip.
Chapter 5. Evaluation

Figure 5.34: Throughput of Conditional matchers as function of value size and pipeline depth.

5.7.2 Resource Consumption

As Table 5.13 shows, the resource consumption of the comparison-based filtering units is dominated by the registers required for storing the 64B memory lines as they pass through. One aspect of the design that can be optimized in future iterations of the project is that currently the BRAMs used for buffering run-time parameters and the output of the conditional matching pipeline are somewhat over-sized (as can be seen in the BRAM consumption column for the multi-step setups).

Table 5.13: Resource consumption of the comparison unit (Virtex7, 156MHz).

<table>
<thead>
<tr>
<th>Component</th>
<th>Slice LUTs</th>
<th>Slice Reg.</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single comparison</td>
<td>1030</td>
<td>1655</td>
<td>0</td>
</tr>
<tr>
<td>4 Conditionals</td>
<td>4125</td>
<td>6816</td>
<td>12</td>
</tr>
<tr>
<td>6 Conditionals</td>
<td>6226</td>
<td>10126</td>
<td>18</td>
</tr>
<tr>
<td>8 Conditionals</td>
<td>8300</td>
<td>13440</td>
<td>24</td>
</tr>
</tbody>
</table>

Result summary

The conditional matchers have modest logic footprint. While their BRAM usage is also not very high, it could be further optimized.
5.8 Focus on Decompression

### Purpose of the experiments

Decompression is an iterative process and therefore introduces latency in the pipeline. In the following we quantify this latency and highlight what factors influence it. In terms of resource consumption we show that its footprint is fairly modest and this allows us to combine multiple “cores” to achieve higher throughput.

---

#### 5.8.1 Latency Overhead

Latency added by the decompression module increases linearly with the size of the decompressed value because its output is generated one byte at a time. Figure 5.35 shows the exact values for increasing value sizes and two different clocking rates for this module.

This experiment illustrates one drawback that FPGA-based designs face: due to the low clock-rates, iterative logic will introduce latencies comparable to the actual data transmission time. In the case of decompression, however, the trade-off is acceptable because the main goal of storing compressed data is to save storage capacity not to minimize latencies. Nonetheless, the latencies can be reduced by a fixed factor by opting for a higher clock rate – this shows that in FPGAs, often it is desirable to clock the circuit as high as possible.

![Figure 5.35: Decompression latency increases linearly with decompressed size of values.](image-url)
As discussed in the implementation section, compressed data contains two types of “words”: literal and copy. The first type represents a byte transferred as-is from input to output, whereas the second one requires copying a number of bytes from the sliding window to the output. The current implementation introduces one cycle delay when encountering a copy word and in case the input is composed predominantly of copy words this could reduce the effective throughput of the decompression unit. In Figure 5.36 we show how latency and throughput evolve for decompressing a 128B value depending on its compressed structure. In case the compressed values is only literal words (0 on the X-axis) throughput is 1B/cycle. If the input is, however, a series of literal bytes, and then just copy words, the smaller amount each copy writes to the output, the larger the overhead becomes. When designing this module, we targeted input that is not “pathological”, but its design could be adjusted to remove the overhead introduced on copy operations by adding more complex logic to the input processing.

![Decompression latency in case of “pathological” inputs that contain copy operations only.](image)

**Result summary**

The latency introduced by the decompression units is mostly dependent on the length of the output, but with the presence of many “copy words”, additional cycles are introduced. These can lead to a 50% increase in latency if they are copying pathologically short sequences.
5.8.2 Resource Consumption

The resource consumption of a single decompression core is similar to that of a conditional matcher, because they both have to register at least two full memory words. In the case of the decompression, on the input side one is used to iterate through the compressed bytes, while on the output side one is used to act as a shift register to compose the decompressed data. Half of a BRAM is used to hold the window data structure.

Table 5.14: Resource consumption of a decompression core (Virtex7, 312MHz).

<table>
<thead>
<tr>
<th>Component</th>
<th>Slice LUTs</th>
<th>Slice Reg.</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decompression</td>
<td>1223</td>
<td>1123</td>
<td>0.5</td>
</tr>
<tr>
<td>FIFO in/out</td>
<td>64</td>
<td>174</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Result summary

Resource requirements are very modest (<1% of the FPGA resources per decompression core) and this allows us to combine multiple cores for higher aggregate throughput.

5.9 Power Consumption

Purpose of the experiments

First we quantify the power consumption of our prototype to show that Caribou has a modest energy footprint.

Second, we use modeling to show in what cases it will be worth it to replace regular CPUs with specialized hardware, to gain in energy efficiency.

We measured CaribouCH’s power consumption with a wall power meter to be on average 29Ws with moderate load (using a single, directly connected machine as load generator). We know from previous work [14] that in networking-related scenarios the load-dependent power increase is very small (<10%). For a design with decompression, four condition matchers and 16 regular expression matchers running at double the network clock, the Xilinx Power Analyzer reported an estimated 11Ws power consumption of the FPGA chip alone. Starting from the DRAM models presented in the survey by Dayarathna et al. [27]
and Micron’s guidelines\footnote{https://www.micron.com/-/media/documents/technical-note/dram/tn41_01ddr3_power.pdf} we estimate the two SODIMMS on the VC709 board to consume 4W each. This leaves 10Ws for the rest of the board.

Our hypothesis is that, with the exception of the case when a single server machine has hundreds of GBs of DRAM as backing storage, using a specialized micro-server design instead of regular server will lead to meaningful power reduction. To back this idea up with numbers, we will compare Caribou to an “ideal” server machine, targeting a 10Gbps key-value store workload with light in-storage processing.

For the server we assume the following: it is composed of a CPU, motherboard (that contains the NIC as well) and DRAM or Flash for storage. Based on the book by Barroso et al. \cite{9} we estimate the motherboard and NIC to consume 35Ws. We will consider two CPU options, as explained below, that represent two ends of the spectrum for server machines:

- **Low-power server – 2x Intel Atom** – This assumes that the storage nodes run lower power CPUs. Based on the whitepaper implementing memcached on an Atom server \cite{47}, we assume that a dual-socket system is needed with e.g. an Intel Atom C2000 CPU to handle load at 10Gbps for small requests.

- **High-power server – 1x Intel Xeon E5** – From the work of Li et. al \cite{73} we know that when using user-space networking, two cores of a beefy server CPU are enough to handle load from clients at 10Gbps using a protocol similar to UDP. Even though handling TCP traffic is much more expensive\footnote{This is visible for instance in the increased response times when using TCP vs. UDP in memcached in Figure 5.5.}, for the sake of this experiment we will assume that it can be handled at line-rate on a large multi-core with some leftover cycles for light in-storage processing.

Table 5.15 provides a breakdown of the power assumed for different parts in our model, based on following sources \cite{27, 9} and part specifications.

If we assume that a storage node built with an FPGA will have the same storage medium attached to it as the regular servers, we can plot the expected improvement compared to software. We express improvement in terms of reduction of the per node power consumption. As seen in Figure 5.37, using specialized storage node architecture pays off if there is less than 64Ws used by the memory because in this case even when compared to a
5.9. Power Consumption

Table 5.15: Power consumption estimate per component.

<table>
<thead>
<tr>
<th>Component</th>
<th>Power (W)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caribou FPGA</td>
<td>11</td>
<td>Virtex7 VX690T (156/312MHz)</td>
</tr>
<tr>
<td>Caribou Board</td>
<td>10</td>
<td>Includes SFP+</td>
</tr>
<tr>
<td>Caribou Memory</td>
<td>8</td>
<td>2x4GB</td>
</tr>
<tr>
<td>Server Motherboard</td>
<td>35</td>
<td>Includes NIC</td>
</tr>
<tr>
<td>Server CPU (Atom)</td>
<td>40</td>
<td>2x Intel Atom C2750</td>
</tr>
<tr>
<td>Server CPU (Xeon)</td>
<td>130</td>
<td>1x Intel Xeon E5-2697 v2</td>
</tr>
<tr>
<td>Server Memory</td>
<td>4</td>
<td>Per DIMM (8 or 16GB)</td>
</tr>
<tr>
<td>SATA Flash</td>
<td>7</td>
<td>Intel DC S3710 (1.2TB), 600MB/s</td>
</tr>
<tr>
<td>SAS Flash</td>
<td>9.5</td>
<td>Toshiba PX04s (1.6TB), 2GB/s</td>
</tr>
<tr>
<td>NVMe Flash</td>
<td>18</td>
<td>Toshiba PX04P NVMe (400GB), 3.1GB/s</td>
</tr>
<tr>
<td>PCIe Flash</td>
<td>24</td>
<td>Seagate Nytro XP6500 (1.5TB), 4GB/s</td>
</tr>
</tbody>
</table>

Figure 5.37: Estimate of Caribou’s power consumption improvement over regular servers as a function of Watts spent on storage.

A low-power server we can save 2x on power consumption. This would translate to using up to two high-performance flash devices or less than 128GBs of DRAM per node (assuming 16GB DIMMs). With more than 64Ws consumed by the memory, using a specialized server is likely less beneficial from a power efficiency aspect alone. It is important to note, however, that the models we used are somewhat pessimistic towards the FPGA’s benefit
because both the motherboard and CPU power is more a lower bound than a typical value.

<table>
<thead>
<tr>
<th>Result summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>We can conclude that it is very likely that in cases where each storage node contains DRAM in the tens of gigabytes, or a handful of flash drives, using specialized servers instead of regular ones will pay off in terms of energy efficiency. For larger capacity, case-by-case analysis will be necessary.</td>
</tr>
</tbody>
</table>

## 5.10 Comparison to Related Work

### 5.10.1 Key-value Stores

We compare *Caribou* to a set of representative designs in more detail in Table 5.16 (see Section 2.5.1 for a detailed presentation of these systems). The comparison includes functionality, performance, and energy consumption. Our prototype platform consumes 29 Ws, including 2 DDR3 SODIMMs. While this device is not necessarily tailored to the intelligent storage use-case, it provides an upper bound on how much power an FPGA-based solution for this use-case would consume (with more DRAM or Flash this would increase, but these have the same power requirements in all systems).

The other FPGA-based solutions provide similar throughput and latency but lack the functionalities provided by *Caribou*. The throughput of software solutions is in the same range as *Caribou* but they have higher latencies (with the exception of MICA [73, 75]). MICA achieves an order of magnitude higher throughput than *Caribou*$_{CH}$, which makes it a very energy efficient system, demonstrating 378 kRPS/Watt. This number is comparable to our prototype which reaches as high as 380 kRPS/Watt (with replication enabled this is reduced to 267 kRPS/Watt, explained mostly by the consensus messages consuming part of the network bandwidth). It is however important to note that *Caribou* runs general purpose TCP/IP while MICA uses UDP and multiple network cards in the same machine. It also pushes complexity to the clients. In contrast, *Caribou* is able to perform predicate evaluation operations or decompression on the data without reducing performance. This shows the different trade-offs that hardware and software face and why we consider specialized hardware a viable way for managing and computing on large disaggregated storage.
Tellstore [90] is a key-value store designed for mixed get/put and scan workloads. It is part of a larger project, TellDB, a distributed, shared data database. Tellstore implements near-data processing in a flexible way: functions to execute on a scan are compiled at runtime with LLVM. For networking it relies on RDMA over Infiniband. Internally it relies on a set of different data structures (row-major, column-major or log-structured) each of which are well suited to different operation types. It demonstrates a scan rate of 4 to 9 GB/s. Tellstore’s get/put throughput is also high, in the 2-2.5MRPS range. This number is, however, lower than what Caribou can achieve, even though it uses slower networking and runs TCP/IP.

5.10.2 Effect of Complexity on Software

Sitaridi and Ross [101] use GPUs to accelerate substring search in strings, however their work assumes that the data is already in the GPU’s memory and has a very specific layout which suits the GPU’s execution pattern. When running on string data that has not been organized specially for the GPU the performance drops from 60-70 GB/s to around 20 GB/s, comparable to CPU-based execution. Additionally, the more wild cards the expression has, the slower the matching becomes. In contrast, our specialized hardware solution provides complexity-independent performance on unmodified data.

In hardware it is possible to add processing stages to the global pipeline without reducing performance, whereas in software, once the execution becomes compute bound, any added functionality decreases performance. We illustrate this effect with a simple experiment for which we use a modified version of MemC3 [37]. We chose it because it has good performance (almost 4.5MRPS for a 16 thread setup), and is easy to benchmark and extend. We augmented the code performing a GET operation so that it would a) check for a hard coded array of bytes in the value (“Cond. match” using `strstr`), b) check for a string search with wild cards (“Substring match” using GNU C Regexp) and combined these with the same decompression as in hardware (using the zlib library). We ran the server on a single thread and used 8 machines with 8 threads each to generate load. The key is 16 B and the uncompressed values are 512 B. Figure 5.38 shows the relative performance degradation with more features added (100% amounts to 310k read requests/s). Figure 5.39 illustrates how Caribou delivers network-bound, or close to network bound throughput with or without added features (the workload has 8B keys and 512B values with 100% selectivity, and results are normalized to 1.99MRPS). The main purpose of this experiment is not to
focus on the absolute numbers but to illustrate that in software “there is no free lunch”. Unfortunately it also does not exist in hardware with the trade-off being in terms of chip space vs. complexity rather than performance vs. complexity.

Figure 5.38: MemC3 performance with increasing complexity

Figure 5.39: Impact on adding more processing to Caribou and MemC3 has different effects. Throughput is normalized to the platform-specific maximum.
## Table 5.16: Comparison of representative hardware and software key-value store designs

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Properties</th>
<th>Ops/s Per Node</th>
<th>Min. Latency</th>
<th>Power Per Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanaka et al. [103]</td>
<td>FPGA</td>
<td>Single node, UDP</td>
<td>20M (on 20Gbps)</td>
<td>4μs</td>
<td>15W + 100W</td>
</tr>
<tr>
<td>Blott et al. [14, 51]</td>
<td>FPGA</td>
<td>Single node, UDP</td>
<td>13M</td>
<td>4.5μs</td>
<td>26W</td>
</tr>
<tr>
<td>BlueCache [116]</td>
<td>FPGA</td>
<td>Custom dedicated network</td>
<td>4M</td>
<td>&gt;20μs</td>
<td>40W*</td>
</tr>
<tr>
<td>MemC3 [37]</td>
<td>SW</td>
<td>Single node, TCP</td>
<td>4.4M</td>
<td>&gt;20μs</td>
<td>&gt;120W†</td>
</tr>
<tr>
<td>MICA [73]</td>
<td>SW</td>
<td>Single node, UDP</td>
<td>120M (on 4x40Gbps)</td>
<td>&lt;20μs</td>
<td>317W</td>
</tr>
<tr>
<td>FARM [33]</td>
<td>SW</td>
<td>Replicated, Transactions, RDMA</td>
<td>&gt;10M (on 40Gbps)</td>
<td>&lt;10μs</td>
<td>&gt;230W†</td>
</tr>
<tr>
<td>Tellstore [90]</td>
<td>SW</td>
<td>Processing on scans, RDMA</td>
<td>&lt;2.5M (on 40Gbps)</td>
<td>&gt;50μs</td>
<td>&gt;160W†</td>
</tr>
<tr>
<td>Caribou</td>
<td>FPGA</td>
<td>Replicated, Processing on scans, TCP</td>
<td>11M</td>
<td>10-20μs</td>
<td>29W*</td>
</tr>
</tbody>
</table>

Legend: *– includes power drawn by DRAM, not only processing; †– lower bound, the thermal design power (TDP) of the processor used for the original evaluation; ‡ – our measurement
Chapter 6

Conclusion

6.1 Summary

Caribou proves that it is possible to distill all functionality needed for a key-value store into a small footprint specialized device. Even though some aspects were already explored in related work (e.g. streaming processing, hash table designs), Caribou is the first to add features necessary for real-world deployments, such as replication or TCP/IP networking. In doing so it establishes that there is a possibility of building distributed systems using stand-alone specialized hardware nodes. The resulting system can be used as a starting point for exploring both in-storage processing ideas and FPGA-based distributed systems.

Chapter 1 presented the motivation behind the work in this dissertation: data movement bottlenecks in scale-out applications and the observation that most existing smart storage solutions have an unbalanced design. This is mostly due to the fact that, given the compute-bound nature of many filtering operations, it is difficult to avoid over-provisioning the nodes in terms of CPU capacity. Instead, we propose using specialized hardware to implement a system whose network bandwidth is balanced with the data management logic, and processing with the storage bandwidth. Caribou implements complex but flexible functionality in a modest physical footprint, saving on energy consumption as well. It is an exercise in application/hardware co-design and shows that it is possible to design a balanced storage node with the “right amount” of compute.

Chapter 2 describes in detail how data management is implemented in Caribou to provide quick random access to key-value pairs over a simple interface. The main challenge in hardware is to find the data structures that can offer high throughput access to the data.
(both random access and scans) while being able to scale to large capacity. Driven by random access performance, we chose cuckoo hashing, a well studied software hash table that offers amortized constant time for all operations. Scans are handled using bitmap indexes and storage scaling is possible thanks to the slab-based memory allocator. In addition to this general purpose design (Caribou$_{CH}$), we discussed a simpler key-value store design (Caribou$_{MC}$) tailored for caching workloads. Even though the former design comes with significantly more features than the latter, these do not lead to dramatic changes in performance for Caribou.

Chapter 3 presents the replication logic. The main challenge in implementing this functionality was to find a way of encoding the underlying consensus algorithm as a state machine that maps well to the execution model of an FPGA. High throughput for the replication logic is rooted in the choice of the algorithm because it offers pipelining by default. Low latency is mostly a side-effect of using specialized hardware, but to exploit it for such complex operations the data structures that store the state and meta-data of the consensus protocol had to be designed with care. The replication module of Caribou solves at its core a more general problem, namely that of distributed consensus, and, as a result, opens up interesting research questions around building distributed systems using stand-alone FPGAs.

Chapter 4 focused on near data processing, more specifically: one module to transform data (decompress) and two to filter tuples retrieved by software. The latter two showcase different ways of using parallelism in hardware: either to increase throughput by deploying multiple processing “cores” in a data parallel fashion, or as a way to increase complexity by building deeper pipelines. Overall the filtering modules have been designed such that the specific expression they execute is parameterizable at runtime. This adds flexibility and increases the likeliness that a workload that can take advantage of the processing inside Caribou.

Chapter 5, the Evaluation, covers all presented modules and functionality of Caribou in detail and compares its performance to what is achievable today with software key-value stores. The main take-away of this chapter is that Caribou has predictable and stable throughput and latency. This is an important property for a distributed storage solution because it makes it possible to provision the size of the layer more precisely. Caribou achieves network-bound throughput for most workloads, which is not surprising for a specialized hardware device, but at the same time, it offers complex functionality to the clients. Thanks to hardware pipelining, replication rounds can be carried out significantly
faster than in most commodity software systems, and Caribou is even competitive with cutting-edge RDMA-based software, while running on top of commodity TCP/IP sockets. When it comes to the processing modules of Caribou, we showed that it is possible to create high throughput units that are, at the same time, runtime parameterizable and not impacted by expression complexity as long as their tasks fit within the confines of the circuit. This leads to predictable behavior even when processing, an important property for the storage layer.

6.2 Research Outlook

Both the design of datacenters and that of individual server nodes is changing. Heterogeneous hardware is deployed increasingly often and in multiple levels of the architecture in an effort to provide more efficient computation and data movement. Caribou shows that specialized hardware solutions make sense beyond the traditional accelerator use-case: by re-designing the internal architecture of storage nodes, we can push most of the functionality into specialized hardware keeping less or no CPU cores around.

Furthermore, Caribou also shows that even though specialized hardware is often associated with fixed functionality, it is possible to design high-throughput circuits that can change their task at runtime with no overhead. With the wider-spread adoption of FPGAs, it will be important to design flexible circuits and possibly combine them with methods such as partial reconfiguration, to ensure that changing workloads can take full advantage of the benefits of the specialized hardware functionalities.

The work in Caribou opens up paths for exploration in the following directions:

Next-generation micro-servers. There is an emerging field of research around running software on hybrid CPU+FPGA systems (e.g. on the Intel Xeon+FPGA platform [18], or IBM CAPI-enabled servers), but the FPGA is often treated as a limited-scope accelerator instead of an integral part of the application. Especially in the micro-server use-case, it can be beneficial to have a deeper integration between software and hardware.

Even though in this dissertation we built Caribou on stand-alone FPGAs, and demonstrated that it is possible to provide interfaces similar to software using a dedicated hardware device, one of the main design challenges with FPGAs is that all code has to be present on the chip, even parts that are rarely used and these compete with other, more
commonly executed, parts for chip space. Especially when it comes to management or monitoring tasks, a large number of simple functionalities have to be supported. This is the reason why adding small CPU cores to these kind of micro servers could be a more efficient way of implementing this functionality than creating circuits on the FPGA.

Caribou running on a device such as the Xilinx MPSoC or an UltraScale+ device could be a good starting point for exploring new directions in operating system and application design for tightly coupled hybrid architectures. Furthermore, such hybrid devices could also be a good starting point for exploring how software could manage the reconfigurable fabric, using methods such as partial reconfiguration (swapping out parts of the hardware circuit at run-time).

**Exploration of near-data processing ideas.** If we look at main-memory key-value stores such as memcached or Redis, their fundamental design is very similar and what sets them apart is mostly the protocol they implement and the types of processing they provide on top of the data. Additional features, such as replication, are orthogonal to the core key-value store logic. Caribou provides the basic functionality required for building a key-value store in hardware and therefore it acts as a platform for implementing near-data processing solutions on top. Even though we already provide several types of in-storage processing, these could easily be replaced or modified without requiring changes to the hash table design or the memory allocator.

Data science has many examples of data-intensive workloads that could benefit from near-storage compute. In particular in the domain of machine learning, tasks are distributed and require, on the one hand, storing and accessing large amounts of data (larger datasets than most traditional applications) and, on the other hand, fairly complex processing, especially for learning tasks. There is more and more work on using specialized hardware for learning tasks [62], as well as inference [87]. They provide the highest benefit when working on non-standard types, such as reduced precision or quantized data. Given this growing collection of machine-learning related accelerators it is possible to explore several ideas on how the higher level algorithms could benefit from offloading to the storage, and Caribou can be used as a starting point for this. It can store the data needed for the machine learning algorithms, replacing the current distributed data store, and offer offloading at the same time.
**Consensus in hardware.** The ideas presented about the consensus module can be useful beyond the key-value store use-case. Since the consensus block is not dependent on the hash table, it could be used independently to implement a form of coordination between hardware devices or to offload this operation from hosts into the network. By extending, for instance, smart switches [80] or middleboxes with this module, it would be possible to ensure that configuration updates are being applied consistently among a large number of devices. One can also add the consensus module to programmable NICs, such as Microsoft Catapult [19], and this would help with offloading consensus (or at least the common cases) to the network for applications running in software that require low latency consensus among nodes.

While the idea of accelerating parts of consensus protocols in switches and other “smart” network devices is already being explored in related work [25, 71], these approach the acceleration task in a bottom-up approach, implementing in hardware only a subset of the operations. The consensus module from Caribou provides a potential for exploring a top-down approach where the hardware can run the whole protocol, if needed, on its own.
List of Tables

2.1 Properties of the different modules inside the Cuckoo hash table. 33
2.2 The overhead for engine data structures is small. 36

3.1 Structure of the ZAB request header in Caribou 50
3.2 Cost of ZAB operations and the theoretical maximum consensus rounds per second when using a single 10GbE link per node. 56

4.1 Maximum throughput of the regular expression matchers as a function of parallelism. 78
4.2 Measured and projected throughput of the conditional matchers for different value sizes. 82

5.1 We benchmark different types of operations with different versions of the system. 95
5.2 Overheads important when comparing throughput of node-local operations on the different variants. 96
5.3 Implementation details important when comparing the response time of node-local operations of the different variants 96
5.4 Implementation details important when comparing the throughput and response time of replicated operations of the different variants. 96
5.5 Resource consumption of each module in Caribou. 100
5.6 Latencies of different “infrastructure blocks” in KVS$_{Light}$ and Caribou$_{MC}$. All values, except the last row, are independent of our implementation. 104
5.7 Internal latencies of the Caribou$_{MC}$ hash table and value access. 105
List of Tables

5.8 Internal latencies of the CaribouCH hash table and value access. 105
5.9 When designing Caribou, we had emerging memory technologies in mind, that offer similar throughput to NAND flash devices but with higher IOPS and smaller page sizes. 124
5.10 Resource consumption of the different regular expression matcher deployments, and their maximum measured throughput. 126
5.11 Patterns used for evaluation 135
5.12 Resource consumption of basic building blocks (Virtex7, 312MHz). 140
5.13 Resource consumption of the comparison unit (Virtex7, 156MHz). 142
5.14 Resource consumption of a decompression core (Virtex7, 312MHz). 145
5.15 Power consumption estimate per component 147
5.16 Comparison of representative hardware and software key-value store designs 151
List of Figures

1.1 There is an increasing gap between data growth and CPU performance increase. This leads to changes in the way we design computers and in how we use distributed systems for data processing. ......................... 2

1.2 Many data processing applications are designed with decoupled resources in mind. By disaggregating resources inside a rack, more flexibility is available in provisioning different workloads. .................................. 3

1.3 Caribou represents a balanced design point in which in-storage compute matches the network and storage medium bandwidth. .................. 4

2.1 Inside a Field Programmable Gate Array (FPGAs) is a “sea” of gates and memories that can be composed and configured to implement any circuit. (Image credit: B. Ronak) .................................................. 10

2.2 Key-value stores are used in many different settings in the datacenter, especially in web-facing applications. ................................. 12

2.3 Internally, keys can be organized either as a flat structure or as a hierarchy. 13

2.4 A Caribou node is composed of five main building blocks that can be modified independent of each other. ............................... 15

2.5 The two Caribou variants presented in this dissertation. ............... 17

2.6 Many key-value stores store keys and values separately for better flexibility in management. ....................................................... 18

2.7 All pipelined hash tables in hardware are composed of four main steps. The choice of, e.g., hash function or collision resolution strategy results in a specific type of hash table. ................................. 18
List of Figures

2.8 Multiple hash units can work in a data-parallel way to reach the desired bandwidth or to reduce latencies when processing long keys. 21
2.9 Building blocks of CaribouMC 22
2.10 ASCII commands are parsed in a pipeline. 23
2.11 Buckets are preallocated in hardware with a fixed number of slots. Collisions beyond these slots are not resolved. 24
2.12 Each bucket has four slots and each slot spans at most 8 memory lines. 25
2.13 High level view of a CaribouCH node. 28
2.14 In a Cuckoo hash table every item has two locations, determined by two different hash functions. If both of them are taken, one of the two occupying keys is moved to its second location. This continues until every key has a free slot. 29
2.15 Cuckoo hash table implemented as a pipeline in hardware. 31
2.16 Slab based memory allocator. Metadata is stored in form of bitmaps and pointers are stored in in-memory queues. 34
2.17 CaribouCH implements three types of processing in the storage. 37
2.18 If implemented without pipelining, key-value stores on FPGAs will have sub-optimal throughput for small value sizes. 39
3.1 Zookeeper – a distributed fault-tolerant meta-data store. 47
3.2 Zookeeper’s Atomic Broadcast (ZAB) Protocol requires a leader to propose changes to followers. 48
3.3 Read and write requests take different paths: reads go directly to the hash table, while writes are first replicated. 49
3.4 Caribou implements two different network protocols: TCP/IP and a specialized point-to-point one. 51
3.5 From the perspective of the key-value store and replication logic, specialized backbone connections are not different from TCP sockets. 54
3.6 The specialized point-to-point connections act as a backbone interconnect between nodes. These links only carry replication-related traffic. 54
List of Figures

3.7 Overview of hardware-based ZAB implementation. .................................................. 55
3.8 ZAB as a state machine. ............................................................................................. 55

4.1 Example of offloading filtering for an SQL-style query to the storage layer. ........... 66
4.2 Two examples of mapping relational tuples to a key-value stores. ................. 67
4.3 Example NFA matching a regular expression. Transitions between states are taken when encountering specific characters in the input. ......................... 69
4.4 Example NFA matching a regular expression. Characters that are used to match a linear sequence of states have been highlighted as “Tokens”. ........ 69
4.5 Creating a general NFA where any state could transition to any other one provides maximum flexibility, but the circuit size is prohibitive for large state counts. ................................................................. 70
4.6 High level view of a single Regular Expression matching unit. ....................... 71
4.7 Output from character matchers is represented as a vector and AND-ed with the sensitivity list of each state. If the result is not zero, the state is activated. ................................................................. 72
4.8 Mapping the bitmaps used for configuring each part of the regular expression matcher to a single configuration vector. .................................................... 72
4.9 Inside a character matcher. ....................................................................................... 73
4.10 Regex throughput is increased with a data-parallel architecture. Note the clock domain crossing through the input and output FIFOs. ................. 77
4.11 Conditional matchers compare a given offset in the value to an input constant. By chaining multiple of them, more complex conditions can be expressed. ................................................................. 79
4.12 Conditional matchers are pipelined and contain several parameterizable elements (offset, function to compare with and constant to compare to). . . 80
4.13 The condition matcher receives values that can span multiple cycles and can optionally be already tagged with a “last” signal corresponding to the last word of a value (see values with * in the figure). Otherwise, the module creates the last signals based on the length stored within the value. In this example, only Value2 does not match the condition. Ready signals are omitted for simplicity. .................................................... 81
List of Figures

4.14 Decompression units are built with a state machine controlling the input and decoding commands, and a BRAM for storing the sliding window data. 84

4.15 Multiple parallel decompression units are used to reach the desired throughput level. 86

5.1 Overview of the Xilinx VC709 evaluation board. We used the boards stand-alone, powered through a separate power supply not PCIe. (Photo credit: Xilinx) 92

5.2 Our evaluation boards were installed in a hollowed out desktop machine for deployment in the cluster room (here only 4 FPGAs are pictured). The boards are powered through the power supply of the case. 93

5.3 Abstract view of Caribou nodes. In reality all communication happens over a 10Gbps switch. 94

5.4 Throughput of $KVS_{Light}$ for different value sizes as function of read-write mix (dashed lines = 10GbE maximum) 102

5.5 Median response time of $KVS_{Light}$ with increasing load for 90% read workload (light lines = 99th percentiles). 103

5.6 Insert response time increases with the value size (key=8B). Replication adds constant overhead ($Caribou_{CH}$). 107

5.7 Reads and Updates become more costly with large values (key=8B). Replication overhead is the same as for Inserts ($Caribou_{CH}$). 107

5.8 Median and 99th percentile response times are stable even with high load (workload: 50% read 50% update, 64 B value) 108

5.9 Write response times in the different Caribou variants are very similar, the main differentiating factor being UDP vs. TCP networking. 109

5.10 Replicated write response times differ somewhat between Caribou variants due to $Caribou_{CH}$'s TCP optimizations. 109

5.11 $Caribou_{CH}$: Throughput is mostly limited by the network for read-heavy operation mixes. 111

5.12 Comparison of $Caribou_{CH}$ and $KVS_{Light}$ for 128 B values and different operation mixes shows similar behavior in read-mostly mixes. $Caribou_{CH}$ has a higher overhead in the write-heayt part of the graph. 112
5.13 Consensus rounds (writes operations) per second as a function of request size measured on the leader. .................................................. 113
5.14 Replication-only behavior follows a similar trend in CaribouMC and CaribouCH. 114
5.15 Mixed workload throughput measured on leader with replication over different network interfaces (CaribouMC: TCP+Dedicated, CaribouCH: TCP) 115
5.16 Skew has no negative impact on throughput due to the single processing pipeline (10% updates, 64B values) ............................................. 116
5.17 Projected scalability of Caribou with replication to all nodes and different workloads (64B values). Aggregate throughput is considered, in contrast to the leader-only approach of the previous sections. ...................... 118
5.18 Projected scalability of Caribou with replication to all nodes and different workloads (128B values). Aggregate throughput is considered, in contrast to the leader-only approach of the previous sections. ...................... 118
5.19 Response time of operations with 3, 4, and 5 nodes as a function of aggregate throughput. Read response times are on top and replicated write response times are on bottom. (CaribouCH, constant write rate of 300kOps/s, value size = 128B). ................................................................. 120
5.20 When combining single-value reads and selection, network is the limiting factor. Internal bandwidth is clearly over-provisioned ...................... 122
5.21 Scan performance is limited by regex selection (16 units @ 312MHz) for low selectivity, and by the network and client for high selectivity ............ 123
5.22 When scanning the data, there is a fixed overhead in going through the bitmaps and encoding the request and decoding the answer in the client. The larger the data sizes, the closer we get to the theoretical maximum of the filtering modules. The selectivity of the filters in this experiment is 0.1%). 126
5.23 Effect of increasing selectivity on the scan rate for the stock market dataset (and its scaled-up version that is 10 times larger). A single-threaded client measures the runtime including the parsing of response headers. .............. 127
5.24 The maximum scan rate can be somewhat increased in case the regular expression matcher finds a match very early in the string and can discard the rest of the input immediately. Otherwise, throughput is the previously reported maximum regardless of complexity. ........................................ 128
List of Figures

5.25 Scan performance of Caribou, with and without decompression. Selectivity is 0% and the average value size is 128B. ........................................... 129
5.26 Latency of consensus in Caribou vs. Related work .................................... 131
5.27 Consensus round latency on leader (3 nodes, TCP) ....................................... 131
5.28 Latency of consensus on leader when waiting for majority or all nodes, on TCP and direct connections ......................................................... 132
5.29 Load vs. Latency on leader ............................................................................. 132
5.30 Leader election triggered while client issue an all-write workload, nodes connected via TCP links ................................................................. 134
5.31 Number of NFA vs. DFA states for each pattern and the benefit of compressing NFAs through character extraction .............................................. 136
5.32 Throughput as function of pattern complexity for 64 B values. ..................... 138
5.33 The regex module’s throughput is 1 byte/cycle in the worst case (when the hit appears at the end of the value, or not at all). In other cases, it can reach much higher throughput. ............................................. 139
5.34 Throughput of Conditional matchers as function of value size and pipeline depth. ......................................................................................... 142
5.35 Decompression latency increases linearly with decompressed size of values. 143
5.36 Decompression latency in case of “pathological” inputs that contain copy operations only. ............................................................................. 144
5.37 Estimate of Caribou’s power consumption improvement over regular servers as a function of Watts spent on storage. .......................... 147
5.38 MemC3 performance with increasing complexity ........................................... 150
5.39 Impact on adding more processing to Caribou and MemC3 has different effects. Throughput is normalized to the platform-specific maximum. . . 150
Bibliography


[75] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In NSDI’14.


[83] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In USENIX ATC’13.


[99] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency TCP/IP stack for data center applications. In *FPL’16*.


[115] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. Beyond the wall: Near-data processing for databases. In DAMoN’15. ACM.


[118] Y. Yang, W. Jiang, and V. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In ANCS’08.

Appendices
Binary Protocol Format

For the cuckoo version of Caribou we have opted for a binary protocol that encodes operation types and key and value lengths explicitly. The benefit of such a format is that it is, on the one hand, more compact than an ASCII representation, and, on the other hand, easier to parse both in software and hardware. Furthermore, it is a header format that can be merged with the replication-specific headers, so all request use a single format. This is in contrast to the protocol of CaribouMC where replicated writes required additional headers on top of the Memcached protocol. Below we present the general structure of a request:

Byte 0-7: FFFFrererPPPPkkQQ
Byte 8-15: rrrrrrrrrrrrrrrrr
Byte 16-23: KKKKKKKKKKKKKKKK
Byte 24-31: LLLVVVVVVVVVVVV
...
Byte ..-..: VVVVVVVVVVVVVVVV

Legend:
F [2B] = 0xFFFF Magic number at the beginning of a request
r = Reserved for the replication logic
P [2B] = payload (key + value) size in 64bit words
k [1B] = length of key in 64 bit words (can be 01 or 02).
Q [1B] = node-local command code
K [64B/128B] = key
L [2B] = length of value (including these two bytes) in bytes
V [variable] = value (if no value is needed for the operation, stop at K)

Every packet starts with a magic number (0xFFFF) and then the length of the request
Binary Protocol Format

followed by an opcode. The payload length field is computed as the length of the key and the value, ignoring the first 16B of the request which is fixed size and always present. Values have to encode in the first 2 bytes their length so that they can be correctly interpreted when accessed through a scan operation as well.

The benefit of this protocol is that the first 16Bs (first two words sent over 10Gbps Ethernet) encode enough information that the end of the request can be determined by counting down bytes and no parsing is necessary, in contrast to the ASCII protocol. This allows, for instance, the replication logic to treat the payloads as BLOBs and changes to the hash table logic will not break functionality in other parts of the pipeline.

Replicated Writes From the client’s perspective the only important operation is ”replicated set” that will replicate the given key and value to all nodes. These operations are formatted as follows:

Byte 0-7: FFFFxxCCPPPP0000
Byte 8-15: EEEEEEEEEEEEEEEE
Byte 16-23: KKKKKKKKKKKKKKKK
Byte 24-31: LLLLVVVVVVVVVVV
...
Byte ..-..: VVVVVVVVVVVVVVVV

Legend:

x [1B] = reserved to encode node id
C [1B] = opcode of the operation
P [2B] = payload (key + value) size in 64bit words. E.g. 4=4*64bit
E [8B] = reserved to encode epoch, zxid
K [64B] = key
L [2B] = length of value (including these two bytes) in bytes
V [variable] = value (if no value is needed for the operation, stop at K)