Caribou: Intelligent Distributed Storage

Zsolt István David Sidler Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

The ever increasing amount of data being handled in data centers causes an intrinsic inefficiency: moving data around is expensive in terms of bandwidth, latency, and power consumption, especially given the low computational complexity of many database operations.

In this paper we explore near-data processing in database engines, i.e., the option of offloading part of the computation directly to the storage nodes. We implement our ideas in Caribou, an intelligent distributed storage layer incorporating many of the lessons learned while building systems with specialized hardware. Caribou provides access to DRAM/NVRAM storage over the network through a simple key-value store interface, with each storage node providing high-bandwidth near-data processing at line rate and fault tolerance through replication. The result is a highly efficient, distributed, intelligent data storage that can be used to both boost performance and reduce power consumption and real estate usage in the data center thanks to the microserver architecture adopted.

复制

1. INTRODUCTION

The exponential increase of data to be stored and processed in datacenters is a challenge for current systems. An answer to this challenge comes in the form of distributed in-memory database systems [27, 33, 36, 40] that scale out compute and storage capacity beyond a single machine. One common architectural element in such systems is the separation between storage and processing nodes [36, 37]. The storage nodes expose a single large shared pool of network-addressable storage which can be accessed from any processing node. Since data in the storage layer is shared across all processing nodes, flexibility is increased and load balancing at the processing layer is simplified [5, 36]. Additionally, the two layers can be independently provisioned. While there are many benefits to such an architecture, the data movement it requires is an obvious source of inefficiencies.

数据量增长
↓
存储节点/处理节点分开，但共享一个数据存储池
↓
池中数据到各个存储节点/处理节点均存在数据迁移开销，导致性能下降

Data movement can be reduced by pushing operators down to the storage layer. This is already done in shared-disk architectures that rely on traditional servers to provide storage supporting query offloading [62]. Today, however, processing offload can be achieved with much less hardware and resources, for instance, by using SSDs embedded with computational capabilities [10, 14, 23, 24, 64]. 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 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 [6]. For instance, Intel Optane SSDs[1] can be used as a persistent extension of RAM, blurring the boundaries between main-memory and persistent storage.

We use field programmable gate arrays (FPGAs) to create *Caribou*[2], a first prototype of our ideas for near-data processing for database engines. *Caribou* builds on top of earlier projects [19, 21, 50, 64] and consolidates their ideas into a single system. It shows that it is possible to hide the idiosyncrasies of specialized hardware behind regular TCP/IP sockets and a clean, general purpose, key-value interface. Replication is built into the system for fault tolerance. Thanks to the internal dataflow parallelism, *Caribou* nodes deliver rich functionality at a high bandwidth and low latency. We identify a set of features and properties a storage node should have to be suitable for use with relational databases and explain below how *Caribou* fulfills these. To our knowledge, *Caribou* is the first system to integrate all these aspects into a single device:

合并

性质

Low latency and high throughput: We extend the state of the art in hardware-based key-value stores [8, 53, 67], generalizing to use-cases beyond caching. This is done by implementing a cuckoo hash table with a slab-based memory allocator to improve the handling of collisions and various value sizes.

Lookups and scans on the same data: To minimize the amount of data transfered over the network, it is important

[1] https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html

[2] Project website at http://systems.ethz.ch/fpga

Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/07.

to execute scans without having to send multiple requests to the storage. We have augmented the key-value store with bitmap indexes for efficient scan operations.

Near-data computation: We implement selection both for structured and unstructured data in an effort to reduce data movement even further in a wide range of workloads. Our design ensures that in-storage processing has no negative impact on the data retrieval rate as the selection operators are parameterizable at runtime (require no reprogramming for different queries).

检索

Fault tolerance through transparent replication: We demonstrated in earlier work [21] that FPGAs can be used to build replication into a system at high throughput and without slowing down common operations. In this work we integrate this module into a complex processing pipeline.

Simple interface over traditional network: Many hardware solutions today expose either no proper API (the interface is intertwined with software) or require dedicated secondary networks. We acknowledge the importance of easy integration with existing infrastructures, and provide traditional TCP/IP connectivity using our network stack [49].

交织 专用辅助网络

2. BACKGROUND

There are multiple possible internal architectures for near-data processing, ranging from using low-power ARM cores all the way to fully custom hardware, with various steps in between, combining for instance ARMs and FPGAs. In this work we prototype our ideas on FPGAs. These 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. What makes them interesting as prototyping tools is that, in comparison to traditional processors, they allow fine-grained dataflow parallelism due to the fact that all "code" executes in parallel in the device [54]. This allows for exploring implementations that break traditional software tradeoffs of complexity vs. performance.

FPGA的优势

Internally, FPGAs 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. Their energy footprint is an order of magnitude lower than that of server-grade CPUs.

A majority of the FPGA platforms available for development today are intended for networking or accelerator use-cases and have only modest amounts of DRAM (1-8 GBs). The limitation is not fundamental. There are examples of FPGAs with access to tens of GBs of DRAM (e.g. Maxeler MPX2000 or Convey HC-2).

Many of the ideas presented in this paper could be applied to future architectures and the presence of a CPU core would also simplify management tasks. The Xilinx MPSoC line[3], for instance, promises a combination of ARM cores and programmable fabric attached to fast DRAM memory.

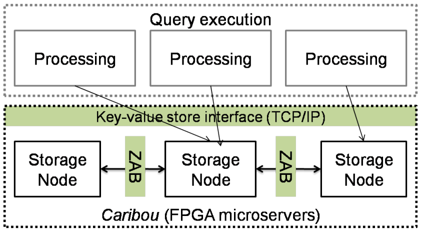


Figure 1: *Caribou* offers replicated intelligent storage for shared-data databases. Replication uses Zookeeper's Atomic Broadcast and is transparent to the layers above.

3. SYSTEM OVERVIEW

3.1 Interface

Caribou implements a distributed key-value store with built-in replication and offload capabilities. Figure 1 shows that *Caribou* exposes a key-value store interface over 10Gbps TCP/IP sockets [48] to the processing nodes (we will refer to these as clients for the remaining of the paper) and runs Zookeeper's Atomic Broadcast protocol [21, 26] between nodes. Replication happens over the same network as the client-serving traffic.

We target row-oriented databases which have a shared-data model. Figure 2 illustrates how relations can be stored in the key-value data structure: the primary key column(s) represent the key for a row and the whole tuple is stored as the value. The system can handle variable sized keys and values, and at its core is agnostic to the schema of the relation. Selection and more advanced processing requires more structural information but, for the purpose of management, tuples are treated as BLOBs in memory. *Caribou* exposes the following operations to clients:

不知道的 概要

Insert – Stores a value in memory, under a key specified in the request. It can be performed both in a replicated or a node-local way. The response encodes the success/failure state of the operation.

Read – Retrieves a value based on a key. If the key is not present in the system it will return failure.

Conditional Read – Retrieves a value corresponding to the key in the request and evaluates a predicate on it. The parameters for the predicate evaluation are provided with the request. If the predicate matches, the value is returned. Otherwise a failure header (16 B) is returned.

谓词

Delete – Removes a key's entry from the hash table and frees the memory holding the value.

Update – Given a key and a value, updates value stored in the system. This operation does not allocate memory if the new value size is the same[4] as the existing one. Updating a non-existent key will fail.

Scan Query – Scans internally over all[5] values in the storage and evaluates a predicate on each. Only the values that fulfill the predicate are sent to the client.

[3] https://www.xilinx.com/support/documentation/product-briefs/zynq-ultrascale-plus-product-brief.pdf

[4] To be more precise, as long as the new value and the old value are the same multiple of 64 Bs

[5] The scan operation retrieves all values stored in the same tablespace. In our current implementation there is only a global tablespace, but as later explained the system can be extended to support multiple tablespaces. This means that scans on one relation will not be influenced by the size of other relations.

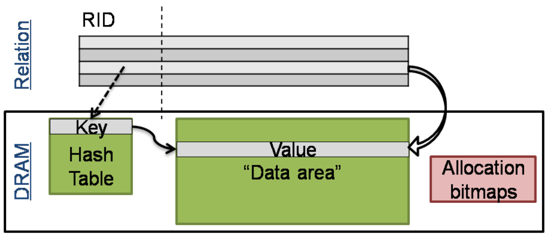


Figure 2: Mapping tables to storage in *Caribou*.

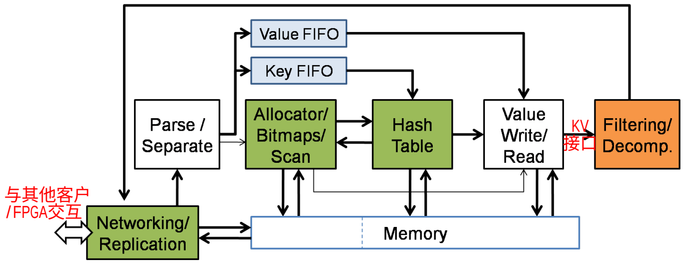


Figure 3: Modules in *Caribou* and their connections.

Keys can be variable size, with a maximum size of 16 B. Values can be of arbitrary length, but we have designed and optimized the system with smaller (32-512 B) values in mind. For larger values, the management tasks become less challenging and can be performed by software as well (as the transmission overhead dominates the cost).

We consider the above set of operations enough to implement a transactional processing layer on top. In the future, depending on the needs of applications, it could be beneficial to augment the interface with compare-and-set (CAS) operations to implement versioning. We showed in our earlier work how CAS operations can be added to a hardware hash-table [21].

3.2 Architectural Elements

Figure 3 depicts the high level architecture of a single *Caribou* node. The logic is centered around a hash table that 1) can handle reads and writes in constant time, so that the storage has predictable performance, and 2) can store variable sized or non-ordered keys, so that *Caribou* can be used in different use-cases. We implement a Cuckoo hash table because it provides constant time lookups. Insertions and deletions might need variable time to be carried out but our implementation can finalize these operations in the background while still providing constant time answers to clients. In order to make sure that memory space is used efficiently, the location of hash table entries and values needs to be completely decoupled. Memory for the values is allocated by a separate module. The same module is responsible for carrying out scans as well. *Caribou* implements a slab-based allocation strategy (similar to the one found in memcached).

Near-memory processing is carried out by the module at the end of the pipeline. It receives values retrieved either as a result of a lookup or a scan operation. This module supports selection on both structured and less structured data. Thanks to hardware parallelism, selection can be combined with other transformations, e.g., decompression. As explained in Section 5, these units exploit parallelism using methods that apply to other data processing operators as well. Functionality could be extended in the future with, e.g., group-by aggregation [64], with statistics [22], skyline queries [63] or complex pattern detection [65].

The networking component of *Caribou* is taken from earlier work on TCP/IP stacks in hardware (more details can be found in [48, 49]). An important feature of the networking stack is that it has low latency (< 10μs RTTs), can saturate 10Gbps even with small packets, and can support up to thousands of connections at the same time. We use the same network interface and the same networking stack to communicate both with clients and with other FPGAs.

We used Zookeeper's atomic broadcast [21, 26] to replicate data from the node that acts as master copy to the replica nodes. Writes are directed to the master copy and they are successful if a majority of nodes perform them. Reads can be directed to any node, but an external commit manager or coordinator [36, 43] should be used by the processing nodes to make sure that they are not reading from nodes with stale data (e.g., after recovering from a network partition). The implementation in hardware comes from previous work [21] and we chose this algorithm because it allows for pipelining of proposal sending and taking commit decisions, fitting well with hardware execution. Furthermore, low latency TCP/IP networking and reliable response times of other FPGA nodes help to maintain high performance while using ordinary sockets instead of special purpose networking or dedicated links.

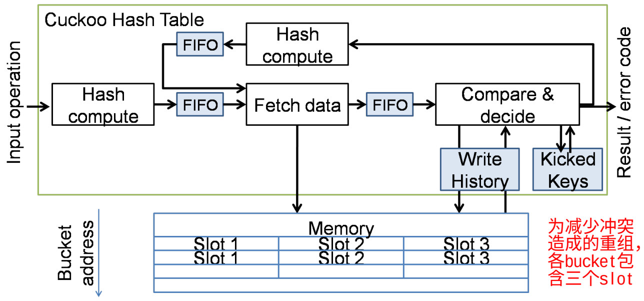
4. KEY-VALUE STORE

There is a rich body of work on implementing key-value stores based on hash tables inside an FPGA. Most of it is focused on caching use-cases [7, 8, 19, 53, 67]. This means that read operations are given more importance than writes and hash collisions are usually not resolved if this would lead to variable response time (hence, the data structures are typically lossy). For *Caribou* we implemented from scratch a cuckoo hash table with slab-based memory allocation to be able to handle update workloads and make more efficient use of memory even with changing value sizes.

4.1 Cuckoo Hash Table

In a cuckoo hash table every item has two possible locations, determined by two different hash functions. When a key is searched, it has to be in either one of its two locations, otherwise it is not in the table. If upon an insert both locations are occupied, a randomly chosen key in one of these locations is "kicked out". This key is then reinserted in the background (done using the connection on top in Figure 4). Even if this leads to an additional reorganization, it has no impact on the throughput of the system since regular lookup/insert requests are processed in parallel to the reinsert. The implementation of the hash table is depicted in Figure 4, and shows how the different operations are pipelined. As a result multiple key lookups can be carried out in parallel.

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) 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.



为减少冲突造成的重组，各bucket包含三个slot

Figure 4: Pipelined cuckoo hash table

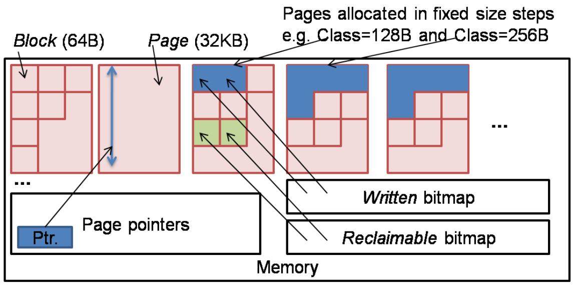


Figure 5: Data structures used for memory allocation

Incoming keys are hashed and the hash value is used to request the corresponding hash table bucket from the memory. We implemented two hash functions for this system: a simple shift-and-XOR function well suited for mostly consecutive binary keys, and a multiplicative hash function similar to the one used in [18] that works well on ASCII strings. The two functions we chose illustrate well the difference in on-chip area required for a low and high compute complexity function (we quantify area requirements in Section 6.3). Alternatively, there are also hash functions that require very little compute (hence minimal logic footprint) but rely on pre-generated tables (leading to more BRAM resource usage) such as the one described by Kara et al. [29].

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 *decide* module implements a small write-through cache that stores recent writes with their address, much like the write buffers present in CPUs. Then, each key is compared either to the data from memory, or if available, its most recent corresponding write stored in this cache. There is also a secondary data structure that holds keys which have been kicked out and are "in flight". This is important to ensure that the same key is not inserted twice or appears to be deleted but then reappears later.

4.2 Choice of Memory Allocator

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 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 [3]. 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.

4.3 Management of Pages and Blocks

调整使适应

We tailored the allocation logic to predefined classes of value sizes (similarly to memcached and other related work [31, 53]). 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 5). This is because our DRAM bus width is 64 B. 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 5 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 5.

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.

4.4 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 tablespace 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 tablespace 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 take longer to scan but, as shown in Section 6.4, the cost of bitmap traversal is much less than that of processing the actual values. For high selectivity performance is always bottlenecked on network bandwidth.

4.5 Data Structure Sizes in Memory

Table 1 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 collocated the hash table and the data area (value storage) on 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.

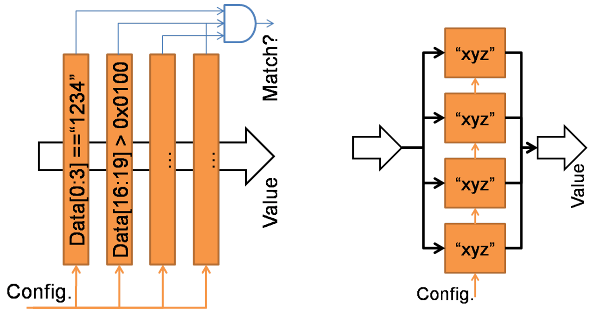
5. PROCESSING: SELECTION OFFLOAD

In *Caribou*, storage and management capabilities are extended to support selection predicates on data before sending it to the processing nodes. One important requirement for selection is that the circuit has to be able to adapt quickly to each different query. We have borrowed ideas from related work [28, 55] and our previous work [20, 50] to implement the operations inside the FPGA in such a way that they require no reprogramming. Instead, client requests parametrize the underlying circuits at runtime.

The benefit of specialized hardware when it comes to processing near the data is that it can execute operations in parallel without degrading performance. Both low and high compute complexity operations can take advantage of this property, though in different ways. Operators that are computationally cheap can be combined in deep pipelines to provide more sophisticated predicates (Figure 6a). Computationally costly operations can be replicated in a data-parallel fashion to increase overall throughput (Figure 6b).

Table 1: The overhead for engine data structures is small.

Type	MBs	% of total
Space for data	**3840**	**93%**
“Reclaimable” bitmap	**8**	**0.2%**
“Written bitmap” (one tablesp.)	**8**	**0.2%**
“Written bitmap” (seven tablespaces)	32	0.8%
Pointer lists (1+4 classes)	**25**	**0.6%**
Hash table (1Mio. @ 75%)	21.3	0.5%
Hash table (12Mio. @ 75%)	**256**	**6.2%**
Total overhead for engine	**297**	**7.2%**



(a) Evaluation pipeline (b) Data-parallel unit

Figure 6: Exploiting parallelism: for adding complexity in a pipeline, or for increasing the throughput of compute-intensive operations

In the following we describe the implementation of three operators: First, simple selection predicates such as those present in most SQL queries, expressed as comparisons. While these are cheap to execute in software, when multiple of them are combined they can become compute bound. Second, we investigate a compute-intensive selection predicate involving substring matching to illustrate how hardware can speed up operations that are compute-bound in software [20]. Third, as another example of a computationally costly operator, we look at decompression of values before executing one of the previously described selection predicates.

5.1 Comparison-based Selection

The simple selection predicate 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. The unit is pipelined, so multiple conditions at different offsets can be combined. In the current prototype, we allow for up to 8 conditions on the same value, and these are combined into a single boolean value, by ANDing them together. Only values matching all conditions will be sent out over the network. Figure 6a shows how the individual condition evaluators are assembled into a pipeline, all of them working in parallel on different parts of a value, or different values.

The pipeline is parametrized at runtime with a *Conditional Read* or *Scan Query* request. This takes one clock cycle and does not degrade performance, even if every lookup uses predicates. These requests encode 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. To give an example a predicate definition corresponding to `where col2=123 and col3<4 and col4='a'` would result in three triplets sent with the *Conditional Read* request, with the offsets of each column depending on the schema of the table.

5.2 Substring-based Selection

This selection predicate matches substrings in the value, separated by a wildcard (e.g., `WHERE name LIKE '%J.%Doe%'`). This unit is based on earlier work [50] with which it shares its internal design, but it has a modified interface. Just as the regular expression matchers presented in [20, 50], it is composed of a series of comparators with small register memories into which the bytes of the strings are loaded in parallel. The circuit is provisioned with 20 characters and the position of the wildcard can be chosen with an extra parameter. This selection predicate is useful if the data stored in memory is unstructured or semi-structured.

Instead of pipelining, in the case of this module, there are multiple parallel units that match on one input value each. This is because the string has to be searched for one byte at a time in the input, which translates to 156MB/s performance per matcher – when 32 of them are deployed in parallel on the other hand, the aggregate processing throughput reaches close to 5GB/s, i.e., four times the 10Gbps network bandwidth (1.25 GB/s). This is important for low selectivity scans because, as opposed to high selectivity ones that are bottlenecked by the network, these achieve speedup due to the additional internal bandwidth.

Figure 6b depicts the way different values are sent to independent string matchers using a round-robin order, where they are buffered in small on-chip memories while the string-matching is performed. The output is gathered using the same round-robin order. This layout trades chip space for higher performance and is a standard technique in hardware design to achieve the desired performance of inherently iterative computational units.

5.3 Decompression

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 the storage hierarchy and network 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 [68] 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. To achieve higher rates, multiple of these units are deployed on the chip in a data-parallel architecture (6b). They add latency linearly with the length of the decompressed data (e.g., 256 clock cycles at 156 MHz for a 256 B value) but do not impact throughput. We deploy 8 units to reach 10Gbps network bandwidth. While reaching higher output rates is possible, on our current platform we are bound by clocking and space constraints. These are however not fundamental and would not apply on larger FPGAs.

6. EVALUATION

6.1 Platform and Experimental Setup

Our prototype platform for *Caribou* is composed of five Xilinx VC709 Evaluation boards[6] with Virtex-7 VX690T FPGAs and 8 GBs of DDR3 memory. They are connected to a conventional 10Gbps switch. 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 (*R3*) or four (*R5*) other nodes. We use up to twelve load generating machines, 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*.

For load generation we used clients written in Go, based on the memcached client library[7]. Each machine is able to generate more than a million requests/s when simulating 64 clients, which allows us to measure 10Gbps performance even for small values on a single instance of *Caribou*. Unless otherwise stated, we read and write to a single node (master copy), with replicated requests then being transmitted transparently to the replica nodes. The key size is always 8 B (simulating 64 bit row IDs) and values contain 2 B of extra metadata encoding their length. By default we deploy *Caribou* with 8 comparison-based selection modules and store uncompressed data. Response time is measured at the client, so it contains the network round-trip time. Throughput numbers are reported as averages of 10 repetitions although we have seen no significant standard deviation in our experiments. Whenever we show "10Gbps limit" on graphs, we compute it by adding 54 B of TCP/IP headers and 12 B of interframe gap to the average size of the response packets. This number is slightly optimistic as it does not account for any retransmissions, but we find it a good enough approximation of the achievable line-rate.

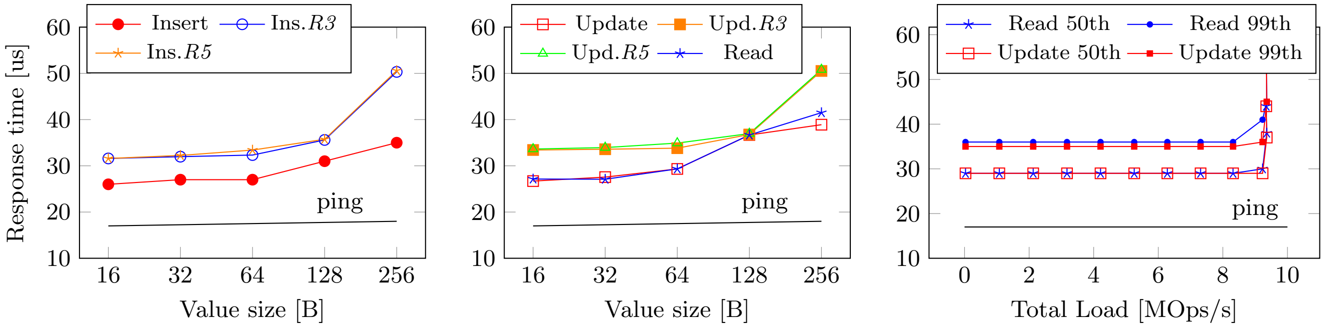
6.2 Micro-benchmarks

As a first baseline we establish the response times for various operations performed on *Caribou*. We measure them by running a single client issuing requests and reporting the median of the time taken per operation (Figures 7a and 7b). 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. We approximate this cost with the ping time between the client machines and *Caribou*: 17-18μs depending on packet size (marked as a line on response time graphs). Decompression, when enabled, adds as many cycles latency as the uncompressed value size (e.g. 1.6μs for 256 Bs). When compared to the total response time, these overheads are acceptable.

Overall *Caribou* has low latency with no large differences between operation types, as long as they are local to the master node. Replication introduces a 6μs overhead but this is less significant the larger the values get. 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.

[6] `https://www.xilinx.com/products/boards-and-kits`

[7] `https://github.com/bradfitz/gomemcache`



(a) Insert response time increases with the value size. Replication adds constant overhead.

(b) Reads and Updates become more costly with large values. Replication overhead is the same as for Inserts.

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

Figure 7: Response times for different operation mixes and load levels

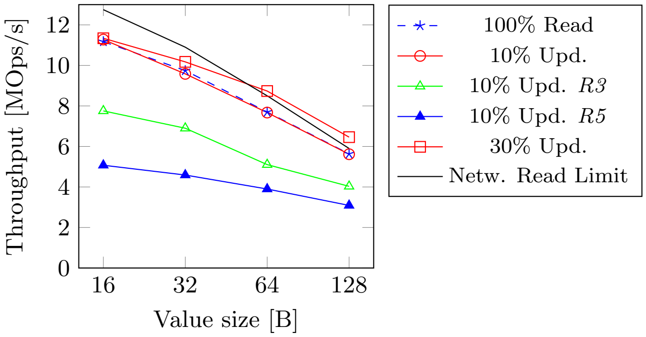


Figure 8: Throughput is mostly limited by the network for read-heavy operation mixes

If we compare the latency of *Caribou* as measured at the nodes to other FPGA-based systems [7, 21, 53] and OS-bypassed software solutions (e.g., MICA [35]) the numbers are in the same range, even though *Caribou* implements additional functionality (Table 3).

6.3 Read/Write effects

To test the throughput of *Caribou* under a more realistic mix of operations [4] we start experiments with a prepopulated store and issue reads and updates from the clients. Figure 8 shows that performance stays close to the maximum read-only throughput achievable over 10Gbps Ethernet and TCP/IP. For small response packets, the system does not achieve the theoretical maximum but it is close to 90% of it. As the value size increases, 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 on the master node is as high as 70% of the non-replicated one, and this amount of overhead is expected as both the replication logic and the TCP/IP stack has to issue and receive additional messages for each replication round. From the perspective of replicas, the overhead of replicated writes is very close to that of local writes. Overall, the performance of *Caribou* is close to be network bound for reads and updates which means that it is able to handle the load that can be received over a 10Gbps Ethernet link.

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. 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.

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 7c 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.

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 where an increasingly small percentage of the key-set is accessed. Figure 9 shows that even when this 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).

6.4 Scan Performance

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

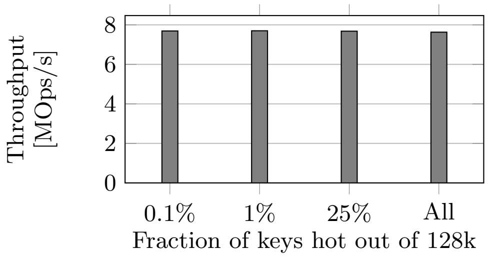


Figure 9: Skew has no negative impact on throughput due to the single processing pipeline (10% updates, 64 B values)

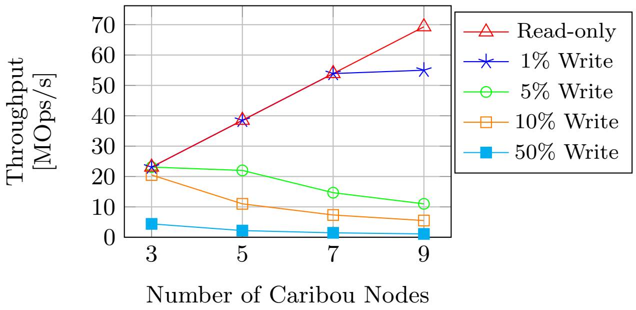


Figure 10: Projected scalability of Caribou with replication to all nodes and different workloads (64 B values)

both the key to lookup and the parameters for the selection computation (48 B both for the string matcher and the 8 conditional expressions), hence 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 11 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.

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 12 depicts the rate at which the data area can be scanned given the selectivity and different value sizes. The trend in the graph is expected because, regardless of the choice of selection types, the processing pipeline of the FPGA is bound at 5 GB/s throughput, which 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. The overhead for scanning the bitmaps which determine what memory lines to retrieve amounts to 400μs for each GB of value storage. This should not impact performance unless the tablespace is very sparsely populated. To counter these cases, in the future these bitmaps could be augmented with pointers to skip over non-allocated areas.

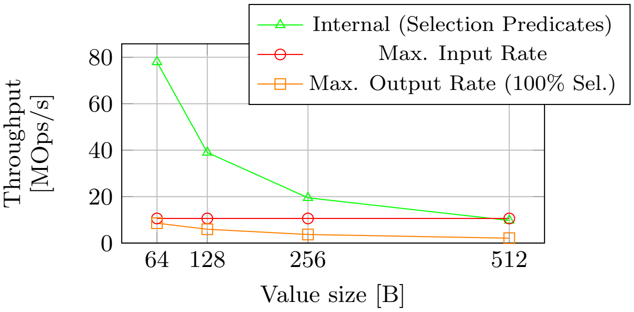


Figure 11: When combining single-value reads and selection, network is the limiting factor. Internal bandwidth is clearly over-provisioned

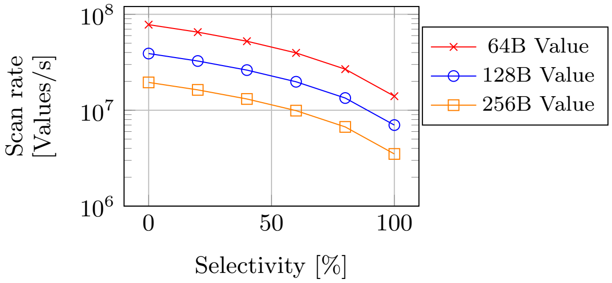


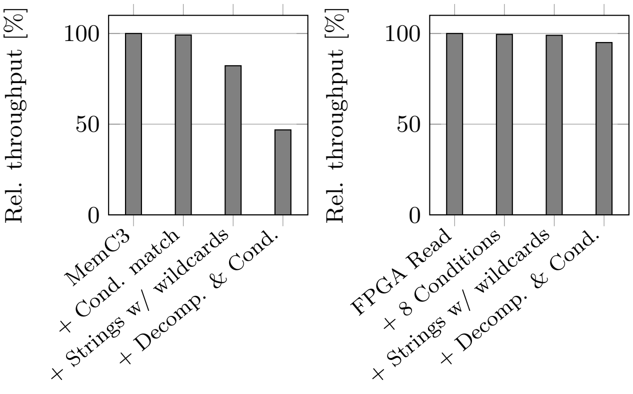
Figure 12: Scan performance is limited by selection (approx. 5GB/s, regardless of type) for low selectivity, and by the network (at most 1.25GB/s) for high selectivity

6.5 Scalability

In our current setup we are constrained by the number of load generators, that are unable to saturate multiple FPGAs for small values, but given our measurements on the 3 and 5 node setup, it is possible to extrapolate for larger deployments of *Caribou*. We assume the presence of an external transaction/commit manager that maps the database-level transaction identifiers to the lower level consensus rounds. This enables the clients to read from all nodes, provided that they contain the updates that have to be visible to a specific transaction. 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 write-only workloads the leader can sustain 2.2M and 1.1M requests/s for *R3* resp. *R5* for 64 B values. Figure 10 sums up these numbers and projects the aggregated throughput of *Caribou* nodes for larger deployments. The decline in performance with more writes is explained by the bottleneck on the leader's output bandwidth and is typical for systems replicating from a single master copy [58]. In larger deployments it could be beneficial to partition the data among multiple smaller groups of *Caribou* nodes, instead of treating it as one logical unit. This way both high performance and fault tolerance could be achieved, even in the presence of write-intensive workloads.

6.6 Effect of Complexity on Software

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



(a) Software

(b) Hardware

Figure 13: Impact on adding more processing to *Caribou* and MemC3 has different effects. Throughput is normalized to the platform-specific maximum.

added functionality decreases performance. We illustrate this effect with a simple experiment for which we use a modified version of MemC3 [16]. 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 hardcoded array of bytes in the value ("Cond. match" using *strstr*), b) check for a string search with wildcards ("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 13a shows the relative performance degradation with more features added (100% amounts to 310k read requests/s). Figure 13b illustrates how *Caribou* delivers network-bound, or close to network bound throughput with or without added features (the workload has 8B keys and 512 B 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.

6.7 Resource Utilization

In hardware the cost for complexity is real estate, which translates into power consumption. However, the power drawn by an FPGA is an order of magnitude lower than that of a server-grade CPU, even when fully utilized. Additionally, with increased chip-area usage, placing and routing of the logic elements on the chip becomes more difficult. In Table 2 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 36 Kb on-chip memory it requires for its operation. These are mostly used for buffers and various caching structures. As depicted in Table 2, 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. Newer FPGA devices have hardened memory controllers that are faster and would not require real estate on the programmable fabric. The string matcher unit has high real estate requirements because it is a combination of 32 smaller units, and the BRAM memories are used as FIFOs for distributing and collecting work. 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, they consume less resources overall. To sum up, even when one would deploy all the features we implemented in the same system, there would be chip space left to add more functionality to *Caribou*.

Table 2: Resource consumption of each module in *Caribou*.

Module	**Logic slices (%)**	**BRAMs (%)**
DRAM Controller	25k (23%)	128 (9%)
Networking	11k (10%)	151 (10%)
Replication	3.5k (3.2%)	65 (4.4%)
Memory allocation	2.9k (2.7%)	8 (<1%)
Hashtable	3.9k (4.4%)	119 (8%)
Total w/o processing	**46.3k (42.9%)**	**471 (32%)**
Pred. Eval: 8 Conditions	4.8k (4.4%)	78 (5.3%)
Pred. Eval: Substring	16.4k (15.2%)	240 (16%)
Decompression	4.5k (4.2%)	64 (4.4%)
Available on device	**108k (100%)**	**1470 (100%)**

7. RELATED WORK

7.1 High-performance Key-value Stores

There is a growing body of performance-oriented key-value stores built in software and hardware. Software approaches [15, 16, 34] focus on two main aspects, 1) reducing the overhead of networking by exploiting user-space network stacks and RDMA, 2) adapting data structures for multi-core scalability and make them aware of the underlying hardware. On the hardware side, novel solutions based on dataflow architectures have been proposed [7, 8, 53, 67]. The goal is to implement complete processing pipelines in hardware which operate at a high-bandwidth and are able to hide the memory access latency.

One software approach, MemC3 [16], optimizes memcached with algorithmic and data structure improvements by replacing the original hash table with a cuckoo-hash [41] based structure to reduce locking conflicts. To achieve even higher throughput in software it is necessary to "specialize" the whole software stack to the key-value store workload. In MICA [34, 35], for instance, Intel's DPDK library is used to access the network from user-space, thereby removing the overhead of the operating system's network stack and reducing latencies. To reduce the overhead of networking further, MICA uses UDP for client-server communication. While this stateless protocol increases throughput, it is less robust than TCP. The results of this work are very promising: the complete system demonstrates 120 MRPS over four 40Gbps NICs in a single machine. It achieves its high access rate through a combination of lossy and lossless data structures. On the downside, in MICA clients perform some of the server's tasks (e.g. hashing the key) and skewed workloads experience slowdowns due to the partitioned nature of the data structures. In *Caribou* there is a single data

Table 3: Comparison of representative hardware and software key-value store designs

System	Type	Properties	Ops/s Per Node	Min. Latency	Power Per Node
Tanaka et al. [53]	FPGA	Single node, UDP	20M (on 20Gbps)	4μs	15W + 100W
Blott et al. [7, 19]	FPGA	Single node, UDP	13M	4.5μs	26W
BlueCache [67]	FPGA	Custom dedicated network	4M	>20μs	40W*
MemC3 [16]	SW	Single node, TCP	4.4M	>20μs‡	>120W†
MICA [34]	SW	Single node, UDP	120M (on 4x40Gbps)	<20μs	317W
FARM [15]	SW	Replicated, Transactions, RDMA	>10M (on 40Gbps)	<10μs	>230W†
Caribou	FPGA	Replicated, Data processing, TCP	11M	10-20μs	29W*

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

structure and key-access skew inside a node has no negative effect on performance (Section 6.3, Figure 9).

Another way of addressing the networking overhead in the traditional stack is to use RDMA [15, 40, 44]. In (one-sided) RDMA the CPU is not involved in the data movement leading to a drastic reduction in latency. In FARM [15], 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 10 MRPS. These systems are however not able to fully saturate the underlying network (Infiniband QDR and 40GbE). For small request sizes they are limited by the packet rate. This illustrates that for software based solutions there is a fixed overhead per packet which would continue to increase when adding further functionality.

Recent work by Xilinx [8] extends the hardware-based caching-KVS design with flash-based storage for more capacity. The work by Tanaka et. al [53] is a memcached replacement with flash storage attached to the FPGA for higher capacity. It uses a hash table design similar to the one in *Caribou*, but lacks near-storage processing. *Caribou* would benefit from a higher storage capacity, as it is currently limited by the DDR memory available on the FPGA board. The techniques used in the previously mentioned works could be used to extend its storage capacity with flash.

BlueCache [67] is a flash-based key-value store for caching. The nodes are connected through a special purpose dedicated network. The performance of a single node reaches up to 4 M operations/s demonstrating that flash-based FPGA designs can achieve excellent performance. In contrast to *Caribou*, BlueCache nodes are not stand-alone and offer no built-in replication or processing. While the dedicated network delivers excellent latency and eliminates virtually all packet loss, we consider TCP a more flexible method of connecting multiple nodes together. This applies even more if we want to provision regular servers and specialized nodes separately inside a rack.

7.2 Near-data Processing

The idea of performing near-data computation is not new, and has been explored both in the context of main memory (active memory [2, 17, 39, 66]) and persistent storage (active disk [1, 14, 23, 46]). What makes *Caribou* different is that it targets distributed systems by design and combines processing with high-level interfaces and fast networking. And while network-attached HDDs and SSDs with key-value interfaces are also available, e.g, from Toshiba and Seagate [8], these do not implement application-specific processing. In *Caribou* we demonstrate that it is possible to perform both data management and data processing in a single, small footprint device efficiently.

Examples of projects using specialized hardware are IBM's Netezza data warehouse appliance[9], Ibex [64], and various others [11, 24, 25, 61]. Thanks to the dataflow parallelism of the devices, complex processing can be performed on the storage side without slowing down data transmission. In fact pre-processing, -selection, and -aggregation can lead to a reduction in the amount of data that has to be transmitted. A clear drawback of such devices is that they are less flexible than traditional software solutions. Examples such as the SmartSSD from UWisconsin [14] or the recent work from Samsung [23] demonstrate the versatility of software, but also show its limitations in compute complexity. In this work we focus on computational capability first, but also try to make the circuit as runtime-parameterizable as possible.

In the domain of databases, multiple distributed DRAM-based storage solutions exist, such as RamCloud [40] or TellStore [42]. Trivedi et al. [57] also explore a similar design, targeting graph processing. Furthermore, various related work explores how relations can be represented with a variety of data structures optimized for low latency random access [9, 13, 32, 33]. It is also becoming more common to design relational engines with a more clear separation between processing and storage layer [5, 36, 40, 47]. We see *Caribou* as a natural fit for these systems.

7.3 Query Processing on FPGAs

There is a significant interest in using specialized hardware to increase the efficiency or performance of database workloads. These works show promising results in either offloading sub-operators (e.g., partitioning [30, 59]) or whole execution pipelines to hardware [12, 38, 45, 51, 60].

Even though the above mentioned works are in the context of accelerators near a CPU, the operator designs could be ported to *Caribou* to offload for instance joins directly to the storage. Furthermore, the body of related work implementing statistics on FPGAs [22, 52, 56] could be used to provide insights about the data to the processing layer at no additional cost.

7.4 Performance Comparison

We compare *Caribou* to a set of representative designs in more detail in Table 3. 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

[8] `https://www.openkinetic.org/technology`

[9] `http://www.ibm.com/software/data/netezza/`

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 [34, 35]). MICA achieves an order of magnitude higher throughput which makes it a very energy efficient system, demonstrating 378k RPS/Watt. This number is comparable to our prototype which reaches as high as 380k RPS/Watt. With replication it can achieve 267k RPS/Watt and this reduction is 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.

8. LESSONS LEARNED

Caribou is an exercise in near-data processing. Its internal design is driven by the goal of keeping up with network line-rate and this influenced the choice of data structures and algorithms. For instance, even though some parts of the hash table could have been run at a higher clock-rate than the network or extended to a wider data-bus, making these units faster would not change performance if it is already limited by the network. The same applies for interfacing the DRAM memory and selection operators. Such co-design also applies to many high performance software key-value stores but what makes hardware different is that there are less "unknowns". Everything between the network interface and the memory controller is under the control of the hardware designer. The circuit on the FPGA can be designed such that it is neither over- nor under-provisioned. This leads to predictable behavior, which is crucial for a storage service. Conversely, in software there are many interacting layers and caches that could introduce jitter in the behavior. This holds even when using DPDK and other user-space libraries.

Another aspect to consider related to systems running on specialized hardware is that while improvements over highly-optimized software solutions might not be very large in individual dimensions, they are significant when considering multiple dimensions at the same time (e.g., *Caribou* delivers lower response time and higher throughput, in an overall lower energy footprint). In addition to the raw performance numbers, specialized hardware will often provide more stable and predictable response times, even when combined with complex processing. In contrast to software, adding functionality to a dataflow pipeline does not reduce throughput. We illustrated this point with a comparison-based selection operator and a string matcher, both working as pipelines at the same rate. In the future, additional functionality such as partial aggregation [64], statistics [22], complex event detection [65], etc., could be added to the circuit.

9. CONCLUSION

We presented *Caribou*, a hardware-managed distributed storage solution with near-data computation. Its design is based on and extends our earlier work [19, 21, 50, 64], and fills gaps that currently exist in similar solutions (such as memory allocation). Thanks to the dataflow parallelism of hardware, *Caribou* delivers competitive performance while also providing rich function offloading and both point-lookup and scan-based access methods. We benchmark the system and demonstrate up to 11MRPS performance even for small value sizes. Response time is low (tens of μs end-to-end), a significant part of which is the software overhead in clients. The ideas in this paper can be used as a starting point for porting or building database engines on top of disaggregated storage that has a small energy footprint, and as a result can scale wider.

Acknowledgments

This work has been partially funded by the Microsoft Swiss Joint Research Center (JRC). We would like to thank Xilinx for their generous donation of the FPGAs used in this work.

10. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11), 1998.

[2] J. Ahn, S. Hong, S. Yoo, et al. A scalable processing-in-memory accelerator for parallel graph processing. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 105–117, 2015.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, et al. FAWN: A fast array of wimpy nodes. In *SOSP'09*. ACM.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, et al. Workload analysis of a large-scale key-value store. In *SIGMETRICS'12*.

[5] P. A. Bernstein, C. W. Reid, and S. Das. Hyder: A transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[6] M. Bjorling, P. Bonnet, L. Bouganim, and N. Dayan. The necessary death of the block device interface. In *CIDR'13*.

[7] M. Blott, K. Karras, L. Liu, et al. Achieving 10gbps line-rate key-value stores with FPGAs. In *HotCloud'13*.

[8] M. Blott, L. Liu, K. Karras, and K. Vissers. Scaling out to a single-node 80gbps memcached server with 40Terabytes of memory. In *Usenix HotStorage'15*.

[9] L. Braun, T. Etter, G. Gasparis, et al. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD'15*.

[10] A. De, M. Gokhale, R. Gupta, et al. Minerva: Accelerating data analysis in next-generation SSDs. In *FCCM'13*.

[11] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *FCCM'13*.

[12] C. Dennl, D. Ziener, and J. Teich. On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In *FCCM'12*.

[13] C. Diaconu, C. Freedman, E. Ismert, et al. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD'13*.

[14] J. Do, Y.-S. Kee, J. M. Patel, et al. Query processing on smart SSDs: opportunities and challenges. In *SIGMOD'13*.

[15] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *NSDI'14*.

[16] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI'13*.

[17] M. Gao and C. Kozyrakis. Hrl: Efficient and flexible reconfigurable logic for near-data processing. In *HPCA'16*.

[18] Z. István, G. Alonso, M. Blott, and K. Vissers. A hash table for line-rate data processing. *ACM TRETS*, 8(2), Mar. 2015.

[19] Z. István, G. Alonso, M. Blott, and K. A. Vissers. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *FPL'13*.

[20] Z. István, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *FCCM'16*.

[21] Z. Istvan, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *NSDI'16*.

[22] Z. István, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *SIGMOD'14*.

[23] I. Jo, D.-H. Bae, A. S. Yoon, et al. YourSQL: a high-performance database system leveraging in-storage computing. *VLDB'16*.

[24] S.-W. Jun, M. Liu, K. E. Fleming, et al. Scalable multi-access flash store for big data analytics. In *FPGA'14*.

[25] S.-W. Jun, M. Liu, S. Lee, et al. BlueDBM: an appliance for big data analytics. In *ISCA'15*.

[26] F. P. Junqueira, B. C. Reed, et al. Zab: High-performance broadcast for primary-backup systems. In *DSN'11*.

[27] R. Kallman, H. Kimura, J. Natkins, et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB'08*.

[28] Y. Kaneta, S.-i. Minato, and H. Arimura. Fast bit-parallel matching for network and regular expressions. In *Proc. 17th SPIRE*. Springer, 2010.

[29] K. Kara and G. Alonso. Fast and robust hashing for database operators. In *FPL'16*.

[30] K. Kara, J. Giceva, and G. Alonso. FPGA based data partitioning. In *SIGMOD'17*.

[31] F. Klein, K. Beineke, and M. Schöttner. Memory management for billions of small objects in a distributed in-memory storage. In *IEEE CLUSTER'14*.

[32] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE'13*.

[33] J. J. Levandoski, D. B. Lomet, S. Sengupta, et al. High performance transactions in Deuteronomy. In *CIDR'15*.

[34] S. Li, H. Lim, V. W. Lee, et al. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM SIGARCH Computer Architecture News*, volume 43. ACM, 2015.

[35] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI'14*.

[36] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD'15*.

[37] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. *SIGMOD'17*.

[38] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *VLDB'13*.

[39] M. Oskin, F. T. Chong, and T. Sherwood. *Active pages: a computation model for intelligent memory*, volume 26. IEEE Computer Society, 1998.

[40] J. Ousterhout, P. Agrawal, D. Erickson, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 2010.

[41] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 2001.

[42] M. Pilman. Tell: An elastic database system for mixed workloads (PhD Dissertation No. 24147), ETH Zürich, Switzerland, 2017. `https://www.systems.ethz.ch/sites/default/files/file/UpcomingPublications/PilmanMarkus2016.pdf`.

[43] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[44] M. Poke and T. Hoefler. DARE: high-performance state machine replication on RDMA networks. In *HPDC'15*.

[45] B. Salami, G. A. Malazgirt, O. Arcas-Abella, et al. AxleDB: A novel programmable query processing platform on FPGA. *Elsevier Microprocessors and Microsystems*, 51, 2017.

[46] S. Seshadri, M. Gahagan, S. Bhaskaran, et al. Willow: a user-programmable SSD. In *OSDI'14*.

[47] J. Shute, M. Oancea, S. Ellner, et al. F1-the fault-tolerant distributed RDBMS supporting Google's Ad business. In *SIGMOD'12*.

[48] D. Sidler, G. Alonso, M. Blott, et al. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *FCCM'15*.

[49] D. Sidler, Z. István, and G. Alonso. Low-latency TCP/IP stack for data center applications. In *FPL'16*.

[50] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD'17*.

[51] B. Sukhwani, H. Min, M. Thoennes, et al. Database analytics acceleration using FPGAs. In *PACS'12*.

[52] Y. Sun, Z. Wang, S. Huang, et al. Accelerating frequent item counting with FPGA. In *FPGA'14*.

[53] S. Tanaka and C. Kozyrakis. High performance hardware-accelerated flash key-value store. In *Non-volatile Memories Workshop (NVMW)*, 2014.

[54] J. Teubner and L. Woods. Data processing on FPGAs. *Morgan & Claypool Synthesis Lectures on Data Management*, 2013.

[55] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *SIGMOD'12*.

[56] D. Tong and V. Prasanna. High throughput sketch based online heavy hitter detection on FPGA. *ACM SIGARCH Computer Architecture News*, 43, 2016.

[57] A. Trivedi, P. Stuedi, B. Metzler, et al. Rstore: A direct-access DRAM-based data store. In *ICDCS'15*. IEEE.

[58] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*.

[59] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *FPL'15*.

[60] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on opencl-based fpgas. In *FPL'16*, 2016.

[61] M. Wei, J. D. Davis, T. Wobber, et al. Beyond block I/O: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013.

[62] R. Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[63] L. Woods, G. Alonso, and J. Teubner. Parallelizing data processing on FPGAs with shifter lists. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2), 2015.

[64] L. Woods, Z. István, and G. Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11), July 2014.

[65] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *VLDB'10*.

[66] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *DAMoN'15*. ACM.

[67] S. Xu, S. Lee, S.-W. Jun, et al. Bluecache: A scalable distributed flash-based key-value store. In *VLDB'16*.

[68] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3), 1977.