Specialize in Moderation – Building Application-aware Storage Services using FPGAs in the Datacenter

Lucas Kuhring
IMDEA Software Institute, Madrid, Spain

Eva Garcia*
Universidad Autónoma de Madrid, Spain

Zsolt István
IMDEA Software Institute, Madrid, Spain

Abstract

In order to keep up with big data workloads, distributed storage needs to offer low latency, high bandwidth and energy efficient access to data. To achieve these properties, most state of the art solutions focus either exclusively on software or on hardware-based implementation. FPGAs are an example of the latter and a promising platform for building storage nodes but they are more cumbersome to program and less flexible than software, which limits their adoption.

We make the case that, in order to be feasible in the cloud, solutions designed around programmable hardware, such as FPGAs, have to follow a service provider-centric methodology: the hardware should only provide functionality that is useful across all tenants and rarely changes. Conversely, application-specific functionality should be delivered through software that, in a cloud setting, is under the provider's control. Deploying FPGAs this way is less cumbersome, requires less hardware programming and flexibility increases overall.

We demonstrate the benefits of this approach by building an application-aware storage for Parquet files, a columnar data format widely used in big data frameworks. Our prototype offers transparent 10Gbps deduplication in hardware without sacrificing low latency operation and specializes to Parquet files using a companion library. This work paves the way for in-storage filtering of columnar data without having to implement file-type and tenant-specific parsing in the FPGA.

1 Introduction

With the wide-spread deployment of big data workloads in the cloud, it is important to increase the efficiency of distributed storage. Most recent work in this area focuses on either highly-tuned software solutions [1–3] to exploit fast networks, or pushes all functionality into specialized hardware nodes [4–6] that promise an order of magnitude improvement in energy efficiency. Hardware solutions, however, face two impediments: *1)* deploying them in the cloud is uneconomical unless high utilization can be achieved and *2)* they are limited in the operations they can efficiently support due to small on-chip memories and the fact that functionality is typically translated to pipelines or circuitry with an upper bound on size.

Field Programmable Gate Arrays (FPGAs) are overcoming the first impediment in part thanks to machine learning and similar compute-intensive workloads that are often bound by the performance of general-purpose CPUs. As a result, programmable accelerators are becoming common and can be rented in the public cloud of companies such as Amazon, Baidu and Huawei. Furthermore, they have been recently successfully used in production [7] at Microsoft where they accelerate networking functions in the Azure Cloud. These uses demonstrate that FPGAs and similar programmable hardware can be economical if they benefit a large number of tenants. Beyond compute kernels and network functions, even complex stateful applications, such as low-latency key-value stores [6, 8], can be implemented on FPGAs, resulting in at least an order of magnitude improvement in energy efficiency.

Given these developments and the growing size and heterogeneity of FPGAs [9], new opportunities emerge in architectural exploration for data-intensive applications. The challenge of providing rich functionality with bounded hardware resources is, however, not solved yet. As an example, in the case of smart storage, even though the FPGA could implement different application-specific modules, it is unclear how to share the device efficiently across tenants. If the device is spatially partitioned, less functionality can be provided for each tenant, and this could lead to wasted resources if not all applications are used equally often. Temporal partitioning is not a perfect solution either, because reconfiguring parts of the FPGA can be done only at coarse granularity [16, 17] and takes tens to hundreds of milliseconds.

Proposal. We make the case that, for specialized-hardware-based storage to be practical, it has to cater to many tenants and use-cases. The way to achieve this is by splitting functionality: keeping the common data or compute-intensive parts of workloads on the hardware and providing additional tenant- and application-specific operations in software libraries. Such

*Work done while employed at the IMDEA Software Institue.

a separation also reduces the hardware programming effort and allows faster adaptation to changing workloads, which is important in the context of today's quickly evolving data-intensive service offering [10–13]. Even though the underlying idea is not new, we believe it is time to revisit it from the perspective of FPGA-based storage and cloud service providers. Furthermore, the common blocks we can identify across applications can be used to guide the process of developing FPGAs tailored to the needs of datacenter workloads.

Applying the Methodology. To demonstrate our approach, we implement a storage solution with line-rate data management and deduplication in hardware and move application-specific operations to a client library without reducing performance. We chose to target Apache Parquet [14] in our prototype, a commonly used columnar file format, but the methodology is generalizable to other formats as well. This is because overall it ensures that the FPGA can be in the role of both a general and an application-aware storage node and, in the future, the free chip space can be used to provide in-storage processing to multiple tenants, even if only a subset of them store Parquet files.

The software library breaks files up based on their internal structure and stores them as multiple smaller entries in the storage node. This reduces the size of buffers on the FPGA and keeps the data management logic general purpose. Since the software library handles file-specific meta-data processing, it is possible to implement in-storage processing without requiring Parquet-specific parsers on the FPGA. This is important because the requirement to parse different file formats can reduce the usable space on the FPGA: we know from earlier work [15] that meta-data parsing and data fetching logic for simple database pages can be comparable in size to a 10Gbps Regex processing module.

2 Background and Related Work

FPGA-based Storage. There is an increasing interest in offering FPGA-based or FPGA-accelerated distributed storage [4–6, 8, 18]. This interest is in large motivated by energy efficiency concerns: when deploying TBs of storage, the energy footprint of individual nodes becomes an important cost factor and FPGA-based solutions offer orders of magnitude better energy efficiency than traditional CPU-based solutions. An other advantage of FPGAs (and other specialized hardware) is that they are well suited to network-facing operation because they can guarantee line-rate operation by design. A drawback of many FPGA-based KVS prototypes is that they are designed with small (16-512 B) key-value pairs in mind, motivated by caching use-cases – when targeting big data workloads, however, this design decision has to be revisited.

Deduplication Strategies. Deduplication has been studied in extensive related work and one important differentiating factor between solutions is the way that files are subdivided (chunked) for fingerprinting. Fixed chunk size solutions [19, 20] are computationally cheaper and offer good deduplication ratios for files that are modified through append operations. Conversely, for files that are changed through random insertions and deletions, variable chunk size hashing [19, 21–23], that computes a running hash of the data to identify content-based "cut" points, is better suited.

Another differentiating factor is the location of deduplication: on the client-side [22, 24–26] or in the storage node/device [26–29]. In the case of the former, the bandwidth requirement when writing to storage is reduced by not sending duplicate data, but latency is increased because multiple round-trips are required to identify which parts of a file are unique. This approach also requires the hashing to happen on the client, which can interfere with the client workload. Performing deduplication inside in the storage node addresses these shortcomings but requires that the underlying hardware guarantees high throughput for deduplication operations.

As related work already explored [23], deduplication methods that take into account the internal structure of files are likely to perform better than general-purpose ones. In a similar vein, we propose a hybrid chunking scheme that first breaks up the files into relatively small pages (in the range of 1 KB – 8 KB) along their internal structure without requiring hash computations on the clients, then stores pages as multiple key-value pairs, depending on the maximum value size supported by the KVS. This results in behavior similar to variable size chunking and in very similar, or better, deduplication ratios.

Parquet Files. Emerging machine learning workloads have made column-based file formats, such as Apache Parquet [14], wide-spread. Depending on the computation pipeline, users often access only a subset of the columns in these files, and having a storage solution that can efficiently handle such partial reads reduces data movement bottlenecks. Furthermore, Parquet files are often transformed in ways that touch only specific columns or derive new ones in the feature engineering step of machine learning pipelines [30], which leads to partially duplicate data and therefore potential for deduplication.

3 Design Overview

Our prototype system, *Multes++*, shows that by combining software and specialized hardware using a service-oriented methodology, it is possible to deliver application-specific improvements without reducing the efficiency benefits of the hardware nodes. It brings the following properties:

1. Line-rate read and write behavior in an energy efficient package – enabled by the use of specialized hardware.
2. Structure-aware storage and access of Parquet files without having to dedicate hardware resources to file-type-specific parsing – enabled by the use of a software library.
3. Efficient deduplication using a hybrid chunking scheme – enabled by combining software and hardware.

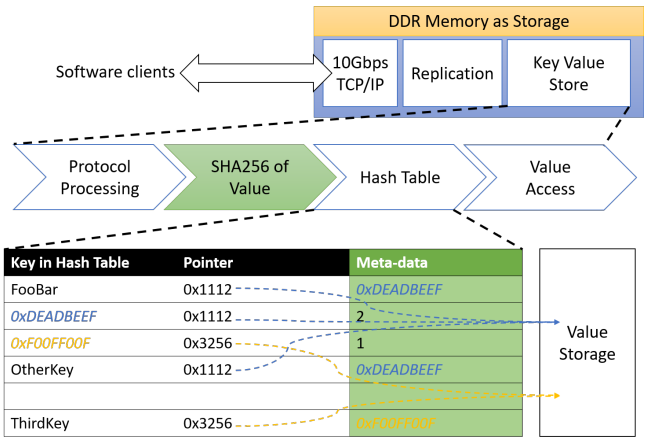

Figure 1: Highlighted in green are the deduplication-specific operations and meta-data added to *Multes*.

Starting Point. This work is based on *Multes* [31], an open source, replicated [32], multi-tenant key-value store (KVS) with performance and data isolation guarantees across tenants. The KVS supports basic set/get/delete operations executed as part of a single pipeline tailored to 10Gbps network speeds and parameterizable to match the performance characteristics of emerging NVM devices (for details see [8]). Internally, a high throughput Cuckoo hash table [33] is combined with a slab-based storage allocator. The hash table stores keys and value-pointers and the actual values are stored separately in a "value area" that is managed by the allocator. It has five slab classes, increasing by factor of two until 1KB, and ensures that variable sized values can be stored without too much internal fragmentation.

Multes stores data in DRAM, which is a placeholder for non-volatile media. The design allows for using durable storage that is fast enough for 10Gbps operation (e.g. to NVM DIMMs) by resizing on-chip buffers but without having to change the KVS design.

Hardware Implementation. We achieved the goals stated at the beginning of this section by extending the internal pipeline of *Multes* with deduplication. For this, we had to modify the key management logic, but this does not slow down the system nor changes the way its multi-tenancy and replication mechanisms works.

We added a fingerprinting step that precedes the hash table operations (Figure 1), and each value that is part of a set request is hashed to determine its fingerprint. For this, we use open-source SHA256 cores[1] in a data-parallel way. Each core takes 66 clock cycles at 156.25 MHz to hash 64B of data, so we deploy nine parallel ones to achieve 10Gbps line-rate (8B/cycle at 156.25 MHz) fingerprinting.

Since the hash table stores all its entries in off-chip memory, it has a large enough capacity to hold both keys and fingerprint entries of values. To achieve this, we changed the internal

[1] https://github.com/secworks/sha256

Algorithm 1 Writing to the KVS with deduplication

```
1:  function SET(Key, Value)
2:      F = SHA256Hash(Value)                          ▷ Compute fingerprint
3:      [Pointer, Count] = GetHashTable(F)  ▷ Read #1 – NULL returned if not found
4:      [_, F_Key] = GetHashTable(Key)      ▷ Read #2 – NULL returned if not found
5:      if F_Key!=NULL and F!=F_Key then
6:          DeleteHashTable(Key)  ▷ If the key already has a different value, delete it
7:      end if
8:      if F_Key!=NULL and F=F_Key then
9:          SetHashTable(F, [Pointer,Count]) ▷ No change in counts (can be omitted)
10:     end if
11:     if F_Key==NULL and Pointer != NULL then
12:         SetHashTable(F, [Pointer,Count + 1])  ▷ New key with an existing value
13:     else
14:         if Pointer == NULL then
15:             Pointer = Malloc(Size(Value))          ▷ Allocate space for new value
16:             SetHashTable(F, [Pointer, 1])          ▷ First time we store this value
17:         end if
18:     end if
19:     SetHashTable(Key, [Pointer,Fingerprint])                ▷ Update the key
20:     WriteValueArea(Pointer, Value)                          ▷ Write to value area
21: end function
```

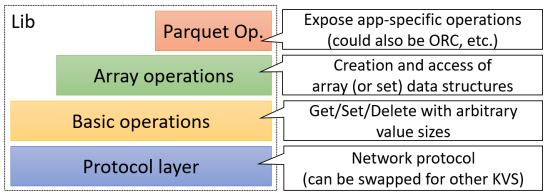

Figure 2: Our library exposes several layers of abstraction, hiding low level communication and FPGA idiosyncrasies.

structure of the hash table entries slightly compared to *Multes* so that each entry holds not only a key and a pointer (which is a combination of a value memory location and the length of the value) but also an additional "Meta-data" field (see bottom of Figure 1). In the case of regular keys, this field is used to store the fingerprint[2] of the values and in the case of fingerprint entries, it is used to store the number of keys with that particular value (i.e., a reference counter).

For reading the value of a key we can use the original hash table logic since we store the pointers to the values with the keys. This means that the read throughput of *Multes++* is identical to that of *Multes*. As Algorithm 1 shows, write operations require an additional memory access per key-value pair, which could result in lower throughput for small values (<128B) but targeting small values for deduplication does not pay off due to the meta-data overhead required. Delete operations behave similarly to writes because they need to update the fingerprint entry as well.

Software Library. We built the software library for *Multes++* in Golang. Beyond the basic get/set operations, the library also provides the ability of storing very large values and arrays transparently (Figure 2 shows its layers). The Parquet file functionality is layered on top of these operations.

One limitation of FPGA designs shows up when dealing

[2] We reduce fingerprints to 64bits in order to fit the current hash table. While false positives should be still extremely unlikely, in a production-grade system with large capacity it would be preferred to store fingerprints in full.

```
addr = '11.1.212.209:2880'
h = pq.connect(addr)
md = pq.open_metadata(h, 'test', schema=0)
airline = pq.get_string_column(h, md, 1)
weight = pq.get_int_column(h, md, 10)
df = pd.DataFrame(data={'a':airline, 'w':weight})
df.sort_values(by=['w'], inplace=True)
print(df.head(5))
```

Figure 3: The library can be called from Python and clients can access specific columns directly. In this snippet, the columns are loaded into a Pandas DataFrame for analysis.

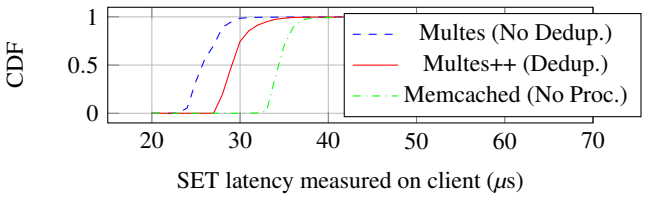

Figure 4: When deduplicating SETs with 512B values, the added latency is visible but not significant for clients. Read operations are not impacted.

with very large request sizes: To handle a multi-KB set request, for instance, the FPGA has to have dedicated buffer space for each tenant. Since on-chip memory is limited, these buffers have to be moved to DRAM, which increases latency and could jeopardize line-rate guarantees. In *Multes++*, we limit key-value sizes to the KB range, so that each logical request will fit in an MTU and fragmentation happens only seldom. This allows *Multes++* to operate with small buffers. We handle larger values by slicing them on the client and storing multiple key-value pairs. To differentiate between user-provided keys and "slice" ones, the library appends index bytes to the key. The first key-value pair encodes how many additional ones follow. To improve throughput, the library batches the access to the slices of the same logical key.

In addition to large value operations, we also implemented an array abstraction in the library. Arrays are accessed through a "directory value" that stores the internal keys pointing to the array entries. These can be single values, long values, or even arrays themselves.

In order to manipulate Parquet files, we embedded an open source Parquet library[3] in our solution to divide the Parquet file into pages. We store all pages as array entries, with file-specific meta-data stored at the first locations in the array. It is also feasible to plug the library into clients written in other languages. In Listing 3 we show, for instance, our experimental go-to-Python binding. Clients can access the Parquet files either as a whole or on a per-column basis. This reduces unnecessary data movement, and opens up opportunities for offloading computation to the storage node in the future.

4 Evaluation

Setup. We use four dual-socket servers with Intel Xeon Silver 4114 CPUs and 10Gbps networking and a Xilinx VC709 Board with a XC7VX690T FPGA and 8GB of DRAM that is used both for the hash table and the value storage.

The client machines ran *Multes++*'s Go-based client library and used Parquet files generated from the datasets available at the San Francisco Open Data Website[4]. We compare *Multes++* with Memcached, that even though not the most optimized software solution, is representative of the performance of general purpose systems. For memcached we change the lowest layer in the library but the others remain the same.

The evaluation focuses on the perspective of individual clients storing and retrieving data from *Multes++*, simulating the behavior of cloud tenants running machine learning jobs, e.g., in Python. In this scenario, even though a tenant can have many parallel tasks, each one is heavily impacted by the data retrieval rate from the storage nodes and can not parallelize across multiple machines.

Resource Consumption. *Multes++* occupies 57% of the logic resources and 42% of BRAM on the chip with an 8 tenant setup (*Multes* needs 51% and 37% for the same), the remaining chip space can be used to implement processing on the reading path. It is important to highlight that this deployment has only a single copy of all data management modules, which are then shared by tenants – clearly, replicating these several times is not an option.

The estimated power consumption of the FPGA when running *Multes++* is 12W, an order of magnitude lower than what a general purpose CPU would use to provide 10Gbps KVS functionality, without additional processing [8].

Latency. When compared to *Multes*, the only step that has been added in *Multes++* that impacts latency is the deduplication logic for set (and delete) operations. The overhead is composed predominantly of the SHA256 hashing that adds in the range of 3μs of latency[5] per 512 B values (Figure 4). The overhead of the additional DDR memory access is negligible.

As for manipulating array data structures, since this operation is entirely implemented in software, it is possible to run it both on top of *Multes++* and Memcached. Figure 5 shows that when reading a single element of an array, two RTT to the server have to be paid: one to retrieve the array meta data and one to read the actual value. If the clients read multiple elements in a single operation, however, the cost of retrieving the meta data is quickly amortized. Memcached shows the same trend as the FPGA, but its base latency is significantly higher than two accesses to *Multes++*.

[3] github.com/xitongsys/parquet-go/
[4] https://datasf.org/opendata/

[5] By increasing the frequency of the hashing cores, it would be possible to improve this metric by a factor of 2, but likely not much more.

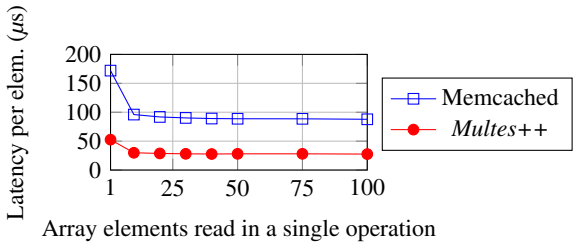

Figure 5: Since array accesses require an additional level of indirection, accessing a single element will incur two RTT latencies. For multi-element access this overhead is amortized.

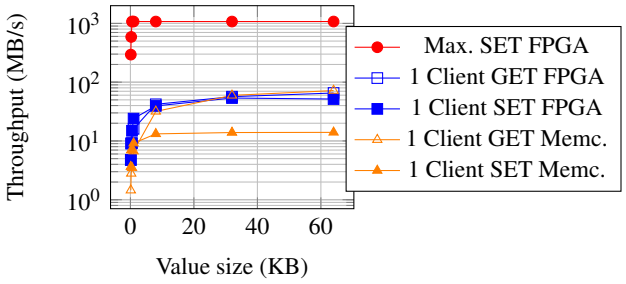

Figure 6: The write throughput of the system reaches line-rate once the value size is >512B. For smaller values it is bound by the hash table logic dealing with deduplication meta data.

Throughput. In Figure 6, we focus on read/write bandwidth as a function of value size ranging from 128B to 64KB (in the Parquet use-case the divided parts of the Parquet file are typically in the 1-8 KB range). The bottom lines in Figure 6 show the read and write bandwidth that a single client experiences when interacting with *Multes++* or Memcached. Not surprisingly, at lower bandwidths, the memcached-backed version can match the FPGA in terms of throughput. The figure also shows that with more clients (128) 10Gbps line-rate is successfully achieved for setting values that are at least 512 B (for smaller values the hash table and memory allocator become a bottleneck). Since *Multes++* does not change the read logic of *Multes* and the latter supports line-rate operation, we omit these measurements for simplicity.

Deduplication. Figure 7 shows storing in *Multes++* the Police dataset in original and then a second time with *a)* half of its rows filtered out, then *b)* with one column removed (without re-encoding the rest of the pages). We use our Parquet-aware storing method, as well as storing key-value pairs determined by the state of the art chunking methods. For this, we use a helper application that computes the running hash of the data in the Parquet file using Rabin fingerprinting [34] to determine variable sized chunks (VCS). Based on a 48B sliding window the tool finds cut points, aiming for 512B and 1KB chunks on average (in practice the average size is around 600B, respectively 1KB).

As Figure 7 shows, our proposal can store the modified file variants in negligible additional storage space. The overhead

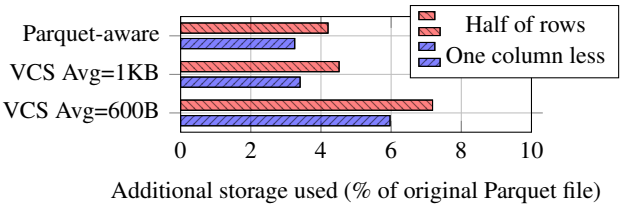

Figure 7: After storing a Parquet file (156 MB), we show the overhead of storing a version with half of the rows (78 MB) and then a column removed (71MB). Our proposed method fairs as good as the state of the art (the overhead includes both data and meta-data).

is mostly dominated by meta-data (the keys and fingerprints describing the new file) and the unique page data only represents a quarter of it. When storing the files using VCS-based chunking, the deduplication results are similar. It is important to point out that when using a chunking configuration that produces values smaller than the FPGA's internal maximum (1KB), there are more key-value pairs to be managed, which in turn increases the size of the meta-data (almost doubling it in our case). The effectiveness of data page deduplication, however, is very similar to the other two variants.

5 Conclusion

Given the increasing availability of programmable hardware in the datacenter, we believe that future distributed storage solutions should aim to utilize them as efficiently as possible. Furthermore, since the size and capabilities of such programmable chips are increasing, it is important to understand what works best for the domain of smart storage – possibly influencing this way the design of future programmable hardware tailored to the datacenter market.

To this end, we make the case that the design of cloud storage systems has to be driven by a service-centric methodology: implement stable multi-tenant functionality in hardware and tenant- and application-specific functionality on the client side in software. This way we can reduce programming complexity in hardware and ensure overall higher flexibility.

Our prototype, *Multes++*, is an FPGA-based key-value store that offers line-rate in-storage deduplication and, thanks to its software library, can store and manipulate Parquet files directly. The service provider could implement further library versions for different columnar formats with minimum effort even though the underlying service remains general purpose.

An important benefit of storing Parquet files in a divided way is that it leads to at least as high deduplication ratios as the state of the art, while it also enables future processing of columnar file formats inside the FPGA-based storage node, without having to dedicate resources to file-type specific parsing logic for each tenant (this could be even impossible due to on-chip resource constraints).

6 Discussion

Even though it has been shown that it is possible to build complex systems entirely using specialized hardware, in production environments the presence of general purpose CPU cores would be welcome. In the case of *Multes*++, these could help with control and configuration of the FPGA-based service. For instance, reconfiguring replication groups after a failure requires branching logic that can't efficiently take advantage of FPGA resources. Therefore, our first question revolves around the choice of hardware for a *Multes*++-like system. Should we use a regular FPGA attached to a regular server machine that can handle management tasks on the side (e.g., similar to Catapult [7]), or rather a stand alone FPGA with a small embedded CPU (e.g., Xilinx Zynq devices[6])? Even though there is a clear benefit in pushing both networking and data management tasks into the FPGA (which suggests a large-FPGA-small-CPU configuration), is there a scenario, in the distributed storage context, where using a large-CPU-small-FPGA configuration would be more useful (e.g., Intel XEON+FPGA [35])?

Second, it is also a question what emerging operations could be offloaded into FPGA-based storage in a multi-tenant setting. For the reasons explained in the paper, choosing processing functionality that benefits only a small subset of tenants and workloads is not economical, so instead we have to devise operators that either benefit a wide cross-section of workloads (e.g. compression) or that are parameterizable at runtime to adjust their behavior. In our earlier work, we explored, for instance, how the same regular expression matching module could be used to filter strings, binary data, help with parsing or be used for decisions of set membership. As a trade-off, the resource requirements of the module were larger than that of a hard-coded regular expression matching unit, but smaller than deploying each of the previously listed modules individually.

Third, this work builds on the assumption that it is the service provider who controls the FPGA and the software library as well. This is motivated by the wide-spread usage of comprehensive frameworks such as Tensorflow [10] and Microsoft ML.NET [11]. In such frameworks it is realistic to assume that the service provider can specialize a part of the infrastructure and code they are executing and is already happening with the Google TPUs and Microsoft's Brainwave accelerators for machine learning. Are there compelling use-cases for application-aware distributed storage outside of the obvious area of machine learning/big data analytics? Or are these the only candidates thanks to their data intensive nature?

[6] https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

Acknowledgments

We would like to thank our shepherd, Tim Harris, as well as Mark Silberstein for their valuable feedback on the work. We also thank Xilinx for their generous donation of software tools used in this project and G. Sutter, S. Lopez-Buedo and their group at UAM Madrid for lending us the FPGA board.

References

[1] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, 2014.

[2] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 476–488.

[3] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 7, 2015.

[4] M. Blott, L. Liu, K. Karras, and K. A. Vissers, "Scaling out to a single-node 80gbps memcached server with 40terabytes of memory." in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, 2015.

[5] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: high-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, 2017, pp. 137–152.

[6] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks *et al.*, "Bluecache: A scalable distributed flash-based key-value store," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 301–312, 2016.

[7] A. M. Caulfield, E. S. Chung, A. Putnam *et al.*, "A cloud-scale acceleration architecture," in *Annual IEEE/ACM International Symposium on Microarchitecture (Micro'16)*, 2016.

[8] Z. István, D. Sidler, and G. Alonso, "Caribou: intelligent distributed storage," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1202–1213, 2017.

[9] K. Vissers, "Versal: The xilinx adaptive compute acceleration platform (acap)," in *Proceedings of the*

2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19). ACM, 2019, pp. 83–83.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016, pp. 265–283.

[11] M. Interlandi, S. Matusevych, S. Amizadeh, S. Zahirazami, and M. Weimer, "Machine learning at microsoft with ml. net," 2018.

[12] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *Annual IEEE/ACM International Symposium on Microarchitecture (Micro'18)*, vol. 38, no. 2, pp. 8–20, 2018.

[13] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of data (Sigmod'15)*. ACM, 2015, pp. 1917–1923.

[14] J. Kestelyn, "Introducing Parquet: Efficient columnar storage for Apache Hadoop," *Cloudera Blog*, vol. 3, 2013.

[15] Z. István, D. Sidler, and G. Alonso, "Runtime parameterizable regular expression operators for databases," in *24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, 2016.

[16] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for fpgas using opencl," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 111–1117.

[17] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018, pp. 107–127.

[18] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA memcached appliance," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 245–254.

[19] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 4, pp. 424–448, 2006.

[20] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig, "Opportunistic use of content addressable storage for distributed file systems." in *USENIX Annual Technical Conference (ATC'03), General Track*, vol. 3, 2003, pp. 127–140.

[21] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 174–187.

[22] Y. Xing, Z. Li, and Y. Dai, "Peerdedupe: Insights into the peer-assisted sampling deduplication," in *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. IEEE, 2010, pp. 1–10.

[23] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful... to deduplication," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, 2015.

[24] D. Cannon, "Data deduplication and tivoli storage manager," *Tivoli Storage, IBM Software Group (September 2007)*, 2009.

[25] J. Buffington, "Why storeonce federated deduplication matters to hp — and should to you, too," http://www.hp.com/hpinfo/newsroom/press_kits/2014/HPDiscover2014/ESG_WP_HP_StoreOnce_Federated_Deduplication_June_2014.pdf.

[26] L. DuBois and R. Amatruda, "Backup and recovery: Accelerating efficiency and driving down it costs using data deduplication," *EMC Corporation*, 2010.

[27] B. K. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory." in *USENIX Annual Technical Conference (ATC'10)*, 2010, pp. 1–16.

[28] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "idedup: latency-aware, inline data deduplication for primary storage." in *USENIX Conference on File and Storage Technologies (FAST'12)*, vol. 12, 2012, pp. 1–14.

[29] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." in *USENIX Conference on File and Storage Technologies (FAST'09)*, vol. 9, 2009, pp. 111–123.

[30] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[31] Z. István, G. Alonso, and A. Singla, "Providing multi-tenant services with FPGAs: Case study on a key-value store," in *Proceedings of the The International Conference on Field-Programmable Logic and Applications (FPL 2018)*, 2018.

[32] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, 2016, pp. 425–438.

[33] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[34] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.

[35] N. Oliver, R. Sharma, S. Chang *et al.*, "A reconfigurable computing system based on a cache-coherent fabric," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig'11)*, 2011.