Providing Multi-tenant Services with FPGAs: Case Study on a Key-Value Store

Zsolt István Gustavo Alonso Ankit Singla
Systems Group, Department of Computer Science, ETH Zürich, Switzerland
{first.lastname}@inf.ethz.ch

***Abstract*—FPGAs can be used to speed up computation and data management tasks in various application domains. In cloud settings, however, high utilization is as important as high performance. In software it is common to co-locate different tenants' workloads on the same servers to increase utilization. Sharing an FPGA is more complex because applications take up physical space on the chip. Even though it is possible to physically partition the FPGA, tenants can have widely different requirements and their needs can also fluctuate over time. In this paper, we take a different approach and provide flexibility to the tenants who are interested in the same type of application but have different workloads and quality of service requirements. We demonstrate our approach of multi-tenant design using a key-value store service but the ideas generalize to other network-facing services as well.**

A key challenge of multi-tenancy is to efficiently share the underlying hardware while enforcing strict data and performance isolation between tenants. In this paper we demonstrate that, by following a single-pipeline design principle, it is possible to control each tenant's share of network bandwidth and computational resources even for complex, distributed operations. Furthermore, we show how state-machine based logic on the FPGA can be made tenant-aware without introducing significant context-switching overhead. Finally, our hardware design provides flexibility for changing per-tenant shares, allowing the same circuit to be used by one or multiple tenants without performance loss.

I. Introduction

Stagnating multi-core CPU performance has led to a greater push for hardware acceleration. FPGAs, for instance, are being used for both data processing and data management tasks at various levels of the datacenter architecture [1]. The adoption of FPGAs in the cloud, however, brings challenges, including keeping the utilization high to ensure economic feasibility.

One way of sharing FPGAs is by time-multiplexing or partitioning the logic resources among users. This has lead to a line of research into "virtualized" FPGAs [2]–[4] that use a software framework, e.g., OpenStack, and a harness on the FPGA to expose several programmable regions. However, this approach incurs significant resource overhead on the chip, especially in case the tenants use the same building blocks. Further, splitting up the FPGA into several regions reduces the flexibility of providing different levels of service or functionality to tenants.

An alternative way of tackling high utilization is to take a service-centric view, exposing the joint implementation of an application (or the service) to a number of tenants, as sketched in Figure 1. Services such as storage and machine learning are widely used, and accelerating them with FPGAs

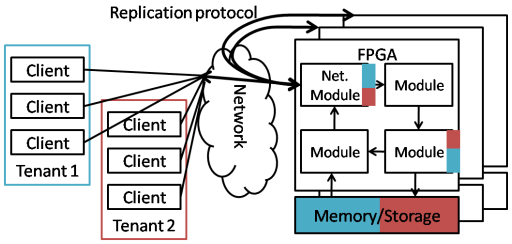

Fig. 1. A multi-tenant design on the FPGA contains only a single instance of the application, composed of several modules, some of which keep per-tenant information. The memory/storage attached to the FPGA is partitioned between tenants. Each tenant can have many clients issuing requests.

is attractive both for clients and cloud providers. Benefiting from this approach requires addressing two key challenges: first, isolation between tenants, both in terms of data and performance, and second, runtime flexibility in dividing the available bandwidth and compute resources among tenants.

In this work we show how the above challenges can be tackled for a distributed key-value store built with FPGA. We demonstrate how to use a single, multi-tenant application to service concurrent tenants while, at the same time, utilizing the FPGA resources efficiently and offering rich functionality. Our prototype system, *Multes*, is based on Caribou [5], an open source replicated key-value store that provides high throughput, low latency access to data. The added multi-tenant functionality does not reduce performance and requires only modest logic resources in exchange for flexible sharing of the FPGA among tenants. The techniques used in *Multes* beyond our example application and could be used in other network-facing services constrained by data movement as well. They can be summarized as follows:

- Re-architecting the application into a single pipeline enables the use of common algorithms from networking (token buckets) to ensure that each tenant uses only their allocated share of the resources. This applies both to client-facing operations (e.g., key lookups) and to distributed operations as well (e.g., the replication protocol).
- For modules that implement event-driven state machines (e.g., protocol controllers), multi-tenancy can be achieved without circuit duplication by externalizing tenant-specific state into registers or on-chip memory.
- Modules expose parameters that are not hard-coded and can be modified at runtime, offering flexibility in the number of active tenants and replication policies.

978-1-5386-8517-4/18/$31.00 ©2018 IEEE
DOI 10.1109/FPL.2018.00029

II. Background

A. FPGAs in the Cloud and Datacenter

There are several options for deploying FPGAs in the datacenter and the cloud. One is to use the FPGA as a PCIe-attached accelerator, such as the ones offered in Amazon F1 instances. This deployment option is well suited for implementing traditional accelerators, but does not allow the FPGAs to access the network directly. Intel's Xeon+FPGA platform [6] and IBM's CAPI solutions are similar to the accelerator model but, because they offer cache-coherent access to the CPU's memory, they allow much finer-grained interaction between hardware and software.

An other option for deploying FPGAs, and the one we focus on in this work, is to expose them directly over the network, either as stand-alone nodes [4], or coupled with a host machine. The latter option is already in widespread use in Microsoft Azure for offloading virtual network tasks: in the Catapult project [7] FPGAs act as a bump-on-the-wire for the server machine's networking interface while also being accessible over PCIe. Even today, the leftover bandwidth and logic area of the FPGAs can be used to provide additional services (for instance AI acceleration [8] or data caching [9]).

B. Key-value Stores and Caribou

Almost all distributed data processing applications require either a storage or a caching layer. As a result, key-value stores (KVSs) such as memcached and Redis and object stores such as Amazon S3, are widely used in the cloud. Most KVSs are built around a random access data structure that holds keys and pointers to values. These values can reside on disk or in main memory, and can be of various sizes. While different KVSs may offer different features, they all need to support read and write (get and set) operations to manipulate key-value pairs. Although there is already much work on using FPGAs to accelerate these applications, ranging from partial offloading [10], to standalone solutions [5], [9], these works do not address multi-tenancy as part of the circuit design.

We build *Multes* by extending Caribou [5], an open source system that offers the functionality needed for stand-alone deployments: reliable networking to many clients using TCP/IP [11], a flexible hash table with memory allocator, and replication among Caribou nodes for fault tolerance [12]. Fault tolerance is a requirement in the cloud, to ensure that even in the face of node failures or network partitions, no user data is lost. Caribou implements a leader-based replication scheme that uses an atomic broadcast protocol [13] to ensure that all participating nodes have the same state. This operation is latency-sensitive as it requires multiple network round-trips.

The interface to Caribou consists of operations to read and write (get and set) a value corresponding to a key, to read the value and apply a filtering operation, or to retrieve all data in the storage (scan) and apply a filtering operation on it. Its implementation is optimized for smaller value accesses (32-512 B), thus the multi-tenant variant has to be able to switch between tenants with high frequency to achieve line-rate performance.

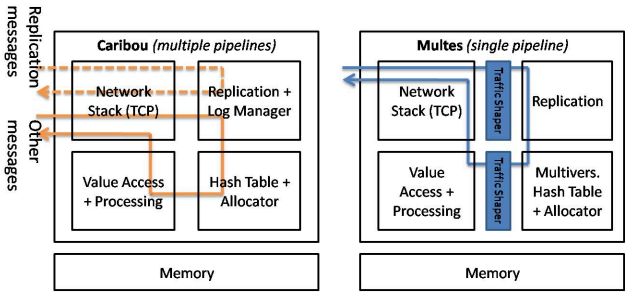

Fig. 2. In the original Caribou design, depending on the operation type, messages might be sent directly from the replication module to the networking stack, making traffic shaping more difficult. In *Multes* all requests follow the same internal path and traffic shaping techniques can be applied consistently.

III. System Overview

Multes provides functionality similar to that of Caribou but in a multi-tenant design. The FPGA is configured with a bitstream that has been provisioned for a maximum number of tenants (design-time parameter) and at runtime the behavior of the key-value store can be adjusted without reprogramming. Tenants can be added or removed, their allocations can be modified, their dataset can be reset and their replication groups can be configured independently.

We modified and extended Caribou in three ways: 1) modified the hash table to store keys belonging to different tenants in different parts of the memory, 2) modified the replication module to support different replication groups for each tenant by switching the state of the algorithm on a per-request basis, 3) added traffic shapers to ensure that no tenant uses more bandwidth or has more outstanding requests than it has been allotted. For space reasons, in this paper we focus on network bandwidth allocation and sharing, but is possible to apply similar policies on DRAM or flash memory bandwidth as well inside the KVS.

IV. Designing for Multi-tenancy

A. Applications as a Single Pipeline

In a multi-tenant setting our goal is to ensure that the application on the FPGA and the networking/memory resources are shared fairly among requests belonging to different tenants. In order to achieve this, we need to limit the rate at which requests of each tenant can enter the application, the output network bandwidth they generate, and the number of outstanding requests they each have inside the application. Caribou's original design, however, makes it difficult to reason about these, mainly because its internal modules form multiple pipelines where messages can pass through (Figure 2). The replication module is connected directly to the networking block, and can send and receive replication-related messages bypassing other modules. Furthermore, a single replication message can result in sending multiple network packets, possibly carrying large payloads. In *Multes*, we reorganized the key-value store logic into a single pipeline that handles both

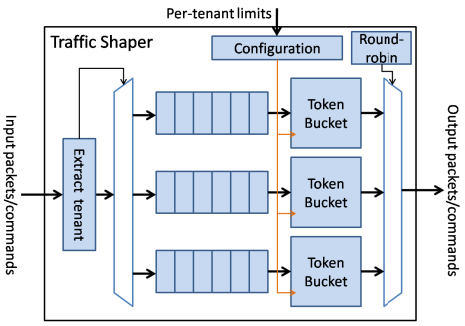

Fig. 3. Each traffic shaping module is composed up of FIFO buffers and Token Buckets. Each tenant's share can be configured independently, and data is dequeued with a round-robin discipline.

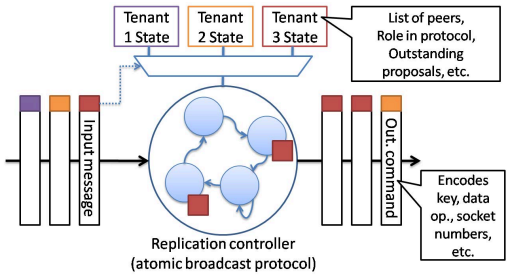

Fig. 4. The original atomic broadcast state machine has been augmented to multiplex between the state of different tenants on a per event/message basis. The switching happens without additional overhead in terms of latency.

replicated and non-replicated operations. This reorganization of logic blocks enables reasoning about the amount of data entering and leaving each pipeline stage for each tenant.

B. Traffic Shaping with Token Buckets

We added two traffic shapers to the pipeline that ensure that both the incoming traffic (and queued requests) and outgoing traffic respect each tenant's allocated share. The first traffic shaper is after the TCP module's output, accepting and queuing messages received over the network. The second traffic shaper is positioned before the value management unit. Most operations that send data out over the network arrive at the value manager as small command words, pointing to data to be retrieved from memory. Because on FPGAs the on-chip buffer space is much smaller than the off-chip memory, it is more efficient to queue these command words and not the full network packets after the value manager.

Traffic shaping is achieved by a module built from multiple token buckets (Figure 3). Each token bucket belongs to a tenant and is composed of a queue and logic to decide whether letting a request through would violate the tenant's allocated share. This decision is taken based on the availability of tokens. Each token corresponds to an 8 byte transfer over the network, and they are refilled periodically. Tokens are accumulated only up to a limit that determines the largest burst a tenant can send out over the network. The refill rate and the limit can be changed on a control interface at run-time.

Each request (command word) encodes how much data it will send over the network (either by already having data associated with it, or by containing a pointer to the value area). This information is used by the token bucket and is provided by *(a)* the networking module (for incoming packets), *(b)* the hash table (for get operations), or *(c)* the replication control state machine (for proposal and write operations). Since there are separate queues per tenant, reordering can happen across tenants, but each tenant will see the same behavior as without the token buckets. Requests are dequeued only if there are enough tokens available. Requests are multiplexed onto the output in a round-robin manner. In case a tenant's queue fills up, requests are dropped until there is enough space in the queue. This can only happen in the first traffic shaper module, because the FIFOs of the second module are sized as to hold the maximum number of requests per tenant allowed to enter the pipeline.

Our current prototype uses a special operation in the key-value store to encode the new allocation for a specific tenant. The two traffic shapers can also be configured independently. This makes it possible to switch the key-value store from multi-tenant to single tenant mode, or to provide a "premium" service to a specific tenant. Furthermore, this could be combined with datacenter-wide load balancing or traffic management decisions, similar to the approach in [14].

C. Running Multiple Replication Groups

In order to make the key-value store multi-tenant without losing its reliability guarantees, in *Multes*, the replication module had to be modified to handle different tenants at the same time because the original design only allows participating in a single replication group.

The replication module is based on our earlier work on atomic broadcast [12] and its behavior is driven by a large state machine. We extended the control state machine with tenant information keeping its core algorithms unchanged. This is feasible because the controller of the atomic broadcast module is event-driven, enabling the use on-chip memory and registers to implement no-overhead "context switching" between tenants as shown in Figure 4.

An additional reason why the original design is ill-suited for sharing is that it relies on a data structure to store pending proposals that is separate from the hash table and value area. Originally this data structure is append-only and does not require memory allocation. With multiple tenants either memory allocation has to be implemented for this data structure (to avoid memory waste) or their proposals have to be interleaved. The latter option is undesirable because the behavior in one group could negatively impact other groups. In *Multes* we move most of the functionality of this data structure into the hash table by implementing multi-versioned entries: Each key in the hash table has a "current" and a "next" pointer to the value area. This allows staging pending changes and, upon a commit, making them visible and freeing the memory location of the previous value. The data structure

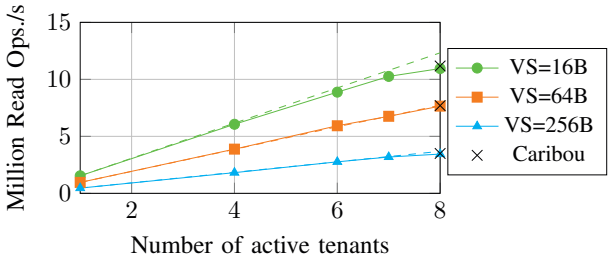

Fig. 5. Supporting multiple tenants does not introduce significant overhead in throughput, regardless of the value size (VS). The throughput with 8 tenants is close to Caribou and the theoretical maximum over 10Gbps (dashed lines).

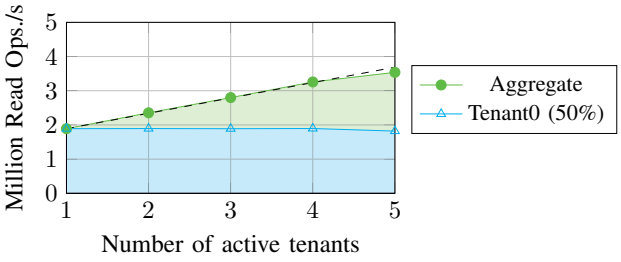

Fig. 6. Even with unequal tenant shares, no bandwidth is wasted (Shares: T0=50%, others=12.5% each, Value: 256B, Dashed line: theoretical max.)

that keeps track of proposals in the replication block becomes much simpler (as it only needs to store fixed sizes entries the modified key), fits in small BRAM blocks, and can be looked up in constant time.

Even though this redesign of the way proposed values are stored will result in keeping only the most recent version, this is compatible with the non-versioned nature of the service being exposed to the clients. When a newly joined node needs to be synchronized, the state of they key-value store can be copied over by retrieving all keys of the particular tenant and inserting them on the other node.

D. Network Traffic and Data Isolation

Tenants are separated in the TCP/IP module by opening several ports to listen on. Requests are tagged with the port number they are received on. Authentication and access control is orthogonal to this work and network-level rules can be used to, for instance, ensure that only clients belonging to a specific tenant can access the FPGA nodes over a specific TCP port. The design does not require changes to the client protocol and tenants can use different port numbers on different nodes.

We achieve data separation by modifying the hash table module to partition the memory area used for keys among tenants. In our design, each tenant receives a partition of the hash table with an equal number of slots for keys. Even though separation could be also achieved by appending the tenant ID to the key, mixing different tenant's keys in the same data structure would make it less efficient to copy or reset data on a per-tenant basis. Furthermore, hash collisions could impact performance negatively.

To ensure that the traffic generated by the replication blocks is also subject to the tenant's limits, instead of using a single replication-specific TCP/IP port and adding meta-data to requests to represent which tenant they belong to, the replication module sets up connections for each replication group using the port assigned to the particular tenant. As a result, any replication-related messages that are sent between FPGAs are also managed by the traffic shapers.

V. Evaluation

We evaluated *Multes* using Xilinx VC709 boards connected to a 10Gbps switch. Each board has a Xilinx XC7VX690T-2 FPGA and 8 GBs of DDR3 memory. Software clients are written in Go (based on the clients of Caribou) and run on Debian Linux on up to 9 machines with dual-socket Xeon E5-2630 v3 CPUs processors.

A. Performance Isolation

One of the biggest challenges in providing a multi-tenant service is ensuring that the network bandwidth is efficiently utilized even if tenants issue concurrent commands. We measure the throughput of the system with increasing number of tenants and demonstrate that the token bucket mechanism does not introduce significant overhead. In Figure 5 we show the behavior for an eight tenant setup with a read-only workload where each tenant (T0-T7) is allowed to use 12.5% of the bandwidth. As expected, throughput increases linearly following the ideal trend-line. There is an exception for the smallest value size (16B), but the overhead doesn't come from the token buckets. The source of the overhead is in Caribou's original design and, as seen in Figure 5, the points measured for Caribou coincide with *Multes*.

Even if tenants are allocated different fractions of the bandwidth, the behavior follows the same trend. In Figure 6 we show throughput for a read-only workload, where the first tenant (T0) has access to 50% of the output bandwidth, and the others only 12.5% each. As we increase the number of tenants the used bandwidth of the node increases and with five tenants we reach the same throughput as in Figure 5.

In addition to isolating tenants in terms of throughput, the design of *Multes* is successful in reducing interaction between response times as well. The reason for this is twofold. On the one hand, the first traffic shaper limits the number of outstanding requests per tenant to ensure that an incoming request does not need to queue behind too many operations in the hash table.

On the other hand, the second traffic shaper ensures that tenants can send their share of responses. To demonstrate response time isolation, we executed an experiment in which T0's clients are sending replicated write requests to *Multes*, while the other tenants are reading at their maximum rate. We chose replicated writes because they are more sensitive to latency fluctuations than simple read requests. We measured latencies on the leader node to remove artifacts of the client's software stack. As Figure 7 shows, even if all other tenants

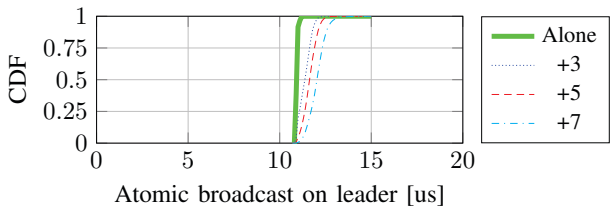

Fig. 7. The time for a atomic broadcast round increases only marginally when Tenant0's leader node is also handling read requests for up to seven other tenants, at their maximum allowed bandwidth.

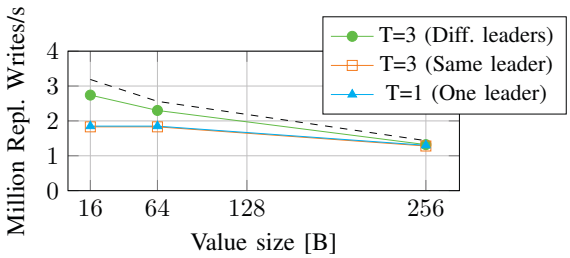

Fig. 8. *Multes* is limited in throughput for single-tenant (single leader) setups by the replication logic. If the leaders reside on different nodes, throughput close to the theoretical maximum (dashed line) can be achieved.

fully utilize their share on the leader node (reading at an aggregated 8.75Gbps), the atomic broadcast times stay within microseconds of the isolated case.

B. Multiple Replication Groups

We measured the maximum atomic broadcast rate (replicated writes) achievable by *Multes*, to ensure that it delivers similar performance to the original implementation in Caribou. We first measure performance using a single tenant (T=1) with 100% bandwidth allocation that issues replicated write requests of increasing sizes for a group of three nodes (one node is the leader, the others followers). As shown in Figure 8, the atomic broadcast rate reaches almost 1.9M operations/s for small values (16 B and 64 B), which is within 80% of the numbers measured for Caribou for a similar key-value size combination. This difference is expected because our modified design performs more memory allocations/deallocations. For larger values, *Multes* is able to saturate the full network bandwidth and achieves equal throughput to Caribou.

We repeat the experiment with three tenants (T=3), each of them with access to 33% of the bandwidth on all nodes, but using the same node as a leader for all three. The throughput is bottlenecked by the leader, and is identical to the single-tenant case. If we configure the replication groups so that their leaders reside on different nodes (each node is the leader for one group, and follower for two others), the performance is bottlenecked by the network allocations, as the replication control units are fast enough to keep up with the load. This demonstrates that the same nodes can be used in multiple roles without reducing performance.

TABLE I
RESOURCE CONSUMPTION OF EACH MODULE WITH 8 AND 16 TENANT CONFIGURATIONS, COMPARED TO THE NON-MULTI-TENANT DESIGN.

Module	**Logic slices**	Overhead	**BRAMs**	Overhead
DRAM controller	25k (23%)	0	128 (9%)	0
Networking	11k (10%)	0	151 (10%)	0
Memory allocation	2.9k (2.7%)	0	8 (<1%)	0
Hashtable	1.8k (1.7%)	-2.1k	150 (10%)	+31
Replication (T=8)	4.3k (4%)	+1.1k	18 (1.2%)	-47
Traffic Shapers (T=8)	3k (2.8%)	n/a	60 (4.1%)	n/a
Replication (T=16)	5.8k (5.4%)	+2.6k	18 (1.2%)	-47
Traffic Shapers (T=16)	5.7k (5.3%)	n/a	112 (7.6%)	n/a

C. Resource Consumption

The overhead of adding token buckets to the design and making the replication module tenant-aware is small and increases linearly with the supported number of tenants. In Table I we break down how the different modules changed in size compared to Caribou. The size of the hash table in *Multes* is smaller than in Caribou because we disabled the scan functionality. Currently, without processing, the entire design consumes 48k (53k) logic slices and 515 (610) BRAMs for 8 (16) tenants.

A single token bucket inside a traffic shaper consumes around 170 logic slices (less than 0.16% of the evaluation device). The state kept inside the replication module takes up in the order of 150 slices (less than 0.15%) per tenant. These latter data structures are mapped to distributed RAM in our current prototype to allow single cycle read-update operations, but in the future, if more space for computation is needed, they could be organized into BRAMs without reducing the performance of the replication control machine significantly.

VI. Related Work

There is increasing interest in providing multiple users with concurrent access to the same set of FPGAs in a cloud setting, with most of the related work focusing on "virtualizing" FPGAs [2]–[4]. Byma et al. [2], for instance, demonstrated that it is possible to offer FPGAs as a generic cloud resource and configure them remotely using OpenStack, with the simplicity of booting up a regular virtual machine.

The work of Fahmy et al. [3] exposes four programmable regions on a PCIe-attached FPGA. Their proposal is evaluated with various data analytics applications, but the overhead of virtualization reduces the energy efficiency (performance/W) of the chosen FPGA five-fold when compared to a non-virtualized baseline.

Centaur [15] also exposes several programmable regions of the FPGA on the Intel Xeon+FPGA platform. It is designed with domain-specific use in mind, namely database acceleration. As a result, its on-chip management overhead is small and it exposes the programmable regions through a simple "hardware thread" interface.

Sharing an FPGA accelerator between multiple software threads is explored in the recent work by Chen et al. [16], that exposes acceleration as a service to Apache Spark. In

that work, however, all threads belong conceptually to a single tenant and load isolation is not required.

Overall, virtualization of FPGAs allows for arbitrary applications per tenant, but chip space is not guaranteed to be used efficiently. Instead, this work explores an approach where tenants share a common application with rich functionality that has been designed to with multi-tenancy in mind, using chip space more efficiently.

Software systems have long explored the challenges of providing multi-tenancy, but usually their quality of service guarantees are coarser than what an FPGA can offer. Multi-tenancy has also been explored in the context of distributed storage, e.g., in Pisces [14] and Moirai [17]. Typically, these systems combine multiple techniques to isolate tenants from each other and ensure weighted sharing including both node-level load balancing techniques and the use of a higher level controller that monitors the distributed state and re-balances load if necessary. Using such a controller would also be beneficial for *Multes* to orchestrate the nodes and manage the tenant-specific bandwidth allocations.

Memshare [18] focuses on an other challenge: sharing memory in key-value stores used for caching. Its internal data structures ensure that each tenant can be guaranteed a certain amount of memory space, and uses the leftover space to increase combined cache hit rates for all tenants. This highlights that beyond isolating tenants from each other, there are also opportunities in using left-over resources to benefit all tenants. *Multes* uses a static allocation of memory space for keys, and could benefit from a more elastic approach.

VII. Future Exploration

A. Application-aware TCP Stack

We designed the application pipeline such that it enforces each tenant's fair share both when receiving and sending out data. The TCP stack, however, is external to our design and, in case packets are dropped or incast congestion happens, it could end up sending more messages for particular tenants, violating the pre-determined bandwidth limits. One solution to this limitation is to integrate the TCP/IP more tightly with the application on the FPGA, capturing the retransmission events of the network stack in our traffic shaper and resending data from memory location accessible from the application.

B. Partial Reconfiguration

Multes solves two large challenges of multi-tenancy, namely data separation and performance isolation but, in the current design, tenants have to share the same set of pre-defined filtering units on the FPGA. There is an opportunity for tenant-specific processing using partial reconfiguration where tenants could offload to the FPGA processing snippets of their choice. These snippets only need to implement a simple streaming interface and therefore can be expressed in high level languages, and their behavior can be verified easier than general purpose FPGA logic. This approach would benefit from the advantages of both a "virtualized" FPGA and a multi-tenant application.

VIII. Conclusion

We explored how flexible multi-tenant services can be provided using a single application pipeline on the FPGA. Using a key-value store as an example, we have shown that it is possible to apply simple traffic shaping techniques from networking to a complex distributed application. The solution is flexible: tenant limits can be changed at runtime without requiring reprogramming, and the nodes can be configured with different replication roles by different tenants, making replication groups independent across tenants.

Multes provides performance very similar to that of Caribou, the system it extends and modifies, reaching more than 10 million KVS operations per second. Even when replicating writes to three nodes from a single leader, it reacher almost 2 million ops/s. In contrast to Caribou, in this work we show that when different tenants use different nodes as leaders for replication, close to line-rate throughput can be achieved. Providing multi-tenant behavior with 16 tenants requires less than 10% of additional chip space when compared to Caribou.

Acknowledgments

Part of this work is funded by Microsoft through the MR-ETHZ-EPFL Joint Research Center. We would like to thank Xilinx for the generous donation of the FPGA boards used in the paper.

References

[1] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *FPL'16*. IEEE, 2016.
[2] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *FCCM'14*. IEEE, 2014.
[3] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *Cloud Computing Technology and Science (CloudCom'15)*. IEEE, 2015.
[4] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *UIC-ATC-ScalCom*. IEEE, 2015.
[5] Z. István, D. Sidler, and G. Alonso, "Caribou: intelligent distributed storage," *Proc. of the VLDB Endowment*, vol. 10, no. 11, 2017.
[6] P. Gupta, "Accelerating datacenter workloads," in *FPL'16*, 2016.
[7] A. M. Caulfield, E. S. Chung, *et al.*, "A cloud-scale acceleration architecture," in *MICRO'16*. IEEE, 2016.
[8] D. Lo *et al.*, "Accelerating persistent neural networks at datacenter scale," in *ML Systems Workshop at NIPS'17*, 2017.
[9] B. Li, Z. Ruan, *et al.*, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.
[10] K. Lim, D. Meisner, *et al.*, "Thin servers with smart pipes: designing soc accelerators for memcached," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013.
[11] D. Sidler, G. Alonso, *et al.*, "Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware," in *FCCM'15*. IEEE, 2015.
[12] Z. István, D. Sidler, *et al.*, "Consensus in a box: Inexpensive coordination in hardware." in *NSDI*, 2016.
[13] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *DSN'11*.
[14] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage." in *OSDI'12*, 2012.
[15] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid CPU-FPGA databases," in *FCCM'17*. IEEE, 2017.
[16] Y.-T. Chen, J. Cong, and other, "When Apache Spark meets FPGAs: a case study for next-generation DNA sequencing acceleration," in *USENIX HotCloud'16*, 2016.
[17] I. Stefanovici, E. Thereska, *et al.*, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015.
[18] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: a dynamic multi-tenant key-value cache," in *Usenix ATC*, 2017.