LaKe: The Power of In-Network Computing

Yuta Tokusashi
Keio University
tokusasi@arc.ics.keio.ac.jp

Hiroki Matsutani
Keio University
matutani@arc.ics.keio.ac.jp

Noa Zilberman
University of Cambridge
noa.zilberman@cl.cam.ac.uk

***Abstract*—In-network computing accelerates applications natively running on the host by executing them within network devices. While in-network computing offers significant performance improvements, its limitations and design trade-offs have not been explored. To usefully and efficiently run applications within the network, we first need to understand the implications of their design. In this work we introduce LaKe, a Layered Key-Value Store design, running as an in-network application. LaKe is a scalable design, enabling the exploration of design decisions and their effect on throughput, latency and power efficiency. LaKe achieves full line rate throughput, while maintaining a latency of 1.1μs and better power efficiency than existing hardware based memcached designs.**

***Index Terms*—Energy Efficiency, Key-value store, FPGA, In-network computing**

I. Introduction

In-network computing is an emerging area in computing, where applications natively running on the host are accelerated by running them on network devices. While hardware acceleration is typically done on stand-alone programmable platforms [1], in-network computing executes the applications on programmable network devices, such as network interface cards (NICs) or switches [2], [3]. These network devices provide both the networking functionality and the execution of an application at the same time [4].

In-network computing has been shown to provide throughput and latency improvement of orders of magnitude [2], [4]. Furthermore, the use cases are far from being limited to networking functions; examples include consensus [5], data processing [6], machine learning [7] and more. The most popular use case of in-network computing for cache-based applications (e.g., [2]). The placement of the in-network computing device within the network saves traversals of the network by-design [4], and is ideal for handling frequently repeated requests for information. In this work we focus on one class of caching applications, the caching of key-value store (KVS), to study design trade-offs in in-network computing.

Online services such as e-commerce and social networks, mostly running in the cloud [8], are commonly using KVS. KVS deployments in datacenters are often scaled-out in order to increase performance [9], which leads in turn to an increased power consumption. One of the limitations of KVS is that it is very sensitive to latency, in the order of tens of microseconds, end-to-end [10]. Using in-network computing has the potential to significantly improve the performance of KVS-based applications.

While in-network computing has attracted a lot of attention over the last few years, most of the work has focused on ASIC-driven implementations [2]–[4], [6]. The design trade-offs in building in-network computing platforms, and in particular those implemented using FPGAs, have to the best of our knowledge, not been explored.

In this paper, we present LaKe: a Layered Key-value store application, implemented using an FPGA and used for in-network computing. LaKe is energy efficient, and provides low latency, accelerated KVS. Operating either as a native NIC or a switch, LaKe provides in-network computing functionality both at the edge and within the network. More importantly LaKe is a highly modular design, using multiple cache layers, combined with multiple processing cores, to achieve high throughput. Unlike other specialized designs [11], LaKe's building blocks conform with memcached and do not require a specialized application.

LaKe explores trade-offs in design and performance by building upon its modularity and leveraging multiple types of on-chip and on-board memories: on-board DRAM as a large data store, on-board SRAM for slab allocation and on-die memory for caching and concealing latency of the external memory devices. LaKe further uses a multi-processor architecture to explore scalability and latency trade-offs. LaKe is implemented on NetFPGA-SUME [12], and detailed throughput, latency and power consumption evaluations are provided, as well as a comparison to the state-of-the-art in KVS acceleration.

In this paper we make the following contributions:

- We introduce LaKe: an in-network computing design, providing KVS acceleration in parallel to networking functionality.
- We describe the modular architecture of LaKe, using multiple processing cores, several layers of cache and hardware/software co-design.
- We provide a detailed evaluation of LaKe, implemented on NetFPGA-SUME, and show that it can reach full line rate, while providing 1.1 μs latency and $\times$5.1 better power efficiency than an existing hardware based memcached system.
- We explore in-network computing design trade-offs, show the impact on performance and power efficiency of memory, caching and processing cores.

The rest of this paper is organized as follows. We first describe LaKe's architecture in Section II. Section III describes the integration of LaKe on NetFPGA-SUME platform. We

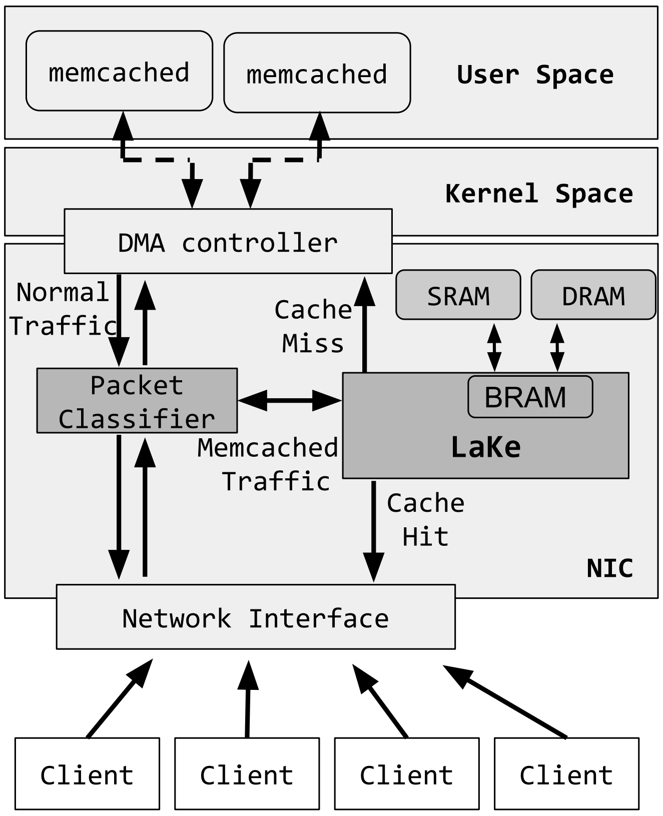

Fig. 1: The high level architecture of LaKe.

evaluate LaKe in Section IV. Section V explores design trade-offs and Section VI discusses related work. We conclude in Section VII.

II. Architecture

LaKe is an in-network computing **La**yered **Ke**y-value store architecture, focused on memcached. LaKe is FPGA based, and provides both network-switching functionality and KVS-acceleration. It achieves significant performance improvement by using multiple layers of cache. Each cache layer provides a trade-off between performance (latency, throughput) and memory size. The performance achieved by LaKe reduces by an order of magnitude the number of servers required in the data center. In this section we explore its architecture, as shown in Figure 1.

A. Background: Key-value Store

KVS is traditionally used for web cache and as backend storage for web applications, using pairs of key-and-value, stored in the host's main memory or storage (e.g., SSD or HDD). KVS provides simple APIs and scalability, using consistent hashing, in comparison with relational database management system (RDBMS). The APIs consist of the primitive GET(key), SET(key,value) and DELETE(key), issuing a read request, a write request and a delete (writing zero) request, respectively. Upon a new query, the KVS calculate a hash value of the key, which allows it to retrieve a descriptor to a respective key-value data entry. Generally, a hash function (e.g., lookup3 [13], [14], md5, cityhash [15]) is used for the hash calculation. Once the KVS finds an entry in the table that matches the requested key, the value paired with the key is returned to the client.

B. High Level Architecture

The LaKe architecture combines a hardware component and a software component. The software component is the memcached host software, modified to support UDP binary protocol. The hardware component, which is the focus of this paper, is a combined design of a networking device and a memcached accelerator running on a single platform.

The architecture of LaKe is shown in Figure 1. While LaKe can operate either as a switch or a network interface card (NIC), let us assume for clarity that it is used as a NIC. Traffic arrives to LaKe from multiple sources. A packet classifier is used to distinguish between memcached queries and any other types of traffic; general traffic will be sent to the host, as in a standard NIC, while memcached queries will be sent to the LaKe module. Queries that are a miss in LaKe's cache and memory, are sent to the host.

We implement LaKe on the NetFPGA-SUME platform [12]. The data plane is based on the NetFPGA Reference Switch project, which can also operate as a NIC, and we amend it with logic enabling memcached operation, as shown in Figure 2. Modules unique to LaKe are marked in dark gray. Incoming traffic from multiple ports is fed into the data plane using an arbitration module (Input Arbiter). A packet classifier, unique to our design, identifies the type of the packet, and sends memcached packets to the LaKe module, described later in this section. Non memcached traffic continues in the pipeline, where it is merged (using a second arbiter) with packets returning from the memcached module: both reply packets, going back to clients, and missed queries, forwarded to the host. The destination of the packet is set in an output port lookup module, and packets wait in an Output Queues module to their turn to be transmitted.

C. LaKe Module

To overcome performance bottlenecks and enable scalability across different platforms, LaKe adopts a multi-core processor approach for query processing. The architecture of the LaKe module is shown in Figure 3.

Incoming queries are spread between a set of processing elements (PEs), using a multiplexing and demultiplexing PE-network. Each PE receives and processes queries. Once a query is processed, the PE accesses a shared memory network (using a second AXIS interconnect core). Three types of memories are connected to the memory network: DRAM, containing the hash table bucket and data store chunks (Section II-D, Section II-E), SRAM, containing chunk information (Section II-E), and CAM, serving as a look up table (LUT) for retrieving key-value pairs (Section II-F).

Figure 4 illustrates the request-response process of a query in LaKe. As a new query arrives, the PE parses the packet and extracts the command, key and value. Next, the hash of the extracted key is calculated. In our implementation, CRC32 is

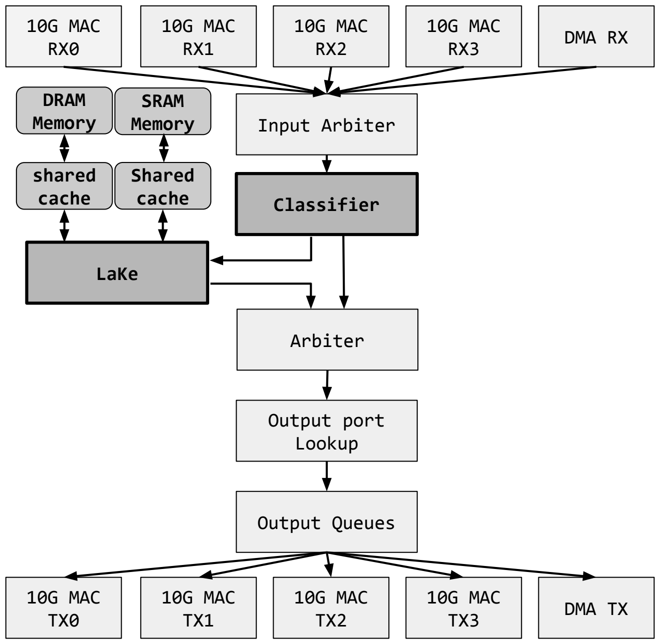

Fig. 2: The block design of LaKe integrated with NetFPGA Reference Switch.

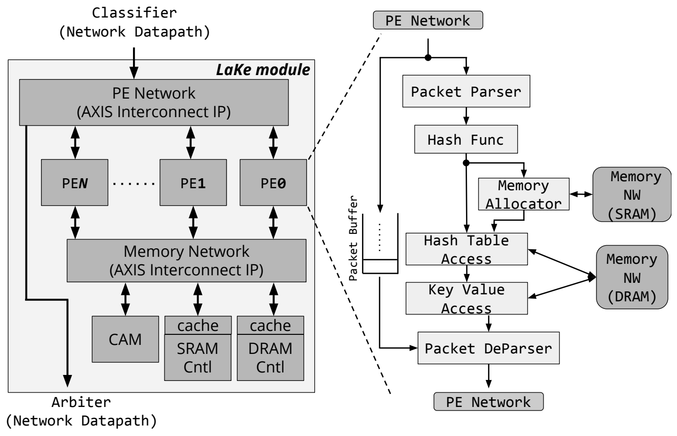

Fig. 3: LaKe module architecture. The architecture of each PE is shown on the right.

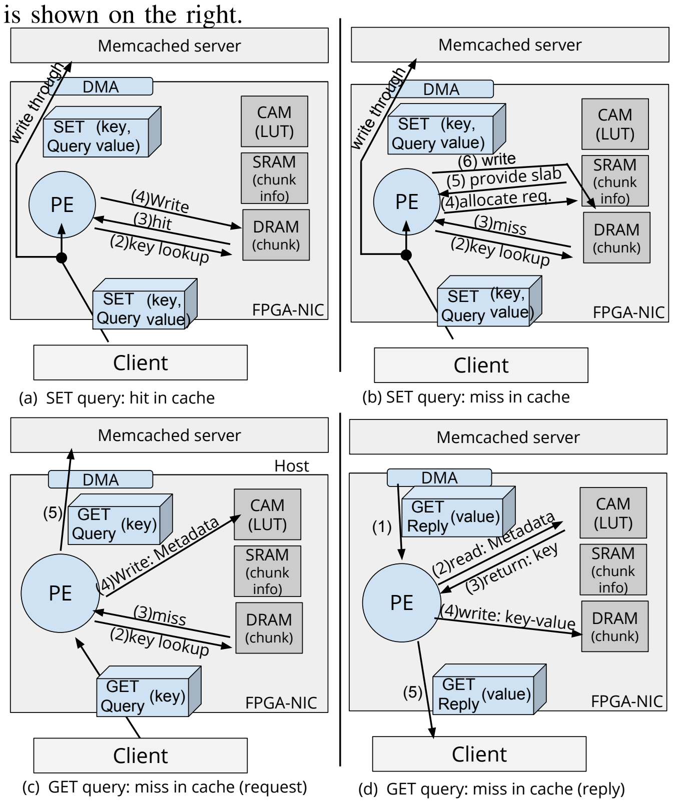

Fig. 4: The request-response process of a query in LaKe.

used as the hash function. The hash value serves as a pointer to an address in the DRAM, holding a descriptor (hash table bucket) pointing to the key-value pair in the memory. If a key exists in LaKe's memory, it is considered a *hit*, otherwise it is a *miss*. Upon a SET command that is a hit (Figure 4(a)), both the hash table and the key-value pair data are updated in the DRAM. If a SET command arrives with a new key (miss, Figure 4(b)), the PE assigns it to a chunk using a list of free descriptors stored in the SRAM and pointing to empty chunks. As a write through policy is used, any SET query is also sent to the host's memcached server.

For a GET query that is a hit, a reply is prepared in the *Packet Deparser* and returned to the client. Otherwise, the request is forwarded to the host memcached server through the switch datapath (Figure 4(c)) and using a DMA engine [12]. The host then sends a reply to LaKe (Figure 4(d)), which updates the key and value in the cache and DRAM before sending the reply to the user.

D. Hash Table

The hash table is used to store descriptors pointing from a hashed key to the address in memory of the actual key-value pair. As such, it is a critical component in the design. The data structure of the descriptors in the hash table is shown in Figure 5. The descriptor size is 64bit, which is performance optimized: the DDR3 SoDIMM on the board uses a bus width of 64bit and a burst size of eight, which leads in turn to a bus width from the DDR3 controller of 512bit. This allows in a single access to read eight descriptor entries, enabling 8-associativity. To reduce the number of accesses to the DRAM, a key's length is compared to the key's length in the descriptor, and only if they match the PE attempts to access the DRAM and read the key-value chunk.

E. Memory Management

Memcached builds upon a slab allocator to efficiently use the memory [9], [16]. This approach is also taken in hardware based designs, as well as in LaKe, enabling to handle variable key- and value-length.

A slab allocator is implemented using an SRAM-based memory, storing addresses of unused chunks. To reduce access time to the SRAM, LaKe uses a small cache (implemented as a FIFO), which pre-loads the next available addresses from the memory. The number of entries in the SRAM can be calculated using the following formula: $\sum_{k=i}^{n} S_k N_k \leq C_{mem}$, where S_k, N_k and C_{mem} denotes the size of chunk, the

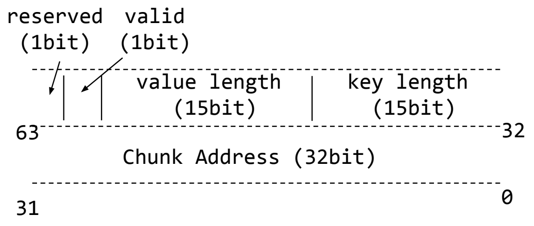

Fig. 5: Hash table format. Fixed size entries are used.

number of chunks and SRAM capacity, respectively. We use multiplications of 64B as slab size, and support 64B, 128B, 256B and 512B chunk sizes in our prototype. The minimum size slab is determined by the width of the memory network datapath: 512bit.

Shared cache To conceal DRAM access latency, LaKe uses a shared cache for each data context: hash table and data store. These caches are located in front of the DRAM controller, rather than inside a PE, in order to enable PE scalability and avoid holding data context (such as CPU process) inside a PE. In this manner, frequent key and value are immediately returned from the caches. Each cache has in our prototype 1024 entries, implemented using a BRAM. We employ write through as the update policy, thus cache coherency is maintained for both cache and DRAM.

DRAM access: Random access to the DRAM has a non negligible and variable latency, which can stall PEs. To attend to this latency we integrate a small cache before the DRAM (e.g., 64B cache line, write through, direct-map, total capacity 64kB). Without the cache, we measure the DRAM controller's access latency (using Xilinx MIG, running at 933.33MHz) to be around 115ns in a zero load test, and to be up to 650ns under high load.

PE scalability: LaKe applies a modular, scalable approach to KVS acceleration. The number of PEs supported by the design starts at one and scales up, with five PEs sufficient to support per-port full line rate. Beyond physically implementing a variable number of PEs, LaKe also allows to control on-the-fly the number of PEs used, balancing workload and power efficiency.

F. Memcache Protocol

Memcached systems [9] generally use the memcache protocol. There are two memcache protocol variants: ASCII based and binary based. Our implementation uses the binary variant. The challenge using the memcache protocol is that key-value pairs cannot be identified in responses from the host. For instance, a GET request missed in the hardware and sent to the host will have a query response returning with the value but without the key. Thus, cache systems cannot handle only response packets; it is required to learn and save a request query's information.

To associate a key with a returned value, we use memcache protocol's opaque field and source UDP port number. Memcache protocol uses a 32-bit opaque field, and memcached systems use the same opaque value in both request and reply. We use a lookup module to match returned values from a host with their paired keys. The LUT is implemented using a CAM, where we query using the opaque value and the source UDP port, and the reply is the original query's key. The keys are updated every time a GET query is a miss in the hardware and forwarded to the host.

III. Implementation

Our target board is NetFPGA-SUME [12], which is equipped with Xilinx Virtex-7 690T FPGA, 8GB DDR3 SDRAM modules (4GB$\times$2, upgradable to 16GB$\times$2), three QDRII SDRAM modules (27MB) and more. The NetFPGA Reference Switch project is the baseline datapath, integrated with the memcached subsystem, as shown in Figure 2. The project is implemented in Verilog HDL, using Xilinx Vivado 2016.4 design flow[1].

The current implementation of LaKe supports up to thirteen PEs, though only five PEs are required to achieve full line rate. The limitation on the number of PEs is due to the number of slave interfaces available on the AXI-Stream interconnect cores, used by the PE interconnect and the memory interconnect (16 slaves, where SRAM, DRAM and CAM must always be connected). The core clock frequency is 200MHz.

Using five PEs, the fully implemented prototype consumes only 35.65% of the Block RAM (BRAM) and 52.33% slice utilization[2]. On a higher end FPGA this number is significantly smaller, allowing scalability to higher data rates: using 13 PEs a query rate of 42.9Mqps is achievable.

A. Integration with NetFPGA Datapath

LaKe is integrated with the NetFPGA switch/NIC datapath as shown in Figure 2. LaKe gives priority to normal traffic over memcached traffic; if the memcached packet rate is high and over-subscribes the cache sub-system, instead of throttling normal traffic LaKe drops memcached packets. Consequently, normal traffic is not affected by LaKe's performance. Outgoing packets from LaKe module go through an arbiter, which arbitrates between memcached packets and normal packets, forwarding them to an *Output Port Lookup* module.

As an in-network computing platform, LaKe provides caching for memcached. Consequently, SET and DELETE requests need to be updated in the host's memory. In our implementation, SET and DELETE requests are copied to both paths within the FPGA: to LaKe and to normal traffic path. Specifically, while a SET request to LaKe updates or adds new cache contents to the shared-cache and DRAM module on NetFPGA, the SET request sent through the normal traffic path updates or adds new content to the host memory, running the memcached server (software). In addition, a reply from the host to a GET request missed in the hardware, also goes through the normal traffic path and to LaKe's module, updating the local cache contents.

[1]The version supported by the NetFPGA project.

[2]Note that the NetFPGA Reference Switch uses 13.91% of the Block RAM (BRAM) and 17.9% slice utilization

IV. Evaluation

The evaluation of LaKe covers two aspects: absolute performance, and the exploration of design trade-offs. The evaluation results are summarized in Table I.

A. Absolute Performance

We evaluate the absolute performance of LaKe based on several performance metrics: throughput, latency and power efficiency. We compare the performance with memcached (v1.5.1), a software implementation, and Emu's memcached implementation [17], a hardware-acceleration of memcached using the binary protocol. Emu is selected as it is comparable, being available open-source on NetFPGA-SUME, yet it does not support networking functionality, only memcached-acceleration. Emu also supports only an on-chip cache, and cannot forward missed query to a server.

1) Test Setup: The memcached server uses Intel Core i7-4770 CPU, 64GB RAM, running Ubuntu 14.04 LTS (Linux kernel 3.19.0) and a NetFPGA-SUME card running LaKe. OSNT [18] is used for traffic injection. A 10GbE port is connected from OSNT to the LaKe card. GET requests, including 4B key and 8B value, are injected at 10Gbps. Throughput is measured on a second granularity. For comparison with software-based memcached, we amended the memcached software to support binary protocol over UDP.

a) Maximum Throughput: For maximum throughput, we compare all three designs using a warmed cached. LaKe achieves a throughput of 13.1Mqps (query per second) when all the queries are *hit* in the shared-cache, as shown in Table I. This is $\times 6.7$ improvement compared with Emu [17], and $\times 13.6$ improvement compared with memcached running on the host. Note that a request query is 74B in size, hence the maximum theoretical throughput of 10GbE link is 13.297Mqps (taking into account Ethernet's preamble and inter-frame gap). The throughput achieved is equivalent to 10GbE line rate, using the given query size, and requires only 5 PEs.

b) Latency: We use an Endace DAG card 10X2-S (4ns resolution) to measure queries' latency. A software-based client is used to generate queries, and the DAG measures the isolated latency of LaKe, client excluded. Despite supporting both memcached and networking functionality, as well as using the DRAM, LaKe's latency on a *hit* (1.16μs) is better than Emu (1.21μs), thanks to the small shared-cache (64kB) in front of the DRAM. When queries are *miss* in the shared-cache, and *hit* in the DRAM, the latency is 5.6μs. Emu does not support cache misses. LaKe's latency is $\times 205$ better than a host-based memcached on a hit, and $\times 42$ better on a miss in the cache and a hit in the DRAM. A miss in both cache and DRAM means LaKe and a host-based memcached will have about the same latency, as LaKe will forward the query to the host. The only penalty is the first look up in the DRAM of the key.

B. Scalability

LaKe scales up both in throughput and resources.

Area and Resources: We implemented up to six PEs while maintaining 200MHz core frequency, as shown in Figure 6. Each PE utilizes around 3% of chip slices and 2% BRAMs. These values include also the interconnection networks, as each PE is connected with both PE-network and memory switch. The small overhead in resources taken by each PE enables scaling the number of PEs used by LaKe with little effect on resource consumption.

Throughput: We evaluate the throughput scalability of LaKe using OSNT [18]. First, the cache is warmed using a SET request. Next, OSNT generates GET requests, matching the warmed cache, using a 4B key, and returning an 8B value. The throughput scalability as a function of the number of PEs is shown in Figure 7. As the figure shows, LaKe can handle up to 13.1Mqps using five PEs, when the queries are *hit* in the shared-cache in front of the DRAM. Each PE processes up to 3.3Mqps. The bottlenecks on throughput growth are the memory interconnect core and memory bandwidth. The throughput grows linearly with the number of PEs until reaching these bottlenecks. On a platform with more memory interfaces, or with a higher speed memory, a higher throughput can be reached.

Core frequency: LaKe is a pipelined design. As such, its throughput depends on its packet processing rate. This packet processing rate, which is shared for the networking data plane and the LaKe memcached module, is fully achieved at a core frequency of 160MHz, as shown in Figure 9. Below this frequency, the NetFPGA platform has a performance limitation in its 10GbE ports [3]. Mean latency drops as core frequency increase: this is as the number of stages in the pipeline is maintained, but the duration of each clock cycle is reduced.

Hit ratio: The hit ratio in the cache plays a critical role in the performance of an in-network computing design. In Figure 10 we demonstrate the effect of the hit ratio on the performance of LaKe. The x-axis indicates the hit ratio of the keys in the on-chip cache. The y-axis indicates the maximum throughput (left) and mean latency (right). Mean latency is measured at a constant query rate of 10Kqps, for all hit ratios, since as we show in Section V, the latency is subject to change under different query rates. The maximum latency, measured across all hit-ratios, is only $1.9\mu s$. The effect of the hit ratio is mandatory to in-network computing solutions, as the size of the on-chip cache directly affects the performance of the device. In devices where the memory capacity is in the order of megabytes to tens of megabytes [19], this becomes a crucial element.

Figure 11 continues the exploration of hit-ratio effects, by exploring the effect of the hit ratio in the DRAM, and LaKe as a whole. The x-axis in Figure 11 indicates the overall hit ratio in both on-chip cache and DRAM. We fix the hit ratio in the on-chip cache to 10%, and vary the hit ratio in the DRAM, with all queries missed in the DRAM being sent to the host. As the results show, throughput linearly increases with the hit ratio in the DRAM.

[3]https://github.com/NetFPGA/NetFPGA-SUME-live/issues/36

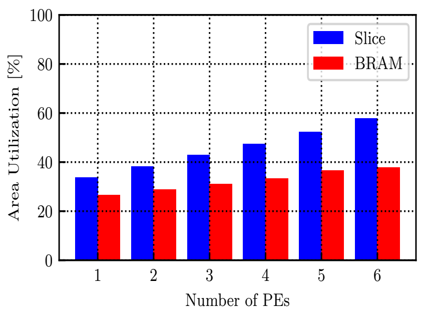

Fig. 6: The area utilization of LaKe implemented on NetFPGA SUME.

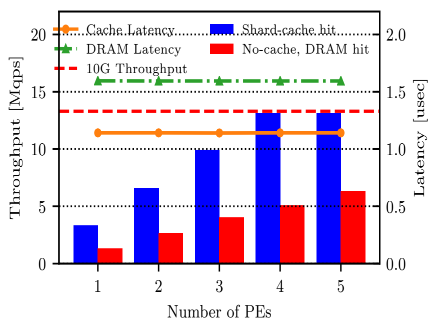

Fig. 7: Throughput and latency as a function of number of PEs.

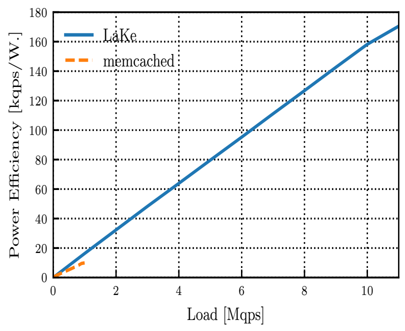

Fig. 8: Power efficiency vs throughput, for LaKe and memcached (bottom left).

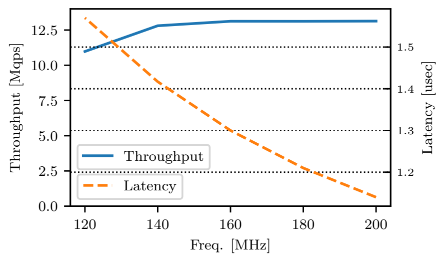

Fig. 9: The throughput and latency of LaKe as a function of core frequency.

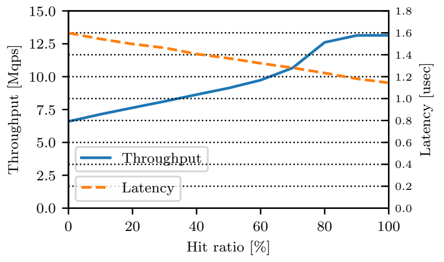

Fig. 10: LaKe's throughput and latency under varying hit ratio in the on-chip cache. Queries missed in the cache are a hit in the DRAM.

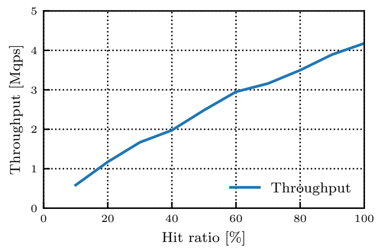

Fig. 11: LaKe's throughput under varying hit ratio in the DRAM, with fixed 10% in the on-chip cache.

C. *Power Efficiency*

We use a wall power meter to measure power consumption. We calculate power efficiency as $E = T/W$, where E, W and T denote power efficiency, power consumption and throughput, respectively. LaKe achieves 242.962 kqps/Watt using five PEs at full line rate. This is $\times 5.1$ improvement compared with Emu.

We investigate dynamic power consumption by measuring how power consumption varies as a function of throughput. Power consumption is normalized to 0W under zero load, i.e. the static power consumption. The dynamic power consumption takes up to 3.4W on our LaKe modules, while software based memcached consumes a maximum of 58.2W dynamically. Thus, LaKe reduces dynamic power consumption by more than an order of magnitude. Moreover, the power efficiency of LaKe scales linearly with throughput (as shown in Figure 8), much better than a host's power efficiency does. To put it in order words, LaKe's power consumption changes very little under load, which means that it is most efficient when the query rate is maximal. Even under a low query rate, LaKe's power efficiency is better than running on a host.

V. Design Trade-Offs

The previous section has introduced the absolute performance of LaKe. In this section we focus on trade-offs in the design of LaKe, and extrapolate from them to in-network computing designs at large.

In-network computing applications tend to implement cache using only on-chip memory [2], [4], [20]. For KVS applications, this leads to a very small percentage of keys that can be cached: in the orders of thousands to tens of thousands on an FPGA, and in the order of hundreds of thousands to a million on an ASIC. For example, NetChain [3] suggests that up to 10MB on a Tofino switch can be used as a cache. This number of cache entries is insufficient for large KVS systems: in Facebook, between a billion and hundred billion unique keys are accessed every hour [21], with 18.4% to 74.7% of these keys accessed within 5 minutes (For Facebook's different workloads [21]). It is therefore important to understand the effect of using external memories on in-network computing performance.

So far the evaluation used a fully-featured LaKe: using BRAM, SRAM and DRAM. Next, we check the effect of each on the performance. Note that for this discussion we employ a single DRAM module (4GB) which utilizes both hash table region (2GB: 268M entries) and a data store region (2GB: 33M entries as 64B chunk), and consumes 4W. When we use BRAM instead of DRAM, the number of hash table entries and data store entries are 4096 entries and 512 entries, respectively. We also employ two SRAM modules (total 18MB) to manage

TABLE I: Performance comparison.

System	Average latency [μs]	Throughput [Mqps]	Power efficiency [kqps/Watt.]
memcached(software)	238.84	0.962	9.938
Emu (hardware) [17]	1.21	1.932	47.121
LaKe (shared-cache)	1.16	13.120	242.962

free-list on slab allocation. When we use BRAM instead of SRAM, the number of free-list addresses stored is 144 entries.

When only the BRAM is used, and the SRAM and DRAM memory controllers are taken out, the maximum power consumption of LaKe is 16W including NetFPGA-SUME card — almost identical to a standalone switch, and the maximum throughput is 13.1Mqps. Under these circumstances we use a BRAM-based 1k entry cache as hash table and data store instead of a DRAM, and use BRAM-based FIFO as slab allocator instead of an SRAM.

Adding the SRAM adds 6W and holds 4.7M chunk addresses, which are updated when a DELETE operation moves a specific chunk to the free list. A BRAM-based FIFO placed in front of the SRAM is used to hide SRAM access latency, but is shallow in comparison with the SRAM. One can therefore trade the 6W SRAM power consumption with the number of available chunks on LaKe. Alternatively, one can use a DRAM to store chunk address: this solution is cheaper and more power efficient than using SRAM, but results in an increased latency and considerably lower throughput.

The use of DRAM as a second level cache increases the number of keys hit in LaKe. However, as can be expected, DRAM access does not provide the same performance as on-chip cache access. As shown in Figure 7, the maximum throughput using DRAM only is 6.3Mqps (using five PEs), lower than using the shared-cache. To understand the throughput of the DRAM, we isolate the DRAM from the LaKe module, and consider its latency under low and high utilization. As Figure 12 shows, while the latency is almost constant without a load, under high utilization the latency almost doubles. As memcached accesses to the memory are random and not sequential, as keys are not requested in a sorted order, this double-latency explains the 6.3Mqps throughput achieved.

While using the DRAM may seem as a disadvantage, it is in fact an advantage: access to the DRAM is only upon a miss in the cache, and replaces an access to the host memory (as in a device without a DRAM). In this manner, significant time ($\times 42$) and dynamic power ($\times 17$) are saved.

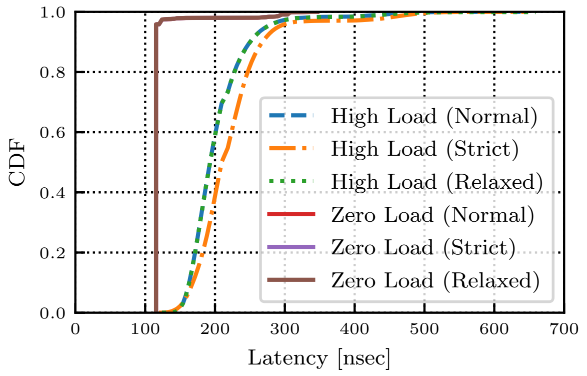

Fig. 12: The cumulative distribution function of READ latency from the DRAM, for a Random access, under zero and high load. Strict, Normal and Relax are the three memory controller access modes [22].

VI. Related Work

In-network computing has emerged as a mean to reduce data processing loads from the host, as data processing demands keep increasing [23]. Use cases included, for example, consensus [4], [5] and coordination [3]. Caching, and KVS in particular is a representative and popular in-network computing use case [2], [24]. Programmable switch ASICs (e.g., Barefoot Networks Tofino), enable these devices to achieve high performance — both throughput and latency wise. Yet until such devices have significant memory resources attached, they will not be able to completely offload KVS applications, as discussed in Section V. To the best of our knowledge, this paper presents the first implementation of KVS as an in-network computing solution.

Hardware-based KVS has been actively researched along with the rise of cloud computing [1], [13], [14], [25]–[33], though not while providing switching functionality. It was previously shown that offloading KVS into dedicated hardware, such as FPGA or ASIC, benefits in terms of latency, throughput and power efficiency. Although hardware based memcached appliances [14], [27] were shown to achieve 10GbE throughput, the cache capacity was small, limited by physical resources and FPGA I/O constraints. In contrast, our work focuses on a layered cache architecture, benefiting from an integration with the host machine. As long as queries are a hit in the FPGA-NIC, the CPU load is significantly reduced. Further, LaKe also serves as a standard network device, avoiding additional hardware required by dedicated acceleration solutions.

KV-Direct [11] demonstrated a SmartNIC achieving a high query rate (e.g., $\sim$180Mqps), but was limited to KVS operations only, was a proprietary solution using 8B query size, batching multiple queries in a single packet, and processing vector queries. LaKe supports the highly popular memcache protocol and different slab sizes, offering a far richer feature set, as well as standard networking operation.

Energy efficiency in KVS was also researched in software-based solutions [15], [24], [34]–[38]. Software-based solutions improve performance and power efficiency by using the CPU more effectively. However, software-based solutions do not improve latency and power consumption, while LaKe has improved latency, throughput and energy efficiency dramatically.

VII. Conclusion

While network bandwidth is ever increasing, computing performance is leveling off. In-network computing is providing hardware acceleration using network devices already existing in the network. While ASIC based in-network computing offers order of magnitude higher throughput than running on a host, we show that for realistic KVS workloads external memories are required, and their cost in power and performance is high. We presented LaKe, a new architecture for energy efficient in-network KVS. LaKe can serve as a switch or a NIC, while presenting a multi-core, multi-level cache architecture, that balances throughput, latency and power efficiency. LaKe achieves $\times 17$ better energy efficiency than running on a host, with $\times 6.7$ to $\times 13.6$ higher throughput, maintaining two orders of magnitude better latency. LaKe does all that without giving up memcached functionality and while supporting a large and scalable number of keys.

Acknowledgment

This work was supported by JSPS Research Fellowship and Keio University Research Grant for Young Researcher's Program. This work was supported by JST CREST Grant Number JPMJCR1785, Japan. We acknowledge the support of the Leverhulme Trust (ECF-2016-289) and the Isaac Newton Trust.

References

[1] S. R. Chalamalasetti *et al.*, "An FPGA Memcached Appliance," in *FPGA*, Feb. 2013, pp. 245–254.
[2] X. Jin *et al.*, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *SOSP*. ACM, 2017, pp. 121–136.
[3] ——, "NetChain: Scale-Free Sub-RTT Coordination," in *NSDI*. USENIX Association, 2018, pp. 35–49.
[4] T. D. Huynh *et al.*, "P4xos: Consensus as a network service," *Technical Report University of Lugano*, May 2018.
[5] H. T. Dang *et al.*, "Netpaxos: Consensus at network speed," in *SOSR*. ACM, 2015, p. 5.
[6] T. Jepsen *et al.*, "Life in the fast lane: A line-rate linear road," in *Proceedings of the Symposium on SDN Research*. ACM, 2018, p. 10.
[7] A. Sapio *et al.*, "In-network computation is a dumb idea whose time has come," in *HOTNETS*. ACM, 2017, pp. 150–156.
[8] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.
[9] R. Nishtala *et al.*, "Scaling memcache at facebook," in *NSDI*. USENIX, 2013, pp. 385–398.
[10] N. Zilberman *et al.*, "Where has my time gone?" in *Passive and Active Measurement*, 2017, pp. 201–214.
[11] B. Li *et al.*, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC," in *SOSP*. ACM, 2017, pp. 137–152.
[12] N. Zilberman *et al.*, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.
[13] E. S. Fukuda *et al.*, "Caching Memcached at Reconfigurable Network Interface," in *FPL*, Sep. 2014, pp. 1–6.
[14] M. Blott *et al.*, "Achieving 10Gbps Line-rate Key-value Stores with FPGAs," in *HotCloud*, Jun. 2013.
[15] H. Lim *et al.*, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in *NSDI*, Apr. 2014, pp. 429–444.
[16] Danga Interactive, "Memcached - A Distributed Memory Object Caching System," http://memcached.org/.
[17] N. Sultana *et al.*, "Emu: Rapid Prototyping of Networking Services," in *ATC*. USENIX Association, 2017, pp. 459–471.
[18] G. Antichi *et al.*, "OSNT: open source network tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, September 2014.
[19] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.
[20] "Barefoot Tofino," https://www.barefootnetworks.com/products/brief-tofino/, 2018.
[21] B. Atikoglu *et al.*, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS*, Jun. 2012, pp. 53–64.
[22] Xilinx, "7 Series FPGAs Memory Resources UG473(v1.12)," Sep 2016.
[23] Cisco, "Global cloud index, forecast and methodology, 2016-2021," *Retrieved July*, 2018.
[24] M. Liu *et al.*, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *ASPLOS*, 2017, pp. 795–809.
[25] M. Lavasani *et al.*, "An FPGA-based In-Line Accelerator for Memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, Jul. 2014.
[26] J. M. Cho *et al.*, "An FPGA Implementation of High-throughput Key-value Store Using Bloom Filter," in *VLSI-DAT*, Apr. 2014, pp. 1–4.
[27] M. Blott *et al.*, "Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory," in *HotStorage*, Jul. 2015.
[28] Y. Tokusashi *et al.*, "A Multilevel NOSQL Cache Design Combining In-NIC and In-Kernel Caches," in *Hot Interconnects*, Aug. 2016.
[29] ——, "Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches," *IEEE Micro*, vol. 37, no. 5, pp. 44–51, 2017.
[30] K. Lim *et al.*, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *ISCA*, Jun. 2013, pp. 36–47.
[31] J. Lockwood *et al.*, "Implementing Ultra Low Latency Data Center Services with Programmable Logic," in *HoT Interconnects*, Aug 2015, pp. 68–77.
[32] S. Xu *et al.*, "Bluecache: A Scalable Distributed Flash-based Key-value Store," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 301–312, Nov. 2016.
[33] J. Choi *et al.*, "Accelerating Memcached on AWS Cloud FPGAs," in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART, 2018, pp. 2:1–2:8.
[34] S. Li *et al.*, "Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform," in *ISCA*, Jun. 2015, pp. 476–488.
[35] ——, "Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform," *ACM Trans. Comput. Syst.*, vol. 34, no. 2, pp. 5:1–5:30, Apr. 2016.
[36] ——, "Achieving One Billion Key-Value Requests per Second on a Single Server," *IEEE Micro*, vol. 36, no. 3, pp. 94–104, May 2016.
[37] H. Lim *et al.*, "Cicada: Dependably Fast Multi-Core In-Memory Transactions," in *SIGMOD*. New York, NY, USA: ACM, 2017, pp. 21–35.
[38] Hyotaek Shim, "PHash: A memory-efficient, high-performance key-value store for large-scale data-intensive applications," *Journal of Systems and Software*, vol. 123, pp. 33 – 44, 2017.