# KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC

Bojie Li[*§†] Zhenyuan Ruan[*‡†] Wencong Xiao[•†] Yuanwei Lu[§†]

Yongqiang Xiong[†] Andrew Putnam[†] Enhong Chen[§] Lintao Zhang[†]

[†]Microsoft Research [§]USTC [‡]UCLA [•]Beihang University

## ABSTRACT

Performance of in-memory key-value store (KVS) continues to be of great importance as modern KVS goes beyond the traditional object-caching workload and becomes a key infrastructure to support distributed main-memory computation in data centers. Recent years have witnessed a rapid increase of network bandwidth in data centers, shifting the bottleneck of most KVS from the network to the CPU. RDMA-capable NIC partly alleviates the problem, but the primitives provided by RDMA abstraction are rather limited. Meanwhile, programmable NICs become available in data centers, enabling in-network processing. In this paper, we present KV-Direct, a high performance KVS that leverages programmable NIC to extend RDMA primitives and enable remote direct key-value access to the main host memory.

We develop several novel techniques to maximize the throughput and hide the latency of the PCIe connection between the NIC and the host memory, which becomes the new bottleneck. Combined, these mechanisms allow a single NIC KV-Direct to achieve up to 180 M key-value operations per second, equivalent to the throughput of tens of CPU cores. Compared with CPU based KVS implementation, KV-Direct improves power efficiency by 3x, while keeping tail latency below 10 $\mu$s. Moreover, KV-Direct can achieve near linear scalability with multiple NICs. With 10 programmable NIC cards in a commodity server, we achieve 1.22 billion KV operations per second, which is almost an order-of-magnitude improvement over existing systems, setting a new milestone for a general-purpose in-memory key-value store.



*SOSP '17, Shanghai, China*



DOI: 10.1145/3132747.3132756

## CCS CONCEPTS

•**Information systems** → **Key-value stores;** •**Hardware** → *Hardware-software codesign;*

## KEYWORDS

Key-Value Store, Programmable Hardware, Performance

**ACM Reference format:**

Bojie Li[*§†] Zhenyuan Ruan[1][‡†] Wencong Xiao[•†] Yuanwei Lu[§†] and Yongqiang Xiong[†] Andrew Putnam[†] Enhong Chen[§] Lintao Zhang[†] . 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017,* 16 pages.

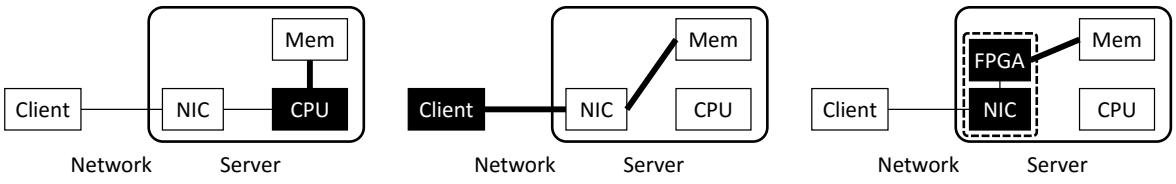
DOI: 10.1145/3132747.3132756

## 1 INTRODUCTION

In-memory key-value store (KVS) is a key distributed system component in many data centers. KVS enables access to a shared key-value hash table among distributed clients. Historically, KVS such as Memcached [25] gained popularity as an object caching system for web services. Large web service providers such as Amazon [17] and Facebook [3, 57], have deployed distributed key-value stores at scale. More recently, as main-memory based computing becomes a major trend in the data centers [18, 58], KVS starts to go beyond caching and becomes an infrastructure to store shared data structure in a distributed system. Many data structures can be expressed in a key-value hash table, *e.g.*, data indexes in NoSQL databases [12], model parameters in machine learning [46], nodes and edges in graph computing [67, 74] and sequencers in distributed synchronization [37]. For most of these applications, the performance of the KVS is the key factor that directly determines the system efficiency. Due to its importance, over the years significant amount of research effort has been invested on improving KVS performance.

Earlier key-value systems [17, 25, 57] are built on top of traditional OS abstractions such as OS lock and TCP/IP stack. This puts considerable stress on the performance of

[*]Bojie Li and Zhenyuan Ruan are co-first authors who finish this work during internship at Microsoft Research.




(a) Software / two-sided RDMA. (b) One-sided RDMA. (c) KV-Direct.

**Figure 1: Design space of KVS data path and processing device. Line indicates data path. One KV operation (thin line) may require multiple address-based memory accesses (thick line). Black box indicates where KV processing takes place.**

the OS, especially the networking stack. The bottleneck is exacerbated by the fact that physical network transport speed has seen huge improvements in the last decade due to heavy bandwidth demand from data center applications.

More recently, as both the single core frequency scaling and multi-core architecture scaling are slowing down [21, 69], a new research trend in distributed systems is to leverage Remote Direct Memory Access (RDMA) technology on NIC to reduce network processing cost. One line of research [36, 37] uses two-sided RDMA to accelerate communication (Figure 1a). KVS built with this approach are bounded by CPU performance of the KVS servers. Another line of research uses one-sided RDMA to bypass remote CPU and shift KV processing workload to clients [18, 55] (Figure 1b). This approach achieves better GET performance but degrades performance for PUT operations due to high communication and synchronization overhead. Due to lack of transactional support, the abstraction provided by RDMA is not a perfect fit for building efficient KVS.

In the meantime, another trend is emerging in data center hardware evolution. More and more servers in data centers are now equipped with programmable NICs [10, 27, 64]. At the heart of a programmable NIC is a field-programmable gate array (FPGA) with an embedded NIC chip to connect to the network and a PCIe connector to attach to the server. Programmable NIC is initially designed to enable network virtualization [24, 44]. However, many found that FPGA resources can be used to offload some workloads of CPU and significantly reduce CPU resource usage [14, 30, 52, 60]. Our work takes this general approach.

We present KV-Direct, a new in-memory key-value system that takes advantage of programmable NIC in data center. KV-Direct, as its name implies, directly fetches data and applies updates in the host memory to serve KV requests, bypassing host CPU (Figure 1c). KV-Direct extends the RDMA primitives from memory operations (READ and WRITE) to key-value operations (GET, PUT, DELETE and ATOMIC ops). Compared with one-sided RDMA based systems, KV-Direct deals with the consistency and synchronization issues at server-side, thus removes computation overhead in client and reduces network traffic. In addition, to support vector-based operations and reduce network traffic, KV-Direct also provides new vector primitives UPDATE, REDUCE, and FILTER, allowing users to define active messages [19] and delegate certain computation to programmable NIC for efficiency.

Since the key-value operations are offloaded to the programmable NIC, we focus our design on optimizing the PCIe traffic between the NIC and host memory. KV-Direct adopts a series of optimizations to fully utilize PCIe bandwidth and hide latency. Firstly, we design a new hash table and memory allocator to leverage parallelism available in FPGA and minimize the number of PCIe DMA requests. On average, KV-Direct achieves close to one PCIe DMA per READ operation and two PCIe DMAs per WRITE operation. Secondly, to guarantee consistency among dependent KV operations, KV-Direct includes an out-of-order execution engine to track operation dependencies while maximizing the throughput of independent requests. Thirdly, KV-Direct exploits on-board DRAM buffer available on programmable NIC by implementing a hardware-based load dispatcher and caching component in FPGA to fully utilize on-board DRAM bandwidth and capacity.

A single NIC KV-Direct is able to achieve up to 180 M KV operations per second (Ops), equivalent to the throughput of 36 CPU cores [47]. Compared with state-of-art CPU KVS implementations, KV-Direct reduces tail latency to as low as 10 $\mu s$ while achieving a 3x improvement on power efficiency. Moreover, KV-Direct can achieve near linear scalability with multiple NICs. With 10 programmable NIC cards in a server, we achieve 1.22 billion KV operations per second in a single commodity server, which is more than an order of magnitude improvement over existing systems.

KV-Direct supports general atomic operations up to 180 Mops, equal to normal KV operation and significantly outperforms the number reported in state-of-art RDMA-based system: 2.24 Mops [36]. The atomic operation agnostic performance is mainly a result of our out-of-order execution engine that can efficiently track the dependency among KV operations without explicitly stalling the pipeline.

# 2 BACKGROUND

## 2.1 Workload Shift in KVS

Historically, KVS such as Memcached [25] gained popularity as an object caching system for web services. In the era of in-memory computation, KVS goes beyond caching and becomes an infrastructure service to store shared data structure in a distributed system. Many data structures can be expressed in a key-value hash table, *e.g.*, data indexes in NoSQL databases [12], model parameters in machine learning [46], nodes and edges in graph computing [67, 74] and sequencers in distributed synchronization [37, 45].

The workload shifts from object cache to generic data structure store implies several design goals for KVS.

**High batch throughput for small KV.** In-memory computations typically access small key-value pairs in large batches, *e.g.*, sparse parameters in linear regression [48, 74] or all neighbor nodes in graph traversal [67], therefore a KVS should be able to benefit from batching and pipelining.

**Predictable low latency.** For many data-parallel computation tasks, the latency of an iteration is determined by the slowest operations [59]. Therefore, it is important to control the tail latency of KVS. CPU based implementations often have large fluctuations under heavy load due to scheduling irregularities and inflated buffering.

**High efficiency under write-intensive workload.** For cache workloads, KVS often has much more reads than writes [3], but it is no longer the case for distributed computation workloads such as graph computation [61], parameter servers [46]. These workloads favor hash table structures that can handle both read and write operations efficiently.

**Fast atomic operations.** Atomic operations on several extremely popular keys appear in applications such as centralized schedulers [63], sequencers [37, 45], counters [76] and short-term values in web applications [3]. This requires high throughput on single-key atomics.

**Support vector-type operations.** Machine learning and graph computing workloads [46, 67, 74] often require operating on every element in a vector, *e.g.*, incrementing every element in a vector with a scalar or reducing a vector into the sum of its elements. KVS without vector support requires the client to either issue one KVS operation per element, or retrieve the vector back to the client and perform the operation. Supporting vector data type and operations in KVS can greatly reduce network communication and CPU computation overhead.

## 2.2 Achilles' Heel of High-Performance KVS Systems

Building a high performance KVS is a non-trivial exercise of optimizing various software and hardware components in a computer system. Characterized by where the KV processing takes place, state-of-the-art high-performance KVS systems basically falls into three categories: on the CPU of KVS server (Figure 1a), on KVS clients (Figure 1b) or on a hardware accelerator (Figure 1c).

When pushed to the limit, in high performance KVS systems the throughput bottleneck can be attributed to the computation in KV operation and the latency in random memory access. CPU-based KVS needs to spend CPU cycles for key comparison and hash slot computation. Moreover, KVS hash table is orders of magnitude larger than the CPU cache, therefore the memory access latency is dominated by cache miss latency for practical access patterns.

By our measurement, a 64-byte random read latency for a contemporary computer is ~110 ns. A CPU core can issue several memory access instructions concurrently when they fall in the instruction window, limited by the number of load-store units in a core (measured to be 3~4 in our CPU) [26, 28, 75]. In our CPU, we measure a max throughput of 29.3 M random 64B access per second per core. On the other hand, an operation to access 64-byte KV pair typically requires ~100 ns computation or ~500 instructions, which is too large to fit in the instruction window (measured to be 100~200). When interleaved with computation, the performance of a CPU core degrades to only 5.5 M KV operations per second (Mops). An optimization is to batch memory accesses in a KV store by clustering the computation for several operations together before issuing the memory access all at once [47, 56]. This improves the per-core throughput to 7.9 MOps in our CPU, which is still far less than the random 64B throughput of host DRAM.

Observing the limited capacity of CPU in KV processing, recent work [18, 55, 70] leverage one-sided RDMA to offload KV processing to clients and effectively using the KVS server as a shared memory pool. Despite the high message rate (8~150 Mops [37]) provided by RDMA NICs, it is challenging to find an efficient match between RDMA primitives and key-value operations. For a write (PUT or atomic) operation, multiple network round-trips and multiple memory accesses may be required to query the hash index, handle hash collisions and allocate variable-size memory. RDMA does not support transactions. Clients must synchronize with each other to ensure consistency using RDMA atomics or distributed atomic broadcast [70], both incurring communication overhead and synchronization latency [18, 55]. Therefore, most RDMA-based KVS [18, 36, 55] recommend using one-sided RDMA for GET operations only. For PUT operations, they fall back to the server CPU. The throughput of write-intensive workload is still bottlenecked by CPU cores.

## 2.3 Programmable NIC with FPGA

Ten years ago, processor frequency scaling slowed down and people turned to multi-core and concurrency [69]. Nowadays,

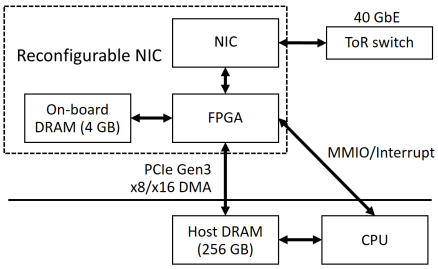
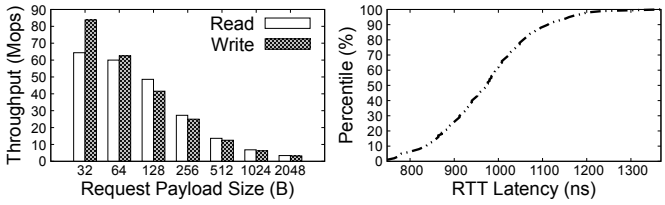



**Figure 2: Programmable NIC with FPGA.**




**(a) Throughput.** **(b) DMA Read Latency.**

**Figure 3: PCIe random DMA performance.**

power ceiling implies that multi-core scaling has also met difficulties [22]. People are now turning to domain-specific architectures (DSAs) for better performance.

Due to the increasing mismatch of network speed and CPU network processing capacity, programmable NICs with FPGA [10, 24, 27, 44] now witness large-scale deployment in datacenters. As shown in Figure 2, the heart of the programmable NIC we use is an FPGA, with an embedded NIC chip to connect to the network. Programmable NICs typically come with on-board DRAM as packet buffers and runtime memory for NIC firmware [44], but the DRAM is typically not large enough to hold the entire key-value store.

## 2.4 Challenges for Remote Direct Key-Value Access

KV-Direct moves KV processing from the CPU to the programmable NIC in the server (Figure 1c). Same as RDMA, the KV-Direct NIC accesses host memory via PCIe. PCIe is a packet switched network with ~500 ns round-trip latency and 7.87 GB/s theoretical bandwidth per Gen3 x8 endpoint. On the latency side, for our programmable NIC, the cached PCIe DMA read latency is 800 ns due to additional processing delay in FPGA. For random non-cached DMA read, there is an additional 250 ns average latency (Figure 3b) due to DRAM access, DRAM refresh and PCIe response reordering in PCIe DMA engine. On the throughput side, each DMA read or write operation needs a PCIe transport-layer packet (TLP) with 26-byte header and padding for 64-bit addressing. For a PCIe Gen3 x8 NIC to access host memory in 64-byte granularity, the theoretical throughput is therefore 5.6 GB/s, or 87 Mops.

To saturate PCIe Gen3 x8 with 64-byte DMA requests, 92 concurrent DMA requests are needed considering our latency of 1050 ns. In practice, two factors further limit the concurrency of DMA requests. First, PCIe credit-based flow control constrains the number of in-flight requests for each DMA type. The PCIe root complex in our server advertises 88 TLP posted header credits for DMA write and 84 TLP non-posted header credits for DMA read. Second, DMA read requires assigning a unique PCIe tag to identify DMA responses which may come out of order. The DMA engine in our FPGA only support 64 PCIe tags, further limiting our DMA read concurrency to 64 requests, which renders a throughput of 60 Mops as shown in Figure 3a. On the other hand, with 40 Gbps network and 64-byte KV pairs, the throughput ceiling is 78 Mops with client-side batching. In order to saturate the network with GET operations, the KVS on NIC must make full use of PCIe bandwidth and achieve close to one average memory access per GET. This boils down to three challenges:

**Minimize DMA requests per KV operation.** Hash table and memory allocation are two major components in KVS that require random memory access. Previous works propose hash tables [9, 18] with close to 1 memory access per GET operation even under high load factors. However, under higher than 50% load factor, these tables need multiple memory accesses per PUT operation on average with large variance. This not only consumes PCIe throughput, but also leads to latency variations for write-intensive workloads.

In addition to hash table lookup, dynamic memory allocation is required to store variable-length KVs that cannot be inlined in the hash table. Minimizing hash table lookups per KV operation and memory accesses per memory allocation is essential for matching PCIe and network throughput under write-intensive, small-KV workloads.

**Hide PCIe latency while maintaining consistency.** An efficient KVS on NIC must pipeline KV operations and DMA requests to hide the PCIe latency. However, KV operations may have dependencies. A GET following PUT on a same key needs to return the updated value. This requires tracking KV operations being processed and stall the pipeline on data hazard, or better, design an out-of-order executor to resolve data dependency without explicitly stalling the pipeline.

**Dispatch load between NIC DRAM and host memory.** An obvious idea is to use the DRAM on NIC as a cache for host memory, but in our NIC, the DRAM throughput (12.8 GB/s) is on par with the achievable throughput (13.2 GB/s) of two PCIe Gen3 x8 endpoints. It is more desirable to distribute memory access between DRAM and host memory in order to utilize both of their bandwidths. However, the on-board DRAM is small (4 GiB) compared to the host memory (64 GiB), calling for a hybrid caching and load-dispatching approach.

**Table 1: KV-Direct operations.**

| | |
|---|---|
| get $(k) \rightarrow v$ | Get the value of key $k$. |
| put $(k, v) \rightarrow$ bool | Insert or replace a $(k, v)$ pair. |
| delete $(k) \rightarrow$ bool | Delete key $k$. |
| update_scalar2scalar $(k, \Delta, \lambda(v, \Delta) \rightarrow v) \rightarrow v$ | Atomically update the value of key $k$ using function $\lambda$ on scalar $\Delta$, and return the original value. |
| update_scalar2vector $(k, \Delta, \lambda(v, \Delta) \rightarrow v) \rightarrow [v]$ | Atomically update all elements in vector $k$ using function $\lambda$ and scalar $\Delta$, and return the original vector. |
| update_vector2vector $(k, [\Delta], \lambda(v, \Delta) \rightarrow v) \rightarrow [v]$ | Atomically update each element in vector $k$ using function $\lambda$ on the corresponding element in vector $[\Delta]$, and return the original vector. |
| reduce $(k, \Sigma, \lambda(v, \Sigma) \rightarrow \Sigma) \rightarrow \Sigma$ | Reduce vector $k$ to a scalar using function $\lambda$ on initial value, and return the reduction result $\Sigma$. |
| filter $(k, \lambda(v) \rightarrow$ bool$) \rightarrow [v]$ | Filter elements in a vector $k$ by function $\lambda$, and return the filtered vector. |

In the following, we will present KV-Direct, a novel FPGA-based key-value store that satisfies all aforementioned goals and describe how we address the challenges.

# 3 DESIGN

## 3.1 System Architecture

KV-Direct enables *remote direct key-value access*. Clients send *KV-Direct operations* (§3.2) to KVS server while the programmable NIC processes the requests and sending back results, bypassing the CPU. The programmable NIC on KVS server is an FPGA reconfigured as a *KV processor* (§3.3). Figure 2 shows the architecture of KV-Direct.

## 3.2 KV-Direct Operations

KV-Direct extends one-sided RDMA operations to key-value operations, as summarized in Table 1. In addition to standard KVS operations as shown in the top part of Table 1, KV-Direct supports two types of vector operations: Sending a scalar to the NIC on the server and the NIC applies the update to each element in the vector; or send a vector to the server and the NIC updates the original vector element-by-element. Furthermore, KV-Direct supports user-defined update functions as a generalization to atomic operations. The update function needs to be pre-registered and compiled to hardware logic before executing. KV operations with user-defined update functions are similar to *active messages* [19], saving communication and synchronization cost.

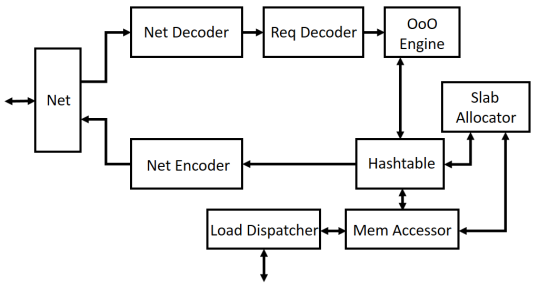



**Figure 4: KV processor architecture.**

When a vector operation update, reduce or filter is operated on a key, its value is treated as an array of fixed-bit-width elements. Each function $\lambda$ operates on one element in the vector, a client-specified parameter $\Delta$, and/or an initial value $\Sigma$ for reduction. The KV-Direct development toolchain duplicates the $\lambda$ several times to leverage parallelism in FPGA and match computation throughput with PCIe throughput, then compiles it into reconfigurable hardware logic using an high-level synthesis (HLS) tool [2]. The HLS tool automatically extracts data dependencies in the duplicated function and generates a fully pipelined programmable logic.

Update operations with user-defined functions are capable of general stream processing on a vector value. For example, a network processing application may interpret the vector as a stream of packets for network functions [44] or a bunch of states for packet transactions [68]. Single-object transaction processing completely in the programmable NIC is also possible, *e.g.*, wrapping around S_QUANTITY in TPC-C benchmark [16]. Vector reduce operation supports neighbor weight accumulation in PageRank [61]. Non-zero values in a sparse vector can be fetched with vector filter operation.

## 3.3 KV Processor

As shown in Figure 4, the KV processor in FPGA receives packets from the network, decodes vector operations and buffers KV operations in the reservation station (§3.3.3). Next, the out-of-order engine (§3.3.3) issues independent KV operations from reservation station into the operation decoder. Depending on the operation type, the KV processor looks up the hash table (§3.3.1) and executes the corresponding operations. To minimize the number of memory accesses, small KV pairs are stored *inline* in the hash table, others are stored in dynamically allocated memory from the slab memory allocator (§3.3.2). Both the hash index and the slab-allocated memory are managed by a unified memory access engine (§3.3.4), which accesses the host memory via PCIe DMA and caches a portion of host memory in NIC DRAM. After the KV operation completes, the result is sent back to the out-of-order execution engine (§3.3.3) to find and execute matching KV operations in reservation station.

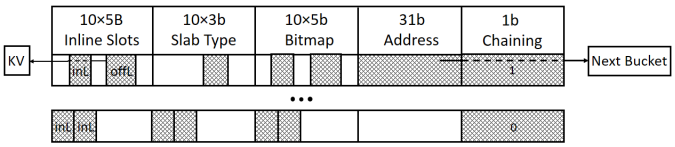



**Figure 5: Hash index structure. Each line is a hash bucket containing 10 hash slots, 3 bits of slab memory type per hash slot, one bitmap marking the beginning and end of inline KV pairs and a pointer to the next chained bucket on hash collision.**

As discussed in §2.4, the scarcity of PCIe operation throughput requires the KV processor to be frugal on DMA accesses. For GET operation, at least one memory read is required. For PUT or DELETE operation, one read and one write are minimal for hash tables. Log-based data structures can achieve one write per PUT, but it sacrifices GET performance. KV-Direct carefully designs the hash table to achieve close to ideal DMA accesses per lookup and insertion, as well as the memory allocator to achieve < 0.1 amortized DMA operations per dynamic memory allocation.

*3.3.1 Hash Table.* To store variable-sized KVs, the KV storage is partitioned into two parts. The first part is a hash index (Figure 5), which consists a fixed number of *hash buckets*. Each hash bucket contains several *hash slots* and some metadata. The rest of the memory is dynamically allocated, and managed by a slab allocator (§3.3.2). A *hash index ratio* configured at initialization time determines the percentage of the memory allocated for hash index. The choice of hash index ratio will be discussed in §5.1.1.

Each hash slot includes a pointer to the KV data in dynamically allocated memory and a secondary hash. Secondary hash is an optimization that enables parallel inline checking. The key is always checked to ensure correctness, at the cost of one additional memory access. Assuming a 64 GiB KV storage in host memory and 32-byte allocation granularity (a trade-off between internal fragmentation and allocation metadata overhead), the pointer requires 31 bits. A secondary hash of 9 bits gives a 1/512 false positive probability. Cumulatively, the hash slot size is 5 bytes. To determine the hash bucket size, we need to trade-off between the number of hash slots per bucket and the DMA throughput. Figure 3a shows that the DMA read throughput below 64B granularity is bound by PCIe latency and parallelism in the DMA engine. A bucket size less than 64B is suboptimal due to increased possibility of hash collision. On the other hand, increasing the bucket size above 64B would decrease hash lookup throughput. So we choose the bucket size to be 64 bytes.

*KV size* is the combined size of key and value. KVs smaller than a threshold are stored *inline* in the hash index to save the additional memory access to fetch KV data. An inline KV may span multiple hash slots, whose pointer and secondary

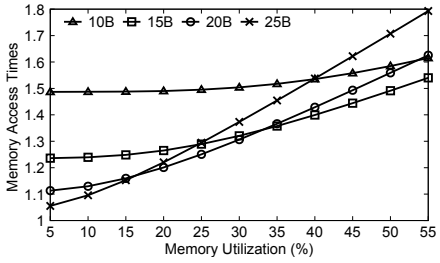



**Figure 6: Average memory access count under varying inline thresholds (10B, 15B, 20B, 25B) and memory utilizations.**

hash fields are re-purposed for storing KV data. It might not be optimal to inline all KVs that can fit in a bucket. To minimize average access time, assuming that smaller and larger keys are equally likely to be accessed, it is more desirable to inline KVs smaller than an *inline threshold*. To quantify the portion of used buckets in all buckets, we use *memory utilization* instead of load factor, because it relates more to the number of KVs that can fit in a fixed amount of memory. As shown in Figure 6, for a certain inline threshold, the average memory access count increases with memory utilization, due to more hash collisions. Higher inline threshold shows a more steep growth curve of memory access count, so an optimal inline threshold can be found to minimize memory accesses under a given memory utilization. As with hash index ratio, the inline threshold can also be configured at initialization time.

When all slots in a bucket are filled up, there are several solutions to resolve hash collisions. Cuckoo hashing [62] and hopscotch hashing [29] guarantee constant-time lookup by moving occupied slots during insertion. However, in write-intensive workload, the memory access time under high load factor would experience large fluctuations. Linear probing may suffer from primary clustering, therefore its performance is sensitive to the uniformity of hash function. We choose *chaining* to resolve hash conflicts, which balances lookup and insertion, while being more robust to hash clustering.

*3.3.2 Slab Memory Allocator.* Chained hash slots and non-inline KVs need dynamic memory allocation. We choose slab memory allocator [7] to achieve $O(1)$ average memory access per allocation and deallocation. The main slab allocator logic runs on host CPU and communicates with the KV-processor through PCIe. Slab allocator rounds up allocation size to the nearest power of two, called *slab size*. It maintains a *free slab pool* for each possible slab size (32, 64, ..., 512 bytes), and a global *allocation bitmap* to help to merge small free slabs back to larger slabs. Each free slab pool is an array of *slab entries* consisting of an address field and a slab type field indicating the size of the slab entry. The free slab pool can be cached on the NIC. The cache syncs

with the host memory in batches of slab entries. Amortized by batching, less than 0.07 DMA operation is needed per allocation or deallocation. When a small slab pool is almost empty, larger slabs need to be split. Because the slab type is already included in a slab entry, in *slab splitting*, slab entries are simply copied from the larger pool to the smaller pool, without the need for computation. Including slab type in the slab entry also saves communication cost because one slab entry may contain multiple slots.

On deallocation, the slab allocator needs to check whether the freed slab can be merged with its neighbor, requiring at least one read and write to the allocation bitmap. Inspired by garbage collection, we propose *lazy slab merging* to merge free slabs in batch when a slab pool is almost empty and no larger slab pools have enough slabs to split.

*3.3.3 Out-of-Order Execution Engine.* Dependency between two KV operations with the same key in the KV processor will lead to data hazard and pipeline stall. This problem is magnified in single-key atomics where all operations are dependent, thus limiting the atomics throughput. We borrow the concept of dynamic scheduling from computer architecture and implement a *reservation station* to track all in-flight KV operations and their *execution context*. To saturate PCIe, DRAM and the processing pipeline, up to 256 in-flight KV operations are needed. However, comparing 256 16-byte keys in parallel would take 40% logic resource of our FPGA. Instead, we store the KV operations in a small hash table in on-chip BRAM, indexed by the hash of the key. To simplify hash collision resolution, we regard KV operations with the same hash as dependent, so there may be false positives, but it will never miss a dependency. Operations with the same hash are organized in a chain and examined sequentially. Hash collision would degrade the efficiency of chain examination, so the reservation station contains 1024 hash slots to make hash collision probability below 25%.

The reservation station not only holds pending operations, but also caches their latest values for *data forwarding*. When a KV operation is completed by the main processing pipeline, its result is returned to the client, and the latest value is forwarded to the reservation station. Pending operations in the same hash slot are checked one by one, and operations with matching key are executed immediately and removed from the reservation station. For atomic operations, the computation is performed in a dedicated execution engine. For write operations, the cached value is updated. The execution result is returned to the client directly. After scanning through the chain of dependent operations, if the cached value is updated, a PUT operation is issued to the main processing pipeline for cache write back. This data forwarding and fast execution path enable single-key atomics to be processed one operation per clock cycle (180 Mops), eliminate head-of-line blocking

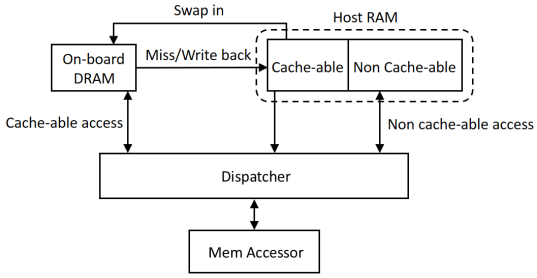



**Figure 7: DRAM load dispatcher.**

under workload with popular keys, and ensure consistency because no two operations on the same key can be in the main processing pipeline simultaneously.

*3.3.4 DRAM Load Dispatcher.* To further save the burden on PCIe, we dispatch memory accesses between PCIe and the NIC on-board DRAM. Our NIC DRAM has 4 GiB size and 12.8 GB/s throughput, which is an order of magnitude smaller than the KVS storage on host DRAM (64 GiB) and slightly slower than the PCIe link (14 GB/s). One approach is to put a fixed portion of the KVS in NIC DRAM. However, the NIC DRAM is too small to carry a significant portion of memory accesses. The other approach is to use the NIC DRAM as a cache for host memory, the throughput would degrade due to the limited throughput of our NIC DRAM.

We adopt a hybrid solution to use the DRAM as a cache for a fixed portion of the KVS in host memory, as shown in Figure 7. The cache-able part is determined by the hash of memory address, in granularity of 64 bytes. The hash function is selected so that a bucket in hash index and a dynamically allocated slab have an equal probability of being cache-able. The portion of cache-able part in entire host memory is called *load dispatch ratio* ($l$). Assume the cache hit probability is $h(l)$. To balance load on PCIe and NIC DRAM, the load dispatch ratio $l$ should be optimized so that:

$$\frac{l}{tput_{DRAM}} = \frac{(1-l) + l \cdot (1 - h(l))}{tput_{PCIe}}$$

let $k$ be the ratio of NIC memory size and host memory size. Under uniform workload, cache hit probability $h(l) = \frac{\text{NIC memory size}}{\text{cache-able corpus size}} = \frac{k}{l}$ when $k \le l$. Caching under uniform workload is not efficient. Under long-tail workload with Zipf distribution, assume $n$ is the total number of KVs, approximately $h(l) = \frac{\log(\text{NIC memory size})}{\log(\text{cache-able corpus size})} = \frac{\log(kn)}{\log(ln)}$ when $k \le l$. Under long-tail workload, the cache hit probability is as high as 0.7 with 1M cache in 1G corpus. An optimal $l$ can be solved numerically, as discussed in §6.3.1.

# 4 IMPLEMENTATION

Our hardware platform is built on an Intel Stratix V FPGA based programmable NIC (§2.3). The programmable NIC is attached to the server through two PCIe Gen3 x8 links in

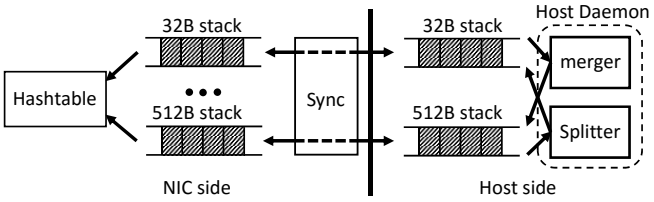
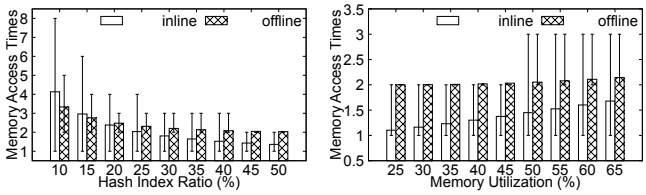



**Figure 8: Slab memory allocator.**




**(a) Fix memory utilization 0.5.** **(b) Fix hash index ratio 0.5.**

**Figure 9: Memory access count under different memory utilizations or hash index ratio.**

a bifurcated x16 physical connector, and contains 4 GiB of on-board DRAM with a single DDR3-1600 channel.

For development efficiency, we use Intel FPGA SDK for OpenCL [2] to synthesize hardware logic from OpenCL. Our KV processor is implemented in 11K lines of OpenCL code and all kernels are fully pipelined, *i.e.*, the throughput is one operation per clock cycle. With 180 MHz clock frequency, our design can process KV operations at 180 M op/s if not bottlenecked by network, DRAM or PCIe.

Below highlights several implementation details.

**Slab Memory Allocator.** As shown in Figure 8, for each slab size, the slab cache on the NIC is synchronized with host DRAM using two double-ended stacks. For the NIC-side double-ended stack (left side in Figure 8), the left end is popped and pushed by the allocator and deallocator, and the right end is synchronized with the left end of the corresponding host-side stack via DMA. The NIC monitors the size of NIC stack and synchronizes to or from the host stack according to high and low watermarks. Host daemon periodically checks the size of host-side double-ended stack. If it grows above a high watermark, slab merging is triggered; when it drops below a low watermark, slab splitting is triggered. Because each end of a stack is either accessed by the NIC or the host, and the data is accessed prior to moving pointers, race conditions would not occur.

**DRAM Load Dispatcher.** One technical challenge is the storage of metadata in DRAM cache, which requires additional 4 address bits and one dirty flag per 64-byte cache line. Cache valid bit is not needed because all KVS storage is accessed exclusively by the NIC. To store the 5 metadata bits per cache line, extending the cache line to 65 bytes would reduce DRAM performance due to unaligned access; saving the metadata elsewhere will double memory accesses. Instead, we leverage spare bits in ECC DRAM for metadata storage. ECC DRAM typically has 8 ECC bits per 64 bits of data. For Hamming code to correct one bit of error in 64 bits of data, only 7 additional bits are required. The 8th ECC bit is a parity bit for detecting double-bit errors. As we access DRAM in 64-byte granularity and alignment, there are 8 parity bits per 64B data. We increase the parity checking granularity from 64 data bits to 256 data bits, so double-bit errors can still be detected. This allows us to have 6 extra bits which can save our address bits and dirty flag.

**Vector Operation Decoder.** Compared with PCIe, network is a more scarce resource with lower bandwidth (5 GB/s) and higher latency (2 $\mu$s). An RDMA write packet over Ethernet has 88 bytes of header and padding overhead, while a PCIe TLP packet has only 26 bytes of overhead. This is why previous FPGA-based key-value stores [5, 6] have not saturated the PCIe bandwidth, although their hash table designs are less efficient than KV-Direct. This calls for *client-side batching* in two aspects: batching multiple KV operations in one packet and supporting vector operations for a more compact representation. Towards this end, we implement a decoder in the KV-engine to unpack multiple KV operations from a single RDMA packet. Observing that many KVs have a same size or repetitive values, the KV format includes two flag bits to allow copying key and value size, or the value of the previous KV in the packet. Fortunately, many significant workloads (*e.g.* graph traversal, parameter server) can issue KV operations in batches. Looking forward, batching would be unnecessary if higher-bandwidth network is available.

## 5 EVALUATION

In this section, we first take a reductionist perspective to support our design choices with microbenchmarks of key components, then switch to a holistic approach to demonstrate the overall performance of KV-Direct in system benchmark.

We evaluate KV-Direct in a testbed of eight servers and one Arista DCS-7060CX-32S switch. Each server equips two 8 core Xeon E5-2650 v2 CPUs with hyper-threading disabled, forming two NUMA nodes connected through QPI Link. Each NUMA node is populated with 8 DIMMs of 8 GiB Samsung DDR3-1333 ECC RAM, resulting a total of 128 GiB of host memory on each server. A programmable NIC [10] is connected to the PCIe root complex of CPU 0, and its 40 Gbps Ethernet port is connected to the switch. The programmable NIC has two PCIe Gen3 x8 links in a bifurcated Gen3 x16 physical connector. The tested server equips SuperMicro X9DRG-QF motherboard and one 120 GB SATA SSD running Archlinux (kernel 4.11.9-1).

For system benchmark, we use YCSB workload [15]. For skewed Zipf workload, we choose skewness 0.99 and refer it as *long-tail* workload.

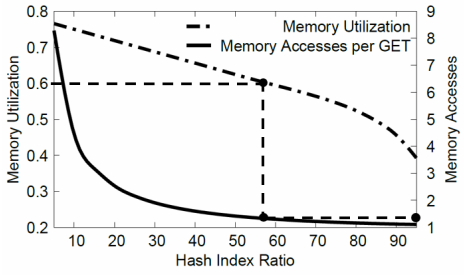
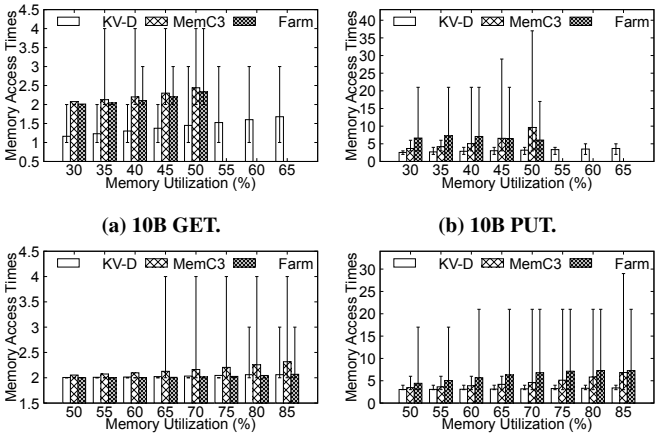



**Figure 10: Determine the optimal hash index ratio for a required memory utilization and KV size.**




**(a) 10B GET.** **(b) 10B PUT.**

**(c) 254B GET.** **(d) 254B PUT.**

**Figure 11: Memory accesses per KV operation.**

## 5.1 Microbenchmarks

*5.1.1 Hash Table.* There are two free parameters in our hash table design: (1) inline threshold, (2) the ratio of hash index in the entire memory space. As shown in Figure 9a, when hash index ratio grows, more KV pairs can be stored inline, yielding a lower average memory access time. Figure 9b shows the increase of memory accesses as more memory is utilized. As shown in Figure 10, the maximal achievable memory utilization drops under higher hash index ratio, because less memory is available for dynamic allocation. Consequently, aiming to accommodate the entire corpus in a given memory size, the hash index ratio has an upper bound. We choose this upper bound and get a minimal average memory access times, shown as the dashed line in Figure 10.

In Figure 11, we plot the number of memory accesses per GET and PUT operation for three possible hash table designs: chaining in KV-Direct, bucket cuckoo hashing in MemC3 [23] and chain-associative hopscotch hashing in FaRM [18]. For KV-Direct, we make the optimal choice of inline threshold and hash index ratio for the given KV size and memory utilization requirement. For cuckoo and hopscotch hashing, we assume that keys are inlined and can be compared in parallel, while the values are stored in dynamically allocated slabs.

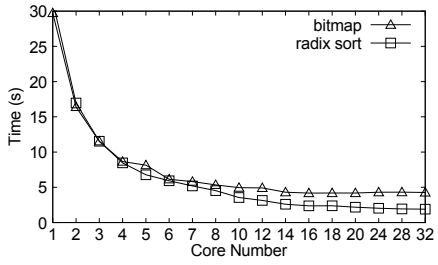



**Figure 12: Execution time of merging 4 billion slab slots.**

Since the hash table of MemC3 and FaRM cannot support more than 55% memory utilization for 10B KV size, the three rightmost bars in Figure 11a and Figure 11b only show the performance of KV-Direct.

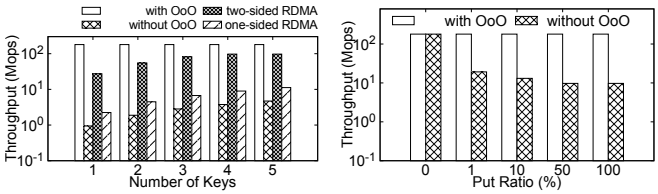
For inline KVs, KV-Direct has close to 1 memory access per GET and close to 2 memory accesses per PUT under non-extreme memory utilizations. GET and PUT for non-inline KVs have one additional memory access. Comparing KV-Direct and chained hopscotch hashing under high memory utilization, hopscotch hashing performs better in GET, but significantly worse in PUT. Although KV-Direct cannot guarantee worst case DMA accesses, we strike for a balance between GET and PUT. Cuckoo hashing needs to access up to two hash slots on GET, therefore has more memory accesses than KV-Direct under most memory utilizations. Under high memory utilization, cuckoo hashing incurs large fluctuations in memory access times per PUT.

*5.1.2 Slab Memory Allocator.* The communication overhead of slab memory allocator comes from the NIC accessing available slab queues in host memory. To sustain the maximal throughput of 180M operations per second, in the worst case, 180M slab slots need to be transferred, consuming 720 MB/s PCIe throughput, *i.e.*, 5% of total PCIe throughput of our NIC.

The computation overhead of slab memory allocator comes from slab splitting and merging on host CPU. Fortunately, they are not frequently invoked. For workloads with stable KV size distributions, newly freed slab slots are reused by subsequent allocations, therefore does not trigger splitting and merging.

Slab splitting requires moving continuous slab entries from one slab queue to another. When the workload shifts from large KV to small KV, in the worst case the CPU needs to move 90M slab entries per second, which only utilizes 10% of a core because it is simply continuous memory copy.

Merging free slab slots to larger slots is rather a time-consuming task, because it involves filling the allocation bitmap with potentially random offsets and thus requiring random memory accesses. To sort the addresses of free slabs and merge continuous ones, radix sort [66] scales better to multiple cores than simple bitmap. As shown in Figure 12,




(a) Atomics. (b) Long-tail workload.

**Figure 13: Effectiveness of out-of-order execution engine.**

merging all 4 billion free slab slots in a 16 GiB vector requires 30 seconds on a single core, or only 1.8 seconds on 32 cores using radix sort [66]. Although garbage collecting free slab slots takes seconds, it runs in background without stalling the slab allocator, and practically only triggered when the workload shifts from small KV to large KV.

*5.1.3 Out-of-Order Execution Engine.* We evaluate the effectiveness of out-of-order execution by comparing the throughput with the simple approach that stalls the pipeline on key conflict, under atomics and long-tail workload. One-sided RDMA and two-sided RDMA [37] throughputs are also shown as baselines.

Without this engine, an atomic operation needs to wait for PCIe latency and processing delay in the NIC, during which subsequent atomic operations on the same key cannot be executed. This renders a single-key atomics throughput of 0.94 Mops in Figure 13a, consistent with 2.24 Mops measured from an RDMA NIC [37]. The higher throughput of RDMA NIC can be attributed to its higher clock frequency and lower processing delay. With out-of-order execution, single-key atomic operations in KV-Direct can be processed at peak throughput, *i.e.*, one operation per clock cycle. In MICA [51], single-key atomics throughput cannot scale beyond a single core. Atomic fetch-and-add can be spread to multiple cores in [37], but it relies on the commutativity among the atomics and therefore does not apply to non-commutative atomics such as compare-and-swap.

With out-of-order execution, single-key atomics throughput improves by 191x and reaches the clock frequency bound of 180 Mops. When the atomic operations spread uniformly among multiple keys, the throughput of one-sided RDMA, two-sided RDMA and KV-Direct without out-of-order execution grow linearly with the number of keys, but still far from the optimal throughput of KV-Direct.

Figure 13b shows the throughput under the long-tail workload. Recall that the pipeline is stalled when a PUT operation finds any in-flight operation with the same key. The long-tail workload has multiple extremely popular keys, so it is likely that two operations with the same popular key arrive closely in time. With higher PUT ratio, it is more likely that at least one of the two operations is a PUT, therefore triggering a pipeline stall.

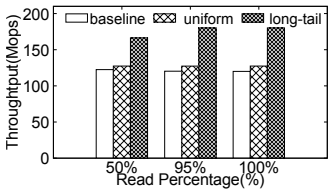
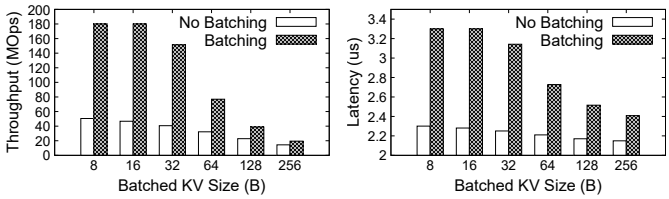



**Figure 14: DMA throughput with load dispatch (load dispatch ratio = 0.5).**

| Vector Size (B) | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| Vector update with return | 11.52 | 11.52 | 11.52 | 11.52 | 11.52 |
| Vector update without return | 4.37 | 4.53 | 4.62 | 4.66 | 4.68 |
| One key per element | 2.09 | 2.09 | 2.09 | 2.09 | 2.09 |
| Fetch to client | 0.03 | 0.06 | 0.12 | 0.24 | 0.46 |

**Table 2: Throughput (GB/s) of vector operations with vector update or alternatives.**



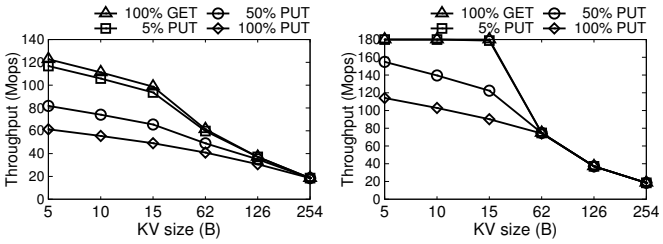

(a) Throughput. (b) Latency.

**Figure 15: Efficiency of network batching.**

*5.1.4 DRAM Load Dispatcher.* Figure 14 shows the throughput improvement of DRAM load dispatch over the baseline of using PCIe only. Under uniform workload, the caching effect of DRAM is negligible because its size is only 6% of host KVS memory. Under long-tail workload, ~30% of memory accesses are served by the DRAM cache. Overall, the memory access throughput for 95% and 100% GET achieves the 180 Mops clock frequency bound. However, if DRAM is simply used as a cache, the throughput would be adversely impacted because the DRAM throughput is lower than PCIe throughput.

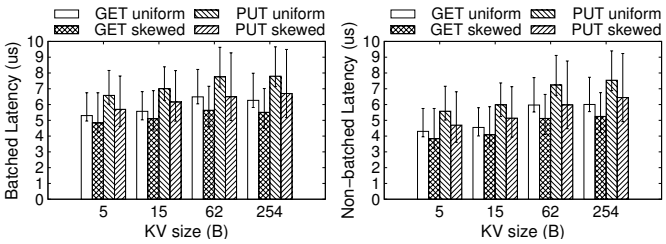
*5.1.5 Vector Operation Decoder.* To evaluate the efficiency of vector operations in KV-Direct, Table 2 compares the throughput of atomic vector increment with two alternative approaches: (1) If each element is stored as a unique key, the bottleneck is the network to transfer the KV operations. (2) If the vector is stored as a large opaque value, retrieving the vector to the client also overwhelms the network. Additionally, the two alternatives in Table 2 do not ensure consistency within the vector. Adding synchronization would incur further overhead.

KV-Direct client packs KV operations in network packets to mitigate packet header overhead. Figure 15 shows that network batching increases network throughput by up to 4x, while keeping networking latency below 3.5 $\mu$s.




(a) Uniform. (b) Long-tail.

**Figure 16: Throughput of KV-Direct under YCSB workload.**




(a) With batching. (b) Without batching.

**Figure 17: Latency of KV-Direct under peak throughput of YCSB workload.**

## 5.2 System Benchmark

*5.2.1 Methodology.* Before each benchmark, we tune hash index ratio, inline threshold and load dispatch ratio according to the KV size, access pattern and target memory utilization. Then we generate random KV pairs with a given size. The key size in a given inline KV size is irrelevant to the performance of KV-Direct, because the key is padded to the longest possible inline KV size during processing. To test inline case, we use KV size that is a multiple of slot size (when size $\leq$ 50, *i.e.* 10 slots). To test non-inline case, we use KV size that is a power of two minus 2 bytes (for metadata). As the last step of preparation, we issue PUT operations to insert the KV pairs into an idle KVS until 50% memory utilization. The performance under other memory utilizations can be derived from Figure 11.

During benchmark, we use an FPGA-based packet generator [44] in the same ToR to generate batched KV operations, send them to the KV server, receive completions and measure sustainable throughput and latency. The processing delay of the packet generator is pre-calibrated via direct loop-back and removed from latency measurements. Error bars represent the $5^{th}$ and $95^{th}$ percentile.

*5.2.2 Throughput.* Figure 16 shows the throughput of KV-Direct under YCSB uniform and long-tail (skewed Zipf) workload. Three factors may be the bottleneck of KV-Direct: clock frequency, network and PCIe/DRAM. For 5B~15B KVs inlined in the hash index, most GETs require one PCIe/DRAM access and PUTs require two PCIe/DRAM accesses. Such tiny KVs are prevalent in many systems. In PageRank, the KV size for an edge is 8B. In sparse logistic regression, the KV size is typically 8B-16B. For sequencers and locks in distributed systems, the KV size is 8B.

Under same memory utilization, larger inline KVs have lower throughput, due to a higher probability of hash collision. 62B and larger KVs are not inlined, so they require an additional memory access. Long-tail workload has higher throughput than uniform workload and able to reach the clock frequency bound of 180 Mops under read-intensive workload, or reach the network throughput bound for ≥62B KV sizes. Under long-tail workload, the out-of-order execution engine merges up to ~15% operations on the most popular keys, and the NIC DRAM has ~60% cache hit rate under 60% load dispatch ratio, which collectively lead to up to 2x throughput as uniform workload. As shown in Table 3, the throughput of a KV-Direct NIC is on-par with a state-of-the-art KVS server with tens of CPU cores.

*5.2.3 Power efficiency.* When the KV-Direct server is at peak throughput, the system power is 121.4 watts (measured on the wall). Compared with state-of-the-art KVS systems in Table 3, KV-Direct is 3x more power efficient than other systems, being the first general-purpose KVS system to achieve 1 million KV operations per watt on commodity servers.

When the KV-Direct NIC is unplugged, an idle server consumes 87.0 watts power, therefore the combined power consumption of programmable NIC, PCIe, host memory and the daemon process on CPU is only 34 watts. The measured power difference is justified since the CPU is almost idle and the server can run other workloads when KV-Direct is operating (we use the same criterion for one-sided RDMA, shown at parentheses of Table 3). In this regard, KV-Direct is 10x more power efficient than CPU-based systems.

*5.2.4 Latency.* Figure 17 shows the latency of KV-Direct under the peak throughput of YCSB workload. Without network batching, the tail latency ranges from 3~9 $\mu$s depending on KV size, operation type and key distribution. PUT has higher latency than GET due to additional memory access. Skewed workload has lower latency than uniform due to more likelihood of being cached in NIC DRAM. Larger KV has higher latency due to additional network and PCIe transmission delay. Network batching adds less than 1 $\mu$s latency than non-batched operations, but significantly improves throughput, which has been evaluated in Figure 15.

*5.2.5 Impact on CPU performance.* KV-Direct is designed to bypass the server CPU and uses only a portion of host memory for KV storage. Therefore, the CPU can still run other applications. Our measurements find a minimal impact on other workloads on the server when a single NIC KV-Direct is at peak load. Table 4 quantifies this impact at the

| KVS | Comment | Bottleneck | Tput (Mops) | | Power Efficiency (Kops/W) | | Avg Delay ($\mu$s) | |
|---|---|---|---|---|---|---|---|---|
| | | | GET | PUT | GET | PUT | GET | PUT |
| Memcached [25] | Traditional | Core synchronization | 1.5 | 1.5 | ~5 | ~5 | ~50 | ~50 |
| MemC3 [23] | Traditional | OS network stack | 4.3 | 4.3 | ~14 | ~14 | ~50 | ~50 |
| RAMCloud [59] | Kernel bypass | Dispatch thread | 6 | 1 | ~20 | ~3.3 | 5 | 14 |
| MICA [51] | Kernel bypass, 24 cores, 12 NIC ports | CPU KV processing | 137 | 135 | 342 | 337 | 81 | 81 |
| FaRM [18] | One-sided RDMA for GET | RDMA NIC | 6 | 3 | ~30 (261) | ~15 | 4.5 | ~10 |
| DrTM-KV [72] | One-sided RDMA and HTM | RDMA NIC | 115.2 | 14.3 | ~500 (3972) | ~60 | 3.4 | 6.3 |
| HERD'16 [37] | Two-sided RDMA, 12 cores | PCIe | 98.3 | ~60 | ~490 | ~300 | 5 | 5 |
| Xilinx'13 [5] | FPGA (with host) | Network | 13 | 13 | 106 | 106 | 3.5 | 4.5 |
| Mega-KV [75] | GPU (4 GiB on-board RAM) | GPU KV processing | 166 | 80 | ~330 | ~160 | 280 | 280 |
| **KV-Direct (1 NIC)** | Programmable NIC, two Gen3 x8 | PCIe & DRAM | 180 | 114 | 1487 (5454) | 942 (3454) | 4.3 | 5.4 |
| **KV-Direct (10 NICs)** | Programmable NIC, one Gen3 x8 each | PCIe & DRAM | 1220 | 610 | 3417 (4518) | 1708 (2259) | 4.3 | 5.4 |

**Table 3: Comparison of KV-Direct with other KVS systems under long-tail (skewed Zipf) workload of 10B tiny KVs. For metrics not reported in the papers, we emulate the systems using similar hardware and report our approximate measurements. For CPU-bypass systems, numbers in parentheses report power difference under peak load and idle.**

| KV-Direct status → | | Idle | Busy |
|---|---|---|---|
| Random Latency | CPU0-0 | 82.2 ns | 83.5 ns |
| | CPU0-1 | 129.3 ns | 129.9 ns |
| | CPU1-0 | 122.3 ns | 122.2 ns |
| | CPU1-1 | 84.2 ns | 84.3 ns |
| Sequential Throughput | **CPU0-0** | **60.3 GB/s** | **55.8 GB/s** |
| | CPU0-1 | 25.7 GB/s | 25.6 GB/s |
| | CPU1-0 | 25.5 GB/s | 25.9 GB/s |
| | CPU1-1 | 60.2 GB/s | 60.3 GB/s |
| Random Throughput | 32B read | 10.53 GB/s | 10.46 GB/s |
| | 64B read | 14.41 GB/s | 14.42 GB/s |
| | 32B write | 9.01 GB/s | 9.04 GB/s |
| | 64B write | 12.96 GB/s | 12.94 GB/s |

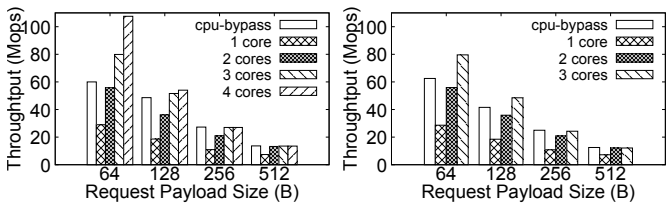
**Table 4: Impact on CPU memory access performance when KV-Direct is at peak throughput. Measured with Intel Memory Latency Checker v3.4.**

peak throughput of KV-Direct. Except for sequential throughput of CPU 0 to access its own NUMA memory (the line marked in bold), the latency and throughput of CPU memory accesses are mostly unaffected. This is because 8 channels of host memory can provide far higher random access throughput than all CPU cores could consume, while the CPU can indeed stress the sequential throughput of the DRAM channels. The impact of the host daemon process is minimal when the distribution of KV sizes is relatively stable, because the garbage collector is invoked only when the number of available slots for different slab sizes are imbalanced.

# 6 EXTENSIONS

## 6.1 CPU-based Scatter-Gather DMA

PCIe has 29% TLP header and padding overhead for 64B DMA operations (§2.4) and the DMA engine may not have enough parallelism to saturate the PCIe bandwidth-delay product with small TLPs (§4). Larger DMA operations with up to 256-byte TLP payload is supported by the PCIe root complex




**(a) Read.** **(b) Write.**

**Figure 18: Scatter-gather performance.**

in our system. In this case, the TLP head and padding overhead is only 9%, and the DMA engine has enough parallelism (64) to saturate the PCIe link with 27 in-flight DMA reads. To batch the DMA operations on PCIe link, we can leverage the CPU to perform scatter-gather (Figure 19). First, the NIC DMAs addresses to a request queue in host memory. The host CPU polls the request queue, performs random memory access, put the data in response queue and writes MMIO doorbell to the NIC. The NIC then fetches the data from response queue via DMA.

Figure 18 shows that CPU-based scatter-gather DMA has up to 79% throughput improvement compared to the CPU-bypassing approach. In addition to the CPU overhead, the primary drawback of CPU-based scatter-gather is the additional latency. To save MMIOs from the CPU to the NIC, we batch 256 DMA operations per doorbell, which requires 10 $\mu$s to complete. The overall latency for the NIC to access host memory using CPU-based scatter-gather is ~20 $\mu$s, almost 20x higher than direct DMA.

## 6.2 Multiple NICs per Server

In some cases, it may be desirable to build a dedicated key-value store with maximal throughput per server. Through simulation, [47] showed the possibility of achieving a billion KV op/s in a single server with four (currently unavailable) 60-core CPUs. As shown in Table 3, with 10 KV-Direct NICs

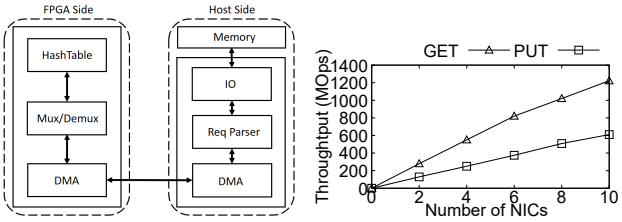



**Figure 19: Scatter-gather architecture.**

**Figure 20: Performance of multiple NICs per server.**

| | 1/1024 | 1/256 | 1/64 | 1/16 | 1/4 | 1 |
|---|---|---|---|---|---|---|
| 1/2 | 0.366 | 0.358 | 0.350 | 0.342 | 0.335 | 0.327 |
| 1 | 0.583 | 0.562 | 0.543 | 0.525 | 0.508 | 0.492 |
| 2 | 0.830 | 0.789 | 0.752 | 0.718 | 0.687 | 0.658 |
| 4 | 1 | 0.991 | 0.933 | 0.881 | 0.835 | 0.793 |
| 8 | 1 | 1 | 1 | 0.995 | 0.937 | 0.885 |

**Table 5: Optimal load dispatch ratio for long-tail workload under different NIC DRAM/PCIe throughput ratio (vertical) and NIC/host memory size ratio (horizontal).**

on a server, the one billion KV op/s performance is readily achievable with a commodity server. The KV-Direct server consumes 357 watts power (measured on the wall) to achieve 1.22 Gop/s GET or 0.61 Gop/s PUT.

In order to saturate the 80 PCIe Gen3 lanes of two Xeon E5 CPUs, we replace the motherboard of the benchmarked server (Sec. 5) with a SuperMicro X9DRX+-F motherboard with 10 PCIe Gen3 x8 slots. We use PCIe x16 to x8 converters to connect 10 programmable NICs on each of the slots, and only one PCIe Gen3 x8 link is enabled on each NIC, so the throughput per NIC is lower than Figure 16. Each NIC owns an exclusive memory region in host memory and serves a disjoint partition of keys. Multiple NICs suffer the same load imbalance problem as a multi-core KVS implementation. Fortunately, for a small number of partitions (*e.g.* 10), the load imbalance is not significant [47, 51]. Under YCSB long-tail workload, the highest-loaded NIC has 1.5x load of the average, and the added load from extremely popular keys is served by the out-of-order execution engine (Sec. 3.3.3). Figure 20 shows that KV-Direct throughput scales almost linearly with the number of NICs on a server.

## 6.3 Discussion

*6.3.1 NIC hardware with different capacity.* The goal of KV-Direct is to leverage existing hardware in data centers to offload an important workload (KV access), instead of designing a special hardware to achieve maximal KVS performance. We use programmable NICs, which usually contain limited amount of DRAM for buffering and connection-state tracking. Large DRAMs are expensive in both die size and power consumption.

Even if future NICs have faster or larger on-board memory, under long-tail workload, our load dispatch design (Sec. 3.3.4) still shows performance gain over the simple design of partitioning the keys uniformly according to NIC and host memory capacity. Table 5 shows the optimal load dispatch ratio for long-tail workload with a corpus of 1 billion keys, under different ratio of NIC DRAM and PCIe throughput and different ratio of NIC and host memory size. If a NIC has faster DRAM, more load will be dispatched to the NIC. A load dispatch ratio of 1 means the NIC memory behaves exactly like a cache of host memory. If a NIC has larger DRAM, a slightly less portion of load will be dispatched to the NIC. As shown in Table 6, even when the size of NIC DRAM is a tiny fraction of host memory, the throughput gain is significant.

| | 1/1024 | 1/256 | 1/64 | 1/16 | 1/4 | 1 |
|---|---|---|---|---|---|---|
| 1/2 | 1.36 | 1.39 | 1.40 | 1.37 | 1.19 | 1.02 |
| 1 | 1.71 | 1.77 | 1.81 | 1.79 | 1.57 | 1.01 |
| 2 | 2.40 | 2.52 | 2.62 | 2.62 | 2.33 | 1.52 |
| 4 | 3.99 | 4.02 | 4.22 | 4.27 | 3.83 | 2.52 |
| 8 | 7.99 | 7.97 | 7.87 | 7.56 | 6.83 | 4.52 |

**Table 6: Relative throughput of load dispatch compared to simple partitioning. Column titles are same as Table 5.**

The hash table and slab allocator design (Sec. 3.3.1) is generally applicable to hash-based storage systems that need to be frugal of random accesses for both lookup and insertion.

The out-of-order execution engine (Sec. 3.3.3) can be applied to all kinds of applications in need of latency hiding, and we hope future RDMA NICs to support that for atomics.

In 40 Gbps networks, network bandwidth bounds non-batched KV throughput, so we use client-side batching (Sec.4). With higher network bandwidth, batch size can be reduced, thus reducing latency. In a 200 Gbps network, a KV-Direct NIC could achieve 180 Mop/s without batching.

KV-Direct leverages widely-deployed programmable NICs with FPGAs [10, 64]. FlexNIC [40, 41] is another promising architecture for programmable NICs with Reconfigurable Match-action Tables (RMT) [8]. NetCache [35] implements a KV cache in RMT-based programmable switches, showing potential for building KV-Direct in an RMT-based NIC.

*6.3.2 Implications for real-world applications.* Back-of-the-envelope calculations show potential performance gains when KV-Direct is applied in end-to-end applications. In PageRank [61], because each edge traversal can be implemented with one KV operation, KV-Direct supports 1.22G TEPS on a server with 10 programmable NICs. In comparison, GRAM [73] supports 250M TEPS per server, bound by interleaved computation and random memory access.

KV-Direct supports user-defined functions and vector operations (Table 1) that can further optimize PageRank by offloading client computation to hardware. Similar arguments hold for parameter server [46]. We expect future work to leverage hardware-accelerated key-value stores to improve distributed application performance.

## 7 RELATED WORK

As an important infrastructure, the research and development of distributed key-value store systems have been driven by performance. A large body of distributed KVS are based on CPU. To reduce the computation cost, Masstree [53], MemC3 [23] and libcuckoo [48] optimize locking, caching, hashing and memory allocation algorithms, while KV-Direct comes with a new hash table and memory management mechanism specially designed for FPGA to minimize the PCIe traffic. MICA [51] partitions the hash table to each core thus completely avoids synchronization. This approach, however, introduces core imbalance for skewed workloads.

To get rid of the OS kernel overhead, [31, 65] directly poll network packets from NIC and [34, 54] process them with the user space lightweight network stack. Key-value store systems [39, 47, 51, 58, 59] benefit from such optimizations for high performance. As a further step towards this direction, recent works [1, 36, 36, 37, 37] leverage the hardware-based network stack of RDMA NIC, using two-sided RDMA as an RPC mechanism between KVS client and server to further improve per-core throughput and reduce latency. Still, these systems are CPU bound (§2.2).

Another different approach is to leverage one-sided RDMA. Pilaf [55] and FaRM [18] adopt one-sided RDMA read for GET operation and FaRM achieves throughput that saturates the network. Nessie [70], DrTM [72], DrTM+R [13] and FaSST [38] leverage distributed transactions to implement both GET and PUT with one-sided RDMA. However, the performance of PUT operation suffer from unavoidable synchronization overhead for consistency guarantee, limited by RDMA primitives [37]. Moreover, client-side CPU is involved in KV processing, limiting per-core throughput to ~10 Mops on the client side. In contrast, KV-Direct extends the RDMA primitives to key-value operations while guarantees the consistency in server side, leaving the KVS client totally transparent while achieving high throughput and low latency even for PUT operation.

As a flexible and customizable hardware, FPGA is now widely deployed in datacenter-scale [10, 64] and greatly improved for programmability [4, 44]. Several early works have explored building KVS on FPGA. But some of them are not practical by limiting the data storage in on-chip (about several MB memory) [50] or on-board DRAM (typically 8GB memory) [11, 32, 33]. [6] focuses on improving system capacity rather than throughput, and adopts SSD as the secondary storage out of on-board DRAM. [11, 50] limit their usage in fixed size key-value pairs, which can only work for the special purpose rather than a general key-value store. [5, 43] uses host DRAM to store the hash table, and [71] uses NIC DRAM as a cache of host DRAM, but they did not optimize for network and PCIe DMA bandwidth, resulting in poor performance. KV-Direct fully utilizes both NIC DRAM and host DRAM, making our FPGA-based key-value store system general and capable of large-scale deployment. Furthermore, our careful hardware and software co-design, together with optimizations for PCIe and networking push the performance to the physical limitation, advancing state-of-art solutions.

Secondary index is an important feature to retrieve data by keys other than the primary key in data storage system [20, 42]. SLIK [42] supports multiple secondary keys using a B+ tree algorithm in key-value store system. It would be interesting to explore how to support secondary index to help KV-Direct step towards a general data storage system. SwitchKV [49] leverages content-based routing to route requests to backend nodes based on cached keys, and NetCache [35] takes a further step to cache KV in the switches. Such load balancing and caching will also benefit our system. Eris [45] leverages network sequencers to achieve efficient distributed transactions, which may give a new life to the one-sided RDMA approach with client synchronization.

## 8 CONCLUSION

In this paper, we describe the design and evaluation of KV-Direct, a high performance in-memory key-value store. Following a long history in computer system design, KV-Direct is another exercise in leveraging reconfigurable hardware to accelerate an important workload. KV-Direct is able to obtain superior performance by carefully co-designing hardware and software in order to remove bottlenecks in the system and achieve performance that is close to the physical limits of the underlying hardware.

After years of broken promises, FPGA-based reconfigurable hardware finally becomes widely available in main stream data centers. Many significant workloads will be scrutinized to see whether they can benefit from reconfigurable hardware, and we expect much more fruitful work in this general direction.

## ACKNOWLEDGEMENTS

We would like to thank Kun Tan, Ningyi Xu, Ming Wu, Jiansong Zhang and Anuj Kalia for all technical discussions and valuable comments. We'd also like to thank the whole Catapult v-team at Microsoft for support on the FPGA platform. We thank our shepherd, Simon Peter, and other anonymous reviewers for their valuable feedback and comments.

## REFERENCES

[1] 2000. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association.
[2] 2017. Altera SDK for OpenCL. (2017). http://www.altera.com/.
[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[4] David F Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63.

[5] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *The 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, San Jose, CA.

[6] Michaela Blott, Ling Liu, Kimon Karras, and Kees A Vissers. 2015. Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory.. In *HotStorage '15*.

[7] Jeff Bonwick and others. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator.. In *USENIX summer*, Vol. 16. Boston, MA, USA.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 99–110.

[9] Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. 2016. Horton tables: fast hash tables for in-memory data-intensive computing. In *USENIX ATC '16*.

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, and others. 2016. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[11] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA memcached appliance. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*. ACM, 245–254.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[13] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Eurosys '16*. ACM.

[14] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. 2016. Invited - Heterogeneous Datacenters: Options and Opportunities. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 16, 16:1–16:6 pages.

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[16] TPC Council. 2010. tpc-c benchmark, revision 5.11. (2010).

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: fast remote memory. In *NSDI '14*.

[19] TV Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active messages: a mechanism for integrated communication and computation. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*. IEEE, 256–266.

[20] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 25–36.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 365–376.

[22] Hadi Esmaeilzadeh, Emily Blem, Renée St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2013. Power challenges may end the multicore era. *Commun. ACM* 56, 2 (2013), 93–102.

[23] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI '13*. 371–384.

[24] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI '17*. Boston, MA, 315–328.

[25] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[26] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1992. *Hiding memory latency using dynamic scheduling in shared-memory multiprocessors*. Vol. 20. ACM.

[27] Albert Greenberg. 2015. SDN for the Cloud. In *Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication*.

[28] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 195–206.

[29] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *International Symposium on Distributed Computing*. Springer, 350–364.

[30] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. 2016. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 456–469.

[31] DPDK Intel. 2014. Data plane development kit. (2014).

[32] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2013. A flexible hash table design for 10gbps key-value stores on fpgas. In *23rd International Conference on Field programmable Logic and Applications*. IEEE, 1–8.

[33] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2015. A hash table for line-rate data processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8, 2 (2015), 13.

[34] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *NSDI '14*. 489–502.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP '17*.

[36] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 295–306.

[37] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC '16*.

[38] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *OSDI '16*. 185–201.

[39] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 9.

[40] Antoine Kaufmann, Simon Peter, Thomas E Anderson, and Arvind Krishnamurthy. 2015. FlexNIC: Rethinking Network DMA.. In *HotOS '15*.

[41] Antoine Kaufmann, Simon Peter, Navven Kumar Sharma, and Thomas Anderson. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21th International Conference on Architectural*

*Support for Programming Languages and Operating Systems*.

[42] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. 2016. SLIK: Scalable low-latency indexes for a key-value store. In *USENIX ATC '16*.

[43] Maysam Lavasani, Hari Angepat, and Derek Chiou. 2014. An FPGA-based in-line accelerator for Memcached. *IEEE Computer Architecture Letters* 13, 2 (2014), 57–60.

[44] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *SIGCOMM '16*. ACM, 1–14.

[45] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *SOSP '17*.

[46] Mu Li, David G Andersen, and Jun Woo Park. 2014. Scaling Distributed Machine Learning with the Parameter Server.

[47] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, Sukhan Lee, Pradeep Dubey, and others. 2016. Full-Stack Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform. *ACM Transactions on Computer Systems (TOCS)* 34, 2 (2016), 5.

[48] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Eurosys '14*. ACM, 27.

[49] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. 2016. Be fast, cheap and in control with SwitchKV. In *NSDI '16*.

[50] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang. 2016. Memory efficient and high performance key-value store on FPGA using Cuckoo hashing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4.

[51] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI '14*. 429–444.

[52] Xiaoyu Ma, Dan Zhang, and Derek Chiou. 2017. FPGA-Accelerated Transactional Execution of Graph Workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, Jonathan W. Greene and Jason Helge Anderson (Eds.). ACM, 227–236.

[53] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 183–196.

[54] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 175–186.

[55] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC '13*. 103–114.

[56] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions.. In *OSDI '14*, Vol. 14. 511–524.

[57] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. 2013. Scaling memcache at facebook. In *NSDI '13*.

[58] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, and others. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.

[59] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, and others. 2015. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015).

[60] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-defined accelerator for large-scale DNN systems. *2014 IEEE Hot Chips 26 Symposium (HCS)* 00 (2014), 1–23.

[61] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[62] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[63] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 307–318.

[64] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, and others. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 13–24.

[65] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[66] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 351–362.

[67] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 505–516.

[68] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM, 15–28.

[69] Herb Sutter. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal* 30, 3 (2005), 202–210.

[70] Tyler Szepesi, Bernard Wong, Ben Cassell, and Tim Brecht. 2014. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *First International Workshop on Rack-scale Computing*.

[71] Yuta Tokusashi and Hiroki Matsutani. 2016. A multilevel NOSQL cache design combining In-NIC and In-Kernel caches. In *High-Performance Interconnects (HOTI '16)*. IEEE, 60–67.

[72] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP '15*. ACM, 87–104.

[73] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 408–421.

[74] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. TuX2: Distributed Graph Computation for Machine Learning. In *NSDI '17*.

[75] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.

[76] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, and others. 2015. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 479–491.