A Multilevel NOSQL Cache Design Combining In-NIC and In-Kernel Caches

Yuta Tokusashi and Hiroki Matsutani
Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa, JAPAN 223-8522
Email: {tokusasi, matutani}@arc.ics.keio.ac.jp

***Abstract*—Since a large-scale in-memory data store, such as key-value store (KVS), is an important software platform for data centers, this paper focuses on an FPGA-based custom hardware to further improve the efficiency of KVS. Although such FPGA-based KVS accelerators have been studied and shown a high performance per Watt compared to software-based processing, since their cache capacity is strictly limited by the DRAMs implemented on FPGA boards, their application domain is also limited. To address this issue, in this paper, we propose a multilevel NOSQL cache architecture that utilizes both the FPGA-based hardware cache and an in-kernel software cache in a complementary style. They are referred as L1 and L2 NOSQL caches, respectively. The proposed multilevel NOSQL cache architecture motivates us to explore various design options, such as cache write and inclusion policies between L1 and L2 NOSQL caches. We implemented a prototype system of the proposed multilevel NOSQL cache using NetFPGA-10G board and Linux Netfilter framework. Based on the prototype implementation, we explore the various design options for the multilevel NOSQL caches. Simulation results show that our multilevel NOSQL cache design reduces the cache miss ratio and improves the throughput compared to the non-hierarchical design.**

***Keywords*-FPGA, Key-value store, NOSQL, Multilevel cache**

I. Introduction

Green computing that maximizes energy efficiency of computers is recognized as one of primary concerns toward a sustainable planet. With an increase in the popularity of cloud computing and social networking services, data centers that accommodate 1k to 100k servers are still expanding in the world. As data center facilities consume a significant amount of power for computers and cooling systems, their energy efficiency should be improved.

Especially, CPUs consume 42% of total power in such data center servers, as reported in [1]. A common and promising approach to significantly improve their energy efficiency is to replace a part of software with an application specific custom hardware. To build such a custom hardware, recently, Field-Programmable Gate Arrays (FPGAs) have been widely deployed for data center applications [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], due to the reconfigurability, low power consumption, and a wide set of IP cores and I/O interfaces supported. For example, Microsoft proposes to take advantage of FPGAs to accelerate the Bing search engine in the next-generation data centers [2]. Another key application in data centers is a distributed data store, such as memcached [13], used as a large-scale data store and memory caching system. Because FPGA devices can be tightly coupled with I/O subsystems, such as high-speed network interfaces [14], [15], their application to memcached has been extensively studied recently [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [16]. An experimental result in [5] shows that an FPGA-based standalone (i.e., FPGA board only) memcached accelerator improves the performance per Watt by 36.4x compared to an 8-core Intel Xeon processor. Even with a host, it improves the performance per Watt by 15.2x. However, a serious limitation of such FPGA-based memcached accelerators is that their cache capacity is limited by DRAM capacity mounted on the FPGA boards. As a DRAM module typically has more than 200 I/O pins (e.g., 204 pins for DDR3 SO-DIMM package), the number of DRAM modules handled by a single FPGA cannot be increased easily, as mentioned in [12]. As DRAM capacity for a host main memory is growing, the capacity gap between host memory and FPGA-based NIC should be addressed.

In addition to such FPGA-based solutions, software-based optimizations have been studied to improve the performance of data stores [17], [18], [19], [20], [21], [22], [23]. A latency breakdown of memcached reported in [4] shows that a packet processing time for NIC and kernel network protocol stack is longer than that spent in memcached software. Actually these software-based optimizations mainly focus on how to reduce the processing time for kernel network protocol stack and they can be classified into two approaches: kernel bypassing and in-kernel processing. The first approach bypasses the kernel network protocol stack by a dedicated software framework, such as Intel DPDK [24] and netmap [25], so that a data store application running on user-space can access NIC devices directly. The second approach moves the data store processing from user-space to kernel-space [20]. Both the approaches can utilize a large host main memory and remove the kernel overheads related to network protocol stack and system calls. However, the FPGA-based in-NIC cache solutions [4], [5], [6] where memcached operations are processed inside NICs without any software processing, would be advantageous in terms of efficiency.

In this paper, we propose a multilevel NOSQL [1][26] cache architecture that utilizes both the FPGA-based in-NIC cache and the in-kernel key-value cache in a complementary style. The former cache is referred to as Level-1 (L1) NOSQL cache and the latter is referred to as L2 NOSQL cache. Since memcached workload has a strong data access locality as analyzed in [27], the proposed multilevel NOSQL cache can fully exploit high energy-efficiency of in-NIC processing, while addressing the capacity limitation by the in-kernel cache that utilizes a huge host main memory. This is the first paper that focuses on the multilevel NOSQL cache architecture that combines L1 in-NIC cache and L2 in-kernel cache systems for energy-efficient data centers. Our contributions are summarized as follows.

- We propose a multilevel NOSQL cache design.
- We explore various design options for the multilevel NOSQL cache, such as cache write and inclusion policies between L1 and L2 NOSQL caches.
- We implemented a prototype system of the multilevel NOSQL caches using NetFPGA-10G board and Linux Netfilter framework.
- Simulation result based on the prototype shows that the

[1]NOSQL is referred as "Non relational" or "Not Only SQL" data store. It includes key-value stores, column-oriented stores, document-oriented stores, and graph databases. This paper mainly focuses on key-value stores.

TABLE I
SUMMARY OF IN-NIC PROCESSING APPROACHES.

Ref.	Type	Platform	GET operation	SET operation	Storage	Parallelism
[4]	Standalone	FPGA+1GbE	In-NIC	In-NIC	NIC DRAM	Two cores
[3]	Standalone	Dedicated SoC	In-NIC	Embedded CPU	Host DRAM	Single accelerator is depicted
[5], [6]	Standalone	FPGA+10GbE	In-NIC	Host CPU assisted	NIC DRAM	Deep pipeline
[12]	Standalone	FPGA+10GbE	In-NIC	In-NIC	NIC DRAM+SSD	Deep pipeline
This work (L1)	Cache	FPGA+10GbE	In-NIC	In-NIC microcontroller	NIC DRAM	Many cores (crossbar connected)

multilevel NOSQL cache reduces the cache miss ratio and improves throughput compared to non-hierarchical design.

The rest of this paper is organized as follows. Section II introduces related work. Section III proposes the multilevel NOSQL cache architecture that combines L1 in-NIC cache and L2 in-kernel cache and Section IV illustrates our implementation. Section V evaluates the multilevel NOSQL cache and its design options and Section VI summarizes this paper.

II. RELATED WORK

Various approaches have been studied for the performance improvements on memcached, and their solutions can be classified into in-NIC processing approaches and kernel bypassing approaches. We will survey some of them in this section, as the proposed multilevel NOSQL cache that combines the in-NIC and in-kernel caches relies on these approaches.

A. In-NIC Processing Approach

This approach first appeared in [4], which proposes an FPGA-based standalone memcached appliance that utilizes DDR2 memory modules and a 1GbE network interface on an FPGA board. The memcached appliance consists of dedicated hardware modules. Both GET and SET requests are processed by the dedicated hardware modules. It can be parallelized by duplicating the memcached appliance cores.

Another memcached accelerator is designed as a dedicated SoC in [3]. It leverages the hardware prototype proposed in [4] for GET requests, while it relies on general-purpose embedded processors for the remaining functionalities, such as memory allocation, key-value pair eviction and replacement, logging, and error handling.

Another notable FPGA-based standalone memcached accelerator was proposed in [5], [6]. It leverages DDR3 memory modules and a 10GbE network interface on an FPGA board. To fully exploit the parallelism of memcached requests, the request parsing, hash table access, value store access, and response formatting are pipelined deeply. GET requests are processed by the pipelined hardware, while a host CPU assists with memory management functionalities required for SET requests. To handle key collisions by hardware, up to eight keys mapped to the same hash table index are looked up in parallel (i.e., 8-way). Recently, this design is extended to support SATA3 SSDs in addition to DRAM as storage [12] so that key-value pairs are stored in SSD or DRAM regions, depending on the value length.

Table I summarizes the above-mentioned existing designs and our L1 cache design of the multilevel NOSQL cache. The existing designs can be used as a standalone memcached server, while we will use the dedicated hardware as an L1 cache of the proposed multilevel NOSQL cache. Since we assume that complete NOSQL servers are running on an application layer, the dedicated hardware is operated just as an L1 cache and sophisticated functionalities (e.g., logging, error handling, and data replication) can be left to the software NOSQL servers. In the existing designs, GET requests are processed by a dedicated hardware, while SET requests are processed by software or hardware, depending on how a complicated memory management is implemented. In our L1 NOSQL cache, a microcontroller is implemented inside an FPGA NIC to process SET requests without any host CPU assistance. We will illustrate our L1 cache design of the multilevel NOSQL cache in Section III-A.

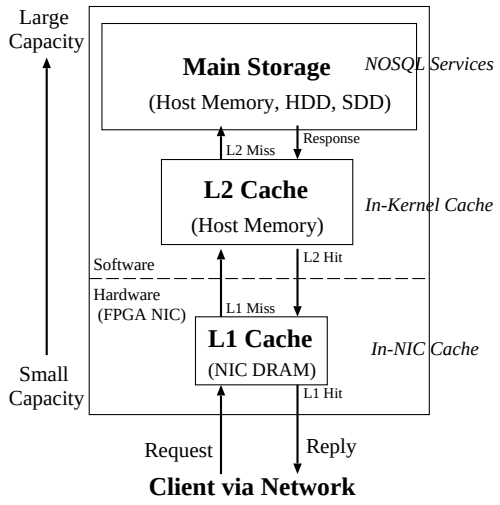


Fig. 1. Relationship between L1 and L2 NOSQL caches.

B. Kernel Bypassing Approach

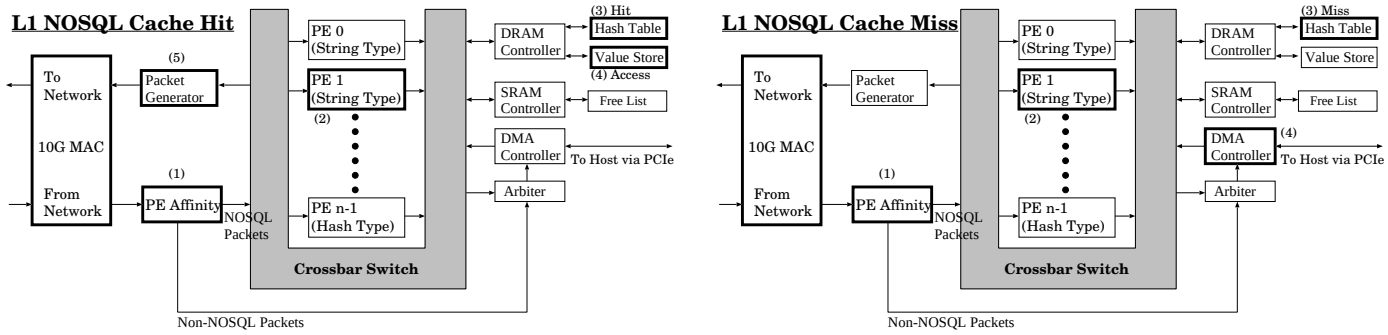
To improve the throughput of key-value store (KVS), a holistic approach that includes the parallel data access, network access, and data structure design was proposed in [17]. As the network access, Intel DPDK is used so that the server software can directly access NIC by bypassing the network protocol stack to minimize the packet I/O overhead. We mainly take into account the network access optimization (e.g., kernel bypassing) in this paper. The other optimizations are orthogonal to the multilevel NOSQL cache and can be applied for further efficiency.

As proposed in [20], moving the KVS into the OS kernel is an alternative approach to remove most of the overhead associated with the network stack and system calls. In a kernel layer, received packets are hooked by Netfilter framework and only KVS queries are retrieved. The retrieved queries are processed inside the kernel with an in-kernel hash table. The response packet is generated and sent back to the device driver.

In our multilevel NOSQL cache architecture, since we assume that complete NOSQL servers are running on an application layer and thus sophisticated functionalities, such as data replication, are left to the NOSQL servers, the in-kernel cache approach is embedded as an L2 cache in the proposed NOSQL cache hierarchy. L2 cache design will be illustrated in Section III-B.

III. NOSQL CACHE ARCHITECHTURE

Figure 1 illustrates our multilevel NOSQL cache architecture that complementally combines the in-NIC and in-kernel caches in order to fully exploit high energy-efficiency of in-NIC processing, while addressing the capacity limitation by the in-kernel cache that utilizes a huge host main memory. We assume that one or more NOSQL databases, such as KVS, column-oriented store,



(a) Cache hit case (b) Cache miss case

Fig. 2. Heterogeneous multi-PE design of L1 NOSQL cache and its behavior (two cases).

and document-oriented store, are running as software servers on a machine, where FPGA-based network interfaces (FPGA NICs) are mounted for receiving and responding NOSQL queries. On-board DRAM of the FPGA NICs is used as the L1 NOSQL cache, while a host main memory allocated by the kernel module is used as the L2 NOSQL cache.

When the FPGA NIC receives a packet, the received packet header is examined, and if it is a NOSQL query, it is processed inside the FPGA NIC; otherwise it is transferred to an Ethernet device driver as well as common TCP/IP packets. For NOSQL queries, a key-value pair is extracted from the packet and the corresponding key is looked up from a hash table in the FPGA NIC. If the key is found in the hash table, the value stored in the on-board DRAM is returned to the requestor (i.e., L1 NOSQL cache hit); otherwise it is transferred to an Ethernet device driver as well as common TCP/IP packets (i.e., L1 NOSQL cache miss). In the Ethernet device driver of our L2 NOSQL cache, the received packet header is examined again, and if it is a NOSQL query, it is processed inside the in-kernel cache module; otherwise it is transferred to a standard network protocol stack as well as common packets. For NOSQL queries, a key-value pair is extracted from the packet and the corresponding key is looked up from a hash table in the in-kernel cache. If the key is found in the hash table, the value stored in the in-kernel cache is returned to the requestor (i.e., L2 NOSQL cache hit); otherwise it is transferred to a network protocol stack as usual (i.e., L2 NOSQL cache miss). In the case of L2 NOSQL cache miss, the NOSQL query is transferred to an application layer and processed by a corresponding NOSQL software server.

A. L1 NOSQL Cache

Figure 2 shows a datapath of our L1 NOSQL cache implemented in an FPGA NIC. Only packets with NOSQL queries (i.e., NOSQL packets) are extracted based on their service destination port number and only NOSQL packets are transferred to dedicated PEs (e.g., PE1) for Hash Table lookup. The other packets are transmitted to a host machine with a DMA controller via PCI-Express. Thus, an arbiter is implemented in front of the DMA controller to arbitrate two input sources: NOSQL packets which are not hit in Hash Table (i.e., L1 NOSQL cache miss packets) and non-NOSQL packets.

L1 NOSQL cache stores key-value pairs in the on-board DRAM modules of FPGA NIC. The key and value parts are typically processed as variable-length data. As surveyed in Section II, existing in-NIC KVS accelerators [5], [6] employ a single deep pipeline in which value parts are processed as variable-length binary data. However, a wide diversity of value lengths (e.g., 4B to 1MB) is observed in a memcached workload analysis [27]. Various value types, such as string, list, hash, set,

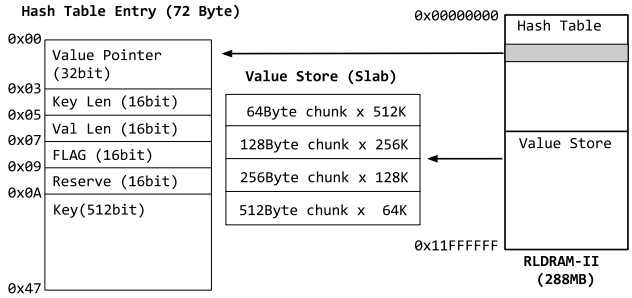


Fig. 3. Hash Table and Value Store implemented on DRAM.

and sorted set [28], are useful for practical applications. Because their processing cycles differ significantly depending on their value types, we built an optimized PE core for each value type, rather than a single deep pipeline where various key-value pairs are uniformly processed. Although we have already implemented various PE cores, since scope of this paper is the multilevel cache design, here we instantiate only the string PE core optimized for string-type value.

As shown in Figure 2, multiple PE cores are connected to a crossbar switch. PE Affinity module is in charge of packet classification. It receives packets from Ethernet MAC (Media Access Control) and checks their destination port number. If the destination port number of a received packet is matched to one of NOSQL service port numbers, PE Affinity module passes the packet to one of PEs which are currently idle. A DRAM controller is also connected to the crossbar switch. It is accessed by the PE cores when they access the Hash Table and Value Store in DRAM modules on the FPGA NIC board.

Memory management is in charge of allocating free memory chunks and freeing unused ones. For example, a write request (e.g., SET operation) needs to allocate a free memory chunk in Value Store to store the new value. We employ Slab Allocator for memory management in Value Store. Slab Allocator manages fixed-length memory chunks of several sizes where values are stored, as shown in Figure 3.

When it receives a SET query that stores a new value, an unused chuck with minimum size where the new value can be fit is removed from the Free List and then used. Assuming, for example, a SET query contains a 24B new value, Slab Allocator in Figure 3 allocates a 64B chunk to store the 24B new value. Such a memory management is one of complicated functions when we implement it as a dedicated hardware. In our L1 NOSQL cache, Slab Allocator is implemented as a microcode running on a tiny soft-CPU processor in each PE core to access Free List of each chunk size.

Figure 2 also illustrates its behavior of L1 NOSQL cache in the two cases where a requested key is hit and missed on the Hash Table, respectively. Hash Table is used to store pairs of a key and a start address of the chuck where the corresponding value is stored. A hashed value of a key is used as an index (address) of the Hash Table. To read or write the chunks stored in Value Store, a PE core performs the following steps.

1) An index in the Hash Table is calculated by hashing the requested key.
2) A content from the Hash Table is read based on the index. Then the requested key and the key read from the Hash Table are compared.
3) If both the keys are identical, a value is read from the Value Store based on the start address of the chunk (Figure 2 (a)). Otherwise, the requested key does not exist in the Value Store (Figure 2 (b)).

L1 NOSQL cache miss occurs when the hashed value of the requested key does not exist or it is conflicted with that of the other keys. To mitigate the hash conflicts, we will introduce set-associative design in Section III-C.

B. L2 NOSQL Cache

In [20], a key-value data store that uses a host memory as a storage is implemented in Linux kernel space. KVS queries received by the kernel are hooked so that a customized handler is called in order to process the KVS queries inside the kernel. Such in-kernel processing can improve the KVS performance compared to the original user space implementation, because the network protocol processing and related system calls can be eliminated. As a result, about 3.3Mops performance in USR trace is achieved as reported in [20]. In our multilevel NOSQL cache design, as an L2 NOSQL cache, a similar in-kernel cache is implemented in Linux kernel using Netfilter framework.

C. NOSQL Cache Hierarchy

Here, we discuss design options for L1 and L2 NOSQL cache hierarchy. More specifically, the following points are introduced and their pros and cons are discussed in this section. They will be evaluated in Section V in terms of NOSQL cache miss ratio.

- Cache write policies between L1 and L2 NOSQL caches (i.e., write-through vs. write-back)
- Cache associativities on Hash Table of L1 NOSQL cache
- Inclusion policies between L1 and L2 NOSQL caches (i.e., inclusive and non-inclusive)
- Slab size configurations of L1 NOSQL cache (i.e., the number of free memory blocks for each size)

1) Write-Back vs. Write-Through: Cache write policy is an important design choice for multilevel cache design. In our multilevel NOSQL cache, write-back policy can reduce the traffic amount between L1 and L2 NOSQL caches, because written data are not transferred from L1 to L2 NOSQL caches until modified (i.e., dirty) cache blocks in the L1 NOSQL cache are evicted. Although in write-back policy the cache blocks in L1 NOSQL cache are not consistent with those in L2 NOSQL cache internally, such inconsistency is never exposed to applications. In write-through policy, cache blocks in L2 NOSQL cache are updated whenever those in L1 NOSQL cache are updated, and thus the traffic amount between L1 and L2 NOSQL caches increases. The problem of such write-through policy is that the write performance of L1 NOSQL cache is restricted by the L2 NOSQL cache bandwidth. We will evaluate the write-back and write-through policies in Section V-A.

2) Cache Associativities on Hash Table: As illustrated in Section III-A, to access cached key-value data in Value Store, the key is hashed to compute the index in the Hash Table where a start address of the cached key-value data in Value Store is stored. There is a possibility that different keys generate an identical hashed value, but only one of them can be stored in Hash Table and the others will be evicted. Such a situation is called a hash conflict.

To avoid hash conflicts and NOSQL cache misses, we can increase the associativity of Hash Table. That is, in the n-way set associative Hash Table, up to n different keys whose hashed values are identical can be stored in Hash Table. As the number of ways increases, L1 NOSQL cache misses due to the Hash Table conflicts are typically decreased. For example, an 8-way Hash Table is used in Xilinx's FPGA memcached appliance to avoid the hash conflicts [5]. Please note that our design accepts variable-length keys. More specifically, a key with up to 64B size can be stored in a single Hash Table entry, while that larger than 64B is stored in multiple Hash Table entries. We will evaluate the set-associative design of Hash Table in Section V-B.

3) Inclusion vs. Non-Inclusion: Inclusion policy (e.g., inclusive or non-inclusive) is an important design choice for multilevel caches. When our L1 and L2 NOSQL caches are inclusive, cached data in L1 NOSQL cache are guaranteed to be in L2 NOSQL cache. When they are non-inclusive, cached data are guaranteed to be in at most one of L1 and L2 NOSQL caches. Their behaviors are differentiated below.

- Assuming that a GET query is missed at L1 NOSQL cache and hit at L2 NOSQL cache, if non-inclusive policy is enforced, the cached block in L2 NOSQL cache is transferred to L1 NOSQL cache and then the cached block in L2 is deleted. If inclusive policy is enforced in the same situation, the cached block in L2 is not deleted.
- Assuming that an unmodified cached data in L1 NOSQL cache is evicted, if non-inclusive policy is enforced, the cached data in L1 NOSQL cache is transferred to L2 NOSQL cache and then the original cached block in L1 is deleted. If inclusive policy is enforced in the same situation, we do not have to copy the cached block in L1 to L2.
- Assuming that a query is missed at both L1 and L2 NOSQL caches, if inclusive policy is enforced, the requested data retrieved from NOSQL server are required to be cached in both L1 and L2 NOSQL caches. This behavior can be simply implemented, because the requested data retrieved from NOSQL server are naturally transferred to the client machine via Ethernet device driver (i.e., L2 NOSQL cache) and FPGA NIC (i.e., L1 NOSQL cache).

Non-inclusive policy is advantageous in terms of cache efficiency, while inclusive policy is simple to implement in L1 NOSQL cache. However, inclusive policy increases the traffic between L1 and L2 NOSQL caches, which may degrade the throughput. We will evaluate the inclusive and non-inclusive policies in terms of total cache miss ratio when L1 and L2 NOSQL cache sizes are varied in Section V-C.

4) Eviction Policies on Hash Table: In Section III-C2, we have introduced set associativities on Hash Table design to reduce L1 NOSQL cache misses due to hash conflicts. By preparing multiple ways for Hash Table, multiple key-value pairs whose hashed keys are identical can be stored in Hash Table. Assuming that a new key-value pair is going to be stored in Hash Table but all the ways are in use, one of existing ways in Hash Table will be replaced with the new key-value pair. Eviction policy defines which way will be evicted in such situations. For example, the following eviction policies can be used.

TABLE II
L1 NOSQL CACHE DESIGN ENVIRONMENT.

CPU	Intel(R) Core(TM) i5-4460
Host memory	4GB
OS	CentOS release 6.7
Kernel	Linux kernel 2.6.32-504
NIC (FPGA)	NetFPGA-10G

TABLE III
TARGET FPGA BOARD FOR L1 NOSQL CACHE.

Board	NetFPGA-10G
FPGA	Virtex-5 XC5VTX240T
DRAM	288MB RLDRAM-II
SRAM	27MB QDRII SRAM
PCIe	PCIe Gen2 x8
Network I/O	SFP+ x4

- Random: One of ways is randomly selected to be replaced with new key-value pair. A simple implementation without a random generator is that the hashed value of the new key-value pair is used to select one of the ways to be evicted.
- LRU: The least recently used way is selected to be evicted. The design complexity increases when the number of ways is greater than two.

5) Slab Size Configurations in L1 NOSQL Cache: A memcached workload analysis in [27] reports that 90% of the requested data sizes are less than 1KB. The study shows that requested key-value sizes are mostly about 20B in USR workload, while large values with up to 1MB are requested in ETC and APP workloads. As illustrated in Section III-A, in our L1 NOSQL cache, a list of free memory blocks are preliminarily allocated for each value size (e.g., 64B, 128B, 256B). The number of memory blocks preliminarily allocated for each value size should be carefully selected. It can be further optimized if the workload is predictable so that the majority of requested key-value data can be fit to the allocated memory blocks.

L1 NOSQL cache is implemented on an FPGA board. Since L1 NOSQL cache capacity is limited, allocating large memory blocks (e.g., 1MB blocks) may significantly reduce the number of small-sized cache blocks and increases the L1 NOSQL cache miss ratio. Therefore, it may be required to determine the upper limit of memory block sizes allocated. In this case, key-value data larger than the upper limit are not cached in L1 NOSQL cache. We will evaluate various slab configurations of the sizes and numbers of memory blocks in Section V-D.

IV. SYSTEM IMPLEMENTATION

In this section, a prototype implementation of our multilevel NOSQL cache is illustrated. Only the necessary functions of the L1 NOSQL cache are implemented to explore the design choices. More specifically, multiple KVS PEs introduced in Section III-A are implemented in the L1 NOSQL cache.

A. Design Environment

Table II lists the design environment. Our L1 NOSQL cache is implemented on NetFPGA-10G board by partially using the reference NIC design provided by NetFPGA Project [14]. Table III shows the hardware specification. FPGA device used is Xilinx Virtex-5 XC5VTX240T. A 10GbE network interface is used for communication. Hash Table and Value Store in the L1 NOSQL cache are implemented on a 288MB RLDRAM-II memory on the FPGA board [2]. Design tool used is Xilinx ISE 13.4.

[2]The memory capacity of NetFPGA-10G board is very small, but newer FPGA boards have more capacity (e.g., 8GB DDR3 SDRAM for NetFPGA-SUME).

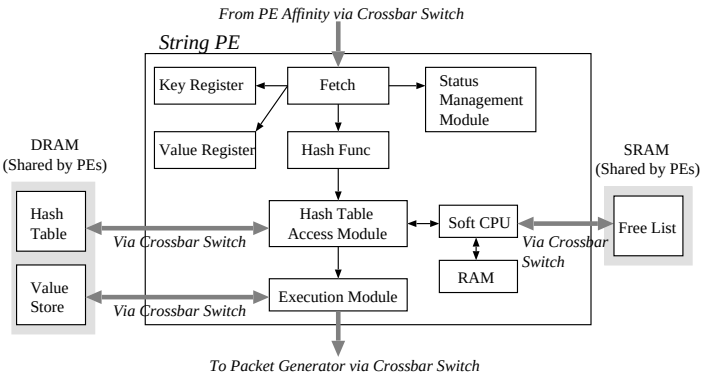


Fig. 4. KVS PE core architecture.

B. Implementation of L1 and L2 NOSQL Caches

We implemented a prototype of an L1 NOSQL cache that consists of KVS PE cores of the string type. The KVS PEs process SET and GET operations. Our design accepts variable-length values, while the key length is fixed to 64B for simplicity. CRC32 is implemented as a hash function. KVS PEs and a DRAM controller are connected via a crossbar switch. A simple fixed-priority arbiter is used for the crossbar switch. Data width of the crossbar is 128bit. We employ UDP as a transport-layer protocol as it is simple and low-overhead [3].

Figure 4 shows a block diagram of a string type KVS PE. Received KVS packets are passed to Fetch module and they are parsed as operation type (e.g., SET, GET, and DELETE), key, and value. In the GET operation, the requested key is examined in the L1 NOSQL cache and if the requested key-value pair does not exist in the cache, the request packet is passed to the crossbar switch and then transferred to the host machine. If the requested key-value pair exists in the L1 NOSQL cache, the response packet is generated by swapping the source/destination IP addresses and source/destination port numbers of the original request packet and adding the requested value.

Slab Allocator is implemented as a microcode running on a MIPS R3000 compatible soft processor. Free List is a list structure that manages unused memory chunks. It is currently implemented as Block RAMs in the FPGA for simplicity, but we will implement it with on-board SRAMs. When the KVS PE executes a SET operation that requires a free memory chunk, it interrupts the soft processor so that the Slab Allocator returns an unused chunk from Free List. In the case of a DELETE operation, Slab Allocator seeks the corresponding chunk and appends it to Free List.

As L2 NOSQL cache, we implemented an in-kernel KVS cache as a loadable kernel module. Netfilter framework is used to process KVS packets in kernel. A customized handler is called when the kernel receives the KVS packets.

C. Area Evaluation

We evaluate the FPGA area utilization of KVS PEs, which are used in L1 NOSQL cache. Horizontal scalability that allows us to add more PEs depending on required performance is an advantage of our heterogenous multi-PE design. Our target device is Xilinx Virtex-5 XC5VTX240T on NetFPGA-10G.

We implemented KVS PEs on the target device. Figure 5 shows the slice utilization of the design. “Reference NIC” shows the slice utilization for the standard 10GbE NIC functions that include four 10G MAC cores and a DMA controller for PCI-Express Gen2 x8. “String PE + Reference NIC” shows that with

[3]UDP is supported in memcached in addition to TCP.

TABLE IV
L1 NOSQL CACHE THROUGHPUT WITH A SINGLE KVS PE.

Query type	Average throughput [Mops]
GET (HIT)	1.42354
GET (MISS)	Determined by L2 NOSQL cache
SET (HIT)	2.20104
SET (MISS)	0.71049

up to eleven PEs and an RLDRAM controller in addition to Reference NIC. Each string PE has a MIPS R3000 compatible processor. Up to eleven PEs can be implemented on the Virtex-5 device or larger devices.

D. Throughput

We evaluate the query processing throughput of L1 NOSQL cache. On the client machine, netmap-based [25] query injector that can fully utilize 10GbE bandwidth is used to generate queries. The following four query types are used for the throughput evaluation. Each type has a key and a value. Their lengths are fixed to 64B.

- "SET (HIT)" generates SET queries which are always hit in L1 NOSQL cache and modify the cache.
- "SET (MISS)" generates SET queries which are always missed in L1 NOSQL cache. In this case, a memory allocation is performed for each query in L1 NOSQL cache and thus the performance will be degraded.
- "GET (HIT)" generates GET queries which are always hit in L1 NOSQL cache by caching the keys to be requested in L1 NOSQL beforehand. The string PE returns response packets that contain the key-value pair requested in the GET queries to the client.
- "GET (MISS)" generates GET queries that never hit in L1 NOSQL cache. Such queries are transferred to L2 NOSQL cache and processed. Thus, its throughput is determined by that of L2 NOSQL cache rather than L1 NOSQL cache.

Table IV shows average throughputs of the four query types with a single KVS PE on L1 NOSQL cache. Throughput of "GET (HIT)" is 1.42Mops (operation per second). Throughput of "SET (HIT)" is 2.20Mops, while that of "SET (MISS)" is 0.71Mops because memory allocation on Value Store is performed for each query. In "SET (MISS)" case, a soft processor running on the KVS PE executes a memory allocation which takes about 200 clock cycles. The KVS PE is stalled during the memory allocation, and thus the throughput is degraded.

We can calculate the expected throughputs when assuming multiple PEs. NetFPGA-10G board is equipped with two RLDRAM-II modules and their aggregate throughput is 38Gbps. Thus, the memory bandwidth including the arbitration for the DRAM controller is not a major performance bottleneck when a given workload is under 38Gbps. An expected aggregated throughput is calculated as $T_{Total} = min(T_{Mem}, T_{Net}, n \times T_{PE})$, where T_{Mem}, T_{Net}, n, and T_{PE} denote the memory bandwdith, network bandwidth, single PE performance, and the number of PEs, respectively. T_{Mem} and T_{Net} are set to 38Gbps and 10Gbps, respectively. T_{PE} is set based on Table IV. Figure 6 shows the result. It shows that 10Gbps network bandwidth is a major source of the performance bottleneck on L1 NOSQL cache. It also shows that seven PEs and nine PEs are required to achieve the 10Gbps GET and SET throughput, respectively.

V. SIMULATION RESULTS

Various design options are available for the proposed multilevel NOSQL cache architecture in terms of the multilevel organizations and cache policies, as proposed in Section III. In this section, we will quantitively explore the proposed design options by using a simulator with real memcached traces so that this paper would be the first guideline to build the multilevel NOSQL cache architecture that utilizes the FPGA-based in-NIC cache and the in-kernel key-value cache.

We generated the memcached traces based on a memcached workload analysis results [27]. In [27], the following five workload classes are analyzed in terms of operation types (e.g., GET, SET, and DELETE) ratio, key size distribution, value size distribution, key appearance, and so on.

- USR : Key sizes are 16B and 21B. Value sizes are 2B.
- SYS : Most key sizes are less than 30B. 70% of value sizes are around 500B.
- APP : 90% of key sizes are 31B. Values are around 270B.
- ETC : Most key sizes are 20-40B. Small amount of values are very large (e.g., 1MB).
- VAR : Key sizes are 32B. 80% of value sizes are 50B.

Since we measured the above-mentioned values from graphs in [27] by hand, these values may contain certain errors.

A. Write-Through vs. Write-Back

Figure 7 compares the write-through and write-back policies in terms of DRAM traffic between L1 and L2 NOSQL caches. In VAR trace, more than 70% of queries are SET operations that update the L1 NOSQL cache. SYS trace also contains a lot of SET operations (i.e., more than 30% of whole queries). In these SET intensive traces, the write-through policy demands a quite high bandwidth, while in the other traces both the policies require a similar bandwidth.

The 10GbE and PCI-Express interfaces would limit the write throughput between these caches. Theoretical DMA transfer capacity is 4GB/s and 7.69GB/s in PCI-Express Gen2 x8 and PCI-Express Gen3 x8, respectively. A PCI-Express Gen3 x8 interface achieves 7.06GB/s throughput in [29]. We assume 10GbE as a network I/O. In the graph, these 10GbE and PCI-Express bandwidth values are shown as horizontal lines.

Although a performance gain of write-back may not be significant in the read-intensive workload, write-back policy can reduce the DMA traffic in write-intensive workload. When assuming a 10GbE network bandwidth, the DMA bandwidth between L1 and L2 NOSQL caches is a bottleneck when write-through policy is used for VAR and SYS traces.

B. Cache Associativity

The multilevel NOSQL caches were simulated by varying the set associativity N of the L1 NOSQL cache, where N = 1, 2, 4, and 8. In all the cases, Hash Table and Value Store sizes of the L1 NOSQL cache are set so as to fit in the RLDRAM-II memory on the FPGA board. Figure 8 shows the simulation results, where X-axis shows the memcached trace and Y-axis shows the L1 NOSQL cache miss ratio. In USR, SYS, and VAR traces, when N is varied from 2 to 8, the cache miss ratio is reduced by 2-7% compared to the direct mapped cache (i.e., N = 1). In ETC trace, the associativity does not affect the miss ratio due to less hash conflicts.

C. Inclusion vs. Non-inclusion

Here we define "L2/L1 capacity ratio" as the ratio of the L2 NOSQL cache capacity against that of L1 NOSQL cache (e.g., the ratio is two when L2 is twice larger than L1). We will discuss inclusion and non-inclusion design options for L1 and L2 NOSQL caches when the L2/L1 capacity ratio is varied. Figure 9 compares the inclusion and non-inclusion options, where X-axis shows the L2/L1 capacity ratio and Y-axis shows the cache miss ratio. When the capacity ratio is 1, the non-inclusion option

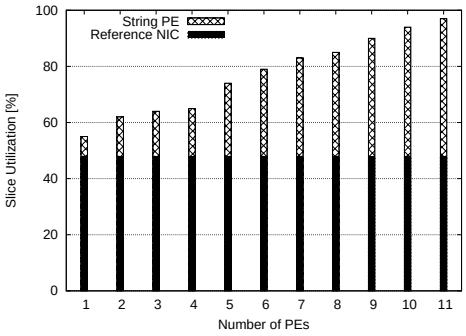


Fig. 5. Area utilization on Virtex-5 XC5VTX240T.

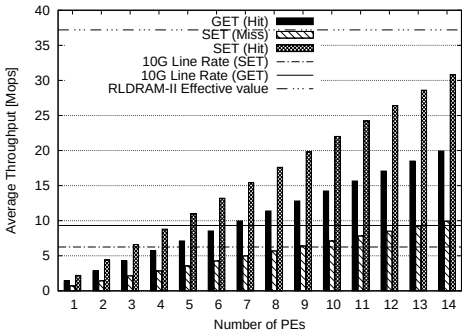


Fig. 6. Aggregated throughput with multiple string PEs on NetFPGA-10G.

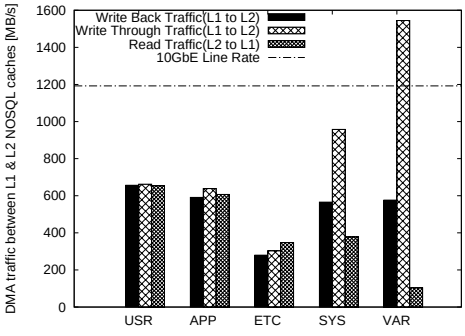


Fig. 7. DMA traffic between L1 and L2 NOSQL caches (write-back vs. write-through).

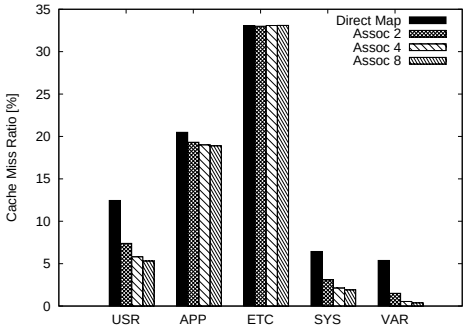
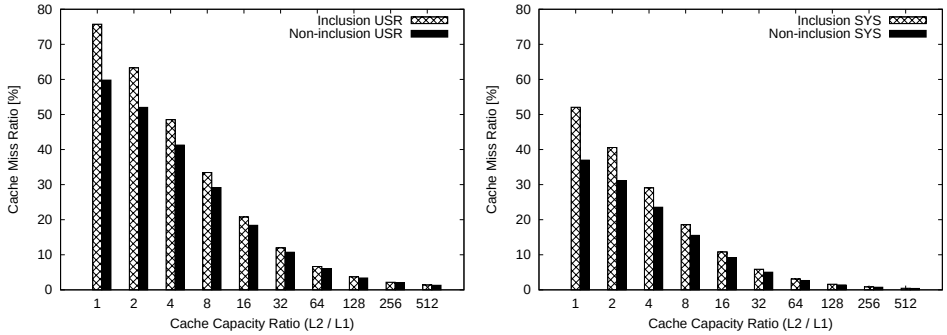


Fig. 8. Cache miss ratio on set-associative L1 NOSQL cache.



(a) USR trace (b) SYS trace

Fig. 9. Inclusive policy vs. non-inclusive policy.

reduces the cache miss ratio by up to 15% compared to the inclusion. Please note that inclusion option when the capacity ratio is 1 implicates the results with only L1 NOSQL cache (no L2 NOSQL cache). In this case, the cache miss ratio is quite high (e.g., 75.8% and 52.1% in USR and SYS, respectively), which demonstrates the necessity of our multilevel NOSQL cache design. When the capacity ratio is over 16, differences between the inclusion and non-inclusion in terms of cache miss ratio become quite small. Assuming L1 NOSQL capacity is 288MB and 8GB, non-inclusion is an efficient option only when L2 NOSQL capacity is less than 4.6GB and 128GB, respectively.

D. Slab Configurations for L1 NOSQL Cache

Our L1 NOSQL cache supports variable-length keys and values. Since the value sizes differ largely, we employ Slab Allocator for the memory allocation. Up to six chunk sizes (a power of 2 from 64B) are supported by configuring the Slab Allocator. Table V shows various slab configurations used in this experiment. Each configuration type has a unique characteristic (e.g., uniform distribution, more small-sized slabs).

Figure 10 shows simulation result of these slab configurations. In USR, more than 90% of queries access small-sized values (e.g., 2B), so the slab configuration that has more small-sized chunks can reduce the cache miss ratio. In APP, the average value size is about 270B. When Type C configuration that does not have large-sized chunks is applied to APP, the cache miss ratio is more than 70% since most requested values cannot be cached in L1 NOSQL cache. In APP, we need 512B chunks to store frequently-requested values; thus Type E that has many large-sized chunks can reduce the cache miss ratio to 40%.

We can improve the cache miss ratio by configuring the chunk sizes and their numbers in L1 NOSQL cache in response to value sizes in an expected workload. In our design, soft-processor in L1 NOSQL cache can configure the chunk sizes and their numbers at a boot time.

TABLE V
CHUNK SIZE CONFIGURATIONS (IN TYPE A, # OF 64B CHUNKS IS 70K).

Type	64B	128B	256B	512B	1kB	2kB	Note
A	70k	70k	70k	70k	70k	70k	Uniform
B	120k	100k	80k	60k	40k	20k	More small sizes
C	160k	140k	120k	0	0	0	No large sizes
D	140k	120k	100k	30k	20k	10k	More small sizes
E	20k	40k	60k	80k	100k	120k	More large sizes

E. Cache Miss Ratio vs. Throughput

Finally, Figure 11 compares the proposed multilevel NOSQL cache and L1 NOSQL only designs in terms of the total throughput when the L1 NOSQL cache miss ratio is varied from 10% to 90%. GET queries are injected in a 10Gbps line rate (i.e., 9.32Mops). In the multilevel NOSQL cache case (Figure 11(a)), L2 NOSQL cache miss ratio is fixed at 15%, which is a conservative assumption. As L1 NOSQL cache miss ratio increases, L2 NOSQL cache and memcached software process more queries. Please note that the throughput decreases quite slowly in the multilevel NOSQL cache case when L1 NOSQL cache miss ratio is less than 40%, while the throughput decreases linearly in the L1 NOSQL only case.

VI. SUMMARY

We proposed a multilevel NOSQL cache architecture that utilizes both the FPGA-based in-NIC cache (L1 NOSQL cache) and the in-kernel key-value cache (L2 NOSQL cache) in a complementary style. We implemented a prototype of our multilevel NOSQL cache using NetFPGA-10G with Virtex-5 XC5VTX240T for L1 NOSQL cache and Linux Netfilter framework for L2 NOSQL cache. Based on the prototype implementation, we showed that 7-9 PEs are needed for a 10Gbps line rate KVS processing. We performed simulations to explore various design options for the multilevel NOSQL cache

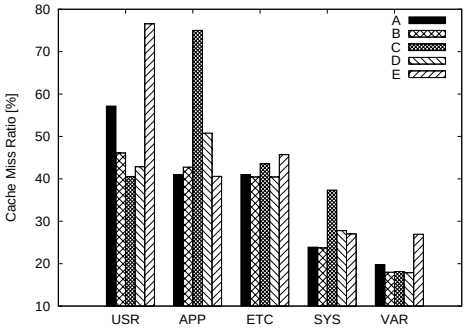
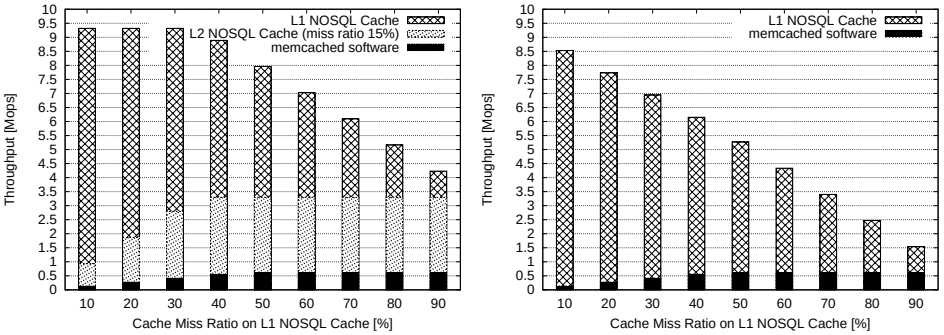


Fig. 10. Slab configurations on L1 NOSQL cache.



(a) Multilevel NOSQL cache (L1 and L2)

(b) Only L1 NOSQL cache

Fig. 11. Multilevel NOSQL cache miss ratio vs. throughput.

architecture, such as the write policies between L1 NOSQL cache and L2 NOSQL cache, hash table design in L1 NOSQL cache, inclusion policies bewteen L1 NOSQL cache and L2 NOSQL cache, slab configurations in L1 NOSQL cache. As a write policy between L1 and L2, the write back policy should be selected to reduce DMA traffic between NIC and host, which is a limiting factor of the throughput. As the slab configuration, the slab sizes distribution should be carefully selected so as to meet an expected workload since L1 NOSQL cache capacity is typically limited. Simulation results demonstrated that our multilevel NOSQL cache design reduced the cache miss ratio and improved the throughput compared to the L1 NOSQL cache only design. Based on the above-mentioned implementation and simulations results, this paper would be the first guideline to build the multilevel NOSQL cache architecture that utilizes both the L1 NOSQL cache and L2 NOSQL cache.

ACKNOWLEDGMENT

This work was supported by SECOM Science and Technology Foundation and JST PRESTO.

REFERENCES

[1] L. A. Barroso and U. Holzle, *The Datacenter as a Computer*, 2nd ed. Morgan & Claypool Publishers, Jul. 2013.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*, Jun. 2014, pp. 13–24.

[3] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*, Jun. 2013, pp. 36–47.

[4] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'13)*, Feb. 2013, pp. 245–254.

[5] M. Blott, K. Karras, L. Liu, K. Vissers, J. Baer, and Z. Istvan, "Achieving 10Gbps Line-rate Key-value Stores with FPGAs," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*, Jun. 2013.

[6] M. Blott and K. Vissers, "Dataflow Architectures for 10Gbps Line-rate Key-value-Stores," in *Proceedings of the IEEE Symposium on High Performance Chips (HotChips'13)*, Aug. 2013.

[7] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based In-Line Accelerator for Memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, Jul. 2014.

[8] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, Sep. 2012, pp. 411–420.

[9] J. M. Cho and K. Choi, "An FPGA Implementation of High-throughput Key-value Store Using Bloom Filter," in *Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT'14)*, Apr. 2014, pp. 1–4.

[10] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura, "Caching Memcached at Reconfigurable Network Interface," in *Proceedings of the International Conference on Field-programmable Logic and Applications (FPL'14)*, Sep. 2014, pp. 1–6.

[11] Z. Istvan, G. Alonso, M. Blott, and K. Vissers, "A Flexible Hash Table Design for 10Gbps Key-value Stores on FPGAs," in *Proceedings of the International Conference on Field-programmable Logic and Applications (FPL'13)*, Sep. 2013, pp. 1–8.

[12] M. Blott, L. Liu, K. Karras, and K. Vissers, "Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, Jul. 2015.

[13] Danga Interactive, "Memcached - A Distributed Memory Object Caching System," http://memcached.org/.

[14] NetFPGA Project, http://netfpga.org/.

[15] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[16] J. Lockwood and M. Monga, "Implementing ultra low latency data center services with programmable logic," in *Proceedings of the IEEE Symposium High-Performance Interconnects (HOTI'15)*, Aug 2015, pp. 68–77.

[17] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Apr. 2014, pp. 429–444.

[18] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *ACM SIGOPS Operating System Review*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[19] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013, pp. 371–384.

[20] Y. Xu, E. Frachtenberg, and S. Jiang, "Building a High-performance Key-value Cache as an Energy-efficient Appliance," *Performance Evaluation*, vol. 79, pp. 24–37, Sep. 2014.

[21] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache," in *Proceedings of the USENIX Annual Technical Conference (ATC'15)*, Jul. 2015, pp. 57–69.

[22] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform," in *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*, Jun. 2015, pp. 476–488.

[23] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, Jun. 2013, pp. 103–114.

[24] Intel, "Intel Data Plane Development Kit (Intel DPDK)," http://www.intel.com/go/dpdk.

[25] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the USENIX Security Symposium (Security'12)*, Aug. 2012, pp. 101–112.

[26] Paramod J. Sadalarge and Martin Fowler, "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence," Aug. 2012.

[27] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, Jun. 2012, pp. 53–64.

[28] S. Sanfilippo, "Redis," http://redis.io/.

[29] D. de la Chevallerie, J. Korinth, and A. Koch, "ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators," in *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'15)*, Jun. 2015.