Ultra-Low-Latency and Flexible In-Memory Key-Value Store
System Design on CPU-FPGA

Yunhui Qiu, Hankun Lv, Jinyu Xie, Wenbo Yin*, Lingli Wang
State Key Laboratory of ASIC and System, Fudan University
Shanghai, China
*E-mail: wbyin@fudan.edu.cn

Abstract—In-memory key-value store (KVS) is critical infrastructure in data centers and is facing challenges in performance and power consumption with the development of the big data technology, which mainly results from the low efficiency of the multi-level memory hierarchy of the CPU-based system. Remote direct memory access (RDMA) technology partly alleviates the problems, but it is still not efficient for KVS, especially for the PUT operation.

In this paper, we present an ultra-low-latency and flexible in-memory KVS system based on the CPU-FPGA heterogeneous architecture, which leverages FPGA to serve as a KVS accelerator. We design a highly parallel accelerator architecture with several novel techniques, including memory pre-allocation, fragmentation processing, and decoupling design, to achieve ultra-low latency, high flexibility, efficiency, and scalability. The system workload can scale up with the storage capacity due to the decoupling design which stores the hash table in onboard DRAM memory and values in the host memory. For each KVS operation, at most one PCIe DMA is needed, which achieves high efficiency. Compared with current hardware-based KVS systems, the proposed one is more flexible, where the supported value range is 4x wider (from 1 byte to 4M bytes). In 10Gbps Ethernet, the peak throughput of the system can reach 13.6 million key-value operations per second (Mops), achieving nearly full utilization of the Ethernet bandwidth. The system latency can achieve as low as 1.2μs for the PUT operation and 1.7μs for the GET operation, which is 3.8x and 2.0x faster respectively than current state-of-the-art KVS systems.

Keywords—in-memory key-value store, fragmentation processing, Memory pre-allocation, CPU-FPGA heterogeneous architecture

I. INTRODUCTION

Key-value store (KVS) is widely used in data centers, such as the Dynamo at Amazon; Redis at GitHub; Memcached at Zinga, Twitter, and Facebook [1]. For a KVS system, data access latency is one of the most important performance metrics, because it strongly affects the user experience. Hence, Memcached, as a memory caching system, has gained great popularity in website services [2]. Further, in-memory KVS systems are developed to achieve lower latency, like Redis [3]. However, with the development of the big data technology, the traditional CPU-based KVS system is facing challenges in power consumption and performance. These challenges bring the energy efficiency of storage systems into focus. The CPU-centric computer architecture is based on a multi-level memory hierarchy. Data stored in the low-speed memory needs to be cached level by level and finally transmitted to the CPU for processing. To some extent, it can speed up data processing. But for data centers, large numbers of data copies between multi-level caches can lead to a tremendous reduction in energy efficiency.

Remote direct memory access (RDMA) technology partly alleviates the problems by reducing the CPU involvement. However, the abstraction provided by RDMA is rather limited and not efficient for KVS [4]–[9].

A new research trend is to adopt FPGA in KVS systems, which offloads KVS operations from CPU to FPGA and decreases data transmission between the client and server. In a hardware-based KVS system, the storage scheme varies with the storage medium, including SRAM, Content Addressable Memory (CAM), DRAM and Flash memory. Examples are as follows:

1. the hash table is stored in SRAM and values in DRAM [10];
2. key-values are stored together in CAM [11];
3. both the hash table and values are stored in onboard DRAM [12] - [17];
4. the hash table is stored in DRAM and values in Flash memory [18];
5. the hash table is stored in onboard DRAM and values in the host memory [4].

Since the storage capacities of both SRAM and CAM are small, the corresponding storage schemes are rather limited in applications. The third and fourth schemes are suitable to establish a KVS system independent of CPU, thus making it hard to interact with the CPU for dynamic memory management. Considering the popularity of in-memory KVS systems, it is more promising to adopt the fifth scheme. However, the supporting value length is a little bit short in current hardware-based KVS systems employing this scheme [4].

In this paper, we present an ultra-low-latency and flexible in-memory KVS system based on the CPU-FPGA heterogeneous architecture, where FPGA serves as a KVS accelerator and can directly access values stored in the host memory. It provides support for standard KVS operations like GET, PUT, and DELETE. Several optimizations are proposed to achieve ultra-low latency in response time, high scalability in storage capacity, great flexibility in the value size, and high efficiency in the KVS operation processing. Firstly, we design a highly parallel accelerator architecture to achieve ultra-low-latency performance. Secondly, we adopt a decoupling design scheme which stores the hash table in onboard DRAM memory and values in the host memory. The workload of the system can scale up with the storage capacity. Thirdly, we design a novel fragmentation strategy which can provide support for super large values. Finally, we propose a new memory allocator, which can hide the latency of the interrupt processing and the memory-mapped I/O (MMIO) of the
PCIe interface through the host memory pre-allocation. It can not only dramatically reduce the latency but also improve the processing efficiency for the PUT operation. Based on the above improvements, the proposed system can achieve high efficiency: at most one PCIe DMA for each GET or PUT operation.

Compared with current hardware-based KVS systems [11][18], the proposed one is more flexible, where the supported value range is 4x wider (from 1 byte to 4M bytes). The system latencies for the PUT and GET operations are as low as 1.2μs and 1.7μs respectively, which is 3.8x and 2.0x faster compared with current state-of-the-art KVS systems [4][9][15]. In 10Gbps Ethernet (GbE), the peak throughput can achieve 13.6 million key-value operations per second (Mops), achieving nearly full utilization of the Ethernet bandwidth.

II. RELATED WORK

The basic storage unit of a KVS system is a key-value pair (KVP), where the value is directly or indirectly associated with the key. According to the value length, there are decoupling and non-decoupling design schemes. When the value length is short, the latter is usually used, where a KVP is stored as a whole [1]. When the value length is long, it is more reasonable to exploit the former, where values are associated with keys by a hash table. The hash table contains keys and the storage addresses of values and can be stored separately from values. This scheme is more economical and efficient on the utilization of storage space compared with the non-decoupling design [10][13]. Since our KVS system provides support for large-size values, we adopt the decoupling design scheme.

In current references, there are also many KVS systems employing this design scheme, but the storage scheme for the hash table and values varies.

Two key metrics are usually used to select the storage mediums of the hash table and values, namely storage capacity and data access latency. Nowadays, there are three common choices: SRAM, DRAM, and Flash memory. Both their data access latency and commercial storage capacity increase sequentially. In general, the storage space needed for the hash table is much smaller than that for values. Therefore, the hash table is usually stored in SRAM or DRAM, while values are generally stored in DRAM or Flash memory. SRAM is used to store the hash table in [10]. It can reach full pipeline processing for KVS operations because of the low data access latency of SRAM, but the small storage capacity of SRAM limits its applications. More systems utilize DRAM to store the hash table [12]-[18]. Besides, values are stored in onboard DDR3 memory in [12] - [17], while in Flash memory in [18]. For hardware-based KVS accelerators, both solutions can build a KVS system independent of CPU, but it is hard to dynamically manage memory utilization and provide software application programming interfaces (APIs) for the system administrators. A more promising storage scheme is to store the hash table in DRAM and values in the host memory so that the hardware-based accelerator can apply to in-memory KVS systems. This scheme is adopted in [4], which leverages programmable network interface card (NIC) to extend RDMA primitives and achieves high performance. We also exploit this scheme in our system.

In KVS systems, the lengths of keys and values can be fixed or variable under different application scenarios. For example, in Facebook’s Memcached server, some pool only supports two key sizes and one value size, while another pool the key size varies from 1 byte to 250 bytes and the value size can reach to 1M bytes [1]. Besides, the length of the string-type value can achieve up to 512M bytes in Redis [3]. However, in most of current hardware-based KVS systems, the value size referred to is less than 4K bytes [4] - [10], [12]- [17], where some even do not provide support for variable-length key-values [10][12]. Few support large-size values (up to 1M bytes) [11][18]. Nevertheless, in [11], the value is divided into multiple parts by software APIs and stored in multiple locations when its length is larger than 352 bits. It will tremendously reduce the storage efficiency and increase the processing latency. In [18], values are stored in Flash memory and the maximum value size referred to is 1M bytes. Our KVS system provides hardware-level support for larger size values, up to 4M bytes.

The hash collision is a tough issue in KVS systems, with solutions mainly including cuckoo hashing [19], hopscotching hashing [22], chained hashing [14]. Chained hashing is based on the chained list, and the latency of operations increases as the workload of the hash table rises. Hopscotching hashing consumes constant time on average for lookup, but it may need to search several buckets for each operation and is more complex than cuckoo hashing. Cuckoo hashing can guarantee constant time for lookup and achieve high storage utilization [23]. In this paper, we employ the cuckoo hashing to resolve hash collisions and further improve it to fully utilize the bandwidth of the DRAM storing the hash table.

III. CPU-FPGA HETEROGENEOUS KVS SYSTEM

As shown in Fig. 1, the proposed in-memory KVS system is based on the CPU-FPGA heterogeneous architecture, where the FPGA card is connected with the host computer through the PCIe interface. The FPGA card serves as a KVS accelerator, which can bypass the host CPU and directly process all standard KVS operations. There are two main modules in the KVS accelerator: the KVS engine (KVSE) and the network offload engine (NOE). KVSE is used for the processing of KVS operations. Its functions are as follows:

1. It receives KVS requests from NOE and returns the processed results.
2. It builds and maintains a hash table, which is stored in onboard DDR4 memory.
3. It controls the transmission of values between FPGA and the host DRAM memory through the PCIe DMA.
4. It communicates with the host CPU through the interrupt and the MMIO of the PCIe interface, so that the host can configure the running parameters of the KVS accelerator and provide a certain amount of pre-allocated memory space to it.
NOE is both a parser and a packer for the user datagram protocol (UDP) implemented on FPGA. It is responsible for transmitting KVS request and response messages between the client and KVSE over the 10GbE.

In this paper, we define a novel application layer protocol, named key-value store message (KVSM) protocol, which is used to transmit the KVS request and response messages. As shown in Fig. 2, it can be divided into two parts, namely a header and a KVP. There are six entries in the header. The `kv_id`, `type`, and `key_size` indicate the KVSM identifier, the KVS operation type, and the length of the key respectively. The `value_size`, `value_total_size`, and `value_offset` indicate the value length in the current KVSM, the total length of the value associated with the key, and the offset of the fragmented value respectively. They are used for fragmentation processing of large-size values. The length of the key is up to 8 bytes by default, and the value length in a KVSM varies arbitrarily within the maximum transmission unit (MTU, 1500 bytes) of the Ethernet. If the total value length is larger than MTU, multiple KVSMs have to be utilized to transmit the large value.

IV. IMPLEMENTATION

The kernel of the KVS accelerator is KVSE, whose architecture is illustrated in Fig. 3. KVSM Parser parses the request KVSMs from NOE and then transmits the request headers and values to Hash Engine and C2H Controller separately. After receiving a request header, Hash Engine looks up the hash table and executes corresponding procedures based on the operation type. If it is PUT miss (key not found), Hash Engine submits a request to Segment Management for contiguous memory space to store the whole value. A DMA descriptor in the direction of card-to-host (C2H) is then transmitted to C2H Controller, which executes fragmentation processing and submits fragmented descriptors and values to DMA Controller. In case of GET hit, Hash Engine submits a host-to-card (H2C) DMA descriptor to H2C Controller which also executes fragmentation processing; submitting fragmented descriptors and packing received fragmented values from DMA Controller. In Segment Management, there is a set of pre-allocated host memory segments, so that it can quickly provide memory space to Hash Engine when needed. Besides, the host CPU can update the memory segments through the PCIe MMIO. Event Controller receives event messages and generates interrupt requests (IRQs) to inform the host CPU for corresponding processing. The events include PUT miss, GET hit, DELETE hit, and segment update. DMA Controller controls the transmission of values from or to the host memory, the interrupt processing, and the PCIe MMIO. KVSM Packer generates response KVSMs and transmits them to NOE.

A. KVSM Parser & Packer

KVSM Parser parses received request KVSMs. It extracts `kv_id`, `type`, `key_size`, `value_total_size` and `key` to form a new header and then transmits it to Hash Engine. For all fragmented KVSMs associated with a key, only one header is submitted, so that only one hash table entry is created for a KVP. Besides, the parsed values are transmitted to C2H Controller.

KVSM Packer formats the processed results of KVS operations into response KVSMs and then transmits them to NOE. There are six results from three sources: Hash Engine, C2H Controller, and H2C Controller. The results include PUT hit/miss, GET hit/miss, DELETE hit/miss. Hash Engine might generate four results: PUT hit, GET miss and DELETE hit/miss. C2H Controller and H2C Controller generate the other two results, PUT miss and GET hit respectively.
B. Hash Engine

In our system, the hash table is stored in DDR4 memory. Each hash table entry is defined as 16 bytes, as shown in Fig. 4, where value total size and value addr indicate the total length and storage address of the value.

We employ the cuckoo hashing to resolve hash collisions. Jenkins hash function [20] is selected in the design due to its excellent performance and high popularity in the open source community. To make full use of the bandwidth of DDR4, 64 bytes per cycle, we improve the cuckoo hashing algorithm: using two hash functions, each corresponding to four buckets, as shown in Fig. 5. Leveraging the parallelism available in FPGA, the eight hash table entries can be read out in two DDR4 clock cycles, and then the request key can be compared with those contained in the eight entries simultaneously. When a new hash entry needs to be inserted into the hash table, a poller mechanism is used to select the inserting and kick-out position, which is in ascending order of the numbers as shown in Fig. 6. If these eight buckets are not all used, an empty bucket will be selected to insert the new entry. Otherwise, an old hash entry will be replaced with the new one and then re-inserted into the hash table.

The processing for a KVS operation can be roughly divided into three steps, as shown in Fig. 7. Firstly, calculate hash addresses by hash functions. Secondly, search the hash table according to hash addresses and then compare keys. Finally, execute corresponding procedures based on the comparison result and the operation type. If it is PUT miss or DELETE hit, the hash table needs to be updated, and a hash table entry may be kicked out. However, both may lead to data consistency issues, such as the read-after-write consistency problem. One solution is to cache key-value data during the processing period [10]. But large amounts of data have to be cached due to the high data access latency of DDR4. This substantial additional overhead makes it difficult to be implemented efficiently. More importantly, the response time of memory request may increase with the memory utilization rate rising, thus resulting in uncertainty of the amount of cached data. Therefore, we propose a semi-pipeline design: the PUT and DELETE operations stall the pipeline, while the GET operation does not. Fortunately, the long transmission time of large-size values can hide the processing time of Hash Engine. So only if the value size is small, less than about 256 bytes, it may reduce the throughput of the system.

C. C2H&H2C Controller

C2H Controller and H2C Controller are mainly used to execute fragmentation processing, cooperating with KVSM Parser and Hash Engine. For a single KVP, KVSM Parser submits only one header, and Hash Engine generates only one hash table entry. In case of PUT miss and GET hit, Hash Engine submits a total DMA descriptor to C2H Controller and H2C Controller respectively. These procedures can be regarded as a merging process, which can increase the storage efficiency and reduce the processing time.

D. Segment Management

Segment Management is designed to provide host memory space to Hash Engine for storing values. Traditionally, to insert values into the host memory, requests for memory space must be sent to the host CPU through the interrupt, and then CPU will allocate memory space and provide it to the requester. This process will dramatically increase the latency for the PUT operation. We propose a novel memory allocation scheme to hide the latency. The host CPU pre-allocates a set of memory space and provides it to the requester for each request. This scheme can greatly reduce the latency and improve the throughput of the system.
segments and provides them to Segment Management through the PCIe MMIO. Hence Segment Management can directly provide memory space for the value storage. When the utilization of the segments reaches a certain threshold, an event message is generated and transmitted to Event Controller, which informs the host CPU to update the used segments by IRQ. This can guarantee that there are always available segments in Segment Management unless the host CPU cannot allocate new memory segments. Therefore, the long processing time of interrupt and memory allocation can be hidden, and the latency for the PUT operation can be greatly reduced. Actually, the module only needs to manage the addresses of the segments. When a value is inserted, the remained memory space can be marked by updating the offset address.

We utilize dual-port BRAM memory as the segment buffer, as shown in Fig. 9. Each entry in the buffer identifies a 4M bytes segment, which is the maximum size of contiguous memory space the host CPU can allocate. The segment address is represented with segment_addr. And the lower 22 bits can be used as the offset address indicating the used space. Besides, one port of the buffer is used to provide memory space for values, and the other is used to initialize and update segments by the host CPU.

V. EVALUATION

A. Platform and Experimental Setup

The KVS system is based on the CPU-FPGA heterogeneous architecture, while its kernel, KVSE, and NOE are implemented on FPGA. The hardware platform is a Xilinx Kintex UltraScale FPGA KCU105 evaluation kit, which has an onboard 2GB DDR4 DRAM and two 10GbE interfaces. Through a PCIe gen3 x8 interface, the FPGA board is connected to a host computer which has an Intel i5-2400 quad-core CPU, a set of DDR3 DRAM with a total capacity of 12GB, and a 256GB SSD. The operating system in the host is CentOS 7.

In our KVS system, the onboard DDR4 DRAM is used to store the hash table, running at 300MHz. Values are stored in the host DDR3 DRAM and communicated with the KVS accelerator via PCIe DMA. UDP protocol is adopted in NOE based on the 10GbE. The KVS accelerator runs at 250MHz, while NOE at 156.25MHz.

To evaluate the performance and analyze the bottleneck of the KVS system, we deploy two test schemes: KVS system with NOE (System A) and KVS system without NOE (System B), as shown in Fig. 10. For each scheme, we utilize KVSM generator and KVSM receiver to simulate the behavior of the client. KVSM generator can generate request KVSMs of different modes, such as PUT/GET/DELETE/Random operations for fixed/variable-length values. KVSM receiver can receive and verify the response KVSMs from the KVS system. In System A, we use two NOEs to evaluate the performance of the KVS system in the practical network environment. Each NOE contains a 10GbE and is connected with each other via a fiber optic cable. In System B, we remove the NOE to evaluate the performance of the KVS accelerator separately. By comparing the two system performances, we can identify where the bottleneck is.

B. Throughput

1) Theoretical Maximum Throughput

The theoretical throughput of the proposed system is determined by both NOE and KVSE. Suppose the network bandwidth is fully utilized, we can calculate the theoretical maximum throughputs of NOE, KVSE, and the KVS system based on the following factors.

(1) Network transmission bandwidth $B$, which consists of the running frequency $f$ and the data width $w$.

$$B = f \times w$$  \hfill (1)
KVSM length $L$, where the value length $l$ is variable and the sum length of the header and the key $l_0$ is fixed. If KVSM contains the value data, then
$$L = l_0 + l.$$  
(2)

Otherwise,
$$L = l_0.$$  
(3)

The success rate of KVS operations $\alpha$, which only affects the response KVSM length for the GET operation. If it is a success, the length is $l_0 + l$, otherwise $l_0$. Therefore, the average length is
$$\alpha(l_0 + l) + (1 - \alpha)l_0 = l_0 + \alpha l.$$  
(4)

Pipeline stalling cycles $n_0$ due to the PUT operation.

For NOE, the theoretical maximum throughput can be calculated as follows. Given the running frequency $f_s$, the data width $w_s$, the throughputs $V_n$ for the PUT and GET operations are shown as (5) and (6) respectively.
$$V_n = \frac{f_s w_s}{l_0 + l}.$$  
(5)
$$V_s = \frac{f_s w_s}{l_0 + \alpha l}.$$  
(6)

For KVSE, the theoretical maximum throughput can be calculated as follows. Given the running frequency $f_s$, the data width $w_s$, the throughput $V_s$ for the PUT operation is
$$V_s = \min \left\{ \frac{f_s w_s}{l_0 + l}, \frac{f_s w_s}{l_0 + \alpha l} \right\},$$  
(7)

while the throughput for the GET operation is
$$V_s = \frac{f_s w_s}{l_0 + \alpha l}.$$  
(8)

The theoretical maximum throughput of the KVS system is the minimum throughput of NOE and KVSE. For the PUT operation, the system throughput is
$$V_s = \min \left\{ \frac{f_s w_s}{l_0 + l}, \frac{f_s w_s}{l_0 + \alpha l} \right\}. $$  
(9)

For the GET operation, the system throughput is
$$V_s = \frac{f_s w_s}{l_0 + \alpha l}.$$  
(10)

2) Experimental Throughput

We evaluate the throughputs of both System A and System B under different value sizes. As shown in Fig. 11, the experimental peak throughputs for the PUT and GET operations can reach 4.3Mops and 13.6Mops respectively in System A, while in System B, 4.4Mops and 27.5Mops respectively. When the value size exceeds 256 bytes, both throughputs are inversely proportional to the value length. Meanwhile, the throughput of System B exceeds that of System A, indicating that NOE becomes the bottleneck of the system throughput.

We also calculate the theoretical maximum throughputs of System A and System B using equations (7) - (10), where the parameter values are as follows.

$$w_s = w_s = 64\text{bits}$$
$$f_s = 156.25\text{MHz}$$
$$f_s = 250\text{MHz}$$
$$l_0 = 192\text{bits}$$
$$n_0 = 56$$
$$\alpha = 1$$

The theoretical throughputs for the PUT and GET operations are shown in Fig. 11a and Fig. 11b, which show that the experimental and theoretical throughputs are almost the same for both operations when the value size exceeds 256 bytes. This indicates that our system can achieve full-pipeline performance and make full use of the 10GbE bandwidth. When the value length is less than 256 bytes, the experimental throughput is slightly lower than the theoretical maximum throughput for the GET operation because of the marginal cost needed to provide support for large-size values in the KVS system. Therefore, we can regard the value size of 256 bytes as a turning point, where the throughput of the PUT and GET operations in system A can reach 3.6Mops and 3.7Mops respectively.
Compared with DRAM-only BlueCache [18], which also provides support for large-size values, our peak throughput is the same for the PUT operation and 3.4x higher for the GET operation.

C. Latency

1) Theoretical Minimum Latency

The theoretical latency of the proposed system is also determined by both NOE and KVSE. Suppose the network bandwidth is fully achieved, we can calculate the theoretical minimum latencies of NOE, KVSE, and the system. The latency can be divided into two parts: the constant processing time and the data transmission time. Since the failed KVS operation costs less time, we only calculate the latencies of the successful KVS operations. There are three related factors: the network transmission bandwidth B, KVSM length L, and pipeline blocking cycles \( n_b \), which have been introduced in the previous section.

For NOE, the theoretical minimum latency can be calculated as follows. Given the running frequency \( f_n \), the data width \( w_n \), and the constant processing time \( T_{np} \) for the PUT operation and \( T_{ng} \) for the GET operation, the latencies \( T_n \) for each operation are shown as (11) and (12) respectively.

\[
T_n = T_{np} + \frac{l_n + l}{f_n w_n} \tag{11}
\]
\[
T_n = T_{ng} + \frac{l_n + l}{f_n w_n} \tag{12}
\]

For KVSE, the theoretical minimum latency can be calculated as follows. Given the running frequency \( f_k \), the data width \( w_k \), the pipeline stalling time \( T_{ksh} \), the constant processing time \( T_{kp} \) for the PUT operation and \( T_{kg} \) for the GET operation, the latencies \( T_k \) for each operation are shown as (13) and (14).

\[
T_k = \max \left\{ T_{ksh} + T_{kp} + \frac{l_k + l}{f_k w_k} \right\} \tag{13}
\]
\[
T_k = T_{kg} + \frac{l_k + l}{f_k w_k} \tag{14}
\]

The theoretical minimum latency of the system is related to the maximum latencies of NOE and KVSE. The constant processing time of the system is longer than those of KVSE and NOE. Given the pipeline stalling time \( T_{sh} \), the constant processing time \( T_{sp} \) for the PUT operation and \( T_{sg} \) for the GET operation, the system latencies \( T_s \) for each operation are shown as (15) and (16).

\[
T_s = \max \left\{ T_{sh} + T_{sp} + \frac{l_s + l}{\min \{f_s w_s, f_s w_s\}} \right\} \tag{15}
\]
\[
T_s = T_{sg} + \frac{l_s + l}{\min \{f_s w_s, f_s w_s\}} \tag{16}
\]

2) Experimental Latency

We measure the latencies of both System A and System B under different value sizes. As shown in Fig. 12, the experimental system latency is as low as 1.2\( \mu s \) for the PUT operation and 1.7\( \mu s \) for the GET operation in System A. In System B, the latency is even lower: 0.3\( \mu s \) for the PUT operation and 0.8\( \mu s \) for the GET operation.

We also calculate the theoretical minimum latencies of System A and System B using equations (13) - (16), where the parameter values are as follows.

\( w_n = w_k = 64\text{bits} \)
\( f_n = 156.25\text{MHz} \)
\( f_k = 250\text{MHz} \)
\( l_n = 192\text{bits} \)
\( T_{ksh} = 0.3\mu s \)
\( T_{kp} = 0 \)
\( T_{kg} = 0.8\mu s \)
\( T_{sh} = 1.1\mu s \)
\( T_{sp} = 1.1\mu s \)
\( T_{sg} = 1.7\mu s \)
The theoretical latencies for the PUT operation are shown in Fig. 12a, while for the GET operation in Fig. 12b. The two figures show that the experimental and the theoretical minimum latencies are almost the same for both the PUT and GET operations, indicating that the Ethernet bandwidth can be made full use of and the processing time of the accelerator is robust. Compared with the state-of-the-art KVS systems in current references, as shown in Table 1, our system is 3.8x and 2.0x faster for the PUT and GET operations respectively.

Besides, the value size of our system can reach up to 4M bytes, 4x larger than that of current hardware-based KVS systems [11][18].

VI. CONCLUSION

In this paper, we present an ultra-low-latency and flexible in-memory KVS system based on CPU-FPGA heterogeneous architecture. Due to the design of a highly parallel accelerator architecture, the system can achieve ultra-low-latency performance. We adopt a decoupling design scheme which stores the hash table in the onboard DDR4 memory and values in the host memory, making the storage capacity of the KVS system highly scalable. Through the design and implementation of a novel fragmentation mechanism in the KVS accelerator, the system can support large values, and thus the flexibility is substantially broadened. For a single key-value operation, such as PUT, only one DMA data transmission at most is needed, which can significantly improve the processing efficiency. Besides, we find that the 10GbE is the bottleneck in this system. If exploiting the Ethernet with larger bandwidth, the system can achieve higher performance.

REFERENCES