Caching Memcached at Reconfigurable Network Interface

Eric S. Fukuda*, Hiroaki Inoue†, Takashi Takenaka†, Dahoo Kim*,
Tsunaki Sadahisa*, Tetsuya Asai*, and Masato Motomura*
*Graduate School of Information Science and Technology
Hokkaido University, Sapporo, Hokkaido 060–0814, Japan
Email: {fukuda@lalsie., kim@lalsie., sadahisa@lalsie., asai@, motomura@}ist.hokudai.ac.jp
†NEC Corporation,
1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa 211–8666, Japan
Email: {takenaka@aj, h-inoue@ce}.jp.nec.com

***Abstract*—Memcached is a technology that improves response speed of web servers by caching data on DRAMs in distributed servers. In order to achieve higher performance, memcached has been evaluated on various platforms. Among them, FPGA seems to be the most efficient platform to run memcached, and several research groups are trying to achieve higher throughput with it. However, it is difficult to utilize a large amount of memory (several dozen gigabytes) with an FPGA. Some groups are trying to solve this problem by using an embedded CPU for memory allocation and another group is employing an SSD. Unlike other approaches that try to replace memcached itself on FPGAs, our approach augments the software memcached running on the host CPU by caching its data and some operations at the FPGA-equipped network interface card (NIC) mounted on the server. The locality of memcached data enables the FPGA NIC to have a fairly high hit rate with a smaller memory. We first explore the cache parameters by software simulations and estimate the effectiveness of our approach, and then prototype a system to prove its effectiveness. Through our evaluation with YCSB, a standard key-value store (KVS) benchmarking tool, we estimate that the latency improved by an order of magnitude over software memcached running on a high performance CPU.**

I. Introduction

Web service providers that have tremendous amounts of user and other information are eager to facilitate new technologies that enable their servers to handle more data traffic. One such technology employed by many web service providers is key-value stores (KVSs). A KVS holds data (values) with keys uniquely assigned (key-value pairs; KVPs), and sends them out as the data (value) is requested with the corresponding key. For its speed of finding the requested data in contrast to traditional relational database management systems (RDBMSs), many web service providers are now using KVS databases such as DynamoDB at Amazon [1], Bigtable at Google [2], memcached at Facebook [3], [4], and many others. Memcached [5] is a technology that reduces the latency of data retrieval by storing KVPs in distributed servers' memories instead of fetching from the hard drives of database servers. Its simple data structure and computation have led to its wide adoption by various web service providers.

Memcached is used not only by Facebook, but also by a number of major web service providers such as Wikipedia and YouTube [5]. According to Facebook's research on their own memcached workloads, they use hundreds of memcached servers [3]. In view of such extensive use, improving the memcached performance would have a large impact on web services' response. In fact, researchers have investigated the suitability of various hardware platforms for running memcached, from multiple low power CPUs [6]–[8] to many-core processors [9] and FPGAs [10]. Meanwhile, FPGA-based memcached systems are outperforming high performance CPUs such as Intel® Xeon® by an order of magnitude [11].

Although these efforts have improved the performance of memcached, major challenges remain. One such challenge is that it is difficult for FPGAs to efficiently manage a large memory size. Memcached servers usually have a few dozen gigabytes of memory, and such a memory space is too large for an FPGA to efficiently manage [12]. One group is trying to handle large memory size by utilizing a CPU core that is embedded in the FPGA [11]. The FPGA invokes the CPU to allocate or reallocate some blocks in the memory and stores data there. Another group employed an SSD to enlarge the memory space using a DRAM as a cache [13].

In this paper, we propose a method that makes possible a low latency hardware memcached system with less memory than others require. Our method caches the subset of data stored in software memcached running on the host CPU at the network interface card (NIC) equipped with an FPGA and a DRAM memory. When the server receives a request from a client, the NIC tries to retrieve the data within the DRAM and sends it back if the data is found. If not, the NIC passes the request to the host CPU and the CPU executes the usual memcached operation. Since memcached data has locality, the NIC requires only a fraction of the amount of memory that the host server has. Furthermore, the commands the NIC cache does not support can be delegated to the host CPU; therefore only the frequently used memcached commands have to be supported on the NIC.

Our contributions in this paper are as follows:

- We propose a method that reduces the delay of memcached by caching the memcached data at the NIC mounted on the server.
- We explain how the subset of memcached functionalities should be implemented on the FPGA equipped NIC in order to maintain data consistency between the NIC and the host CPU.
- We verify the improvement of the performance with a standard evaluation tool that is capable of evaluating

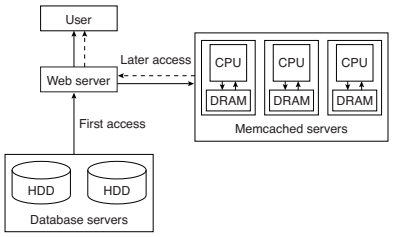

Fig. 1. The operation of memcached.

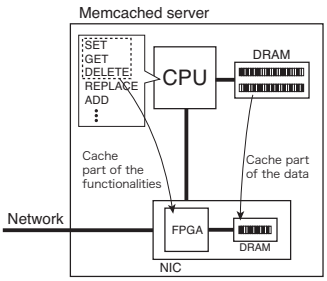

Fig. 2. The basic idea of the proposed method.

various KVSs.

Although we focus on proving the effectiveness of our NIC caching architecture with memcached, it is important to note that this architecture can be applied to many other server applications that require lower latency as long as the data has temporal locality.

II. Background

A. Memcached

Memcached is a kind of KVS database and caches data on memories of distributed servers in the form of key-value pairs. As Fig. 1 shows, memcached servers store the subset of data stored in the database servers, which usually use hard drives, in order to allow faster data access from the web server. Memcached servers often have a few dozen gigabytes of memory each and run in a cluster of several hundred servers. Data are not stored in the memcached server at the beginning, and the web server has to get the data from the database servers. The web server sends the data back to the user and also sends a SET request with a paired key (250 bytes or smaller) and value (1 MB or smaller) to memcached to store the data. When the web server needs the same data later, it sends a GET request with the key to the memcached server, and the memcached server returns the value to the web server. Data that are not accessed frequently on the memcached server are evicted when the capacity is full. If the web server sends a GET request for data that has been already evicted, the memcached server notifies the web server that a cache miss has occurred. The web server will then check the database server and SET the data to the memcached server again.

Table I is a list of memcached commands. GET, SET and DELETE are the commands that are mainly used, and GET is the most frequently used command among them. According to a paper that reports the details of the memcached workloads of Facebook [3], the ratio of GET, SET and DELETE is 30:1:14 (exact ratio of DELETE not being provided in the paper, we estimated it visually from the chart). Therefore, the investigations we look through in Section II-B usually support only the GET, SET and DELETE commands.

TABLE I. Memcached commands

Command	Operation
SET	Store a KVP.
ADD	Store a KVP if the key is not found.
REPLACE	Replace a KVP if the key is found.
APPEND	Append data to a stored value.
PREPEND	Prepend data to a stored value.
CAS	Overwrite a value if the KVP is unchanged since last reference.
GET	Retrieves a value with a key.
GETS	Get a CAS identifier while retrieving a value with a key.
DELETE	Removes an KVP.
INCR/DECR	Increment or decrement a value.
STATS	Get an report of the memcached server statistics.

B. Related Work

Berezecki et al. evaluated the performance of memcached running on Tilera's TILEPro64 processor, which can allocate computations to its 64 cores [9]. Examining several configurations of cores running operations such as Linux kernel, network operations and others, the throughput per watt attained a maximum 2.4-fold increase over Xeon. However, the latency remained the same or worsened slightly from Xeon's 200 - 300 µs to TILEPro64's 200 - 400 µs.

Chalamalasetti et al.'s work was the first to try to utilize FPGA for accelerating memcached [10]. The system mainly consists of two parts, a network processing part and a memcached application part. The network processing part extracts memcached data from incoming packets and gives them to the memcached application part, and also does the reverse. Receiving the data from the network processing part, the memcached application part calculates hashes from the keys in order to determine the memory address at which the KVPs are stored and writes to or read from the memory. The performance of memcached improved dramatically in this scheme: throughput per watt attained 4.3-fold over Xeon and the latency became 2.4 to 12 µs.

Blott's group further improved the performance of memcached running on an FPGA by improving the UDP offload engine and adopting dataflow architecture [11]. They achieved more than 15-fold higher throughput per watt than a Xeon and a latency of 3.5 to 4.5 µs. This work is unique in that it uses a CPU for allocating the memory.

Another approach was proposed by two groups almost coincidently [7], [8]. Through dynamic analysis of memcached codes, they found that instruction cache misses or low branch prediction success rates caused by the frequent call of the network protocol stack, kernel and some library codes were the bottleneck. Their approach to get rid of this bottleneck was to replace the network process and some of the memcached process (GET request handling) software codes with hardware and integrate it into an SoC with a CPU core. This method was evaluated on an FPGA that had an embedded CPU core and yielded 2.3 to 6.1-fold higher throughput per watt than a Xeon. Our method is close to [7] and [8] in the sense that we execute part of the memcached process on hardware. However, we do not share the memory between the memcached hardware and the CPU, and thus the memory control of our method is much simpler.

To gain a larger storage size on hardware memcached, Tanaka and Kozyrakis employed a solid-state drive (SSD) in their FPGA based system [13]. Their approach is to store KVPs in the SSD on the FPGA board, using the DRAM on the same board as a cache. They achieved 14-fold higher throughput, 5-

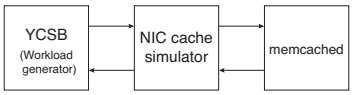

Fig. 3. Connection of software modules.

to 60-fold low latency and 12-fold higher throughput per watt than a Xeon.

Recently, a commercial memcached appliance that can be used in practice has been developed [14]. This appliance achieved 9.7-fold higher throughput than a Xeon by using a CPU and multiple FPGAs while the latency was 500 µs to 1 ms, which is larger than for a Xeon. Its throughput per watt has not been publicly announced.

III. Concept of NIC Cache

The basic idea of our method is to cache part of memcached server's data and functionalities to the NIC mounted on the same computer. According to Facebook's investigation into their own memcached workloads, there is some locality of access to their data [3]. On top of that, Facebook's investigation also indicates that among all memcached commands, SET, GET and DELETE account for most of the requests. This means that reducing the processing latency of only frequently accessed data should have a large impact on the web server's performance. The nearest place to the web server in the server computer on which memcached is running is the network interface. Therefore we try to efficiently reduce the latency by caching frequently used data and functionalities (SET, GET and DELETE) at the NIC and leaving the less frequently used data and functionalities to be handled by the host CPU. The NIC we assume to use has a fast connection to the network (several tens of Gbps), an FPGA, gigabytes of memory, and a fast connection to the host CPU (Fig. 2).

The FPGA on the NIC can be designed to perform various cache behaviors. The following description is an example of the processing policy of the NIC:

SET: The NIC stores the KVP to its DRAM and sends back a reply notifying the web server whether the KVP was properly stored. If a KVP already stored in the DRAM becomes evicted, a SET request with the evicted KVP is sent to the host CPU (write-back, write-no-allocate).

GET: If the key in the request is found in the NIC, the NIC returns a reply message with the corresponding value to the web server. Otherwise, the NIC sends the request to the host CPU, and the CPU searches for the key and returns it to the NIC. After the KVP is cached to its DRAM, it is sent back to the web server (read-allocate). If the key was not found at the CPU, it returns a reply notifying the web server that the key did not exist.

DELETE: If the key in the request is found in the NIC, it is invalidated and the request is sent to the host CPU. The CPU invalidates the data and returns a reply to the web server notifying that the KVP was successfully deleted.

IV. Cache Simulation

In this section, we evaluate the NIC cache concept over software simulation in order to estimate its effectiveness. We implemented a cache simulator that behaves as mentioned in Section III. Test workloads were generated by *Yahoo! Cloud Serving Benchmark* (YCSB), a standard benchmarking tool for KVS [15]. YCSB, cache simulator and memcached was placed as shown in Fig. 3. Requests generated by YCSB are sent to the cache simulator and the cache simulator processes the requests as described in Section III, backed up by real memcached.

A. Testing Tool

YCSB provides workloads that simulate various KVS use cases. Table II, quoted from [15], shows the characteristics of the workloads. Each workload is characterized by the ratio of commands and the record selection distribution.

Read, update and insert operations in the table correspond to memcached's GET, REPLACE and ADD commands respectively. In our experiment, however, we use SET for both update and insert operations. The difference between SET and update and insert is that update (REPLACE) and insert (ADD) check whether or not the data is already stored, and decide to store the data accordingly. Since we have to access the memory before we know whether the same key is stored, we used SET in place of REPLACE and ADD. The delay will be almost the same because checking whether the data is stored or not can be done in parallel with other operations. Regarding Workload E, we do not use it because memcached does not support the scan operation. Thus we use Workload A to D for our evaluation.

There are two record selection distributions: Zipfian and Latest. Zipfian is a distribution in which certain records are popular independent of their insertion order. An intuitive example is Wikipedia, where certain entries such as "Moore's Law" or "Transistor" are frequently viewed even though they were created years ago. On the other hand, Latest is a distribution in which the records added recently are the most popular ones. An example of Latest selection is Facebook's user updates where people mainly view their friend's recent posts.

B. Simulation Results

Fig. 4 and Fig. 5 show the miss rate for GET requests at the NIC cache with various associativity and capacity for FIFO and least recently used (LRU) replacement algorithm respectively. The x-axis is the relative ratio of the NIC cache capacity to the memcached capacity running on the host CPU. Memcached's capacity was set to 512 MB. This is four times as large as the NIC cache size that we implemented as described below in Section V. (The ratio of the cache size to the host memcached size determines the miss rate, rather than the absolute cache size.)

Apparently, the difference in miss rates between the two algorithms is very small. For Workload A to C, the miss rates are a few percent less with LRU than with FIFO when the capacity of the NIC cache is small. Workloads A to C, which use Zipfian distribution, have similar curves, while Workload D with Latest distribution have linearly decreasing miss rates as the NIC cache's capacity increases.

We also carried out an experiment with a read-no-allocate policy, which means that the NIC does not cache the KVP on receiving the GET reply from the host CPU. This policy has the advantage of keeping the data consistency between the NIC cache and the host CPU easier. If a GET miss for a certain key occurs at the NIC and the subsequent request is a SET for the same key, the KVP set by the SET request at the NIC can be overwritten by the GET reply for the GET miss from the host CPU. This problem can be avoided in either of two ways: sending request from the NIC to the host CPU synchronously, or employing a read-no-allocate policy. Since synchronous requests can lead to an increase in average

TABLE II. DESCRIPTION OF YCSB WORKLOADS. (ORIGINALLY SHOWN IN [15].)

Workload	Operations	Record selection	Application example
A–Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B–Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are read tags
C–Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g. Hadoop)
D–Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want read the latest statuses
E–Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

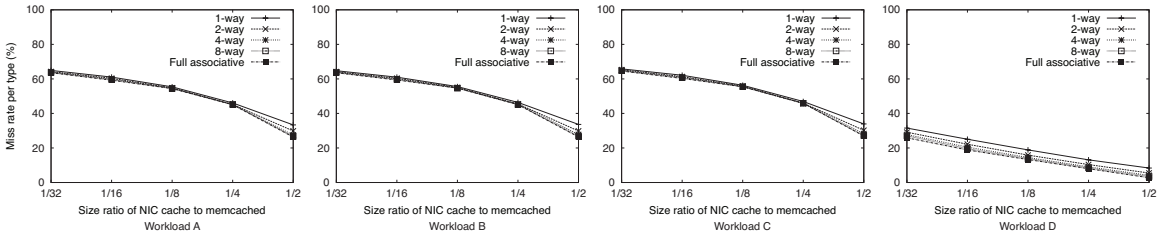

Fig. 4. Miss rate for GET requests with FIFO replacement policy.

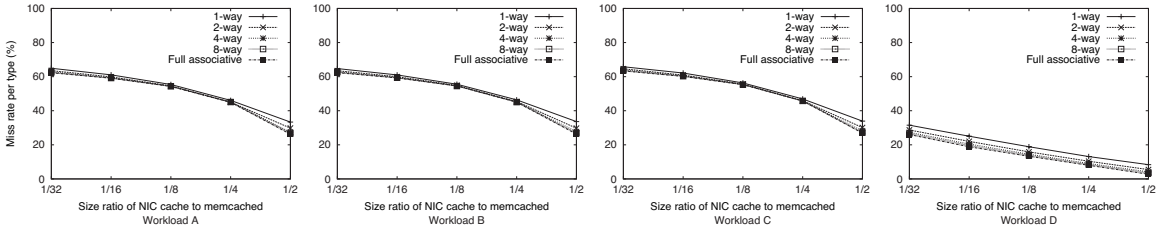

Fig. 5. Miss rate for GET requests with LRU replacement policy.

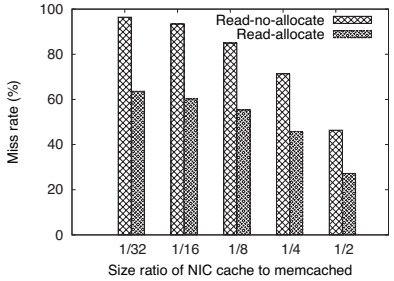

Fig. 6. Miss rates with read-allocate and read-no-allocate for Workload C.

latency, employing a read-no-allocate policy can be beneficial if the miss rate at the NIC does not increase.

We found that the miss rate increased by less than a few percent for Workload A, B and D. For Workload C, however, the miss rate increased by more than ten percent (Fig. 6). This degradation comes from the command mix of Workload C. Unlike Workload A, B and D, Workload C does not send SET requests, so once a popular key is evicted from the NIC during the load phase, it cannot store it again in the transaction phase, and thus the miss rate rises. (YCSB has load phase, which sends SET requests for all the keys for warm up, and transaction phase, which sends requests with the characteristics given in Table II. The results were measured during the transaction phase.)

V. HARDWARE DESIGN

Through our simulation, we confirmed that our concept is effective for various workloads. Although it has a relatively low hit rate for Workload C, as our initial implementation, we implemented the system with a read-no-allocate policy for its simple implementation. Fig. 7 shows the architecture of the NIC cache. The circuit implemented in the FPGA consists of five parts as described below:

Incoming packet handler: Non-memcached packets received from the network side are sent to the CPU without any operations so that the CPU could run not only memcached but also other server applications. On receipt of a memcached packet, the command, the key and the value are extracted from the packet and sent to the memory controller, hash calculator and the hash table. If the command is a GET and the memory controller returns a miss, the packet is sent to the CPU. If the memory controller returns a hit for a GET command, the packet is discarded. If the command is a SET or a DELETE, the packet is sent to the host CPU regardless of hit or miss returned from the hash table.

Outgoing packet handler: Outgoing packet handler does three things. First, it creates a packet in reply to a GET request using the key and the value given from the memory controller. Second, it receives memcached or other packets from the host CPU. Finally, it merges the packets from the two data sources (memory controller and the host CPU) and sends them out to the network. As mentioned in the beginning of this section, in our initial implementation, we do not cache data from the reply packets so as to simplify our implementation. Improving

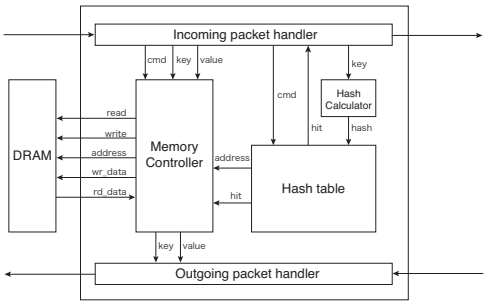

Fig. 7. NIC cache architecture.

this behavior is part of our future work.

Hash calculator: Hash calculator receives a key from the incoming packet handler and calculates a hash with Jenkins's lookup3 function [16]. It produces a 32-bit hash from the key.

Hash table: Hash table manages where in the DRAM memory to store the KVP. More detailed structure is given in Fig. 8. The top 15 bits of the hash given from the hash calculator becomes the index of the hash table, and the lower 17 bits are written to the empty entry in the row, pointed to by the index, as a tag. The table is 8-way associative with a pseudo LRU replacement algorithm. The address of the memory is retrieved uniquely from the column and the row where the tag is stored. The key and the value are stored at the location on the memory where the address points. Memcached originally supports variable sizes of keys and values, but since YCSB supports fixed key and value sizes by default, we use fixed sizes. According to Facebook's investigation, key sizes are mostly less than 50 bytes and value sizes are less than a few hundred bytes. Therefore, we set the key size and the value size to 64 bytes and 448 bytes respectively to keep our hardware implementation of memory addressing simple by setting the size of the KVP to 512 bytes, which is a power of two.

If the command given from the incoming packet handler is a SET, the hash table stores the tag in a certain entry, setting its valid flag. If the command is a GET, the hash value is looked up in the hash table and hit/miss information is sent to the memory controller. Both in the case of SET and GET, the calculated address is sent to the memory controller. For DELETE, the valid flag of the entry is invalidated if the hash value stored in the table matches the hash value given from the hash calculator.

Memory controller: The memory controller receives the command, the key and the value from the incoming packet handler, and also receives the address and the hit/miss information from the hash table. If the command is a GET, it sends a read signal and the address to the memory. Then the two keys from the incoming packet handler and the memory are compared to see whether they match. Since identical hash values can be generated from different key strings, the judgment of hit/miss at the hash table is uncertain. The keys should be checked here so as to make sure they are really identical. Provided that the keys match, the memory controller sends the key and the value to the outgoing packet handler; otherwise it does nothing. If the command is a SET, it writes the key and the value to the memory at the address given from the hash table. If the command is a DELETE, it does nothing.

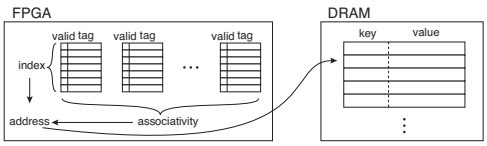

Fig. 8. Correspondence of the hash table and the value storage.

TABLE III. DESIGN SPECIFICATION OF FPGA

Number of used block RAM and FIFO	238 / 324 (86%)
Number of used slice LUTs	60314 / 207360 (29%))
Number of used slice registers	64505 / 207360 (31%)

A. Experimental Conditions

We used UDP protocol for communication between the YCSB server and the memcached server. The two servers were connected with the 10 Gbps interconnect. Although memcached supports both TCP and UDP protocols, to make the packet offloading simple, we used UDP.

Our proprietary platform board consists of two 10 Gbps network interfaces, a Virtex-5 LX330T FPGA, a 1 GB DDR2 SDRAM memory and a PCI Express (Gen1 x8) interface. The host CPU is Intel Xeon E5-1620. How efficiently we can use the memory on the NIC depends on how large a hash table we can implement in the FPGA's block RAMs. The resource usage is shown in Table III.

B. Performance

First of all, we confirmed that our system works for all Workloads A to D. Then we evaluated the latency of our system in three ways: First, to estimate the network latency, we implemented a system on the FPGA of the NIC that returns the request immediately after receiving it from the network. Next, we implemented the system described in Section V, sent GET requests for the same key for several times, and got the minimum latency. Finally, we sent SET requests with different keys several times and got the minimum latency. All the requests were sent from the server connected to the FPGA NIC with a 10 Gbps interconnect. The result is shown in Table IV. According to this table, we can estimate that the latency of the NIC cache was 8 µs (17 µs - 9 µs) and the latency of the host CPU was 242 µs (251 µs - 9 µs)

Based on the minimum latencies and the hit rates, we estimated the maximum improvement of our system for GET requests compared to using only the CPU (Fig. 9). The estimation was done with the following formula.

$$242\mu s/(hit\_rate \times 8\mu s + miss\_rate \times 242\mu s) \quad (1)$$

For workload A and B (Zipfian distribution), the latency improved at a maximum by three to four times. For Workload D (Latest distribution), the latency improved at a maximum by about 15 times. Since the system was implemented with a read-no-allocate policy, the improvement of the latency of Workload C (Zipfian distribution) was a little less than Workload A and B.

VI. DISCUSSION AND FUTURE WORK

In order to keep the implementation simple and avoid data inconsistency between the NIC cache and memcached, we decided to employ a read-no-allocate policy. As a consequence, this leads to a decrease in the hit rate for Workload C, which has only GET requests. However, employing read-allocate will lead to a drop of NIC cache's average latency. Finding a

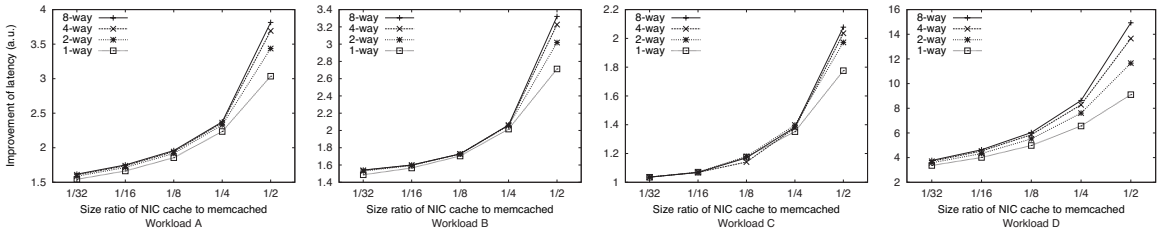

Fig. 9. Latency improvement with various associativity and cache capacity.

TABLE IV. LATENCIES OF THE SYSTEM.

Network	9 µs
Reply from NIC (NIC cache hit)	17 µs
Reply from host CPU (SET)	251 µs

solution to keep data consistency and high performance at the same time is one of the largest tasks remaining.

Another limitation in this paper is that YCSB, the benchmarking tool we used, uses fixed sized keys and values for evaluation. If the web server sends a SET request to our system with variable key and value sizes, while we have fixed sized space to store the KVP as described in this paper, we have to ignore the request at the NIC and leave it to the CPU to handle. This will lead to a decrease in the hit rate of GET requests and thus the performance of the system will degrade. To overcome this problem, we should employ a method to accept any key and value sizes with an efficient memory allocation technique.

The low hit rate for requests with Zipfian selection distribution due to low hit rate at the NIC is also a problem that we have to deal with. Using least frequently used (LFU) algorithm for cache replacement may offer a solution. Since a wider range of keys are requested but the intensively called keys are very few with Zipfian distribution compared to Latest distribution, the LFU will prevent the intensively used keys from being evicted from the cache.

Finally, the relation between the hit rate and the width of the tag stored in the hash table should be investigated. If the width of the tag is smaller, the hash table can hold more entries with the same amount of block memory in the FPGA. However, in such a case, the chance of a KVP being overwritten by a different KVP with the same tag will increase and thus the hit rate will decrease. The balance between larger entries and the risk of being overwritten should be evaluated.

VII. CONCLUSION

In this paper, we proposed a method to improve the latency of memcached by caching its data at the NIC and replying to the client immediately from the NIC when the requested data is found. The evaluation was done with a common KVS evaluation tool, YCSB. With the cache parameters determined through software simulation, the hardware evaluation showed that our method improves the latency by up to 15-fold for GET requests for keys with Latest distribution compared to a Xeon. We will try brushing up our method further by testing other caching policies, supporting variable key and value sizes, and other optimizations. We believe that our approach to improve the performance of the application by caching the data at the NIC is applicable to other applications as well. We will try generalizing our method as a new computation architecture.

REFERENCES

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 205–220.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.

[3] Y. Xu, E. Frachtenberg, S. Jiang, and M. Palecezny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 99, pp. 41–49, 2014.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, 2013, pp. 385–398.

[5] http://memcached.org/.

[6] W. Lang, J. M. Patel, and S. Shankar, "Wimpy node clusters: What about non-wimpy workloads?" in *Proceedings of the 6th International Workshop on Data Management on New Hardware*, 2010, pp. 47–55.

[7] 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 40th Annual International Symposium on Computer Architecture*, 2013, pp. 36–47.

[8] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, vol. 99, pp. 1–4, 2013.

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Proceedings of the 2nd International Green Computing Conference and Workshops*, 2011, pp. 1–8.

[10] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An fpga memcached appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2013, pp. 245–254.

[11] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István, "Achieving 10gbps line-rate key-value stores with fpgas," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013, pp. 1–6.

[12] A. Wiggins and J. Langston, "Enhancing the scalability of memcached," http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0.

[13] S. Tanaka and C. Kozyrakis, "High performance hardware-accelerated flash key-value store," 2014, presented in the 5th Annual Non-Volatile Memories Workshop.

[14] "Convey computer memcached appliance," http://www.conveycomputer.com/files/1813/7998/4963/CONV-13-046_MCD_Datasheet.pdf.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.

[16] B. Jenkins, "Lookup3.c, for hash table lookup," 2006, http://burtleburtle.net/bob/c/lookup3.c.