Regular Paper

Enhancing Memcached by Caching Its Data and Functionalities at Network Interface

Eric S. Fukuda[1,a)] Hiroaki Inoue[2,b)] Takashi Takenaka[2,c)] Dahoo Kim[1,d)]
Tsunaki Sadahisa[1,e)] Tetsuya Asai[1,f)] Masato Motomura[1,g)]

Received: June 30, 2014, Accepted: December 3, 2014

Abstract: Memcached has been widely accepted as a technology to improve the response speed of web servers by caching data on DRAMs in distributed servers. Because of its importance, the acceleration of memcached has been studied on various platforms. Among them, FPGA looks the most attractive platform to run memcached, and several research groups have tried to obtain a much higher performance than that of CPU out of it. The difficulty encountered there, however, is how to manage large-sized memory (gigabytes of DRAMs) from memcached hardware built in 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. In this paper, we describe the architecture of the proposed NIC cache, and evaluate the effectiveness with a standard key-value store (KVS) benchmarking tool. Our evaluation shows that our system is effective if the workload has temporal locality but does not handle workloads well without such a characteristic. We further propose methods to overcome this problem and evaluate them. As a result, we estimate that the latency improved by up to 3.5 times over software memcached running on a high performance CPU.

Keywords: memcached, key-value store, network interface card, cache, FPGA

1. Introduction

Web service providers that have a large number of users 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 [8], Bigtable at Google [6], memcached at Facebook [17], and many others. Memcached [1] is a technology that reduces the latency of data retrieval by storing KVPs in distributed servers' memories instead of fetching them 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 [1]. According to Facebook's research on their own memcached workloads, they use hundreds of memcached servers [14], [17]. 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 [11], [12], [13] to many-core processors [3] and FPGAs [5]. Meanwhile, FPGA-based memcached systems are outperforming high performance CPUs such as Intel® Xeon® by an order of magnitude [4].

Although these efforts have improved the performance of memcached, major challenges remain. One such challenge is that it is difficult for FPGAs to manage efficiently 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 manage efficiently [16]. One group is trying to handle a large memory size by utilizing a CPU core that is embedded in the FPGA [4]. 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 [15].

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.

[1] Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Hokkaido 060–0814, Japan
[2] NEC Corporation, Kawasaki, Kanagawa 211–8666, Japan
[a)] fukuda@lalsie.ist.hokudai.ac.jp
[b)] h-inoue@ce.jp.nec.jp
[c)] takenaka@aj.jp.nec.jp
[d)] kim@lalsie.ist.hokudai.ac.jp
[e)] sadahisa@lalsie.ist.hokudai.ac.jp
[f)] asai@ist.hokudai.ac.jp
[g)] motomura@ist.hokudai.ac.jp

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.

This paper is an extensive work of our recent conference paper [9] whose contributions were as follows:

- We proposed a method that reduces the delay of memcached by caching the memcached data at the NIC mounted on the server.
- We explained 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 verified the improvement of the performance with a standard evaluation tool which is capable of evaluating various KVSs.

Our new contributions in this paper are as follows:

- We apply the least frequently used (LFU) cache replacement algorithm that has a slightly higher hit rate rather than the least recently used (LRU) algorithm for workloads with Zipfian distribution. However LRU is a better choice when taking the hardware implementation cost into account.
- We analyze the effect of our system in relation to the workloads' characteristics in detail.
- We propose a method that enables a large size of cache on the NIC with a small amount of block memory on the NIC's FPGA.

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 a lower latency as long as the data has temporal locality.

2. Background

2.1 Memcached

Memcached is a kind of KVS database that 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.

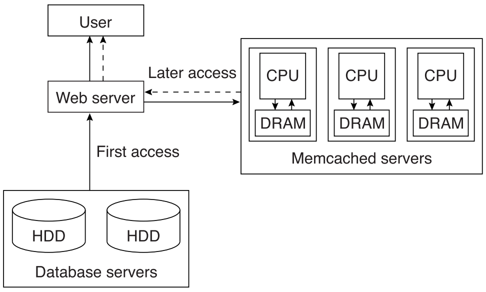


Fig. 1 The operation of memcached.

Table 1 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 [17], 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 2.2 usually support only the GET, SET and DELETE commands.

2.2 Related Work

Berezecki et al. evaluated the performance of memcached running on Tilera's TILEPro64 processor, which can allocate computations to its 64 cores [3]. 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 [5]. 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 reads from

Table 1 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.

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 [4]. They achieved over 15-fold higher throughput per watt than a Xeon and a latency of 3.5 to 4.5 μs. This work also features a CPU for allocating the memory. Their system stores the key in the block RAM and the value in the external DRAM. The system we propose in this paper is different from this: we store only the tag in the block RAM and store the key and the value to the external DRAM. This enables us to store more KVPs in the external DRAM, and as we propose in Section 6, this will further be extended to the hash table compression method.

Another approach was proposed by two groups almost coincidently [12], [13]. 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 was 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 Refs. [12] and [13] 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 simpler.

To gain a larger storage size on hardware memcached, Tanaka and Kozyrakis employed a solid state drive (SSD) in their FPGA based system [15]. 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- 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 [2]. 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.

3. Concept of NIC Cache

The basic idea of our method is to cache part of a 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 [17]. 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 is used 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**).

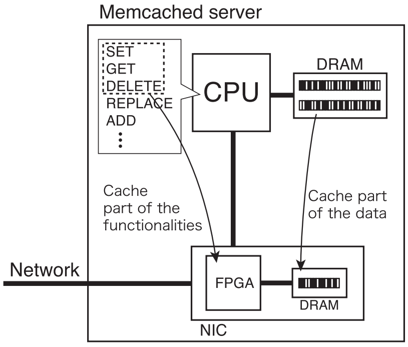


Fig. 2 The image of the proposed method.

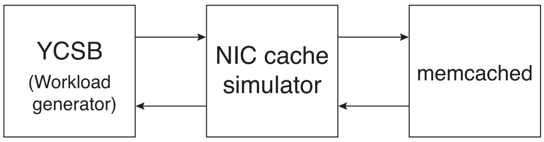


Fig. 3 Connection of software modules.

We assume our system behaves as follows. However, this is an example of adopting the FPGA NIC for memcached, and the behavior can be changed and adapted to various applications.

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.

4. 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 3. Test workloads were generated with *Yahoo! Cloud Serving Benchmark* (YCSB), a standard benchmarking tool for KVS [7]. 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 3, backed up by real memcached.

4.1 Testing Tool

YCSB provides workloads that simulate various KVS use

Table 2 Description of YCSB workloads. (Originally shown in Ref. [7].)

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 to 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)

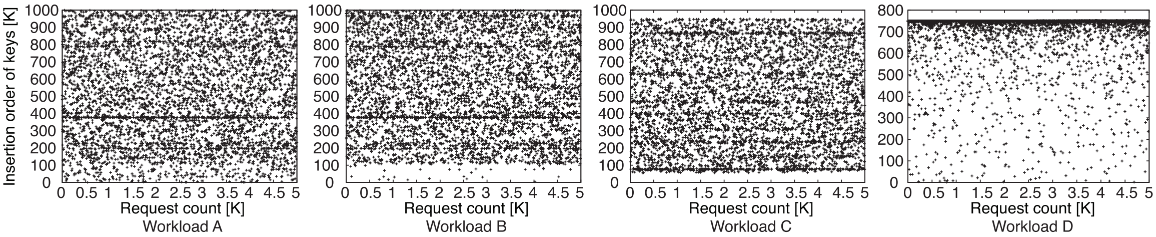


Fig. 4 Key access distribution for the first 5,000 requests.

cases. **Table 2**, quoted from Ref. [7], shows the characteristics of the workloads. Each workload is characterized by the ratio of commands and the record selection distribution. YCSB has a load phase, which sends SET requests for all the keys for the warm up, and transaction phase, which sends requests with the characteristics given in Table 2. All the results provided in this paper are measured during the transaction phase.

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.

Figure 4 shows the access to each key for the first 5,000 requests. The x-axis is the number of requests, which approximately represents the time, while the y-axis shows the keys ordered according to their first appearance. You can see some stripes which are the popular keys in the Zipfian distribution (Workloads A to C), and the key that appeared the last (the top in the chart) is intensively requested in Latest distribution (Workload D).

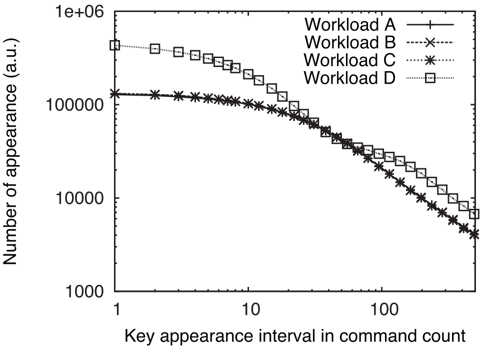


Fig. 5 Appearance interval of same keys for all workloads.

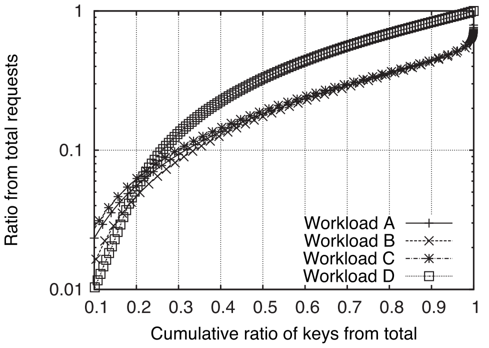
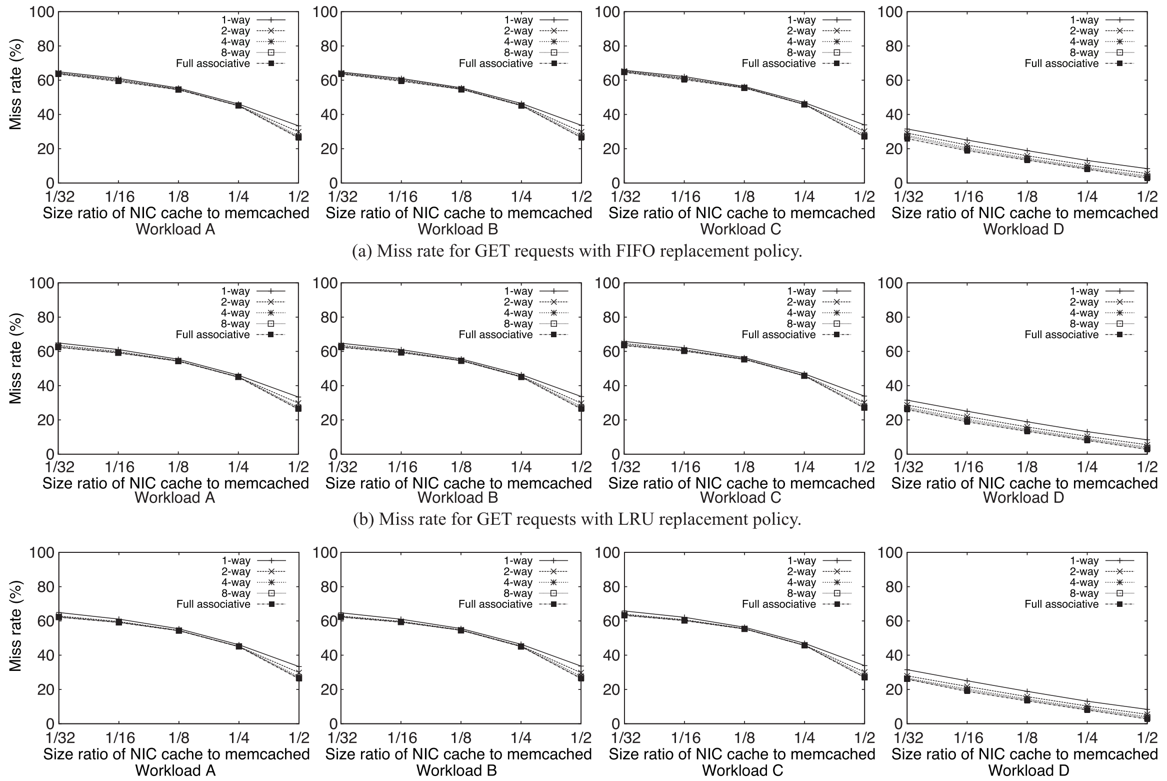


Fig. 6 Cumulative ratio of keys.

Figure 5 shows that Workload D is unique in terms of the key appearance interval. The x-axis denotes the appearance interval of the same keys while the y-axis denotes the total number of each appearance interval. This figure signifies that Workload D has relatively fewer intervals between the same keys compared to Workloads A to C.

Figure 6 shows the appearance of keys in the form of cumulative distribution function (CDF). The x-axis is the ratio of the keys from the total keys, arranged in ascending order of their appearance ratio from total requests. The y-axis is the cumulative ratio of keys from total requests. Figure 6 indicates that Workload D differs from Workloads A to C also in the characteristic



(a) Miss rate for GET requests with FIFO replacement policy.

(b) Miss rate for GET requests with LRU replacement policy.

(c) Miss rate for GET requests with LFU replacement policy.

Fig. 7 Miss rate for GET requests with various replacement policies.

of key appearance. In the figure, the right ends of Workloads A to C's graphs become almost vertical. This means that there is a large gap between the appearance probabilities of the popular keys, which make the dense areas in Fig. 4, and that of the rest of the keys.

4.2 Simulation Results

Figure 7 (a), (b) and (c) show the miss rates for GET requests at the NIC cache with various associativity and capacity for FIFO, least recently used (LRU) and least frequently used (LFU) replacement algorithms respectively. The x-axis is the relative ratio of the NIC cache capacity to the memcached capacity running on the host CPU. We set the memcached capacity to 512 MB and evaluated the cache size parameter from 1/32 to 1/2. The actual cache size we implemented, which we will discuss in Section 5, was 128 MB and this is 1/4 of the memcached capacity. (Rather than the absolute cache size, the ratio of the cache size to the host memcached size determines the miss rate.)

Apparently, the difference in miss rates among the three algorithms is very small. For Workloads A to C, the miss rates are a few percent less with LRU and LRU than with FIFO when the capacity of the NIC cache is small. You can also see that Workloads A to C, which use the Zipfian distribution, have similar curves, while Workload D with Latest distribution have linearly decreasing miss rates as the NIC cache's capacity increases. (For better visibility of this, **Fig. 8** features the miss rates for small cache sizes for Workload A with the three different algorithms.) As we mentioned earlier, workloads with Zipfian distribution have specific keys that are popular independently from when the key has recently been called. This makes us think that LFU, which tries to leave the popular keys in the cache, is a good solution for reducing the cache miss rate. However, LFU had a larger effect, while the difference was still very small, than LRU only when the cache size was small (1/32 or 1/16 of memcached) and the cache associativity was relatively small (2-way or 4-way). This ineffectiveness comes from the very small number of popular keys in Workloads A to C: The LFU has effect only when the ratio of the popular keys is relatively large in the cache.

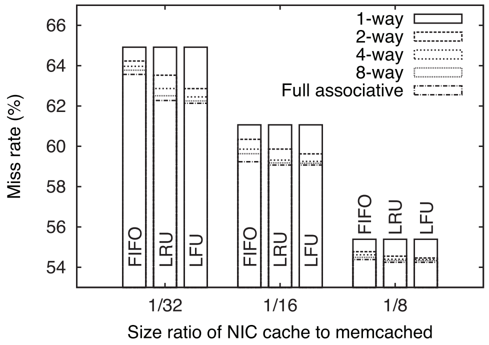


Fig. 8 Miss rates for small cache sizes for Workload A with FIFO, LRU, and LFU cache algorithms.

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 more easily. 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 a request

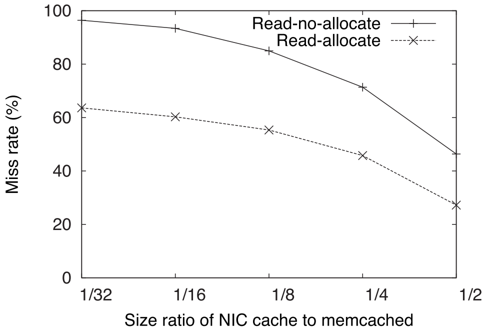


Fig. 9 Miss rates with read-allocate and read-no-allocate for Workload C.

from the NIC to the host CPU synchronously, or employing a read-no-allocate policy. Since synchronous requests can lead to an increase in the average 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. 9**). This degradation comes from the command mix of Workload C. Unlike Workloads 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.

5. Hardware Design

To prove the proposed method works correctly, we designed and implemented the system on an FPGA NIC. Note that the system described below is meant for making sure that the method we proposed above works under a simple one-to-one connection between the server and the client.

Although the cache 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. **Figure 10** 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 whether hit or miss is returned from the hash table.

Outgoing packet handler: The 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 this behavior is a part of our future work.

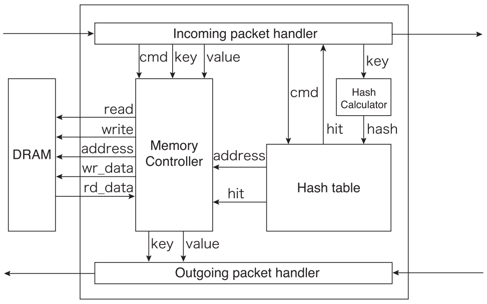


Fig. 10 NIC cache architecture.

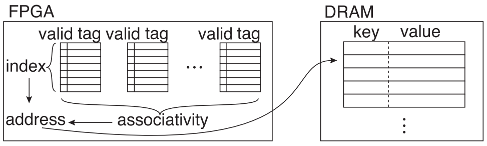


Fig. 11 Correspondence of the hash table and the value storage.

Hash calculator: The hash calculator receives a key from the incoming packet handler and calculates a hash with Jenkins's lookup3 function [10]. It produces a 32-bit hash from the key.

Hash table: The hash table manages where in the DRAM memory to store the KVP. A more detailed structure is given in **Fig. 11**. The top 15 bits of the hash given from the hash calculator become 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. Although LFU have a slightly better hit rate for a small size and low associativity cache, we chose to implement pseudo LRU due to its lower implementation cost. 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. DELETE invalidates the valid flag if the hash value stored in the entry 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. The memory controller also has a cache inside, which reduces the latency of external DRAM access.

5.1 Experimental Conditions

We used UDP protocol for the communication between the computer that runs YCSB and the computer that has the FPGA NIC and runs memcached. 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. **Figure 12** depicts the FPGA NIC mounted on a memcached server. 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. **Table 3** shows the resource usage.

5.2 Latency

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 5, sent GET requests for the same key for several times, and got the minimum average. Finally, we sent SET requests with different keys several times and got the average latency. All the requests were sent from the server connected to the FPGA NIC with a 10 Gbps interconnect. The results are shown in **Table 4**. According to this table, we can estimate that the latency of the NIC cache was 20 µs (29 µs − 9 µs) and the latency of the host CPU was 78 µs (87 µs − 9 µs).

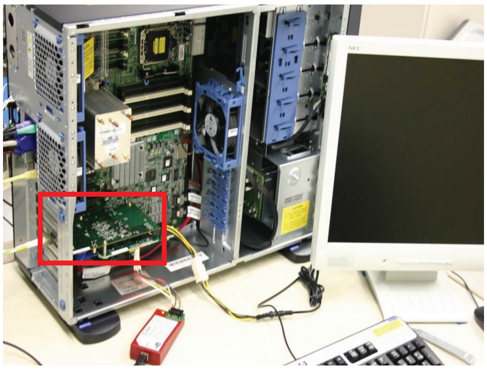


Fig. 12 FPGA NIC mounted on a memcached server.

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. 14**). The estimation was done with the following formula.

$$87\mu s/(hit\_rate \times 20\mu s + miss\_rate \times 87\mu s) \qquad (1)$$

For Workloads A and B (Zipfian distribution), the latency improved at a maximum by about two times, and for Workload D (Latest distribution), the latency improved at a maximum by 3.5 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 for Workloads A and B.

5.3 Throughput

Next, we evaluated the throughput of our system. We used RTL simulation to estimate the throughput. We gave a SET-only workload, a GET-only (100% hit) workload, and a GET-only (100% miss) workload as input. **Table 5** shows the results. The numbers have certain ranges whose lower bound corresponds to 0% hit at the memory controller cache and upper bound to 100% hit. Since our DRAM access module in the memory controller is not optimised, the throughput is relatively slow compared to other works [4].

Based on these numbers, **Fig. 13** estimates the throughput with different miss rates at the NIC cache. The vertical lines indi-

Table 3 Design specification of FPGA.

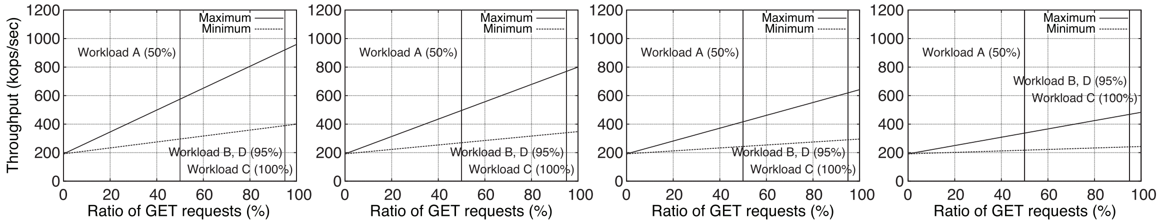
Number of used block RAM and FIFO	238 / 324 (86%)
Number of used slice LUTs	60314 / 207360 (29%)
Number of used lice Registers	64505 / 207360 (31%)

Table 4 Latencies of the system.

Network	9 µs
Reply from NIC (NIC cache hit)	29 µs
Reply from host CPU (SET)	87 µs

Table 5 Throughputs of the system based on RTL simulation.

Command	Throughput (ops/sec)
SET	191,424
GET (100% hit)	399,680-958,772
GET (100% miss)	138,465-165,480



(a) Throughput for 0% miss rate (b) Throughput for 20% miss rate (c) Throughput for 40% miss rate (d) Throughput for 60% miss rate

Fig. 13 Throughput of the system with various hit rates.

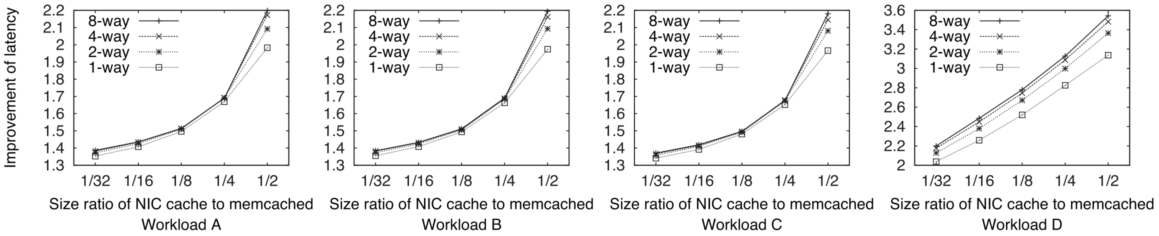
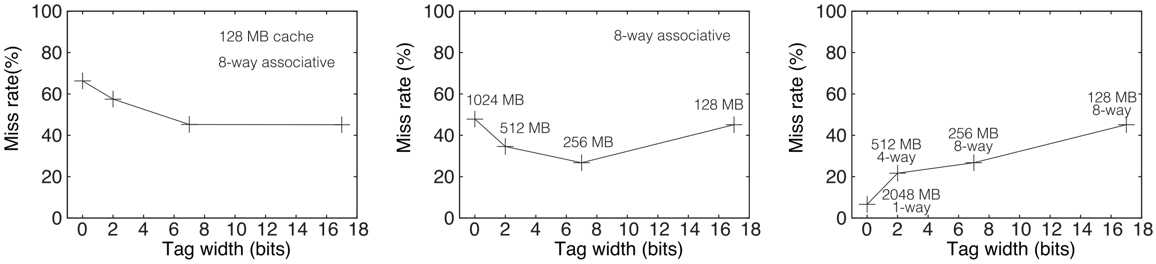


Fig. 14 Latency improvement with various associativity and cache capacity.



(a) Miss rates with variable tag width, constant cache size, and constant associativity.

(b) Miss rates with variable tag width, variable cache size, and constant associativity.

(c) Miss rates with variable tag width, variable cache size, and variable associativity.

Fig. 15 Miss rates with constant block memory size.

cate the command mixes of the workloads; therefore the crossing points show the actual throughputs. In each graph, the throughput increases as the ratio of GET requests to the whole (SET and GET) requests increases. The graphs that are labeled "maximum" are the results for when the cache in the memory controller had 100% hit, and the "minimum" are for 0% hit. The actual throughput will be between the maximum and the minimum depending on the hit rate at the memory controller cache. For example, Fig. 13 (d) shows that the throughput for workload A with cache size of 1/16 of the host DRAM (the miss rate is around 60% according to Fig. 7 (b)) is between 210,000 to 350,000 ops/sec.

6. Cache Size Maximization

The cache size of our system is determined by the number of the entries in the hash table implemented on the block memories in the FPGA. In other words, the size of the available block memories can become a bottleneck if a larger cache size is required. In fact, as shown in Table 3, we have already used 86% of the block memories for having 128 MB cache, so it is difficult to have a larger cache size. In this section, we consider and evaluate a method that enlarges the cache size with a limited amount of block memories by narrowing the tag width.

In our system, we store tags, which are the lower 17 bits of the hash value calculated from the key, instead of storing the key itself in the hash table. (Along with the 17-bit tag, the cache uses 1-bit valid bit and 1-bit MRU bit. MRU bit is used for implementing pseudo-LRU. Throughout this paper, we do not include the valid bit and the MRU bit in the term "tag.") Therefore, on a SET, a certain KVP in the cache can be overwritten by another KVP which has a different key but has the same index and tag. On a GET however, the system ensures that it will not return a KVP that was not requested by checking whether the key in the retrieved KVP from the DRAM matches the requested key. Conversely, it is possible to reduce the block memory usage by making the width of the tag smaller as long as the retrieved key is checked. For such a purpose, we investigated the relations between the tag width and the miss rate.

First, we investigated the relation between the tag width and the miss rate. **Figure 15** (a) shows the miss rate with tag widths of 0, 2, 7 and 17. (As we mentioned above, these numbers do not include the valid bit and the MRU bit.) We used Workload A and an 8-way associative hash table for this evaluation. The figure shows that there is little difference in miss rates between the cases of 17-bit and 7-bit tags. As the tag width gets narrower towards the left, the miss rate becomes larger due to the increase of the chances of overwriting the keys with different keys.

Next, we evaluated the the effect of narrowing the tag and enlarging the cache size with constant block memory size (Fig. 15 (b)). We used Workload A and an 8-way associative hash table also in this evaluation. As the tag width gets narrower, there is more room in the block memories for increasing the hash table's index size, and therefore the cache size can be increased. When the tag is narrowed from 17 bits to 7 bits, the miss rate decreases because the cache size increases while the chance of KVPs being overwritten does not increase. For smaller tag widths, however, the negative effect of overwriting the key becomes larger than the positive effect of larger cache sizes, and result in an increase of the miss rate.

The block memories can further be exploited. Figure 15 (c) shows the hit rates of when the block memories are fully exploited for each tag width. Although if the hash table is 8-way associative, only four different tags can be stored in a single row when the tag width is two ($4 = 2^2$). Instead, we reduced the associativity to four and doubled the number of the index when the tag width is two. In the case of 0-bit-width tags, in other words, in the case of no tags, not only the tags but also the MRU bits are no longer necessary since the associativity is virtually 1-way. The hash table can then consist only of valid bits. Therefore the cache size can be twice as large as that of when the MRU bit still exists. Although the chances of keys overwritten by different keys increase, the effect of increase in cache size outraces such effect and therefore the miss rate decreases when the tag width becomes smaller. Throughout these experiments, it can be said that the cache size can be increased without worsening the miss rate even though the tag width is narrowed.

7. 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 instead will lead to a drop of NIC cache's average latency. Finding a 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.

Although there is only 128 MB cache on our NIC, it can be expanded in two ways. First, as we discussed in Section 6, the cache size can be doubled without increasing the miss rate by narrowing the tag width. (The cache size is proportional to the size of the hash table.) Second, some of the recent FPGAs like Vertex UltraScale have more than ten times of block RAMs than the one we used has. Altogether, there can be a 20 times larger cache (2.5 GB) than the current size (128 MB) on the NIC. In this case, our evaluation considers 5 to 80 GB cache.

Compared to CPU caches, our NIC cache has higher miss rates. We found out in our previous work [9] that the NIC cache does not show high hit rate for workloads with Zipfian key distribution. Therefore in this paper, we tried LFU, a cache replacement algorithm that leaves the popular keys in the cache, expecting the hit rate to improve. However, LFU had almost no effect. This is because the popular keys in the YCSB's workloads with Zipfian distribution is so few that they could remain in the cache even with other cache replacement algorithms. Our next goal is to improve the hit rates for workloads with Zipfian distribution.

8. 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 3.5-fold for GET requests for keys with the Latest distribution compared to a Xeon. Our further investigation showed that the size of the block RAM on the FPGA is less likely to become the bottleneck of the cache size if the tag width of the cache is narrowed. We simplified our method by fixing the sizes of the key and the value, and the hit rate might drop if we adopt variable key and value sizes. We will try improving the hit rate by employing a better cache algorithm and by utilizing the DRAM with an efficient memory allocation method. Furthermore, 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] Memcached - a distributed memory object chaching system, available from ⟨http://memcached.org/⟩.

[2] Convey Computer Memcached Appliance, available from ⟨http://www.conveycomputer.com/files/1813/7998/4963/CONV-13-046_MCD_Datasheet.pdf⟩.

[3] Berezecki, M., Frachtenberg, E., Paleczny, M. and Steele, K.: Many-core Key-value Store, *Proc. 2nd International Green Computing Conference and Workshops*, pp.1–8 (2011).

[4] Blott, M., Karras, K., Liu, L., Vissers, K., Bär, J. and István, Z.: Achieving 10Gbps Line-rate Key-value Stores with FPGAs, *Proc. 5th USENIX Workshop on Hot Topics in Cloud Computing*, pp.1–6 (2013).

[5] Chalamalasetti, S.R., Lim, K., Wright, M., AuYoung, A., Ranganathan, P. and Margala, M.: An FPGA Memcached Appliance, *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp.245–254 (2013).

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Comput. Syst.*, Vol.26, No.2, pp.4:1–4:26 (2008).

[7] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proc. 1st ACM Symposium on Cloud Computing*, pp.143–154 (2010).

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store, *Proc. 21st ACM Symposium on Operating Systems Principles*, pp.205–220 (2007).

[9] Fukuda, E.S., Inoue, H., Takenaka, T., Kim, D., Sadahisa, T., Asai, T. and Motomura, M.: Caching Memcached at Reconfigurable Network Interface, *Proc. 24th International Conference on Field Programmable Logic and Applications* (2014).

[10] Jenkins, B.: LOOKUP3.C, for hash table lookup (2006), available from ⟨http://burtleburtle.net/bob/c/lookup3.c⟩.

[11] Lang, W., Patel, J.M. and Shankar, S.: Wimpy Node Clusters: What About Non-wimpy Workloads?, *Proc. 6th International Workshop on Data Management on New Hardware*, pp.47–55 (2010).

[12] Lavasani, M., Angepat, H. and Chiou, D.: An FPGA-based In-line Accelerator for Memcached, *IEEE Computer Architecture Letters*, Vol.99, pp.1–4 (2013).

[13] Lim, K., Meisner, D., Saidi, A.G., Ranganathan, P. and Wenisch, T.F.: Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached, *Proc. 40th Annual International Symposium on Computer Architecture*, pp.36–47 (2013).

[14] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T. and Venkataramani, V.: Scaling Memcache at Facebook, *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation*, pp.385–398 (2013).

[15] Tanaka, S. and Kozyrakis, C.: High Performance Hardware-Accelerated Flash Key-Value Store (2014). Presented in the 5th Annual Non-Volatile Memories Workshop.

[16] Wiggins, A. and Langston, J.: Enhancing the Scalability of Memcached, available from ⟨http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0⟩.

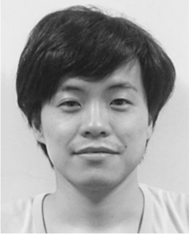
[17] Xu, Y., Frachtenberg, E., Jiang, S. and Palecezny, M.: Characterizing Facebook's Memcached Workload, *IEEE Internet Computing*, Vol.99, pp.41–49 (2014).

Eric S. Fukuda received his B.S. degree in electronic engineering from Hokkaido University in 2007 and received his M.S. in information science from the University of Tokyo in 2009. After working at ACCESS as a software engineer, he is currently a Ph.D. candidate at the Graduate School of Information Science and Technology, Hokkaido University. His interests include reconfigurable architectures and data mining. He is a student member of IEICE and IPSJ.

Hiroaki Inoue received his B.S., M.E. and Ph.D. degrees from Keio University in 1997, 1999 and 2009, respectively. He joined NEC Corporation in 1999, and is now a Manager of Corporate Technology Division. From 2007 to 2008, he was a visiting scholar of Stanford University. His current research interests include real-time computing platforms. He is a senior member of IEEE, and a member of IEICE.

Takashi Takenaka received his M.E. and Ph.D. degrees from Osaka University in 1997 and 2000 respectively. He joined NEC Corporation in 2000 and is currently a principle researcher of NEC Corporation. He was a visiting scholar of the University of California, Irvine from 2009 to 2010. His current research interests include system-level design methodology, high-level synthesis, formal verification, and stream processing. He is a member of IEEE, IEICE and IPSJ.

Dahoo Kim received his B.S. degree in electronic engineering from Hokkaido University in 2013. He is currently a M.S. candidate at the Graduate School of Information Science and Technology, Hokkaido University. His interests include reconfigurable architecture and low power computer design.



Tsunaki Sadahisa received his B.S. degree in electronic engineering from Hokkaido University in 2014, and he is currently a M.S. student at Hokkaido University. His interests include reconfigurable architectures and data mining.

Tetsuya Asai received his B.S. and M.S. degrees in electronic engineering from Tokai University, Japan, in 1993 and 1996, respectively, and his Ph.D. degree from Toyohashi University of Technology, Japan, in 1999. He is now an Associate Professor in the Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan. His research interests are focused on developing nature-inspired integrated circuits and their computational applications. Current topics that he is involved with include intelligent image sensors that incorporate biological visual systems or cellular automata in the chip, neurochips that implement neural elements (neurons, synapses, etc.) and neuromorphic networks, and reaction-diffusion chips that imitate vital chemical systems.

Masato Motomura received his B.S. and M.S. degrees in physics and Ph.D. degree in electrical engineering in 1985, 1987, and in 1996, all from Kyoto University. He was with NEC and NEC Electronics from 1987 to 2011, where he was engaged in the research and business development of dynamically reconfigurable processor and on-chip multi-core processor. Now a professor at Hokkaido University, his current research interests include reconfigurable and parallel architectures and low power circuits. He has won the IEEE JSSC Annual Best Paper Award in 1992, IPSJ Annual Best Paper Award in 1999, and IEICE Achievement Award in 2011, respectively. He is a member of IEICE and IEEE.