MultiMQC: A Multilevel Message Queuing Cache Combining In-NIC and In-Kernel Memories

Koya Mitsuzuka, Yuta Tokusashi, and Hiroki Matsutani
Dept. of ICS, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
Email: {koya,tokusasi,matutani}@arc.ics.keio.ac.jp

***Abstract*—Message queuing systems that deliver messages from publishers to subscribers play an important role to collect data from IoT devices. Traditional message queuing systems have improved their performance in the context of transferring log data from publishers such as Web servers to subscribers that analyze the log data. In this case, both publishers and subscribers have been assumed to have enough buffer capacity and can transfer data as jumbo frame packets for high efficiency. In recent IoT applications, however, publishers are small sensors or edge devices with low-power processors and limited memory capacity. Vast numbers of such publishers produce relatively small packets. Such a lot of small messages significantly decrease the efficiency of conventional message queuing systems. To address this issue, a dedicated message queuing logic can be implemented on FPGA-based network interface card (FPGA NIC). However, a serious issue of such in-NIC approach is a limited memory capacity on the FPGA NIC. To handle message overflow of the in-NIC cache, in this paper, it is combined with a large in-kernel software cache. More specifically, we propose a multilevel message queuing cache combining in-NIC and in-kernel memories, called MultiMQC. The multilevel cache improves the read performance. Regarding the write performance, MultiMQC introduces a batch transfer that packs small incoming messages into a single batch. We implemented MultiMQC using NetFPGA-SUME board as in-NIC cache and Linux Netfilter framework as in-kernel cache. The experimental results demonstrate that the write throughput is increased in proportion to the batch size. When pull requests hit in the in-NIC cache, the read throughput reaches 95.8% of 10GbE line rate in four 10GbE interfaces.**

I. Introduction

Utilizing big data collected from vast numbers of IoT (Internet of Things) devices will make positive impacts on our society. Because these IoT devices transmit a lot of small messages to datacenters, their communication imposes a significant impact on the datacenters. Message queuing systems have been used as an infrastructure for loosely-coupled asynchronous communications between publishers and subscribers. That is, their communications are decoupled for higher flexibility and efficiency. For example, AWS (Amazon Web Services) provides sophisticated message queuing services as a part of AWS IoT [1].

Traditional message queuing systems have improved their performance in the context of transferring log data from publishers such as Web servers to subscribers that analyze the log data. In this case, both publishers and subscribers have been assumed to have enough buffer capacity and can transfer data as jumbo frame packets for high efficiency. In recent IoT applications, however, publishers are small sensor or edge devices with low-power processors and limited memory capacity. Vast numbers of such publishers produce small packets, which significantly decrease the efficiency of conventional message queuing systems.

To address this issue, in this paper, first, we propose a dedicated message queuing cache implemented on an FPGA-based network interface card (FPGA NIC). This approach is referred as in-NIC cache. However, a serious issue of the in-NIC cache approach is a limited DRAM capacity on the FPGA board, because its small cache capacity causes cache misses for read requests from subscribers. To improve the cache hit rate, in this paper, the in-NIC cache is complemented by a software cache working at Linux kernel space, referred as in-kernel cache. More specifically, we propose a multilevel message queuing cache combining in-NIC and in-kernel memories, called MultiMQC. This multilevel cache improves the read performance of message queuing systems. Regarding the write performance, MultiMQC introduces a batch transfer that packs multiple incoming messages from publishers into a single batch to reduce the number of write operations. In the batch transfer, messages waiting for the batch write operation can be read by subscribers so that latency overhead for the batch transfer can be hidden. We implemented MultiMQC using NetFPGA-SUME board as in-NIC cache and Linux Netfilter framework as in-kernel cache. MultiMQC provides the similar APIs as the original message queuing middleware; in other words it is transparent to publishers and subscribers, while it can improve the message queuing latency and throughput.

The rest of this paper is organized as follows. Section II surveys message queuing systems and related work. Section III proposes MultiMQC and Section IV illustrates the implementation. Section V evaluates it in terms of read and write performance and Section VI concludes this paper.

II. Background

A. Message Queuing Systems

Major message queuing systems, such as Apache ActiveMQ [2], employ Advanced Message Queuing Protocol (AMQP) for high reliability. RabbitMQ [3] is a lightweight message queuing system using AMQP. The reliability can be improved by introducing some advanced data cleansing techniques. Apache Kafka [4] is widely used as an efficient message queuing system. It does not employ AMQP in order to improve the efficiency. Although in this paper we assume a Kafka-like system that prioritizes the efficiency as a baseline message queuing system, our MultiMQC can be applied to more reliable message queuing systems by modifying the in-NIC and in-kernel caches to comply with reliable protocols, such as AMQP.

B. Online and Offline Subscribers

Subscribers receive interested messages via a message queuing system and use them for various purposes. For example,

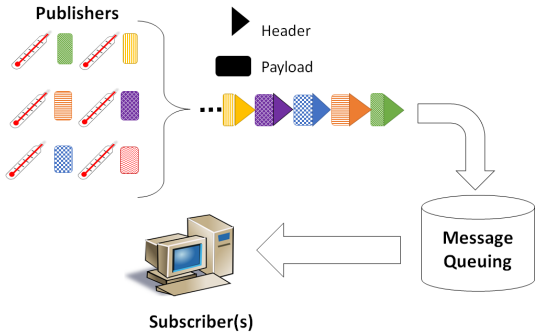


Fig. 1. Publishers, message queueing system, and subscribers

they can monitor messages to detect outliers and alert the anomalies. Also they can perform statistical analysis or data mining on the messages. The former application (referred as online subscriber) imposes a severe latency requirement. Regarding the latter application (referred as offline subscriber), although the latency requirement is not severe, high-volume data from a lot of IoT devices will be kept in the message queuing system for a certain time, requiring a large capacity for the storage.

C. Message Queuing for IoT Applications

As shown in Figure 1, publishers such as IoT devices publish messages to a message queuing system. The messages are read from the message queuing system by subscribers such as data analysis processes. Although their message format varies depending on applications, the messages typically include publisher ID, timestamp, and the value. The value size is quite short in the case of sensor nodes that periodically transmit sensor data. It also tends to be short for resource-limited IoT devices that cannot accumulate data or when immediate responses are required with respect to their data. As shown in Figure 1, although each IoT device produces a low-volume packet stream, the message queuing system receives a high-volume stream from a lot of publishers. Such a communication pattern imposes a significant impact on the typical software-based message queuing system running with standard network protocol stack.

Typical message queuing systems are quite simple; their major functions are temporary message buffering and message transfer. In the proposed MultiMQC, these functions are optimized under the following assumptions.

Append-Only Write: Unlike most database systems, published messages are never modified. More precisely, instead of updating an already-published message, its new version is appended to the message queue. Thus, messages are written in a sequential and append-only manner in the message queuing systems.

Usually Read-Once: Basically, after a subscriber receives a message, the message is no longer requested again unless some failures or rollbacks occur. Please note that a long term buffering is necessary for offline subscribers for statistical analysis or data mining, but most subscribers consume messages in a certain time period and do not request the same messages again. Especially, subscribers for realtime applications tend to consume the latest message as soon as possible. We thus design MultiMQC to store fresh messages in fast and small in-NIC cache for low-latency and volume of messages in large in-kernel cache for favorable cost-performance.

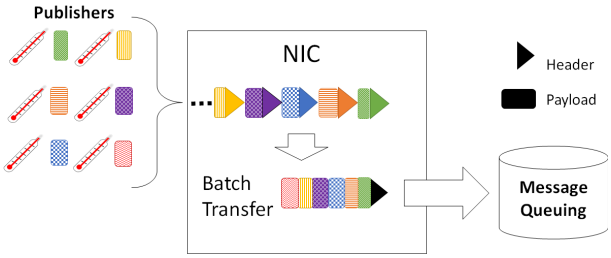


Fig. 2. In-NIC batch processing for incoming messages

D. Network Application Accelerations

Performance of network applications is often limited by packet processing overheads of NIC and kernel network protocol stack. For example, memcached application is reported that processing time for NIC and kernel network protocol stack is longer than that for the software [5]. Various approaches have been studied to overcome the inefficiency of the conventional network protocol stack. One approach is to modify the operating system to mitigate the overheads [6][7]. Another approach is the kernel bypassing that mitigates the packet processing overheads by bypassing the network protocol stack [8][9]. In this case, applications are required to use dedicated APIs or modifications are needed on the network protocol stack. To avoid the negative impacts of a lot of small packets, MultiMQC introduces a batch processing using in-NIC and in-kernel caches, as shown in Figure 2.

Offloading whole or a part of network applications to FPGA NIC drastically improves the packet processing efficiency. For example, a standalone memcached appliance is implemented in an FPGA NIC for high throughput and high energy-efficiency [10]. This is referred as an in-NIC approach. However, it suffers a limited DRAM capacity on the FPGA board. Also, implementing whole or a part of applications to an FPGA requires a high design cost. To address this issue, a multilevel cache combining in-NIC and in-kernel approaches is proposed for NoSQL data stores [11]. Different from the standalone approach, the cache approach is transparent to existing message queuing software. In this paper, we apply the multilevel cache approach to message queuing for high efficiency without modifying the existing message queuing software. The proposed multilevel approach can strike a good balance between the latency for online subscribers and the capacity for offline subscribers.

III. MultiMQC Design

A. Multilevel Organization

The proposed MultiMQC combines in-NIC cache and in-kernel cache. In general it consists of n cache layers. For example, the prototype system implemented in Section IV consists of three cache layers: two layers in NIC and one layer in kernel space (please see Table I). They are denoted as layer-1, layer-2, and layer-3. Figure 3 shows a general structure of MultiMQC that has three cache layers. Please note that MultiMQC works as a multilevel cache for an original message queuing middleware (e.g., Apache Kafka) running on user space. In this paper, it is denoted as layer-4 or last-level

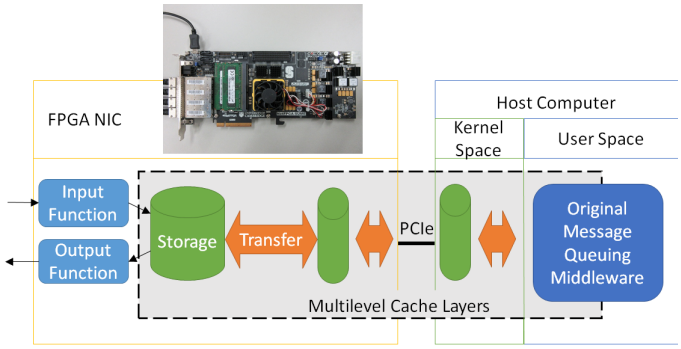


Fig. 3. MultiMQC general structure

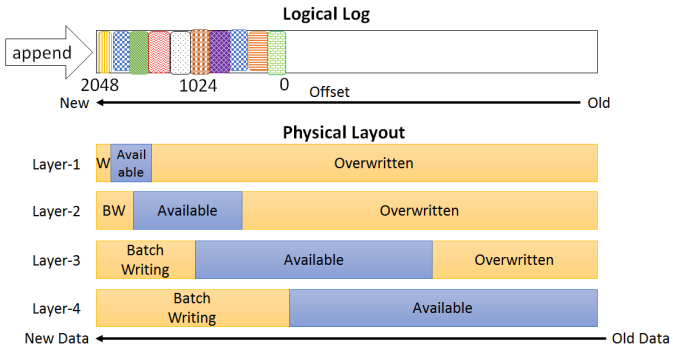


Fig. 4. Logical and physical storage layout in MultiMQC

cache. In addition, application-specific logic for serialization and de-serialization can be implemented in the FPGA NIC as Input Function and Output Function.

MultiMQC employs a simple storage layout like Apache Kafka [4]. Messages are classified into particular topics or types, each of which is corresponding to a logical log. Messages are sequentially stored in the corresponding logical log. Each message is addressed by its offset in the logical log, as shown in the upper part of Figure 4. Storages typically have a trade-off between performance and cost per capacity, e.g., capacity is limited in a faster storage. To strike a balance between low latency and cost efficiency, the log is physically stored in multilevel storages, as shown in the lower part of Figure 4.

B. Write Operation by Publishers

Publishers such as IoT devices publish messages to message queuing system. When a published message arrives at MultiMQC, it is stored in the fastest layer-1 cache (i.e., BRAMs in the FPGA) and then it becomes available for subscribers. A ring buffer is implemented on each cache layer. The oldest messages in layer-i will be overwritten when new messages arrive, depending on the new messages size. Please note that they are asynchronously copied to the layer-$(i+1)$ cache before they are overwritten.

Although write latency in a higher-layer cache is typically higher than that of a lower-layer cache, it can be mitigated by the lower cache layers. Also, capacity of a higher-layer cache is larger than that of a lower-layer cache; thus messages in higher cache layers have a long lifetime (e.g., weeks or months depending on the capacity). The last-level will be a mass storage device attached to host machines with some

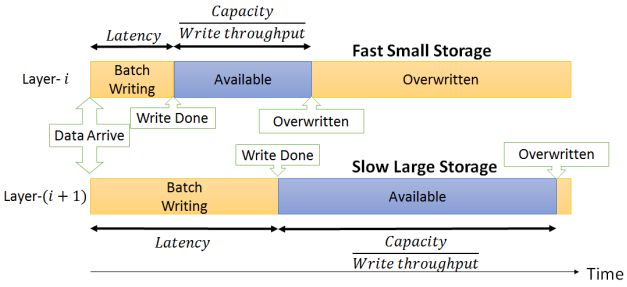


Fig. 5. Example behavior between two layers

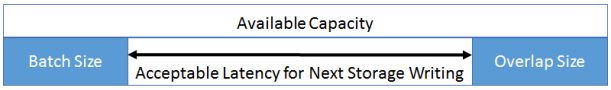


Fig. 6. Relationship between storage parameters and acceptable latency

sophisticated data management techniques (e.g., RAID, page cache, anti-caching [12]).

Figure 5 illustrates an example of lifetime of a message on two cache layers with some performance assumption. Here, the latency is defined as the largest time until a message becomes ready to be read by subscribers after a push request of the message arrives at the NIC. The capacity is the size of ring buffer on each layer and it defines how long a message can reside in the layer. For example, assuming the capacity of layer-i cache is 1GB and write throughput is 100MBps, a message will be overwritten in 10 seconds. Before the message is overwritten, it is transferred to layer-$(i+1)$ cache and gets ready to be read by then. Subscribers can read messages from the lowest layer in which the target messages are alive. The write latency to layer-$(i+1)$ does not affect the subscribers because layer-i keeps the messages while writing them to layer-$(i+1)$.

To minimize the overall latency, latency of layer-$(i+1)$ must be small enough by considering the capacity of layer-i. Figure 6 illustrates how to calculate the maximum acceptable latency of layer-$(i+1)$ to compensate for the capacity of layer-i, based on Equation (1).

$$\text{Latency} < \frac{\text{Capacity} - (\text{Batch Size} + \text{Overlap Size})}{\text{Write Throughput}} \quad (1)$$

Messages are transferred to the next cache layer in a batch manner to improve write throughput. The batch size can be tuned by users. In general, a larger batch size improves the throughput. The overlap size in Equation (1) is defined as the smallest overlap size between layer-i and layer-$(i+1)$ caches. It is set to the same as the MTU (Maximum Transmission Unit) which is approximately 1500 Bytes in Ethernet. In this case, any 1500-Byte ranges reside in either layer-i cache or layer-$(i+1)$ cache when they are cached. If the overlap size is smaller than MTU, requested messages packed in the same packet may span multiple cache layers, which makes the control complicated. Message transfer from layer-i to layer-$(i+1)$ should be completed before the remaining capacity of layer-i is used up. Assuming that a publisher's throughput is 100MBps, the layer-1 cache capacity is 1 MBytes, and the batch size is 512 Bytes, the acceptable latency for the layer-2 cache is $\frac{(1{,}000{,}000-(1500+512))}{100{,}000{,}000} = 9.98$ms. Please note that

such milliseconds-order latency is large enough for the layer-2 cache (i.e., DRAMs on the FPGA board). Those for higher layers will be seconds-order or longer.

C. Read Operation by Subscribers

Subscribers such as data analysis processes subscribe messages in message queuing system. In MultiMQC, they can read messages from any cache layers of MultiMQC. If the range of a pull request is fit in the lowest layer-i cache, requested messages are read from layer-i cache and the reply is sent back to the subscribers. Otherwise, the request is passed to the next layer-$(i+1)$ cache. As long as subscribers continuously consume messages under a condition that the lower-level caches keep all the necessary messages, they can enjoy benefit of faster storages (e.g., BRAMs in FPGA or DRAMs on the FPGA board). In other words, the performance is maximized when subscribers continuously read messages by a certain deadline in order to achieve a minimum gap between data generation and processing. When a subscriber does not require low latency for offline applications, messages may be kept in MultiMQC for a long time. Similarly, when some failures occur or more accurate results are requested, rollback or rereading will be needed, which also demand a long term preservation by higher-level caches.

D. Input and Output Functions

MultiMQC provides Input and Output Functions as shown in Figure 3. In MultiMQC, messages are stored in the same format in all the cache layers and they are addressed by their logical offset. Input Function serializes incoming messages so that they can be placed in the correct offset in the layer-1 cache (i.e., in-NIC BRAM cache). Additional serialization is not needed for data transfer from layer-1 cache to higher-layer caches, because messages are already serialized by Input Function. Because the data format is common in all the layers, subscribers do not have to care which layer the requested message are read from.

In MultiMQC, user-specific Output Functions can be defined. Before a reply is sent back to subscribers, application-specific processing (e.g., extraction and transformation) can be executed on the reply messages. It can be implemented on the FPGA part of in-NIC cache, which is often used for accelerator for stream processing, as shown in Figure 3.

E. Transparency

Each cache layer in MultiMQC provides a common interface for transparency. We can customize structure of the multilevel cache, depending on a given hardware environment. For example, if an in-kernel cache is not available, the in-kernel cache layer can be removed without any changes on in-NIC cache and the original message queuing middleware. On the other hand, when a new storage is available as a part of multilevel cache, the new cache layer can be inserted to improve the cache hit rate and throughput.

To support the above-mentioned transparency, each cache layer should provide the following functions.

- Write function to extract payload of a given push request packet and write it to its storage
- Transfer function to transfer messages to the next layer cache

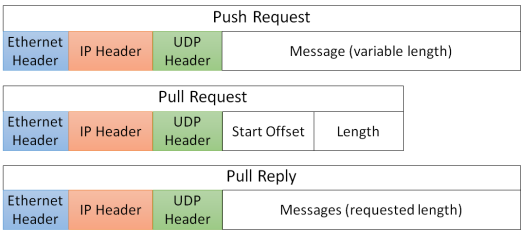


Fig. 7. Packet formats

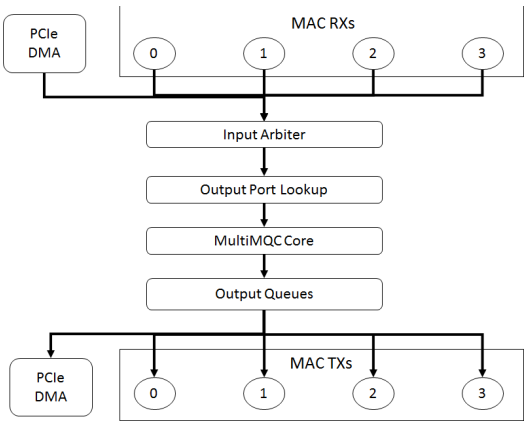


Fig. 8. FPGA modules overview

- Read function to read requested messages from its storage and generates a reply if it caches the requested messages
- Forward function to pass a pull request to the next layer otherwise

The original message queuing middleware is regarded as the last-level cache. It is running as a user-space software and can provide additional functions, such as partitioning and consensus functions supported in advanced message queuing middleware. Also, it can work with sophisticated storage options, such as page cache and anti-caching [12].

IV. Prototype Implementation

As a prototype implementation of MultiMQC, the proposed multilevel cache is organized as shown in Table I.

TABLE I
Cache Layers

i	Layer	Location	Storage
1	In-NIC BRAM layer	FPGA NIC	BRAM (131KB)
2	In-NIC DRAM layer	FPGA NIC	DRAM (8GB)
3	In-kernel layer	Host machine	Memory (kernel space)
4	MQ middleware layer	Host machine	HDD

A. Message Queuing Protocol

We implement a message queuing protocol using UDP for simplicity [1]. Figure 7 shows the packet formats. UDP payload of a push request includes messages to be written. A pull request includes two values: start offset and length in Bytes. UDP payload of a pull reply includes the messages requested.

B. In-NIC Cache

We employ NetFPGA-SUME board as an FPGA NIC that has four 10GbE interfaces and two 4GB DRAMs. We

[1]Although we assumed UDP as a transport layer protocol for simplicity, our concept can be extended to TCP by combining with commercially or freely available FPGA-based 10Gbps TCP cores, such as [13].

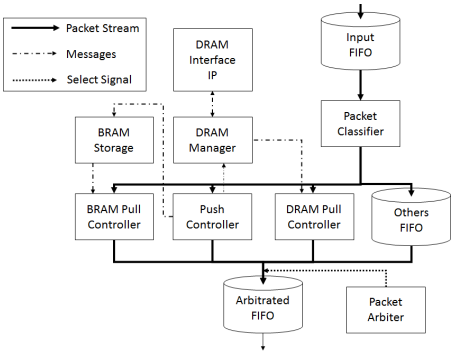


Fig. 9. MultiMQC Core overview

implement the first and second storage layers on the NIC by extending NetFPGA-SUME Reference Switch Lite design [14]. As shown in Figure 8, packets from remote machines via four 10GbE interfaces (MAC RX0-RX3) and host machine via PCIe (PCIe DMA) are merged into a single packet stream by Input Arbiter module. Output Port Lookup module selects the output port for each incoming packet and adds the routing information. According to the information, Output Queues module delivers the packets to one of the four 10GbE interfaces (MAC TX0-TX3) or the host machine via PCIe (PCIe DMA). The proposed MultiMQC Core module is inserted between Output Port Lookup and Output Queues modules. A packet stream goes through the NIC in 32 Bytes per cycle to comply with AXI (Advanced eXtensible Interface) bus of the Reference Switch Lite.

Figure 9 shows MultiMQC Core module overview. When a packet comes, it is classified into three packet types (push request, pull request, and others) by Packet Classifier module. Packets destined to a specific port number (e.g., 19999) are classified as "push request" and forwarded to Push Controller module. Those to another specific port number (e.g., 19998) are classified as "pull request". If the requested range of messages is within in-NIC BRAM cache, it is treated as BRAM pull request and forwarded to BRAM Pull Controller module. If the requested range is not fit within the BRAMs but fit into in-NIC DRAM cache, it is treated as DRAM pull request and forwarded to DRAM Pull Controller module. Otherwise, the request is passed to Others FIFO module so that it is forwarded to the in-kernel cache on the host machine. Packets that are not push nor pull requests are classified as "others" and passed to Others FIFO module. Packet streams from Push Controller, BRAM and DRAM Pull Controllers, and Others FIFO are arbitrated by Packet Arbiter module and merged into a single input stream for Arbitrated FIFO. Then packets in Arbitrated FIFO are asynchronously passed to the original L2 module (i.e., Output Queues).

1) Push Controller: Figure 10 illustrates the packet processing flow of Push Controller module. This module has four functions: 1) Input Function, 2) Write function to the in-NIC BRAM cache, 3) Transfer function to the in-NIC DRAM cache, and 4) Transfer function to the in-kernel cache. Input Function is currently implemented as a simple serializer but user-specific data transformation can be implemented as Input Function. When a packet arrives at this module, the UDP length field is checked and the payload part is written to

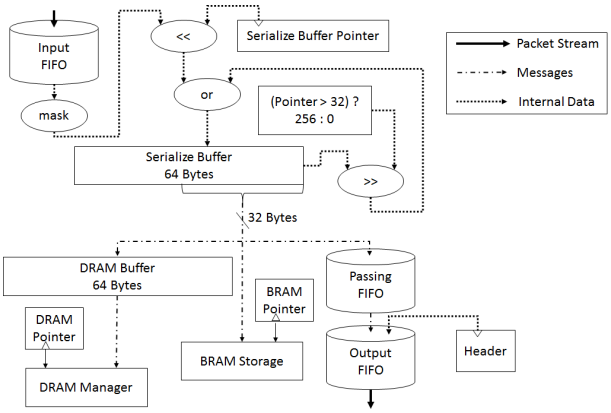


Fig. 10. Push Controller overview

Serialize Buffer.

In MultiMQC, as explained in Section III-B, messages are written to layer-i cache, and they are transferred to layer-$(i+1)$ cache before they are overwritten in layer-i. This basic behavior introduces write and read operations to layer-i cache at the same time. To reduce read accesses on BRAMs and DRAMs, our prototype implementation employs more optimized approach. That is, messages in the Serialize Buffer are transferred to in-NIC BRAM cache, in-NIC DRAM cache, and in-kernel cache at the same time.

The AXI bus transfers 32 Bytes per cycle. The following steps are performed in parallel for each of 32 Bytes in Serialize Buffer.

- Data transfer to in-NIC BRAM cache: The 32 Bytes are copied to BRAMs and BRAM Pointer is incremented.
- Data transfer to in-NIC DRAM cache: DRAM read and write are performed in 64 Bytes. The 32 Bytes are copied to 64-Byte DRAM Buffer. After the 64-Byte DRAM Buffer is filled, the 64 Bytes are copied to DRAMs and DRAM Pointer is incremented.
- Data transfer to in-kernel cache: Data transfer from in-NIC cache to in-kernel cache is done by the predetermined batch size. The 32 Bytes are copied to Passing FIFO. After the messages in Passing FIFO reaches the batch size, a push request from Passing FIFO to in-kernel cache is generated. The request is passed to Output FIFO and when the Push Controller module wins the arbitration, it is passed to Arbitrated FIFO in Figure 9.
- Data shift in Serialize Buffer: Serialize Buffer is right-shifted by 32 Bytes to remove written contents and make a space for the next 32 Bytes. Serialize Buffer Pointer is subtracted by 32 accordingly.

The batch size for the in-kernel cache is stored in a register in the FPGA and configured by the host machine via PCIe.

2) Pull Controller: Figure 11 illustrates the packet processing flow of BRAM Pull Controller module. When a pull request packet arrives at this module, its payload is extracted to check the requested range. As shown in Figure 7, a pull request has two parameters: start offset and length. The requested range in the BRAMs is calculated by these parameters. If the requested range is within the BRAMs, the requested messages are read from the BRAMs and a reply is generated. The reply is passed to Output FIFO and when the BRAM Pull Controller

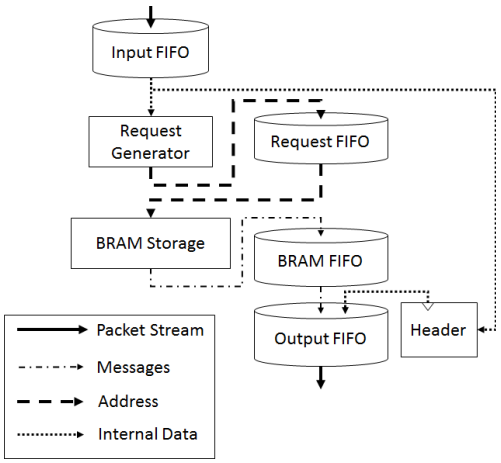


Fig. 11. BRAM Pull Controller overview

module wins the arbitration, it is passed to Arbitrated FIFO in Figure 9. If the requested range exceeds the BRAMs, the pull request is forwarded to DRAM Pull Controller module.

DRAM Pull Controller module is in charge of read from in-NIC DRAM cache. The behavior is similar to BRAM Pull Controller module. If the requested range exceeds the DRAMs, the pull request is forwarded to the host machine.

3) Storages: BRAMs in the FPGA are used as in-NIC BRAM cache that has two read/write ports. One port is used for Push Controller's write and another port is used for BRAM Pull Controller's read. Data width is 32 Bytes so that it can transfer 32 Bytes per cycle, same as AXI bus of Reference Switch Lite. The BRAM capacity is set to 131KB and the address width is 12 bits. The operating frequency is 200MHz.

DRAMs on NetFPGA-SUME board are controlled with an IP generated by Xilinx MIG (Memory Interface Generator) [15]. The operating frequency of interface logic of the IP is set to 200MHz, which is 1/4 of the physical interface running at 800MHz. The number of banks is eight and ordering option is set to NORMAL. Data width is 64 Bytes. DRAM Manager module provides a read and a write interface for the packet control logic. DRAM Manager module serializes the read and write requests based on a round robin manner.

C. In-Kernel Cache

To implement the in-kernel cache, we employ Netfilter framework provided by Linux. The in-kernel cache can be implemented as NIC driver too, but we employ the Netfilter framework, because Netfilter modules can be implemented as kernel modules which are independent of target NICs. When the kernel module is installed, a certain size of memory is allocated to be used as a ring buffer as the in-kernel cache. The memory allocation size can be changed depending on total memory size available in the host machine (e.g., 500GB in [16]). The in-kernel cache is registered as a hook function of PREROUTING property in the Netfilter framework.

When a packet destined to a specific port number (e.g., 19999) arrives, it is detected as a push request and its payload is written to the ring buffer. After accumulated messages in the ring buffer become more than the batch size, a push request for the next layer (e.g., message queuing middleware layer) is generated and sent. The batch size of the in-kernel cache can be larger than Ethernet MTU, because this request does not go through physical layer nor data link layer of TCP/IP stack. When a packet destined to another specific port number (e.g., 19998) arrives, it is detected as a pull request and the requested range is calculated based on its payload. If the range is within the in-kernel cache, the reply is generated and sent back to subscribers. Otherwise the request is forwarded to the message queuing middleware via Linux network protocol stack as a regular packet.

D. Message Queuing Middleware

We implement the last-level cache or message queuing middleware as a simple UDP server process. For publication, a process listening to port 19999 receives push requests and writes their payload to a file. For subscription, another process listening to port 19998 receives pull requests and sends back the replies after reading requested messages from the file. To improve the throughput, we use recvmmsg() system call to receive the requests and sendmmsg() to send back the replies.

V. Evaluations

We evaluate the performance of MultiMQC using a 10GbE hardware packet generator [17]. Although some in-memory technologies (e.g., anti-caching [12]) can be used for the last-level message queuing software, we simply use tmpfs as a RAM disk for storing the messages in the last-level.

A. Write Performance

To evaluate the write performance of each layer, we built a packet generator to send push requests via four 10GbE ports. Figure 12 shows the performance difference between the MQ middleware layer of MultiMQC and the standalone message queuing middleware. The batch size when transfering data from NIC to the host machine is 1472 Bytes, while that transfering from Linux kernel to user space is 32k Bytes. As shown in Figure 12, write performance is significantly improved by batch writing of MultiMQC. The effect of MultiMQC on write performance depends on the request payload size and the batch size. Regarding the payload size, a larger payload decreases the improvement. For example, when the payload size is 4 Bytes, MultiMQC improves the write throughput by 183 times, while when the payload is 32 Bytes the improvement is 51 times. Although a larger payload size improves the effective throughput, more than 32 Bytes payload size cannot increase the throughput in the case of MultiMQC. This is because the last-level cannot keep up with such a high workload and it becomes a bottleneck. However, even if the payload size is 512 Bytes, MultiMQC improves the write throughput by 3 times.

To evaluate the effect of batch size, we run a similar evaluations with various batch sizes. First, the batch size when transfering data from kernel space to user space is fixed at 32k Bytes, and that from NIC to the host machine is varied. Figure 13 shows the result on each layer. X-axis is the batch size to transfer data from NIC to host machine. Y-axis is the written message size per unit time. By increasing the batch size, the number of packets the host machine receives (i.e., packet processing overhead) is reduced and the throughput is improved. When the workload on the host machine is low (e.g., 4-Byte payload in Figure 13(a)), the in-kernel cache and

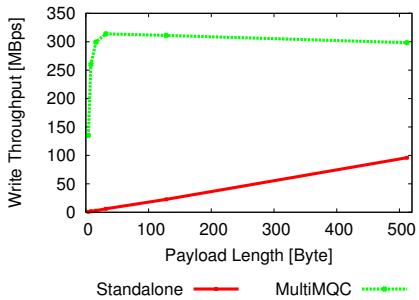


Fig. 12. Write throughput improvement by MultiMQC

the last-level show almost the same performance as the in-NIC caches by 1472-Byte batch transfer. As the payload size increases, the host machine cannot keep up with such a high workload and the performance gap between in-NIC and in-kernel caches is enlarged.

The evaluation results of various batch sizes from kernel space to user space are omitted due to the page limitations.

B. Read Performance

Figure 14 shows the read throughput of each layer. We evaluate three cases where request range is 0-31, 0-511, and 0-1471 Bytes. Similar to the write operation, a larger payload size improves the effective throughput. Packet processing overhead varies depending on each layer. The performance gap becomes large when payload size is small. When the payload size is 32 Bytes, the read throughput of in-NIC BRAM cache, in-NIC DRAM cache, and in-kernel cache are improved by 426, 319, and 2.1 times compared to the last-level message queuing software. When the payload size is 1472 Bytes, both the BRAM and DRAM cache layers improve the throughput by 38 times, and in-kernel cache layer improves the throughput by 2.2 times compared to the last-level. The throughput of both the BRAM and DRAM cache layers reaches 95.8% of 10GbE line rate for 1472 Bytes payload in four 10GbE interfaces.

Figure 15 shows the round-trip time (RTT) until a reply arrives at client NIC after the client NIC transmits a pull request. As the requested size increases, the cache layers on NIC take a longer time to read messages. The cache layers on host machine take almost the same time regardless of the requested size because of high packet processing overheads that hide the requested size differences. When the payload size is 32 Bytes, the BRAM, DRAM, and kernel cache layers reduce the latency by 82, 73, and 1.7 times compared to the last-level, respectively. When the payload size is 1472 Bytes, they achieve 30, 29, and 1.7 times lower latency compared to the last-level, respectively.

Compared to the write performance, a larger performance gain is achieved for the read performance of MultiMQC, because higher-level cache layers do not hold down the gain by lower-level layers. Please note that the read performance is improved only when the request hits in the cache. A pull request will be hit in a cache until the requested messages are overwritten by newly-pushed messages. The available time is defined as a time period during a request hits in the cache. It is calculated by cache capacity and write throughput in the similar way as the maximum acceptable latency, as shown in Equation (1). Table II shows example cases to see how long a message lives in each cache so that the requests for it can hit. As shown in Table II, the available times of the caches are relatively long for online subscribers that keep up with the publication. Regarding offline subscribers, they typically pull large-sized messages and the throughput tends to be high.

TABLE II
AVAILABLE TIME IN CACHE

Layer	Example Capacity	Write throughput		
		1MBps	100MBps	300MBps
In-NIC BRAM layer	1MB	1 sec	10 msec	3 msec
In-NIC DRAM layer	8GB	133 min	1.3 min	27 sec
In-kernel layer	512GB	142 hour	1.4 hour	28 min

C. Resource Utilization

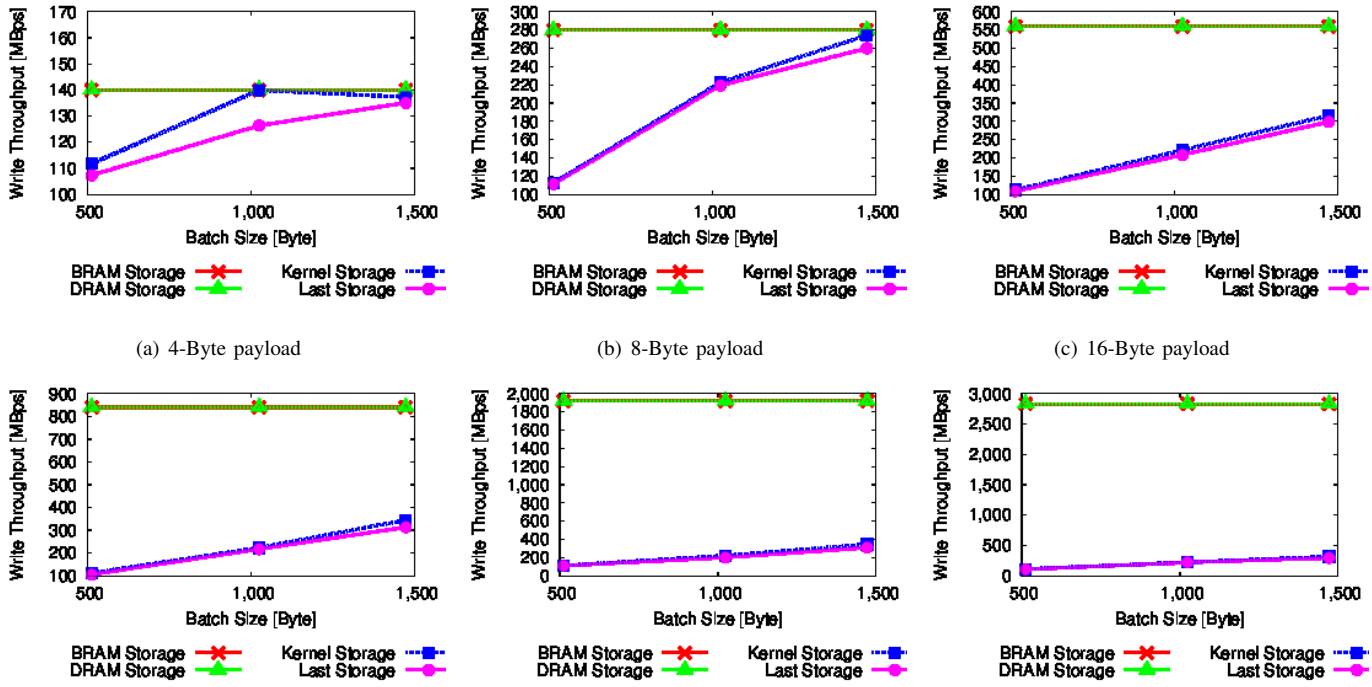
Table III shows resource utilization of the NIC part of MultiMQC. The design tool used is Xilinx Vivado Design Suite 2016.4. The target FPGA device is Virtex-7 XC7VX690T. The target operating frequency is 200MHz. The maximum capacity of the in-NIC BRAM cache is 4M Bytes by considering the available BRAM resource in the FPGA device. However, when the capacity is increased to more than 2M Bytes, the timing constraint could not be met in our design. For write operation, the bottleneck tends to be the host machine and performance overheads by the NIC part can be hidden. Thus, MultiMQC should be implemented by considering the trade-off between read performance versus cache available time.

TABLE III
RESOURCE UTILIZATION

BRAM storage		Utilization		Max Frequency
Address width	Capacity	LUTs	BRAMs	
12	131KB	65,760 (15.18%)	346 (23.54%)	200MHz
13	262KB	65,806 (15.19%)	381.5 (25.95%)	200MHz
14	524KB	66,394 (15.33%)	445.5 (30.31%)	200MHz
15	1MB	66,544 (15.36%)	573.5 (39.01%)	200MHz
16	2MB	66,676 (15.39%)	829.5 (56.43%)	168MHz
17	4MB	67,586 (15.60%)	1,341.5 (91.26%)	151MHz

VI. CONCLUSIONS

Traditional message queuing systems are facing a problem in terms of the efficiency due to a lot of small messages from IoT devices. Toward a high efficiency even with such a small packet stream, we proposed MultiMQC that combines 1) the in-NIC and in-kernel multilevel cache to improve read performance and 2) the batch transfer to improve write performance. MultiMQC strikes a good balance between cache capacity and performance by introducing the multilevel layered cache structure. Evaluation results show that MultiMQC improves write performance by 51 times for 32 Bytes payload packets compared to a standalone message queuing software. MultiMQC also improves the read performance by 426 times and the read latency by 82 times compared to the message queuing software if subscribers always read messages when the messages are alive in the in-NIC cache. When the requests hit in-NIC cache and the reply payload length is 1472 Bytes, read throughput reaches 95.8% of 10GbE line rate in four 10GbE interfaces. As a future work, we are planning to extend MultiMQC to support more reliable protocols, such as AMQP. We will also evaluate Output Functions implemented on FPGA NIC to accelerate user-specific computations for subscribers.



(a) 4-Byte payload (b) 8-Byte payload (c) 16-Byte payload

(d) 32-Byte payload (e) 128-Byte payload (f) 512-Byte payload

Fig. 13. Write throughput of each layer

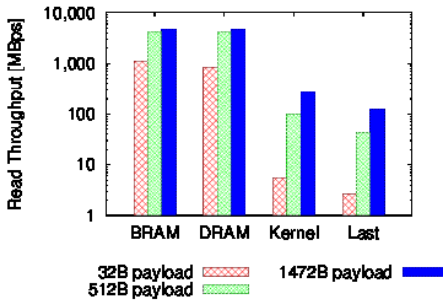


Fig. 14. Read throughput

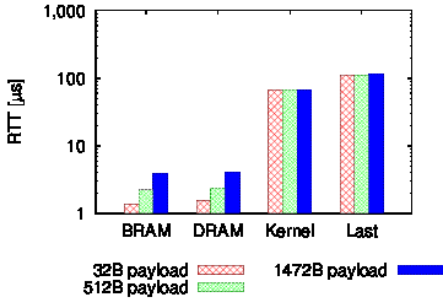


Fig. 15. Read latency

Acknowledgements This work was supported by JST CREST Grant Number JPMJCR1785, Japan.

REFERENCES

[1] "IoT Applications & Solutions," https://aws.amazon.com/iot/.
[2] "The Apache ActiveMQ," http://activemq.apache.org/.
[3] "RabbitMQ - Messaging that just works," http://www.rabbitmq.com/.
[4] "The Apache Kafka," http://kafka.apache.org/.
[5] Mendel Rosenblum, "Low Latency RPC in RAMCloud," https://forum.stanford.edu/events/2011/2011slides/plenary/2011plenaryRosenblum.pdf.
[6] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Oct. 2012, pp. 135–148.
[7] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs," in *Proceedings of the USENIX Conference on Annual Technical Conference (ATC'16)*, Jun. 2016, pp. 43–56.
[8] I. Corporation, "Impressive Packet Processing Performance Enables Greater Workload Consolidation," in *Intel Solution Brief*, 2013.
[9] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *Proceedings of the USENIX Conference on Annual Technical Conference (ATC'12)*, Jun. 2012, pp. 101–112.
[10] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'13)*, Feb. 2013, pp. 245–254.
[11] Y. Tokusashi and H. Matsutani, "Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches," *IEEE Micro*, vol. 37, no. 5, pp. 44–51, Oct. 2017.
[12] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-caching: A new approach to database management system architecture," *Proceedings of the VLDB Endowment*, vol. 6, pp. 1942–1953, Sep. 2013.
[13] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM'15)*, May 2015, pp. 36–43.
[14] "The NetFPGA Project," http://netfpga.org/.
[15] "Memory Interface," https://japan.xilinx.com/products/intellectual-property/mig.html.
[16] K. Tamura and H. Matsutani, "An In-Kernel NOSQL Cache for Range Queries Using FPGA NIC," in *Proceedings of the International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC'16)*, May 2016, pp. 13–18.
[17] "OSNT 10G Home," https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-10G-Home.