PAPER Special Section on Parallel, Distributed, and Reconfigurable Computing, and Networking

Fogcached: A DRAM/NVMM Hybrid KVS Server for Edge Computing

Kouki OZAWA[†], **Takahiro HIROFUCHI**[††], ***Nonmembers***, **Ryousei TAKANO**[††], ***and*** **Midori SUGAYA**[†a)], ***Members***

SUMMARY With the development of IoT devices and sensors, edge computing is leading towards new services like autonomous cars and smart cities. Low-latency data access is an essential requirement for such services, and a large-capacity cache server is needed on the edge side. However, it is not realistic to build a large capacity cache server using only DRAM because DRAM is expensive and consumes substantially large power. A hybrid main memory system is promising to address this issue, in which main memory consists of DRAM and non-volatile memory. It achieves a large capacity of main memory within the power supply capabilities of current servers. In this paper, we propose Fogcached, that is, the extension of a widely-used KVS (Key-Value Store) server program (i.e., Memcached) to exploit both DRAM and non-volatile main memory (NVMM). We used Intel Optane DCPM as NVMM for its prototype. Fogcached implements a Dual-LRU (Least Recently Used) mechanism that seamlessly extends the memory management of Memcached to hybrid main memory. Fogcached reuses the segmented LRU of Memcached to manage cached objects in DRAM, adds another segmented LRU for those in DCPM and bridges the LRUs by a mechanism to automatically replace cached objects between DRAM and DCPM. Cached objects are autonomously moved between the two memory devices according to their access frequencies. Through experiments, we confirmed that Fogcached improved the peak value of a latency distribution by about 40% compared to Memcached.

key words: *Fogcached, KVS, Key-Value-Store, Middleware, edge, edge computing, Dual-LRU, NVM, NVMM, DCPM*

1. Introduction

With the advance of IoT (Internet of Things) technology, a large amount of data is transferred to servers from smart devices and sensors for processing. When IoT technology spreads to every corner of our lives in the future, it will be necessary not only to process data in geographically distant cloud data centers, but also to process data at the edge near users [1]. In edge computing, it is necessary not only to retain data on the distant cloud data center side, but also to retain data on the edge side in order to achieve higher responsiveness [1], [2].

We focused on a key-value-store (KVS) server that allows a large amount of data to be temporarily cached. To realize low-latency access to a large amount of data, the KVS server needs to be equipped with a large memory. However, the capacity of a DRAM module, which is currently used as main memory, is limited, and it is unlikely that it will increase significantly in the future [2], [3]. DRAM is volatile and continuous power supply (i.e., refresh operation) is necessary. It is difficult to equip a computer with a large capacity of DRAM because of its large power consumption. Therefore, we consider improving the capacity of main memory by using a non-volatile main device. It has larger capacity than DRAM and also has energy efficiency enabled by the inherent characteristic of non-volatility. However, its read/write performance is likely inferior to that of DRAM. For example, the read latency of Intel Optane DCPM (Data Center Persistent Memory, i.e., the first commercially available byte-addressable non-volatile memory module) is approximately 3 times slower than that of DRAM for random access and approximately 2 times slower for sequential access [4]. For KVS servers, it is promising to use non-volatile memory device to expand the capacity of the main memory so as to increase its cache hit rate; however, it is necessary to avoid the deterioration of performance due to its inferior performance.

In this study, we propose 'Fogcached', a KVS server program supporting hybrid main memory composed of DRAM and DCPM. It drastically increases the capacity of cached data by taking the advantage of the large capacity of DCPM. It also reduces performance degradation caused by the large latency of DCPM by automatically optimizing the locations of cached data between DRAM and DCPM.

We extend a widely-used KVS server program, Memcached [5], which is originally designed for DRAM-based main memory. Memcached has a segmented LRU (Least Recently Used) algorithm to manage cached objects according to their access frequencies. To keep its powerful functionalities, we keep the segmented LRU for DRAM and add another segmented LRU for DCPM, and loosely couple them with a mechanism to optimize the locations of cached objects between the memory devices. Fogcached automatically moves frequently-accessed cached objects from DCPM to DRAM and infrequently-accessed ones to DRAM to DCPM. It increases the hit ratio to the faster memory device (i.e., DRAM) and decreases performance degradation due to the slower memory device (i.e., DCPM).

Note that Memcached has a mechanism called ext_store [6] that expands the capacity of a KVS server by using a storage device. The ext_store mechanism, however, is designed for flash memory devices, which is not capable in fully taking advantage of byte-addressable non-volatile memory. Byte-addressable non-volatile memory is

Manuscript received January 19, 2021.
Manuscript revised May 22, 2021.
Manuscript publicized August 18, 2021.

[†]The authors are with Faculty Engineering Shibaura Institute of Technology, Tokyo, 135–8548 Japan.
[††]The authors are with National Institute of Advanced Industrial Science and Technology, Tokyo, 135–0064 Japan.
a) E-mail: doly@shibaura-it.ac.jp

DOI: 10.1587/transinf.2021PAP0003

designed to be accessed as the main memory of a computer like DRAM. We aim at avoiding the overhead of storage I/O. Fogcached is natively designed for byte-addressable non-volatile memory.

We conducted experiments using a computer equipped with DCPM. We confirmed that the proposed mechanism correctly works. In tested conditions, the peak value of a latency distribution was improved by approximately 40%, compared to Memcached. Fogcached succeeded in improving performance, thanks to its optimization mechanism.

To promptly report the results and obtain the first feedback from the community, we published an early report of Fogcached [7]. Considering the broader reader's interest, we focus this paper to recount the study and add the detailed explanation of the mechanism related to the performance evaluation. Moreover, we have added the evaluation and discussion of how this work overcame the challenges in the approach.

The structure of this paper is as follows: Sect. 2 describes related work. Section 3 summarizes requirements. Section 4 introduces the proposed design and its implementation. Section 5 presents evaluation. Section 6 concludes this work and discusses issues.

2. Related Work

2.1 Studies on Non-Volatile Main Memory

Non-volatile memory (NVM) based on a new operating principle has appeared in contrast to volatile memory such as SRAM and DRAM used in conventional computer systems. Some types of non-volatile memory devices are being put into practical use. For example, resistance change memory (ReRAM), magnetic memory (MRAM), and phase change memory (PCM) have appeared [7], [8].

These memory devices have non-volatility in which stored data is persistent without electrical supply. It is expected that the capacity of such emerging memory devices will increase in the future. Although the capacity of main memory is expanded, its power consumption will be suppressed. It is also considered that some types of non-volatile memory devices achieve the same performance as DRAM in terms of latency.

Now studies on the use of non-volatile memory devices are being actively conducted. For example, focusing on the fact that non-volatile memory devices have a potential to be used in both main memory and storage, there is a study on a file system that combines DRAM and non-volatile memory [9]. There are also proposals for using non-volatile memory devices for KVS databases. The persistence of data is used to improve the fault tolerance of KVS [10], [11].

2.2 Studies on KVS Systems Using Non-Volatile Memory Devices

In contrast to the fact that some KVS systems traditionally guaranteed persistence by using flash storage, Fei Xia et al. proposed HiKV (Hybrid index Key-Value Store) that guarantees persistence by using non-volatile memory for a KVS system [14]. HiKV provides a mechanism to a construct hybrid index on hybrid memory. The hybrid index is composed of a B+ tree index in DRAM and a hash index in non-volatile memory. It supports the request type of range scan to the KVS server. In contrast, our proposed mechanism exploits the LRU algorithm of Memcached, optimizing its design for ephemeral caching and basic KVS operations to set and get data.

Hao Liu et al. proposed LibreKV [15], which responds to requests from clients at high performance. LibreKV places a hash table in DRAM and non-volatile memory, respectively. When the hash table in DRAM is full, LibreKV merges its data into the hash table in non-volatile memory, in order to continue serving for requests. It also provides the persistence of data. LibreKV is designed as a non-ephemeral database, unlike KVS programs designed as cache servers such as Memcached. LibreKV has no mechanism to delete an object from memory when memory is fully used. LibreKV assumes that the total data size to be saved is smaller than the memory size of a server. This is in contrast to the LRU algorithm of our proposed mechanism, which is designed for a cache server.

Hai Jin et al. proposed a mechanism to extend the basic structure of Memcached [17]. It however adopts the multi-queue algorithm [18] as a replacement algorithm to determine which objects to be selected when optimizing the locations of objects between DRAM and non-volatile memory. Instead, our proposed method reuses the segment LRU algorithm of Memcached and can coexist with its existing functionalities such as ext_store (i.e., a function to expand cache capacity by using a storage device).

We aim at the placement of objects in different memory devices by seamlessly extending the LRU algorithm of the existing KVS. It is intended to achieve high responsiveness.

2.3 Studies on Data Management in Hybrid Main Memory

Wu [21] et al. proposed automatic task memory page movement between memory devices for task-based parallel programming. They have implemented a mechanism that analyzes similar types of tasks from task metadata information and automatically move memory pages of similar types of tasks between DRAM and non-volatile memory according to the information of frequently accessed tasks. However, access patterns in a database can vary greatly from object to object and from time to time. Therefore, it is difficult to predict the behavior of similar types of objects by analyzing them in advance. We consider that it is more appropriate for KVS systems to do prediction by reference frequencies.

3. Requirements

There are three main requirements for KVS servers using hybrid memory. The first requirement is to avoid performance degradation due to the slowness of a non-volatile

memory device. It is possible to increase the hit rate of cached data by expanding memory space by a large size of non-volatile memory device. However, its read/write performance is inferior to that of DRAM. The read performance of DCPM is 3 to 4 times larger than that of 80 ns of DRAM, even though its capacity is 10 times larger. It requires ingenuity to properly use both memory devices in consideration of caching efficiency and performance.

The second requirement is to respond dynamically changing requests to KVS servers. Requests from clients vary from object to object and from time to time. Therefore, it is difficult to determine in advance which object should be stored in a faster memory device (i.e., DRAM). A KVS server needs a mechanism to dynamically move objects between memory devices, according to the access frequencies to objects.

The third requirement is to maximize performance of a byte-addressable non-volatile memory device. Memcached 1.5.4 and later have a mechanism, called ext_store, that holds cached data in a storage device. The design of the ext_store function is intended for flash storage, which is significantly inferior in speed to DRAM. It uses the system calls for file I/O and the operating system performs I/O buffering. On the other hand, byte-addressable non-volatile memory devices are capable of being accessed as a part of main memory in the same manner as DRAM. Its latency is larger than that of DRAM but smaller than that of flash storage. It is expected that maximum performance will not be obtained through a mechanism designed for storage devices.

4. Proposal

4.1 Overview

We propose Fogcached, a KVS server designed for hybrid main memory consisting of both DRAM and DCPM. It is intended to achieve large capacity and high performance together. DCPM provides byte-addressing memory access in its AppDirect mode. Memory access is done in the same way as DRAM. We extended Memcached to support hybrid main memory, keeping its basic mechanism as much as possible. Memcached treats memory objects according to their access frequencies so as to minimize access latency. This basic mechanism is straightforwardly extended for the main memory consisting of the two memory devices.

We expanded the segment LRU (Least Recently Used) algorithm of Memcached, which is its key memory management mechanism. It dynamically moves memory objects between two types of memory devices, depending on the access frequency of each item.

By placing frequently-accessed memory objects in a faster memory device (i.e., DRAM) and other memory objects in a slower device (i.e., DCPM), it is possible to hold a large amount of data on memory as well as alleviate the performance degradation caused by mixing a slower memory device in the main memory.

In this section, we introduce the basic design of Memcached and then describe the design and implementation of Fogcached.

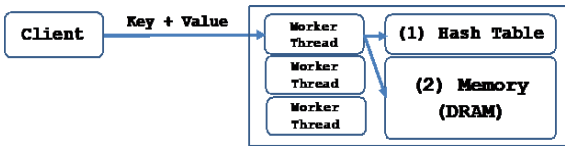


Fig. 1 The high-level overview of the key-value store mechanism of Memcached

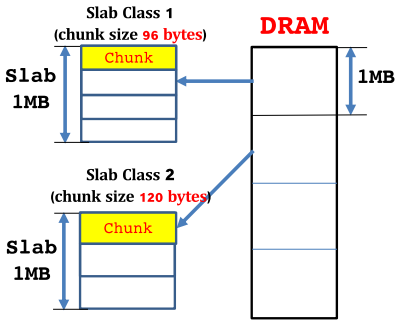


Fig. 2 The basic memory management structure of Memcached, composed of slab classes, slabs and chunks.

4.2 Memcached

Memcached is a typical in-memory KVS server program and is widely used in data centers today. Compared to other KVS servers, its overall structure is relatively simple; we consider that it is suitable as a target for implementing a new idea of memory management. Memcached is a multi-threaded program in which worker threads respond to requests from clients and other threads work for overall memory management. We used the version 1.5.16 of Memcached.

4.2.1 Basic Design

Figure 1 shows the basic operations of Memcached with a client. It receives a key and its value from the client; a worker thread in the server process of Memcached receives them. It firstly updates its hash index table from the key, and secondly saves the set of the key, the value and its metadata (e.g., key/value lengths and ancillary information to track its access frequency) in a region of the main memory.

In Memcached, a set of a key, its value and its metadata is called an item. In order to improve the utilization efficiency of memory space allocated from the main memory, items of similar sizes are grouped and managed together. Memcached allocates memory space from the main memory in a unit called slabs (Fig. 2). The size of a slab is 1 MB. A slab is further divided into chunks. Each slab has its specific chunk size. Slabs are classified by chunk size. For example, slabs with the chunk size of 96 B are managed as slab class 1, and those of 120 B are managed as slab class 2. When saving an item in the main memory, Memcached

selects the slab class whose chunk size the most nearly fits the size of the item.

Memcached monitors the valid period of each item during which the item is intended to be retained in its LRU structure (described later). It deletes an item in the structure when its valid period has expired. If Memcached lacks memory space when adding a new item, it deletes an old item by means of the following algorithm, regardless of its valid periods of items. It then adds the new item to the structure.

4.2.2 Segmented LRU Algorithm

Each slab class has an LRU to manage items according to the frequency of references of each item. The LRU is divided into three segments: Hot, Warm and Cold. Each item has bit flags that change depending on the number of references. It moves between the segments depending on the state of the flags. Just after an item is created, the flags are in the cleared state.

As the item is referenced first, the Fetched flag is set and linked to the head of the Hot segment. When the item is referenced again in the Hot segment, the Active flag is also set. A thread, called maintenance thread, periodically monitors the tail of each segment. If the item at the tail of the Hot segment has the Active flag, it is moved to the Warm segment. Otherwise, it is moved to the Cold segment (i.e., such items not being accessed again in the Hot segment are deemed to be unlikely accessed again). The Active flag is cleared.

The item at the tail of the Warm segment remains at the Warm segment. Otherwise, it is moved to the Cold segment. The item at the tail of the Cold segment is evicted due to the lack of memory space. See Fig. 4 (a) for details.

4.3 Fogcached

We propose Fogcached, an advanced memory management mechanism for the KVS server supporting hybrid main memory. Figure 3 shows the overview of the memory management structure of the proposed mechanism. We added a new group of slab classes dedicated for the memory space allocated from DCPM, while maintaining the original slab classes of Memcached. The structure of the slab classes for DCPM is the same as that of DRAM. We call this structure dual-LRU slab classes. The LRU algorithm composed of the Hot, Warm and Cold segments is implemented for the memory space of each memory device. This design is intended to avoid drastic modification to the current design of Memcached and keep supporting its existing powerful memory management functionalities also for the emerging memory device with minimum efforts.

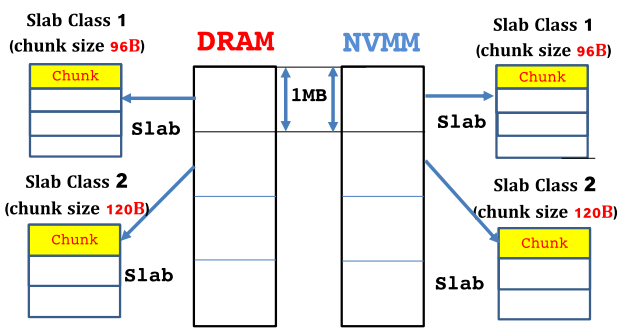


Fig. 3 The proposed memory management structure off Fogcached, called dual-LRU slab class

4.3.1 Extension of the Segment LRU Algorithm

To seamlessly integrate the newly added group of slab classes, we added two new segments to bridge the two segmented LRU structures. See Fig. 4 (b) for details. The Move segment to DCPM holds the items being evicted from DRAM and being placed in DCPM. The Move segment to DRAM holds the items being placed in DRAM. We also added a thread to monitor each Move segment and conduct replacement of items. We call them migration threads.

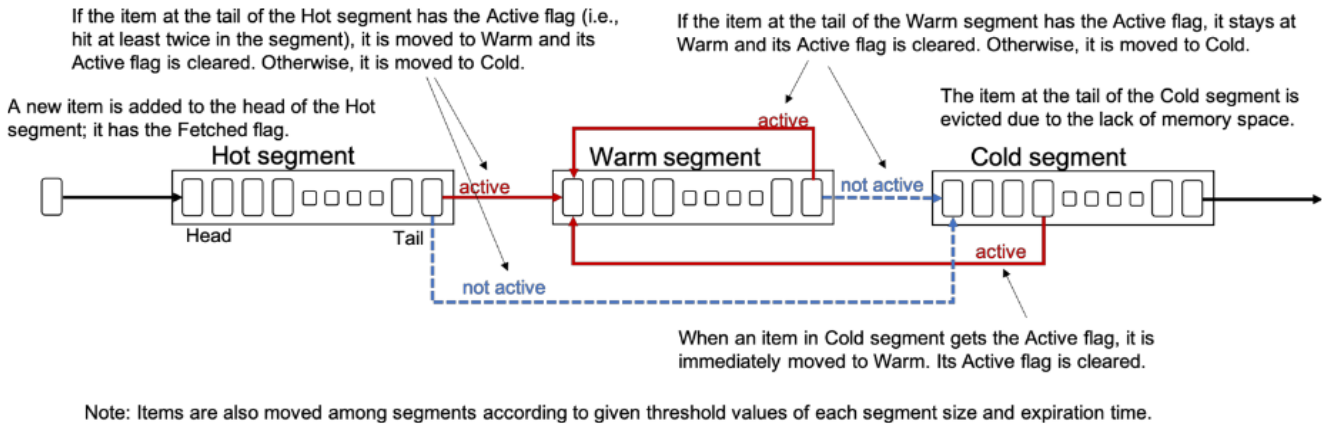
The basic design of the segmented LRU to manage the memory space of each memory device is not modified. The maintenance thread for each segment LRU periodically monitors items in the LRU. By introducing the Move segments and the migration threads, we can loosely couple the two LRU mechanisms, each of which is originally designed just for a single memory device. Each LRU mechanism independently moves items among its segments without interfering the other LRU mechanism. The migration thread independently works for each Move segment. The item migrations to DCPM and those to DRAM asynchronously work.

Because the maintenance thread needs to periodically scan items to find expired items, another merit of having the two maintenance threads is to mitigate performance overhead. The maintenance thread needs to hold an exclusive lock for its LRU segments while scanning items. The performance of worker threads is adversely affected if it needs to hold the lock for a long time.

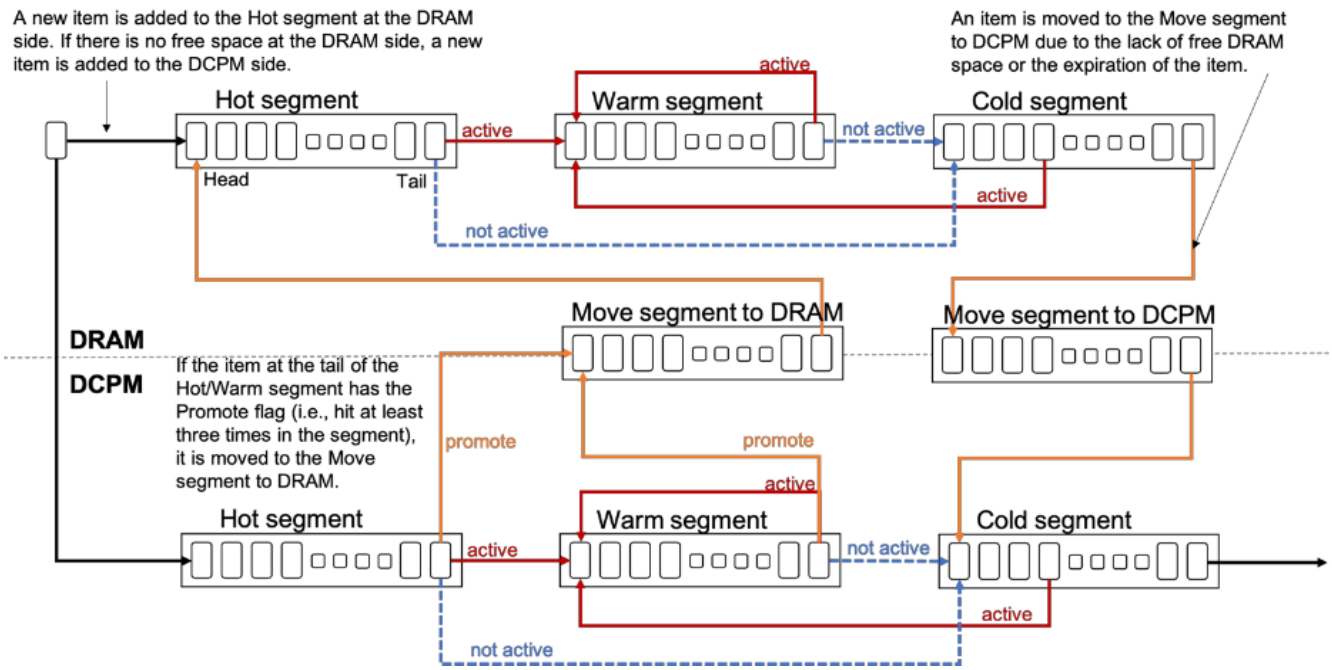
We added the Promote flag to the state of an item to distinguish the items to be moved from DCPM and DRAM. In the current implementation, when an item having the Active flag is referenced again, its Promote flag is set. The meaning of the Fetched and Active flags does not change; items are moved among the Hot, Warm and Cold segments according to the existing flags. By adding the Promote flag, we succeeded in reusing the existing LRU algorithm for the inside of each memory device while seamlessly connecting it for the other memory device.

4.3.2 The Migration of Items between DRAM and DCPM

First, the mechanism of moving frequently-accessed items from DCPM to DRAM works as follows. The maintenance thread for the segmented LRU of the DCPM side monitors the items at the tails of the Hot segment and the Warm segment. If an item at the either tail has the Promote flag, it moves it to the Move segment to DRAM. The migration



(a) The LRU mechanism of each slab class in Memcached. The maintenance thread moves items among the Hot/Warm/Cold segments of each slab class.



(b) The LRU mechanism of each slab class in Fogcached. In addition to the maintenance thread of each memory device, the migration thread is added for each memory device to move items from its Move segment.

Fig. 4

thread of the Move segment to DRAM periodically monitors the segment and moves items in the segment to the Hot segment of the LRU of the DRAM side. It finds a target slab, secures a destination item and copy the data of the source item to it. Finally, it deletes the source item. When securing a destination item, if memory space is not available, it deletes the item at the end of the Cold segment of the LRU of the DRAM side. Overall, the above procedure is summarized as follows:

An item in the Hot and Warm segments in the LRU of the DCPM side gets the Promote flag.

(1) The item arriving at the tail of the either segment is moved to the Move segment to DRAM.
(2) The item in the Move segment is moved to the Hot segment of the LRU of the DRAM side.

The mechanism of moving infrequently-accessed items from DRAM to DCPM is rather simple and it reuses the mechanism of the Cold segment of Memcached. When an item in the Cold segment is evicted due to the lack of free memory space or the expiration of the item, it is moved to the Move segment to DCPM. The migration thread of that

Move segment periodically monitors it and moves items to the head of the Cold segment of the LRU of the DCPM side.

4.4 Implementation

The proposed mechanism is based on the version 1.5.16 of Memcached and implemented for the Linux operating system running on a machine equipped with DCPM. The extension to Memcached is approximately 400 lines in C language. The proposed mechanism uses the Device Dax mode of the NVDIMM driver of Linux to access the memory space of DCPM. In the Device Dax mode, user-space programs can allocate memory pages from DCPM by using the device file of the NVDIMM driver (e.g., /dev/dax). The proposed mechanism call the mmap() function to map memory pages from DCPM to its virtual address space.

It reads and writes them in the same way as memory pages allocated from DRAM.

It should be noted that the NVDIMM drive of Linux also provides a mechanism to use DCPM as a storage device. In experiments, we used this mode to compare the proposed mechanism with the ext_store mechanism of Memcached.

5. Evaluation

5.1 Experimental Settings

Through experiments, we confirmed that Fogcached correctly worked and clarified its basic performance. We prepared for two computers. The one is used for a KVS server program working as a cache server, the other is used for a KVS client program sending requests to the server. The specifications of the server and client computers are shown in Table 1. The KVS server machine has 2 Intel Xeon processors. Each processor has 24 physical CPU cores and 2 memory controllers. Each controller has 3 memory channels. Each memory channel has a DRAM module (16 GB) and a DCPM module (128 GB). The total DRAM size is 192 GB. The total DCPM size is 1.5 TB. All the DCPM modules are set to the AppDirect mode. The machines are connected with 10 Gb Ethernet.

Table 1 The machine specifications used in experiments

KVS server machine	
CPU	Intel Xeon Gold 6230 2.10 GHz x2
DRAM	DDR4 16GB x12 (192 GB)
NVM	Intel Optane DCPM 128GB x12 (1.5 TB)
NIC	10Gbase-T
OS	Fedora
KVS client machine	
CPU	Intel Xeon CPU E5-2630 2.20 GHz
DRAM	DDR4 16GB x8 (128 GB)
NIC	10Gbase-T
OS	Ubuntu

5.2 Performance Evaluation

First, we compared the basic performance of Fogcached with that of Memcached. As a benchmark tool for a KVS server, we used Memaslap [22]. For comparison, experiments were conducted in the following 5 configurations in Table 2:

(1) Memcached was used. 32 GB of DRAM was allocated.
(2) Fogcached was used. 32 GB of DCPM was allocated. No DRAM space was allocated to it.
(3) Fogcached was used. 4 GB of DRAM and 28 GB of DCPM were allocated. Its mechanism to dynamically relocate items between DRAM and DCPM was disabled. It first uses DRAM space to place new items. After using all the allocated DRAM space, it uses DCPM.
(4) Fogcached was used. 4 GB of DRAM and 28 GB of DCPM were allocated. Its mechanism to dynamically relocate items between DRAM and DCPM was enabled.
(5) Memcached was used. 4 GB of DRAM was allocated. The ext_store mechanism of Memcached was enabled. A file of 28 GB was used for the ext_store mechanism, which was created on DCPM.

Memaslap was configured to generate GET requests. The size of a key was 64 B and that of value was 1 KB. The number of key-value pairs was 100 K. It created 256 TCP connections concurrently. In this parameter setting, the total size of cached data reached 30 GB. We had in mind that we emulated a situation where many sensor nodes are connecting to a KVS server located at the edge. Memaslap first performed a warm-up phase in which it sends SET requests of all the key-value pairs to a KVS server. After that, it generated GET requests, choosing keys in a *nearly* random manner. To see the contribution of dynamic relocation, access frequencies of keys should not be completely equal; some keys need to be accessed more frequently and supposedly tends to be placed in DRAM for a longer time. We modified Memaslap to alternately change a range of keys from which it selects a key randomly; once it picks up a key from all the keys, next time it picks up a key from a particular set of 5 K keys. In each experiment, Memaslap generated 20 million GET requests. Figure 5 shows the latency distributions of

Table 2 The configurations of experiments

		DRAM	DCPM
(1)	Memcached with DRAM only	32 GB	0 GB
(2)	Fogcached with DCPM only	0 GB	32 GB
(3)	Fogcached without dynamic relocation	4 GB	28 GB
(4)	Fogcached with dynamic relocation	4 GB	28 GB
(5)	Memcached with the ext_store mechanism enabled	4 GB	28 GB

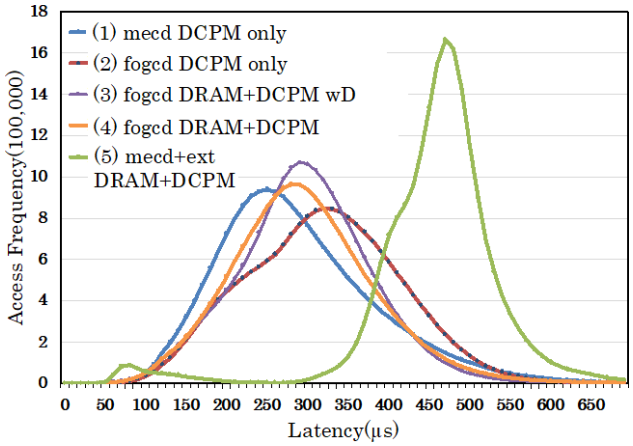


Fig. 5 Latency distributions obtained through the experiments using Memaslap

GET requests obtained in the above configurations (1)-(5).

5.2.1 Comparison between Fogcached and Memcached with its Ext_Store Mechanism

First, we focus on the comparison between (4) Fogcached and (5) Memcached with the ext_store mechanism enabled. Both configurations use the same mixed ratio of DRAM and DCPM (i.e., 4 GB and 28 GB, respectively). The ext_store mechanism of Memcached enables us to apply it to hybrid main memory without any modification to it. Moreover, the use of the ext_store mechanism enables us to allocate any mixed ratio of DRAM and DCPM to Memcached (Fig. 5).

It should be noted that we did not use the Memory mode of DCPM for evaluation. In addition to the AppDirect mode, DCPM also supports the Memory mode in which the memory controller integrates DCPM into main memory and automatically relocates memory pages between DRAM and DCPM. In the Memory mode, the operating system running on the machine virtually sees that main memory is composed of one memory device, not being aware of hybrid main memory. In this work, we however discuss how software-based mechanisms should treat hybrid main memory. Software-based mechanisms enable us to implement application-specific memory management for hybrid main memory. This work picks up a KVS server as an example application and discusses the feasibility of the proposed memory management mechanism. We do not discuss comparison between software and hardware-based mechanisms. In addition, the use of the Memory mode needs to always integrate the entire DRAM with DCPM. It is not possible to flexibly mix two memory devices. This is another reason that we used the ext_store mechanism for experiments.

The results showed that, the average latency of requests with Fogcached was 311 us and that of Memcached was 460 us. The peak value of latency distribution was 290 to 300 us by Fogcached and 470 to 480 us by Memcached. The peak value of latency distribution was improved by approximately 40% by Fogcached. Fogcached, directly accessing DCPM as main memory and not involving complex storage I/O mechanisms, achieved better performance.

5.2.2 Comparison between DRAM and DCPM

The comparison between (1) Memcached with 32 GB DRAM and (2) Fogcached with 32 GB DCPM showed how the performance of a memory device impacts on a KVS server. Note that because Fogcached reuses the segmented LRU of Memcached, if it is configured without DRAM, it behaves as if Memcached was applied to DCPM. The peak value of latency distribution was 250-260 us by Memcached with DRAM, and 330-340 us by Fogcached with DCPM. Thus, for hybrid main memory, it is important to efficiently use DRAM to mitigate performance degradation caused by DCPM.

5.2.3 Contribution of Dynamic Relocation in Fogcached

The peak value of latency distribution was 310 to 320 us when dynamic relocation of Fogcached was disabled. It was 20 us larger than the result of Fogcached with dynamic relocation. Even though Memaslap selects the key of a request in a nearly random manner, dynamic relocation slightly improved the hit ratio to DRAM.

6. Conclusion

We proposed a KVS (Key-Value Store) server program designed for hybrid main memory consisting of DRAM and non-volatile memory. We used Optane DCPM as non-volatile memory for its prototype. Fogcached implements a Dual-LRU (Least Recently Used) mechanism that seamlessly extends the memory management of Memcached to hybrid main memory. Cached objects are automatically moved between the two memory devices according to their access frequencies. Through experiments, we confirmed that Fogcached improved the peak value of a latency distribution by approximately 40% compared to Memcached.

A possible direction of future work is to discuss the performance scalability of a KVS server for a huge size of main memory, which is likely limited to the scalability of its index structure. Holding a large amount of data on the LRU algorithm will increase lock contentions and hash collisions. Other algorithms may achieve better performance. Another direction is to discuss the use of persistency of non-volatile memory for KVS servers. Furthermore, comparison with other Memcached variants is an interesting topic. Evaluation from the viewpoints of other performance metrics such power consumption is also invaluable for edge systems.

Acknowledgments

This research was supported by Japan Science and Technology Agency (JST), CREST, JPMJCR19K1, and JSPS KAKENHI (19H01108) in part.

References

[1] K. Bilal, O. Khalid, A. Erbad, and S.U. Khan, "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers," Computer Networks, vol.130, pp.94–120, 2018.
[2] "International roadmap for device and systems 2018 edition," https://irds.ieee.org/editions/2018
[3] Intel® Optane™ DC Persistent · Memory. https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/optane-dc-persistent-memory.html, 2019-11-7.
[4] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y.J. Soh, Z. Wang, Y. Xu, S.R. Dulloor, J. Zhao, and S. Swanson, Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.
[5] dormando. Memcached wiki. https://github.com/memcached/memcached/wiki, 2019.
[6] "Extstore In The Cloud," https://memcached.org, 2019-11-7.
[7] K. Ozawa, T. Hirofuchi, R. Takano, and M. Sugaya, "fogcached: DRAM-NVM Hybrid Memory-Based KVS Server for Edge Computing," 4th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, Proceedings, pp.50–62, Sept. 18–20, 2020.
[8] J.S. Meena, S.M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," Nanoscale Research Letters, vol.9, no.1, pp.1–33, 2014. http://dx.doi.org/10.1186/1556-276X-9-526
[9] L. Wang, C.-H. Ynag, and J. Wen, "Physical principles and current status of emerging non-volatile solid-state memories," Electronic Materials Letters, vol.11, no.4, pp.505–543, 2015. http://dx.doi.org/10.1007/s13391-015-4431-4
[10] S. Oikawa, "Unification of Non-volatile Main Memory and File System," IPSJ Journal, vol.54, no.3, pp.1153–1164, 2013.
[11] Y. Han and E. Lee, "CRAST: Crash-resilient data management for a key-value store in persistent memory," IEICE Electron. Express, vol.15, no.23, p.20180919, 2018. https://www.jstage.jst.go.jp/article/elex/15/23/15_15.20180919/_article
[12] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM Footprint with NVM in Facebook," Proc. Thirteenth EuroSys Conference, pp.1–13, 2018.
[13] D. Charlie, "Intel's Xpoint is pretty much broken. In their own words it isn't close to the promises," https://semiaccurate.com/2016/09/12/intels-xpoint-pretty-much-broken, 2016.
[14] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," USENIX ATC '17 Proc. 2017 USENIX Conference on Usenix Annual Technical Conference, pp.349–362, 2017.
[15] H. Liu, L. Huang, Y. Zhu, and Y. Shen, Librekv: A persistent in-memory key-value store," IEEE Trans. Emerging Topics in Computing, pp.1–1, 2017.
[16] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang, "NVMcached: An NVM-based Key-Value Cache," Proc. ACM SIGOPS Asia-Pacific Workshop on Systems, pp.1–7, Aug. 2016.
[17] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware Hybrid Memory Management for In-Memory Key-Value Stores," IEEE Trans. Parallel Distrib. Syst., vol.31, no.4, pp.779–792, 2019 DOI 10.1109/TPDS.2019.2945315
[18] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," Proc. USENIX Annual Technical Conference, pp.91–104, June 2001.
[19] F.J. Corbató, M. Merwin-Daggett, and R.C. Daley, "An Experimental Time-Sharing System," Proc. 1962 Spring Joint Computer Conference, 1962.
[20] "EVCache," https://github.com/Netflix/EVCache, 2019 refrence.
[21] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," Proc. International Conference for High Performance Computing, Networking, Storage, and Analysis, pp.31:1–31:13, Nov. 2018.
[22] "ndctl," https://github.com/pmem/ndctl, 2019-1-13.
[23] "Memaslap," http://docs.libmemcached.org/bin/memaslap.html, 2019-12-18.
[24] "Redis," https://redis.io, reference 2019-12-18.
[25] "Open Cache Acceleration Software," reference 2019-12-18.

Koki Ozawa is a student in Information Engineering from Shibaura Institute of Technology from 2016. He is also a student member of IEICE, IPSJ of Japan.

Takahiro Hirofuchi is a senior researcher of National Institute of Advanced Industrial Science and Technology (AIST) in Japan. He is working on system software technologies for non-volatile memory devices. He obtained a Ph.D. of engineering in March 2007 at the Graduate School of Information Science of Nara Institute of Science and Technology (NAIST). He is an expert of operating system, virtual machine, and network technologies.

Ryousei Takano is a senior research scientist of the Institute of Advanced Industrial Science and Technology (AIST), Japan. He received his Ph.D. from the Tokyo University of Agriculture and Technology in 2008. He joined AXE, Inc. in 2003 and then, in 2008, moved to AIST. His research interests include system software and high-performance computing. He is currently exploring an operating system for heterogeneous accelerator clouds.

Midori Sugaya is a professor of the Shibaura Institute of Technology (SIT), Faculty of Science and Engineering, Japan. She received his Ph.D. from the Waseda University in 2010. Previously, she was a researcher of CREST project, and also belonged to the Research and Development Center of the project from 2008-2010. Her research interests include system software and dependable computing, and emotion aware robotics. She is a member of IEEE, ACM. He is also a member of IEICE, IPSJ of Japan.