Designing Persistent Heap for Byte Addressable NVRAM

Taeho Hwang Dokeun Lee Yeonjin Noh Youjip Won
Hanyang University, Seoul, Korea
{htaeh|matureelf|maguyn|yjwon}@hanyang.ac.kr

***Abstract*—NVRAM, such as STT-MRAM and 3D XPoint, enables persistent heap to replace existing file systems for data persistence. The persistent heap can eliminate the overhead of serializing the data structure into the file address space by granting persistence to the data structure. In this paper, we defined a persistent object store for persistent heap, and developed a namespace and persistent object management scheme for persistent object store. The persistent heap technique can impose a burden on the user, such as the use of an interface for a persistent object and consideration of a persistent object that becomes garbage. We developed a code regenerator, garbage collection, software transactional memory-based key value library and swapping technique for persistent heap to eliminate the burden that a user may experience when using persistent heap.**

I. Introduction

NVRAM, such as STT-MRAM and 3D XPoint [12], can be accessed at byte granularity and can hold data without supply of electric power. Considering the characteristics of NVRAM, recent work is being conducted to deploy NVRAM as a persistent heap [10], [3], [17], [8], [13], file system [4], [18], [6], [19], and write back cache [7]. Intel predicted to release a 3D XPoint-based DIMM in 2018 [11]. In addition, Intel is developing a library, which is called NVML [5], that provides a low-level interface for users to use NVRAM. Microsoft developed NTFS that supports DAX function and developed Windows 10 and Windows server 2016 supporting NVRAM [16]. HPE demonstrated a Gen9 server with 8GB DDR4 NVDIMM [9]. The development of the NVRAM platform by companies is accelerating the commercialization and diffusion of NVRAM.

Among the works considering NVRAM, the persistent heap [10], [3], [17], [8], [13] proposes the concept of persistent object store as a new persistent storage space instead of the existing file. The persistent heap provides persistence to the in-memory data structure and provides the ability to remove the serialization process between the in-memory data structure and the file. A user can allocate persistent objects at byte granularity through persistent heap and assign persistence to data structures by constructing data structures with persistent objects. Persistent heap can be applied to key-value stores and improve the performance of libraries and applications. However, the use of persistent heap requires the user to consider new interfaces, garbage objects, consistent updates of data structures, and limited NVRAM capacity.

978-1-5386-1768-7/17/$31.00 © 2017 IEEE

In this paper, we present a persistent heap-based persistent object store and present four element techniques for persistent heap to provide convenience to the user. First, we developed a key-value library based on software transactional memory. A user can allocate a persistent object and construct a data structure with persistent object. However, in the process of updating the data structure, the user can make the data structure inconsistent. The key-value library based on software transactional memory provides the ability to update data structures consistently for the user. Second, we developed a code regenerator. DRAM and NVRAM have different energy consumption in read and write operations. Depending on the object's read and write frequency, user can reduce energy consumption by assigning objects selectively to DRAM or NVRAM. The code regenerator regenerates the code so that the object is assigned to NVRAM according to the memory access characteristics of the object to reduce energy consumption. Third, we developed a garbage collection. At system shutdown, existing objects in the DRAM disappear from memory. On the other hand, persistent objects on NVRAM remain in memory even at system shutdown. When objects and persistent objects become garbage, objects on the DRAM are reclaimed at system shutdown, but persistent objects on NVRAM are not reclaimed and continue to occupy memory space persistently. The garbage collection provides functionality that garbage object can be reclaimed. Fourth, we have developed a swapping technique for NVRAM. Since NVRAM has a lot to consider for commercialization, such as price-to-capacity, early devices using NVRAM are expected to have a limited capacity of NVRAM. User cannot allocate objects from NVRAM because of a small amount of NVRAM. The swapping technique for NVRAM uses NAND flash as the swapping space of NVRAM to provide an unlimited NVRAM capacity to the user. In the remaining sections, we describe a persistent heap-based persistent object store and four element techniques for persistent heap (Section 2), an experiment (Section 3), and a conclusion (Section 4).

II. Design and Implementation

A. Heap-Based Persistent Object Store

We developed Heap-Based Persistent Object Store (HEAPO) [10]. HEAPO consists of a user library that manages the allocation and deallocation of persistent objects, and a kernel layer that manages persistent object stores on NVRAM. HEAPO does not depend on the file system to

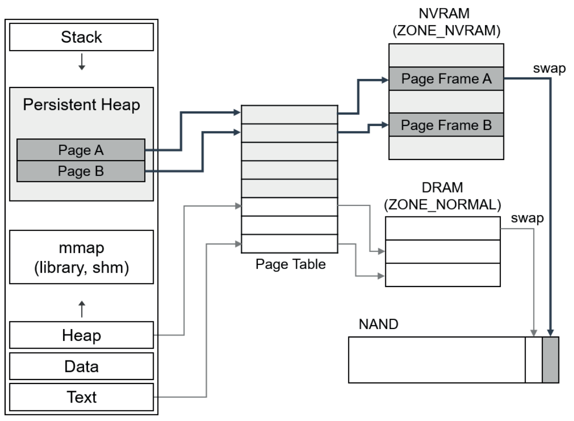

Fig. 1. Physical architecture of HEAPO

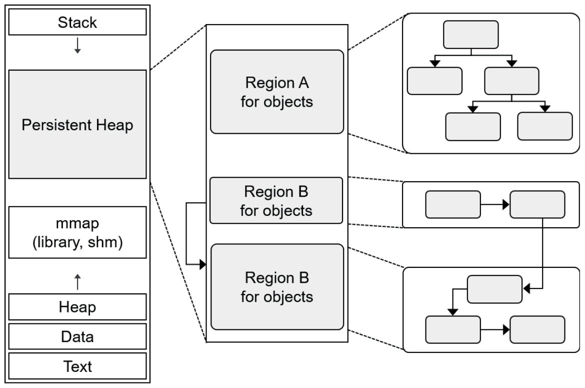

Fig. 2. Logical address space layout of HEAPO

manage the metadata for the persistent object store, but it has its own native layer. HEAPO uses in-memory metadata without duplication of in-memory and on-disk metadata through the native layer. Fig. 1 illustrates the physical architecture of HEAPO, consisting of DRAM, NVRAM, and NAND Flash.

HEAPO has three persistent address space concepts: persistent heap, persistent object store, and persistent object. A persistent heap is an address space in which a persistent object store can reside and has a fixed position in the virtual address space (0x5FFEF800000-0x7FFEF800000, 32TByte). Persistent heap is mapped to NVRAM for the persistence of the data in persistent heap. A persistent object store is an address space with a name. A persistent object store consists of a set of clusters and each cluster consists of consecutive pages. The user who has access to a persistent object store can access the persistent object store after attaching the persistent object store to the user address space. A persistent object is a persistent address space which can be allocated to a user at byte granularity. User can create persistent data structures such as trees and lists that are accessible in-place with persistent objects. Fig. 2 illustrates the layout of persistent heap, persistent object store, and persistent object. There are two persistent object store A and B in the persistent heap. Persistent object store A and B contain a tree and a list, respectively, which consists of persistent objects. Persistent object store B consists of two clusters. HEAPO maintains metadata for persistent heap, persistent object store, and namespace. Metadata for persistent heap and persistent object store correspond to superblocks and inodes in the file system. The metadata for the persistent object store includes location information of the clusters constituting the persistent object store and mapping information with NVRAM.

As the file system manages a namespace consisting of names of files, HEAPO manages the namespace consisting of names of persistent object stores. User can assign a name to a persistent object store when creating it, and specify which persistent object store to access via its name. HEAPO manages the global namespace in the kernel address space, and manages the local namespace in the user address space. Accessing the global namespace in the kernel address space each time when an object is allocated contains a large overload. HEAPO manages attached persistent object stores in the local namespace as file system manages dentries. HEAPO uses a burst trie for the global namespace and a hash table data structure for the local namespace.

HEAPO provides the user with a HEAPO library that performs creation and deletion of persistent object store and allocation and deallocation of persistent object. The library includes heapo_create() to create a persistent object store, heapo_map() to reuse created persistent object store, and heapo_malloc() to allocate persistent object from persistent object store as user interface. A persistent object store is allocated a 4 KB page frame when it is created. The HEAPO library increases the size of the persistent object store when there is not enough free space to allocate persistent objects to the user. If the size of the persistent object store can not be increased continuously, HEAPO maintains a persistent object store consisting of several clusters. Accessing a persistent object store involves three steps: 1) Locating the persistent object store in the virtual address space through the namespace, 2) Attaching persistent object store to virtual address space after confirming user's access right. 3) Mapping between page and page frame by page fault handler, when a user accesses a page in the persistent object store. The HEAPO library maintains a list of free chunks within the persistent object store like glibc which manages the heap address space. Glibc places information for free space in the mmap segment. On the other hand, the HEAPO library places that information in the first page frame that has been created for the persistent object store for the persistence of free space information.

B. Key-Value Library based on Software Transactional Memory

We developed a failure safe software transactional memory (F-STM) that uses HEAPO to process transactions consistently from power failure, and developed a key-value library using F-STM to ensure concurrency and consistency. F-STM is based on the TinySTM library [1], a word-based STM used in volatile memory. TinySTM manages a separate repository

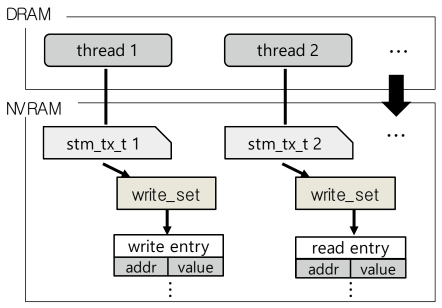

Fig. 3. F_STM structure in NVRAM

for each thread, and records and checks the read and write operations for each thread to keep the shared data consistent. Since data and data structures for existing STM exist in volatile memory, if power failure occurs, all data managed by existing STM disappears. F-STM allocates a write set and write entry from NVRAM that require persistence to perform recovery in case of power failure (Fig. 3).

The F-STM allocates a persistent object through the HEAPO library and uses the persistent object as a write entry. When a transaction terminates normally, all transaction-related data is released, and a log consisting of persistent objects is retained in NVRAM when abnormally terminated. When the user uses F-STM, F-STM checks the log area. If transactional information remains in the log area, F-STM assumes that the previous transaction has terminated abnormally. If an abnormal termination is detected, F-STM performs recovery based on the write entry on NVRAM.

Modern processors have a cache. Although the user code performs a memory write, the corresponding memory write is applied to the cache first, not directly to NVRAM. Modern compilers and processors can also change the order in which user code is executed. F-STM uses the clflush and mfence combination to ensure ordering between memory writes and durability of memory writes.

F-STM uses two methods to verify that logs are atomically written to NVRAM during log generation. The first method is the commit mark method that a flag is added to the log. The method writes the log to NVRAM and then writes the commit value to the flag. When F-STM accesses the logs, it checks flag value of the accessing log. If the flag value is not a commit value, F-STM assumes that the crash occurred while flushing the log to NVRAM, and F-STM does not use the log. This method calls the clflush command after each write. The second method is to add a checksum field in the log to check the log value. The log is validated by comparing the log and checksum values. When F-STM accesses the logs, it calculates the checksum value of the accessing log and compares it with the stored checksum value. If the checksum value is different, F-STM assumes that the crash occurred while flushing the log to NVRAM, and F-STM does not use the log. The checksum method calls a single clflush command. The clflush command, which flushes the cache line and marks the line as invalid, has a large overload, so F-STM uses the checksum field method by default.

F-STM provides two methods to guarantee atomicity of transaction: write-through and write-back. The write-through method stores the original value of the object in an undo log to process the write to the object during the transaction. In order to process rollbacks in reverse order of the order in which undo logs are stored, F-STM records sequence number at each undo log of a transaction. Upon rollback, the F-STM restores the original value stored in the undo log, from the undo log with the largest sequence number to the undo log with the smallest sequence number. The write-back method writes the write value to the redo log to process the write to the object during the transaction. After generating a redo log for all writes, the transaciton turn into a commit state. When a system crash occurs and the transaction status is in the commit state, F-STM reflects the values stored in the redo logs to the object. If the transaction is not in the commit state, the transaction is aborted.

C. Code Regenerator

Through the HEAPO library, the user can be assigned a persistent object on NVRAM. We developed a code regenerator that changes the code that allocates objects from DRAM to the code that allocates object from NVRAM, depending on the usage pattern of the object. Code rengenerator extracts access information for the object through code profiler [15]. The code profiler uses a low-level virtual machine (LLVM) compiler framework to add a callback function for the profiling between load and store commands of the target program. The profile callback function measures the number of reads and writes to some object while the modified program is running. The code profiler calculates the amount of read and write energy if an object is allocated in DRAM or NVRAM based on the number of read and write of the object. The calculated energy consumption determines whether the object is allocated in DRAM or in NVRAM. The code profiler finds the object allocation code which needs to be changed to persistent object allocation code, and generates the output file name and line number associated with the code. Based on the information generated through profiling, the code regenerator changes the object allocation code to the persistent object allocation code of HEAPO library.

In order to allocate a persistent object, a persistent object store must exist in advance. The code regenerator searches the main function of the program and adds the persistent object store creation code to the beginning of the main function. To avoid sharing the persistent object store with other programs, the program's full directory path is used as the name of the persistent object store. If there is a persistent object store in advance, the persistent object store is attached to the user address space.

Since regenerated program allocates not only the temporary object but also the persistent object, the object release code is determined by whether the object is the temporary object or the persistent object. HEAPO uses a fixed virtual address space (0x5FFEF800000-0x7FFEF800000, total 32TByte) as a persistent heap. Depending on the address space in which the object is located, we can distinguish whether the object is a temporary object or a persistent object. If the object's address is in the persistent heap address space, the object is freed through the HEAPO library. If the object's address is not in the persistent heap address space, the object is freed through the legacy object library (malloc/free).

D. Garbage Collection

We developed two-tier garbage collection: Local Garbage Collection (LGC) and Separated Full Garbage Collection (SFGC). We divide persistent object store into two groups according to whether it is attached or not and apply seperate garbage collection to the divided groups. LGC is applied to the attached persistent object store, and SFGC is applied to the remaining unattached persistent object store. SFGC is performed in background once every 5 seconds.

Since the attached persistent object store may be in use by other processes, memory allocation and deallocation requests must be blocked during garbage collection to maintain consistency. The design of the garbage collector to track the processes in which the object store is being used and block their IO can be a complex structure that can cause deadlocks in conjunction with the allocation and deallocation locks of the HEAPO library. Therefore, it is efficient to perform garbage collection on foreground during allocation by using object store lock of HEAPO library. To do this, the process is designed to operate by calling the garbage collection code directly in the process of requesting object allocation of the object store. If object store is not attached, it is not used by some process and is not urgent for garbage collection. It is not appropriate to carry out garbage collection for unattached object store through foreground garbage collection. Therefore, garbage collection for these object stores is performed when some CPU resources are left in the background by automatically mapping a part of the operating daemon to its own address space. The two-tier garbage collection is a structure in which garbage collection is not likely to fail because there is no space that is not subject to garbage collection in any case, and the overhead of garbage collection is dispersed so that the user does not recognize it as much as possible.

Conventional garbage collection algorithms like mark and sweep [20] collect garbage by comparing a set of user-accessible object and a set of allocated object. A set of user-accessible object can be obtained by traversing the user data structure. A set of allocated object can be derived from memory allocator. The problem of this comparison method is that it needs to search as many as the number of objects constituting the data structure because it must be retrieved from the data structure that the user can access each allocated space. Garbage search is the most time-consuming task in garbage collection, and reducing this time affects the performance of the garbage collector. We designed the Fast Scan Algorithm to reduce this overhead. This algorithm is based on the fact that a set of user-accessible object and a set of allocated object match if there is no garbage. By arranging two sets according to object address and comparing the only objects corresponding to the same order, it is possible to detect the garbage by only one round without searching several times.

E. Swapping

HEAPO has a limitation that the number of persistent object store that a process can create is limited to the size of NVRAM as it maps the persistent heap space of the process to NVRAM one to one. This can lead to a spatial burden that HEAPO retains persistent object store in NVRAM that is no longer used or frequently used. To reduce the spatial overhead of NVRAM, a policy is required to swap the persistent object store from NVRAM to a relatively slower storage device like NAND flash. However, existing volatile memory-based page swapping algorithms such as SCLRU and CFLRU are optimized for file system memory management. Such a volatile memory-based swapping policy causes additional operations that do not consider the characteristics of the NVRAM. This additional operation further increases the delay time of the swapping operation, which causes the performance degradation of the overall system. We developed Unmapped-object First LRU (UFLRU), a swapping policy optimized for the characteristics of NVRAM. For UFLRU, two swapping criteria are used in the swapping policy.

First swapping criterion is whether to refer to the process for the swap-out target page. In the case of a page referenced by a process, the reverse mapping tree structure records information about page table entries of processes that refer to the page. Therefore, a page selected as a swap-out target needs to modify the page table information of all the processes that refer to the page by rotating the reverse mapping trees connected to the page. This process involves a considerable amount of time in the process of searching for processes and changing the page table of each process. To prevent this, UFLRU first checks whether a page is referenced. If there is no process that references the page when the page is checked, the speed of the swap-out operation can be reduced because there is no need to traverse and modify the process. Therefore, the UFLRU judges that the priority of the swap out selection criterion of the unreferenced page is higher than that of the page referenced by the process.

Another swapping criterion is whether the page is dirty. A dirty page occurs when the contents of a page written to memory are different from the contents of a page on a storage device. Therefore, if the page selected by swap-out is a dirty page, the page data should be written to the swap partition of the NAND flash memory. Access to other processes is restricted while writing the contents of the page in the swap area. Such a page processing job requires a lot of time costs by accessing and recording the storage device and restricting access to other processes during the operation. On the other hand, if the page is not a dirty page, it does not require access to the storage device, and it can retrieve the page immediately in memory, ensuring fast swap in and out speed. Therefore, the UFLRU judges that the priority of the swap out selection criterion of the clean page is higher than that of the dirty page.

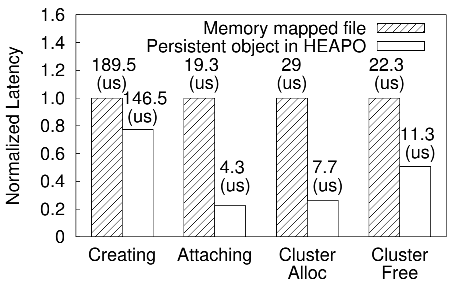

Fig. 4. Metadata operations

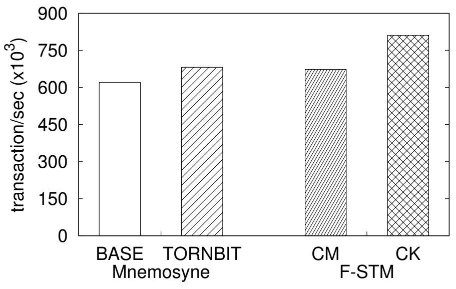

Fig. 5. Performance of insert operation

UFLRU manages all pages for persistent object store as active and inactive page list. A page newly allocated or frequently used by a process is inserted into the head of the active page list. If the inactive page list is smaller than the active page list during the swap-out process for securing the memory for the persistent object store, the page of the active page list is checked from the tail and released to the inactive page list. Whether to insert the page to be evicted into the head or tail of the inactive list is determined according to the UFLRU's swap-out target page selection criteria. In case of referenced or dirty page, it is inserted into the head of the inactive list. If there is no process to be referenced and it is a clean page, it is inserted into the tail of the inactive list. Through such page list management, pages that require less time for page processing are arranged at tail of inactvie page list, and pages that require more time for page processing are arranged at head of inactive page list. UFLRU aims at improving performance by reducing the time costs for page processing through this page list management method.

In order for a process to access a persistent object store that has been swapped out to NAND flash memory, a page fault is generated by referring to the swap entry stored in the page table entry. The page fault handler checks the page table entry that caused the page fault and decides whether to fetch data from the swap cache area or access swapped-out data from the swap area of NAND flash storage.

III. Evaluation

A. Metadata Operation

We measured the metadata performance of HEAPO using in-place metadata for persistent heap and persistent heap based on memory-mapped file. The experiment was performed on AMD Phenom X4 925 Processor (2.8GHz) and 12GB DDR3 DRAM. we located the memory-mapped file in 10Gbyte ramdisk and formatted 10GByte ramdisk with EXT4. In the experiment of metadata operation, creating, attaching, expanding and shrinking object store were performed. Fig. 4 illustrates the latency of each metadata operation. HEAPO showed latency 1.3 times and 4.5 times faster than memory-mapped file method in creating and attaching object store, respectively. Attaching an object store in a memory-mapped file method requires copy and initialization process between in-memory metadata and on-disk metadata.

B. Key-Value Insert Transaction Performance

In this section, we measured the performance of a key-value library that consistently updated the data structure. We have created a microbenchmark that inserts a key-value pair a specified number of times into a data structure such as a list to measure performance.

We compared the performance of F-STM with mnemosyne [17] to ensure consistency of data structure using STM. The experiment was performed on Intel (R) Core (TM) i7-3770 (3.4 GHz) CPU and 12 GB Memory. We measured transaction per seconds (TPS) when inserting 30,000 key-value pairs through microbenchmark. The data structure of the key-value library used in the experiment is a list. Fig. 5 illustrates the insert performance of Mnemosyne base, tornbit, F-STM commit (CM), and checksum (CK). F-STM Checksum is 18.9 % higher than Mnemosyne tornbit.

C. Commit Mark and Checksum Field in F-STM

F-STM uses the commit mark or checksum filed method to verify that the log is atomically written to NVRAM. We compared the performance of the commit method using the clflush command twice and the checksum method using the clflush command once. The experiment was performed on the TUNA board [14], [2], which can emulate NVRAM. TUNA uses the ARM platform and emulates NVRAM by supplying extra power to DRAM. The TUNA platform provides 15 levels of latency for load and store. We measured transaction per seconds (TPS) when inserting 252,000 key-value pairs through microbenchmark. The data structure of the key-value library used in the experiment is a list. Fig. 6 illustrates the insert performance of commit mark method and checksum method. Checksum method is 10% higher than commit mark method.

D. Thread Implication

STM allows multiple threads to enter a ciritical section freely while ensuring concurrency. We measured the performance of the key-value library according to the number of threads in F-STM. The experiment was conducted on the TUNA board. Since the number of cores on the TUNA board is two, we increased the number of threads to two for experiment. For one thread, the transaction per seconds (TPS) was measured when inserting 252,000 key-value pairs through the microbenchmark. When there are two threads, we measure

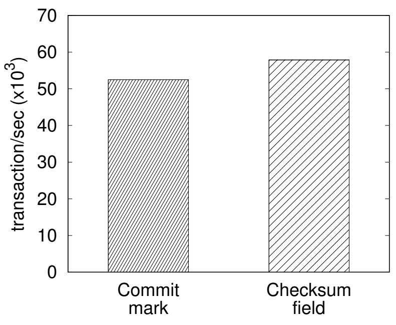

Fig. 6. Performance comparison between commit mark and checksum field

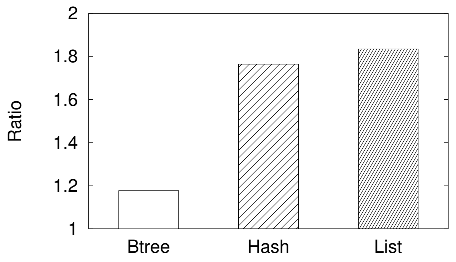

Fig. 7. Performance ratio when the number of thread increases from one to two

transaction per seconds (TPS) when inserting 126,000 key-value pairs in each thread. The data structures of the key-value library used in the experiment are list, hash table, and Btree.

Fig. 7 illustrates how much performance is improved when the number of threads increases from one to two. List is the highest with 1.8 times performance improvement. Btree is the smallest with 1.2 times performance improvement. If there are many store operations, the probability of collision increases, and the overhead for managing the content of STM increases. Since Btree has many store operations in transactions, it shows relatively low performance improvement.

IV. Conclusion

Persistent heap makes it possible to assign persistence to data structures and allow in-place update of data structure. We proposed heap-based persistent object store (HEAPO) that does not go through the file system hierarchy. The use of persistent heap can cause inconsistent updates of data structures, memory leaks, and a limited NVRAM capacity to the user. We proposed STM-based key-value library, code regenerator, garbage collection, and swapping to solve the above problem. With these methods, users can update the data structures consistently through the corresponding techniques, allocate objects with energy efficiency, use NVRAM without memory leaks, and allocate objects over NVRAM capacity.

Acknowledgment

This work was supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device), the BK21 plus program through the National Research Foundation (NRF) funded by the Ministry of Education of Korea, the ICT R&D program of MSIP/IITP (R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services) and Basic Research Laboratory Program through the National Research Foundation (NRF) funded by the Ministry of Science, ICT & Future Planning (MSIP) (No. 2017R1A4A1015498).

References

[1] TinySTM: A lightweight and efficient software transactional memory implementation in c. http://tmware.org/tinystm.

[2] CMALab. Tuna: the platform for emulating nvram. http://opennvram.org/.

[3] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.

[5] Intel Corporation. Persistent Memory Programming, 2015. http://pmem.io/nvml/.

[6] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proc. of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[7] Ziqi Fan, Fenggang Wu, Dongchul Park, Jim Diehl, Doug Voigt, and David HC Du. Hibachi: A cooperative hybrid cache with nvram and dram for storage arrays. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2017.

[8] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2012.

[9] HPE. HPE Persistent Memory for HPE ProLiant Servers, 2016.

[10] Taeho Hwang, Jaemin Jung, and Youjip Won. Heapo: Heap-based persistent object store. *ACM Transactions on Storage (TOS)*, 11(1):3, 2015.

[11] Intel. A New Breakthrough in Persistent Memory Gets Its First Public Demo, 2017. https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/.

[12] Intel and Micron. Intel and micron produce breakthrough memory technology, 2015. https://newsroom.intel.com/news-releases/intel-andmicron-produce-breakthrough-memory-technology/.

[13] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: exploiting nvram in write-ahead logging. In *ACM SIGPLAN Notices*, volume 51, pages 385–398. ACM, 2016.

[14] Taemin Lee, Dongki Kim, Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. Fpga-based prototyping systems for emerging memory technologies. In *Proc. of IEEE International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2014.

[15] Security and Privacy Lab. System design methodology, Newmemory profiler. https://github.com/sdmlab/NVRAMPF.

[16] Tom Talpey. Persistent memory in windows server 2016. In *Persistent Memory Summit*, 2017.

[17] H. Volos, A.J. Tack, and M.M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[18] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[19] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[20] Benjamin Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proc. of ACM conference on LISP and functional programming*. ACM, 1990.