HEAPO: Heap-Based Persistent Object Store

TAEHO HWANG, JAEMIN JUNG, and YOUJIP WON, Hanyang University

In this work, we developed a Heap-Based Persistent Object Store (HEAPO) to manage persistent objects in byte-addressable Nonvolatile RAM (NVRAM). HEAPO defines its own persistent heap layout, the persistent object format, name space organization, object sharing and protection mechanism, and undo-only log-based crash recovery, all of which are effectively tailored for NVRAM. We put our effort into developing a *lightweight* and *flexible* layer to exploit the DRAM-like access latency of NVRAM. To address this objective, we developed (i) a native management layer for NVRAM to eliminate redundancy between in-core and on-disk copies of the metadata, (ii) an expandable object format, (iii) a burst trie-based global name space with local name space caching, (iv) static address binding, and (v) *minimal* logging for *undo-only* crash recovery. We implemented HEAPO at commodity OS (Linux 2.6.32) and measured the performance. By eliminating metadata redundancy, HEAPO improved the speed of creating, attaching, and expanding an object by 1.3×, 4.5×, and 3.8×, respectively, compared to memory-mapped file-based persistent object store. Burst trie-based name space organization of HEAPO yielded 7.6× better lookup performance compared to hashed B-tree-based name space of EXT4. We modified `memcachedb` to use HEAPO in maintaining its search structure. For hash table update, HEAPO-based memcachedb yielded 3.4× performance improvement against original memcachedb implementation which uses `mmap()` over `ramdisk` approach to maintain the key-value store in memory.

Categories and Subject Descriptors: D.4.2 [**Operating System**]: Storage Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Persistent heap, Persistent objects, Nonvolatile memory

ACM Reference Format:
Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-based persistent object store. ACM Trans. Storage 11, 1, Article 3 (December 2014), 21 pages.
DOI: http://dx.doi.org/10.1145/2629619

3

1. INTRODUCTION

The demand for larger main memory has been increasing. Rapid proliferation of microblog sites, such as Facebook and Twitter, and web search engines [Chang et al. 2008; Apache 2009; Chodorow 2010] requires a working set of large size key-value stores and database tables to be in memory to overcome the limited performance of the block

This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device), and partially supported by IT R&D program MKE/KEIT. [No.10035202, Large Scale hyper-MLC SSD Technology Development]. This research was also supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014-H0301-14-1017) supervised by the NIPA (National IT Industry Promotion Agency).
Authors' address: T. Hwang, J. Jung, and Y. Won (Corresponding author), Department of Computer and Software, Hanyang University; emails: {htaeh, jmjung, yjwon}@hanyang.ac.kr.

DOI: http://dx.doi.org/10.1145/2629619

device. Today's servers are loaded with several hundred gigabytes of DRAM, but the size of the entire key-value store is still larger than what a single server can host.

The amount of DRAM that can be installed in a single node is limited due to various reasons, for example, power consumption [Qureshi et al. 2009] and scaling factor [Yoon et al. 2013]. Also, the volatile nature of DRAM makes crash recovery and checkpoint particularly cumbersome in a system with a large size DRAM. A few works proposed combining DRAM and NAND Flash into a single device to provide a large size nonvolatile main memory [Badam and Pai 2011; Diblotechnology 2013; Vikingtechnology 2013]. While these devices address volatility and the size issue of DRAM, they are much more expensive than DRAM and the long-term retention and endurance behavior of their NAND Flash device are yet to be known.

Newly emerging memory devices that can hold data without electricity and access data at the byte granularity have brought forth a new opportunity to address the technical issues that current a DRAM-based computer system faces: volatility, crash recovery overhead, power consumption, serialization overhead, and limited DRAM scale. The use of these devices is expected to bridge the chasm between DRAM (byte-addressable and volatile) and storage (block-addressable and nonvolatile). New Nonvolatile RAM (NVRAM) devices include Spin-Transfer Torque Magnetic RAM (STT-MRAM) [Kim and Lam 2012], Phase-Change RAM (PCRAM) [Kryder and Kim 2009], Ferroelectric RAM (FRAM) [Perez 2012], Resistive RAM (RRAM) [Wang and Amiri 2012], and others. Each of these devices has unique physical characteristics, for example, scalability, energy consumption, access latency, and endurance.

In this work, we aim at developing a *lightweight* and *flexible* management layer for persistent heap. The existing persistent heap approaches, which rely on memory-mapped files [Coburn et al. 2011; Volos et al. 2011], can utilize the rich features of the underlying file system, for example, metadata operations and name space management service. However, we carefully argue that inefficiencies caused by redundant metadata and system call overhead leave much to be desired in terms of fully exploiting the DRAM-like access latency and byte addressability of emergent devices. In this work, we developed a lightweight and robust persistent heap management layer, HEAPO (Heap-based Persistent Object Store). The key design features of HEAPO are as follows.

—*Native management layer for persistent heap*: Via providing a direct management layer for NVRAM, HEAPO does not discretely retain *in-core* and *on-disk* metadata. HEAPO defines its own metadata for persistent object, removing redundancy between in-core and on-disk metadata of a file. HEAPO manages the persistent object with a *single* metadata, eliminating the metadata synchronization overhead from which the mmap-based persistent heap design [Coburn et al. 2011; Volos et al. 2011] may suffer.
—*Global name space with local name space caching*: HEAPO has *trie*-based [Heinz et al. 2002] memory-friendly directory structure. It maintains global name space and defines its own lightweight name space since the existing directory structures, for example, linked list [Cao et al. 2005] and B-Tree [Sweeney et al. 1996; Mathur et al. 2007], are designed for block devices.
—*Extensible object*: Persistent objects in HEAPO, different from the legacy heap object, can dynamically extend. HEAPO expands a persistent object by allocating additional pages directly from the persistent heap. In HEAPO, a persistent object does not have to be allocated to a consecutive address space, which makes the use of a persistent heap segment much more efficient.
—*Static binding*: In HEAPO, a section of the virtual address space is reserved for a persistent heap, and all processes share a global persistent heap. A persistent object

Table I. Comparison of Memories [Kryder and Kim 2009; Perez 2012; Wang and Amiri 2012; Kim and Lam 2012].

Item	DRAM	FRAM	PCRAM	MRAM	STT-MRAM	NAND
Maturity	Product	Product	Product	Product	Prototype	Product
Byte addressable	Yes	Yes	Yes	Yes	Yes	No
Nonvolatile	No	Yes	Yes	Yes	Yes	Yes
Access time (W/R)	10/10ns	50/75μs	100/20ns	12/12ns	10/10ns	200/25μs
Program energy	2pJ	2pJ	100pJ	120pJ	0.02pJ	10nJ
Cell size	$6F^2$	$6F^2$	$5F^2$	$20F^2$	$4F^2$	$4F^2$
Endurance	10^{16}	10^{15}	10^5	10^{16}	10^{16}	10^5

is statically bound to its virtual address, rendering the object sharing and pointer resolution versatile.

HEAPO solves the issues that the memory-mapped file approach fails to address in realizing a persistent heap [Coburn et al. 2011; Volos et al. 2011]: redundant metadata, heavy system call-based name space management, and inflexible object expansion in memory-mapped file.

The results of our performance experiments with HEAPO are promising. Trie-based in-memory directory structure of HEAPO yields 7.6× faster lookup than B-Tree-based directory structure of EXT4. Eliminating metadata redundancy between in-core and on-disk metadata improves the speed of attaching and expanding a persistent object by 4.5× and 3.8×, respectively, compared to a memory-mapped file approach. We developed a key-value library with HEAPO, H-KVLib, which offers B-Tree and hash table-based store. We examined the performance of H-KVLib and Berkeley DB; H-KVLib exhibits 7.5× and 18.1× performance improvement in insert and lookup of a hash table, respectively. We modified Memcachedb [Chu 2008], a distributed key-value store, to adopt H-KVLib in lieu of Berkeley DB. H-KVLib-based Memcachedb exhibits 3.4× performance improvement over Berkeley DB-based Memcachedb in hash table update.

The remainder of this article consists of a brief description of the background (Section 2), the design of HEAPO (Section 3), its implementation (Section 4), its evaluation (Section 6), and a summary of related work (Section 7). Our conclusion is presented in Section 8.

2. BACKGROUND AND PROBLEM ASSESSMENT

2.1. Memory Device Technologies

There is a wide variety of device technologies for byte-addressable nonvolatile memory (Table I) including FRAM [Perez 2012], PCRAM [Kryder and Kim 2009], and STT-MRAM. Each of these devices has unique physical characteristics. FRAM has a scalability issue. The largest FRAM chip available to date is 64Mbit [Kang et al. 2006]. FRAM finds its use as a lower-energy memory device in automotives or sensor devices. PCRAM is the most mature technology [Research 2009]. It is used as a fast storage with small IO latency [Jantunen et al. 2010; Akel et al. 2011], as a cache layer between a main memory and a secondary storage [Liu et al. 2012], or as disparate memory structure [Qureshi et al. 2009]. STT-MRAM aims at directly replacing DRAM. The read and write speeds of STT-MRAM are similar to, or two to three times slower than, those of DRAM [Kryder and Kim 2009]. STT-MRAM effectively addresses the heat and energy consumption issues of DRAM in modern enterprise class servers. It can be used as a memory device.

The software stack for exploiting byte-addressable NVRAM should properly reflect the physical characteristics of individual devices. This work particularly aims

at NVRAM with DRAM-like latency which makes STT-MRAM the right device for HEAPO.

2.2. Persistent Heap versus Byte-Addressable File Ssystem

There are two main approaches to incorporating the NVRAM device into a legacy computer hierarchy: (i) persistent heap [Coburn et al. 2011; Volos et al. 2011] and (ii) byte-addressable file system [Condit et al. 2009; Wu and Reddy 2011]. In the persistent heap approach, the NVRAM device is directly mapped into the process address space. In the process address space, a new type of segment, called *persistent heap*, is allocated to harbor the NVRAM device. The objects in the persistent heap segment are dynamically allocated and deallocated as the objects in the legacy heap. However, the objects in the persistent heap remain orthogonal during the course of the process that created them. The management module of a persistent heap can use a legacy memory management algorithm, e.g., a buddy algorithm. The management module needs to be augmented with naming system, sharing and protection, and pointer resolution features. The persistent heap approach is suitable for NVRAM that has DRAM-like access characteristics, for example, STT-MRAM. Maintaining an object in a heap instead of a block-based storage device removes the burden of serializing and deserializing an object between the block-based storage device and the virtual address space. It also eases the burden of going through complex software stacks, for example, system call, file system, and device driver, to access the objects at the storage device. The existing heap management module needs to be reinforced with name space and mechanisms for protection and sharing.

In the byte-addressable file system approach, the NVRAM device is imposed with file system abstraction [Condit et al. 2009; Wu and Reddy 2011], which exports non-POSIX interfaces to access the file data in subblock granularity. The main advantage of using a file system is that it can inherit various features of the extant file system, for example, the hierarchical name space organization, scalable access control mechanism, and metadata structure, which have evolved over the years and reached sufficient maturity. However, this approach includes a heavy system call, block device, and device driver. Additionally, in accessing file system objects, the context switch-based IO pollutes the Translation Lookaside Buffer (TLB) and Central Processing Unite (CPU) cache and negatively affects the overall system performance. The existing file system separately defines the in-core and on-disk versions of metadata and name space entries. The synchronization overhead becomes rather significant when the storage device bears DRAM-like latency.

2.3. Memory-Mapped File versus Native Heap Management

There are two approaches to maintaining an object in an NVRAM region: memory-mapped files and native heaps. Existing works [Coburn et al. 2011; Volos et al. 2011] depend on memory-mapped files to realize persistent heaps. An *nvheap* in Coburn et al. [2011] and a *region* in Volos et al. [2011] correspond to a single chunk of persistent memory region and are realized using a memory-mapped file. In terms of abstract notions, an *nvheap* and a *region* correspond to a persistent object in HEAPO. Memory-mapped file approach imposes block device abstraction on NVRAM [msdn 2010] and organizes the block device with a commodity file system, for example, EXT2. The objects in the virtual address space are endowed with persistency via being mapped to an NVRAM-based block device. This approach significantly simplifies the implementation because it can exploit all the existing file system primitives, for example, metadata and system calls.

Despite its simplicity, this approach suffers from a number of serious drawbacks, making it practically infeasible. As each persistent object in this approach is basically a

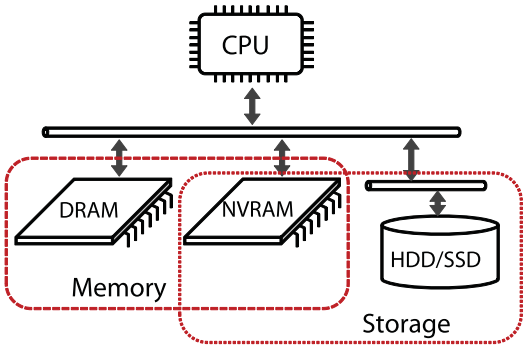

Fig. 1. Architecture of HEAPO.

file, creating and accessing a persistent object involves searching of on-disk metadata, initializing them, and creating in-core metadata (i-node) from the object's on-disk counterpart. The replication overhead of metadata (in-core and on-disk) is crucial. Expanding or shrinking a persistent object requires system calling, which entails crossing the protection boundary and updating metadata, and this is a very expensive operation. In addition, the file must be mapped into consecutive virtual address space, and it is possible that the file cannot grow linearly in the virtual address space because the page that needs to be allocated for the file expansion is already in use. If this happens, the entire file must be remapped to another part of the virtual address space. In a persistent object store, remapping an object to a different virtual address space entails remapping overhead and more importantly, it requires updates on all pointers that point to the remapped object.

2.4. Address Binding: Static versus Dynamic

In static binding, an object is statically bound to a virtual address. In dynamic binding, an object's virtual address may change every time it is mapped to a process' address space. In dynamic binding, the pointers in the persistent object need to be resolved every time the persistent object is mapped to virtual address space. In dynamic binding, the persistent object can be transported across the node into the distributed system. The objective of employing "address relocation" in a modern operating system lied in reusing the limited address space among the processes. If there exists a sufficiently large virtual memory space, static binding will yield better performance. In static binding, heap defragmentation and garbage collection can become more burdensome.

3. HEAP-BASED PERSISTENT OBJECT STORE

3.1. Design

Figure 1 schematically illustrates the hardware organization of the HEAPO system. NVRAM is located on a par with its DRAM counterpart and is directly connected to the system bus. We assume that DRAM and NVRAM form a single physical address space [Freitas and Wilcke 2008]. Figures 2(a) and 2(b) demonstrate the software organization in a legacy computer system and in an NVRAM-enabled computer system hierarchy, respectively.

HEAPO defines a native layer with its own metadata, heap layout, access control, name space, and other parameters without using memory-mapped file over ramdisk. As the name suggests, HEAPO conveys a persistent heap instead of a byte-addressable file system to manage NVRAM. This is primarily to avoid the expensive system call in managing the persistent object. A native management layer eliminates redundancy between the in-core copy and the on-disk copy of the file metadata.

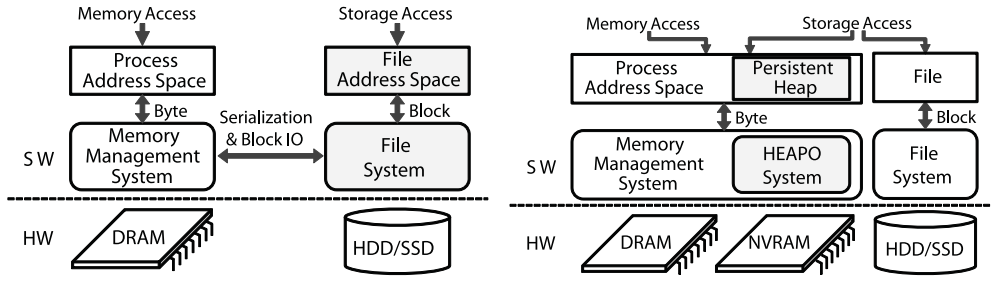

Fig. 2. System layout.

Table II. Design Issues of Persistent Object Store

	HEAPO	NV-Heaps	Mnemosyne	SoftPM
Mapping	Static	Dynamic	Static	Dynamic
Global name space	Native NS	File system NS	None	None
Local name space	Yes	No	No	No
Sharing	Yes	Yes	No	No
Protection	Native ACL	File system ACL	File system ACL	None
Space allocation	Noncontiguous	Contiguous	Contiguous	Contiguous
Persistency	At allocation	At allocation	At allocation	At commit
Address mode	User	User	User	Kernel
Implementation	Native heap	Memory-mapped file	Memory-mapped file	Native heap

Address binding scheme is one of the essential components in persistent heap design. HEAPO allocates a sufficient space (32TByte of virtual address range) to the persistent heap and therefore each object can be assigned a unique location at the time it is created. HEAPO adopts static address binding as in Mnemosyne, wherein each persistent object has a fixed address in the virtual address space. Static address binding relieves HEAPO from the overhead caused by object store relocation, for example, inter- and intraobject store pointer resolution. Through static address binding, sharing of persistent objects becomes much easier.

We compared HEAPO with other recently presented persistent heap proposals: Mnemosyne [Volos et al. 2011], NV-Heaps [Coburn et al. 2011], and SoftPM [Guerra et al. 2012] (Table II). Mnemosyne adopts static address binding, as does HEAPO. NV-heap has its own name space structure, as does HEAPO. The salient features that distinguish HEAPO from the rest are native heap management module, object extensibility, and minimal logging. Table II illustrates the summary of comparisons.

3.2. Persistent Heap Organization

First, let us provide important terminologies in HEAPO. *Persistent heap* is a consecutive address segment in a virtual address space which is mapped to the NVRAM device. A *persistent object* is an object that has a *name*; it is a linked list of the consecutive pages called a *cluster*. The cluster can be thought of as the extent in the extent-based file system.

HEAPO reserves a fixed address range in the virtual address space (`0x5FFEF800000`-`0x7FFEF800000`, total 32TByte) for the persistent heap. HEAPO uses the existing buddy algorithm (Linux 2.6.32) to manage NVRAM. It maintains page frame allocation information on the first page of NVRAM, which enables HEAPO to recover the state of NVRAM across power off and reboot.

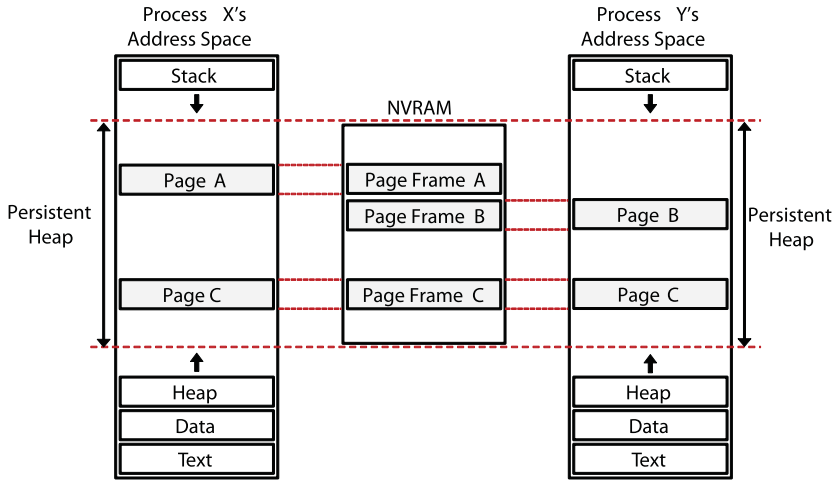

Fig. 3. Virtual address space layout and persistent heap in HEAPO.

Figure 3 illustrates virtual address organization of HEAPO. The persistent object is allocated to a virtual address when it is created. A persistent object is bound to the process' virtual address space when the process opens it; this is called *"attaching"* in HEAPO and it is done to distinguish a persistent object from open system call in the file system. There are three persistent objects in NVRAM—*A*, *B*, and *C*—each of which is allocated to a single page frame. Objects *A* and *B* are attached to process X's address space. Objects *B* and *C* are attached to process Y's address space.

3.3. Persistent Object

We designated the first cluster in the linked list as a special one, called the prime cluster, to contain the metadata of what is stored in the persistent object. The metadata for a persistent object contain a pointer to the prime cluster. A persistent object corresponds to a *file* in the legacy file system.

HEAPO defines primarily three metadata: metadata for the persistent heap, metadata for the persistent object, and name space data. Metadata for persistent heap and persistent objects correspond to superblock and inode, respectively, in the legacy file system. Kernel allocates a persistent region in its address space. Kernel maintains all metadata in the persistent region.

HEAPO also fully implements the notion of persistent variables. A persistent variable in HEAPO is the same as a static variable in C except that a persistent variable retains its data across the invocation of a process, while a static variable does so across the invocation of a function. In HEAPO, a persistent variable in NVRAM is deallocated when the process image that defines the persistent variable is removed from the file system.

Figure 4 shows a sample layout of a persistent heap and persistent objects. There are two objects, *A* and *B*, in the persistent heap; *A* and *B* contain a binary tree and a linked list, respectively, in themselves. Persistent object *A* is comprised of two clusters. The first cluster contains two nodes and the second cluster contains three nodes in the linked list.

3.4. Name Space

Creating, attaching, and expanding a persistent object require name space access. The file system adopts a linked list [Cao et al. 2005], a B-Tree [Sweeney et al. 1996; Mathur

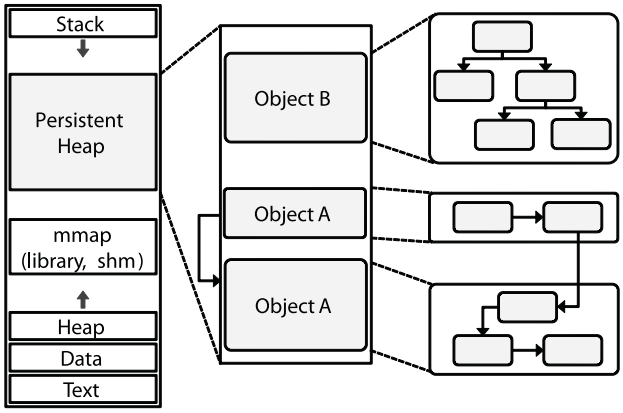

Fig. 4. Persistent object stores and objects in persistent heap.

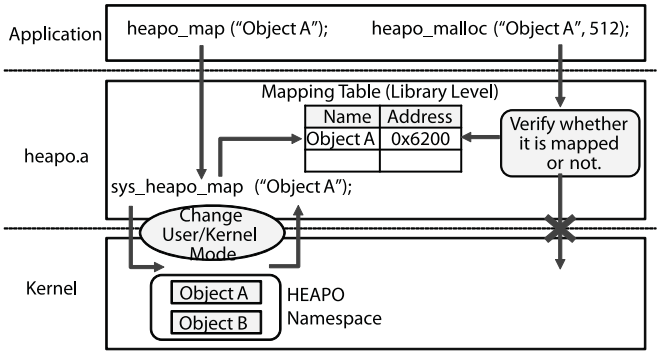

Fig. 5. Library level mapping table.

et al. 2007], and a hash table [Sun Microsystems 2004] to organize the on-disk file system name space and a hash table to organize in-memory file system name space [Lever and Alliance 2000]. When persistent heap is implemented on top of the mmap file system service, the persistent heap layer and the file system layer will maintain their own name space entries for the same persistent object. This redundancy causes significant performance degradation. We used unified organization for HEAPO name space and eliminated the name space redundancy in the legacy operating system. We used an efficient in-memory search structure, burst trie [Heinz et al. 2002], for the global name space and a hash table for the local name space cache.

Because the name space resides in the kernel address space, it may incur significant overhead to examine this data structure for each memory allocation crossing the protection boundary. In HEAPO, each process maintains the list of directory entries that have been attached to its address space. A similar analogy can be found in the *dentry* cache in the legacy file system which is used to expedite the directory accesses. The local name space is organized as a chaining hash table and synchronized with the global name space entry in a transactional manner. Figure 5 illustrates the relationship between the local name space at the user space and the name space at the kernel.

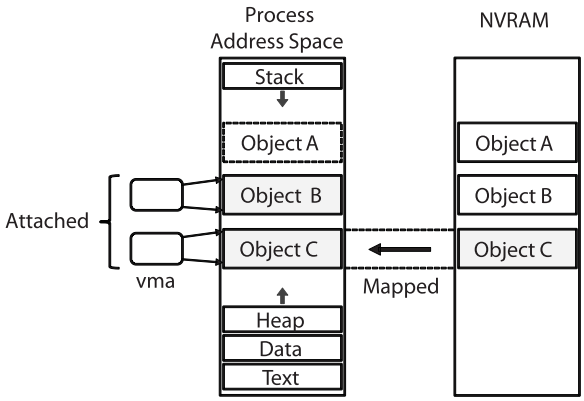

Fig. 6. Attaching before actual mapping.

3.5. Accessing a Persistent Object

Accessing an object store is composed of three major steps: (i) locating the object store, (ii) attaching the object, and (iii) mapping the actual pages (page fault handling). Each of these steps is performed by a kernel module. When a process needs to access a persistent object, it first needs to find the address of the respective object using the HEAPO naming system. HEAPO checks whether a given process has proper access rights to the given object. Each persistent object is protected by an access control mechanism. Figure 6 illustrates the details in the act of attaching and mapping an object store. When the process has proper access rights, HEAPO reserves the page table entries for the respective persistent object and creates the necessary metadata to manage the reserved page table addresses. We call this step "attaching the object store." When the process actually reads or writes the persistent object, a page fault occurs since the respective page table entry has not yet been set. The page fault handler for NVRAM maps the physical page from NVRAM to the respective page table entry. This process is called "mapping."

HEAPO is designed to harbor large size persistent objects, for example, in-memory key-value store [Chu 2008], XML-based document [Harter et al. 2012], etc. It is not fit for maintaining small size kernel objects, for example, inode, socket, file object. A persistent object is initially allocated a 4KByte page. Expanding a persistent object involves attaching one or more pages from the persistent heap to the virtual address space of the caller. The implementation of sharing is subject to the virtual memory architecture of the operating system. In our Linux-based implementation, each process has its own virtual address map. When the persistent object is shared by multiple processes, the virtual address spaces of all sharing processes are immediately updated so that the result becomes visible for the expansion and shrinkage of the persistent object. In allocating additional pages for expansion, the persistent heap management module finds the page frame(s) that are consecutive to the already allocated pages. If it fails, it scans the page table and moves on to the next available page frame. HEAPO uses a *lock* mechanism to synchronize the accesses to the shared object. Software transactional memory has been employed in former Mnemosyne and NV-heap.

The HEAPO library (`libheapo.a`) is a user-level library responsible for manipulating the space within the persistent object. An application requests a chunk of memory in the persistent object via `heapo_malloc()` and frees the chunk of memory via `heapo_free()`. For each persistent object, the HEAPO library maintains the linked lists of free chunks. HEAPO uses the same heap management algorithm of `glibc2.11.1`. There is an

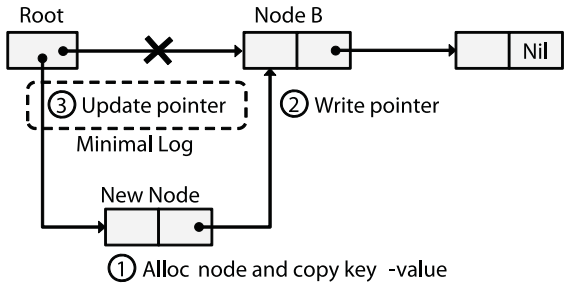

Fig. 7. Example of inserting a node to list data structure.

important difference between the HEAPO library and `glibc` in the way they manipulate the heap segment. For `glibc` implementation of a heap segment, each process maintains free space information of the heap segment in the local address space. In the case of Linux, this information resides at the `mmap` segment. In HEAPO, the persistent object can be shared among the processes. Free space information should persist and should be visible to all processes with proper access rights to that object. In HEAPO, we reserved 2KByte at the beginning of the persistent object to maintain its free space information.

Update consistency for a shared persistent object is guaranteed since the HEAPO library synchronizes the accesses to the shared object metadata with a lock. As each process maintains its own mapping information for the shared object, unmapping a persistent object for a process does not affect others' accessibility. A kernel module provides a `truncate` feature to shrink the persistent object, `sys_heapo_cluster_free()`. The truncate request is granted only when the respective cluster is entirely free.

In HEAPO, `heapo_free()` checks if the cluster to which it belongs is entirely free. If this holds, it returns the respective cluster to the kernel module and the persistent object is automatically shrunk. The virtual address space of the sharing processes is updated altogether.

3.6. Crash Recovery: Minimal Logging

For crash recovery, HEAPO adopts *undo-only logging*; to be precise, HEAPO does not know when to log and what to log. We developed a HEAPO-based key-value store; `KVLib`. `KVLib` implements its insert, delete, and update operations with *undo-only logging*. The logging mechanism of `KVLib` is elaborately crafted to minimize the overhead. In general, software transactional memory logs all the update operations enclosed within the `atomic` keyword in redo (or undo) the transaction for crash recovery as well as consistent update. When it comes to crash recovery, not all this information is needed. For every key-value operation, we carefully identified the *minimal* set of updates required to undo the transaction. We call this logging mechanism *minimal* logging.

Let us provide an example. Figure 7 illustrates the situation where a node is inserted between the root node and node *B*. Making this change consists of three steps: (i) allocating a node and storing the value, (ii) setting the pointer field of the newly allocated node to node *B*, and (iii) setting the pointer field of the root node to the newly allocated node. When these three steps are enclosed in the `atomic` key word, all three updates are logged. However, in this case, only the old value of the root pointer is needed to undo the transaction. In minimal logging, only the old value of the root pointer is solely recorded.

Due to the cache management algorithm of modern processors, the order in which an application issues updates may differ from the order in which the updates are reflected

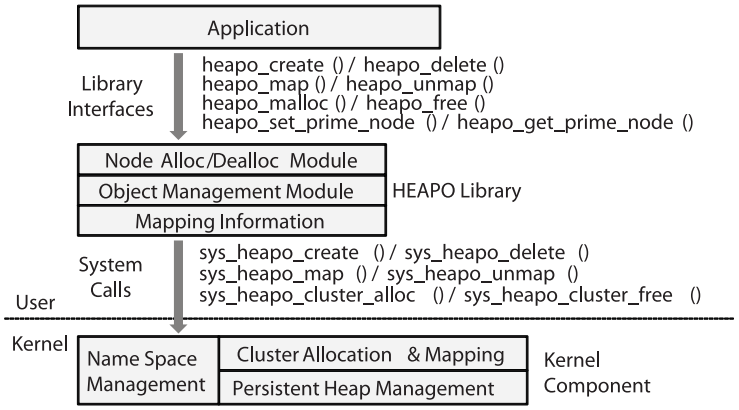

Fig. 8. HEAPO system organization and interfaces.

to memory. In HEAPO, a log record should be written to the storage device before an actual update is made. In current implementations, HEAPO uses `mfence` and `clflush` of $\times 86$ to guarantee that log records always reach the persistent memory before the respective updates are reflected to NVRAM.

4. IMPLEMENTATION

We implemented the prototype of HEAPO using Linux 2.6.32. The total implementation includes 2,800 lines of kernel code and 3,800 lines of user-level library code. Figure 8 summarizes the interface exported by HEAPO.

4.1. NVRAM Zone and Persistent Heap

Linux partitions physical memory into three regions: ZONE_DMA, ZONE_NORMAL, and ZONE_HIGHMEM [Wong et al. 2002]. We added ZONE_HEAPO to harbor the page frames of NVRAM. A page bitmap is used to denote the allocation status of the page frames of NVRAM. The page bitmap for NVRAM resides at the beginning of ZONE_HEAPO. There exists a separate page fault handler to map the persistent heap to NVRAM.

HEAPO adopts the segment organization of Linux OS to organize the persistent heap. The persistent heap is a collection of memory clusters. HEAPO organizes these memory clusters as a linked list as well as an rb-tree, as in Linux. Rb-tree and linked list representations are used to expedite the address search and the scan of the address space, respectively. The persistent heap is represented by a data structure called the `heapo_superblock`. The `heapo_superblock` contains the start and end of the persistent heap in the virtual address space, the number of allocated pages, the location of the name space for persistent objects, pointers to the root of the rb-tree, and a linked list for the persistent heap.

Each process has a local view of the persistent heap which is made up of a set of memory clusters of the persistent objects attached to the process' virtual address space. Figure 9 shows persistent heap implementation.

4.2. Manipulating Persistent Objects

A persistent object is represented by metadata called `heapo_descriptor`. Creating an object involves allocating an object descriptor (`heapo_descriptor`) and a memory cluster descriptor (`heapo_vm_area`). The page fault handler for the persistent heap allocates a page frame from ZONE_HEAPO to the created object. The object is then registered

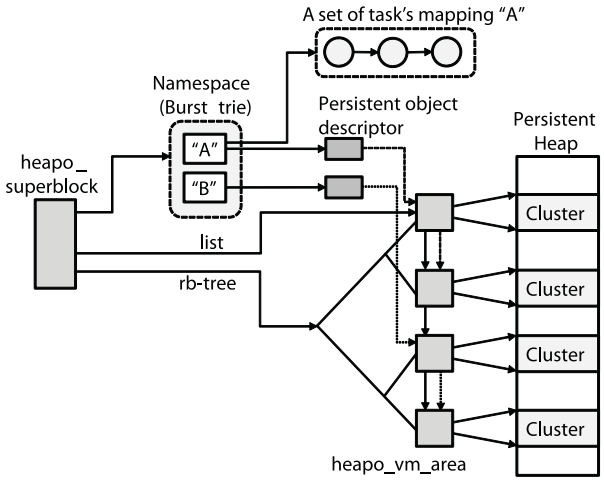

Fig. 9. Global persistent heap.

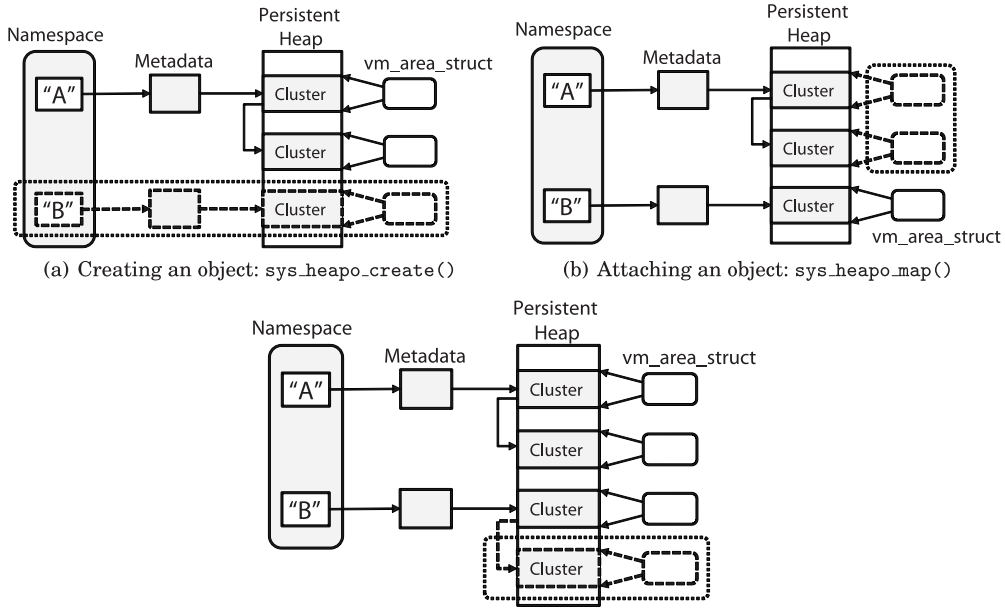

Fig. 10. Details of creating, attaching, and expanding an object.

at the name space and is attached to the creating process' virtual address space. Figure 10(a) illustrates creating a persistent object. Deleting an object is the inverse of the creating process.

Accessing an existing object consists of checking for accessibility, attaching an object, and mapping the pages. Once the access is granted, HEAPO initializes the memory cluster descriptors for the persistent object and inserts them within the process' virtual address space. The physical pages are mapped when the actual read or write operations

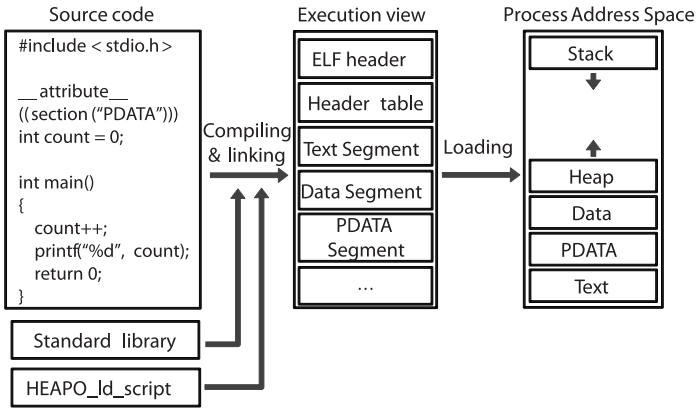

Fig. 11. Process of compiling, linking, and loading a persistent variable.

occur. HEAPO also adds the task ID to the set of processes that maps the persistent object. Figure 10(b) illustrates the details of attaching an object.

Expanding an object indicates allocating more virtual address space, that is, page table entry, to a given persistent object. The persistent heap of individual processes contains the virtual address information of the *mapped* object. One of the remarkable features of HEAPO is a shared update. When a process expands a persistent object, HEAPO updates the persistent heap of all processes that have the expanded persistent object attached to their own address space, as well as the requesting process. The kernel metadata for the persistent heap are updated as follows: (i) the HEAPO library requests more persistent heap area for a given kernel by calling `sys_heapo_cluster_alloc()`; (ii) the kernel allocates a memory cluster, creating the metadata for the memory cluster (the *heapo_vm_area*) and inserting it in the list of memory cluster descriptors for the respective persistent object; (iii) the newly created memory cluster is attached to each of the process' address space that has previously mapped the persistent object; and (iv) metadata for the new memory cluster, `vm_area_struct`, is created and inserted at the virtual address space. Figure 10(c) illustrates the expanding operation.

4.3. Persistent Variables

HEAPO has a *persistent variable* feature. The persistent variable is annotated with `__attribute__` prefix, for example, `__attribute__ ((section(''PDATA'')))  int i`. The compiler coalesces all variables with this prefix into the `PDATA` section and the linker relocates the `PDATA` section to a PDATA segment. The persistent variable feature of HEAPO consists of two main technical ingredients: the persistency mechanism and the loader.

In HEAPO, the life span of the persistent variable is aligned with that of the executable file in the filesystem. When the respective process image is deleted from the filesystem, the segment for persistent variables is deallocated. For this purpose, HEAPO maintains devices and inode numbers of the executable files that contain persistent variables. An entry of this set consists of an inode number and the set of physical pages for the persistent variables in the respective executable file. When a file is deleted, HEAPO searches the persistent file pool and deallocates all physical pages related to that file. The HEAPO loader loads the PDATA segment of the ELF executable into page frames from `ZONE_HEAPO`. Figure 11 shows the compiling, linking, and loading processes for the persistent variable.

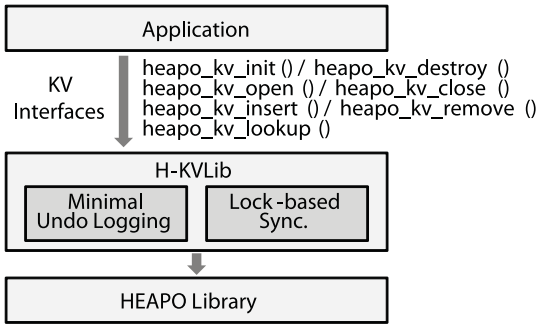

Fig. 12. H-KVLib organization and interfaces.

```
#include < heapo.h >

struct list {
      int data;
      struct list *next;
};

int main(void)
{
      struct list *head, *node;
      int i;

      if (heapo_create ("list") == 0)
            heapo_map ("list");

      head = heap_get_prime_node ("list");
      node = ( struct list *) heapo_malloc ("list", sizeof (struct list));
      node->data = rand();
      node->next = head;
      heapo_set_prime_node ("list", node);

      heapo_unmap ("list");
      return 0;
}
```

Fig. 13. Sample code.

5. H-KVLIB: KEY-VALUE LIBRARY FOR PERSISTENT HEAP

With HEAPO, we developed a key-value library, `H-KVLib`. `H-KVLib` provides two types of key-value store: B-Tree and hash table. The key-value library provides creation and deletion of the key-value store and insert/delete/search operation of the <key, value> pair. Management of the <key, value> pair becomes very efficient in HEAPO since the operations are performed in byte granularity without any metadata redundancy. In `H-KVLib`, a shared object is protected by a lock.

Figure 12 illustrates the organization and interfaces of the HEAPO-based key value library. Each key-value operation is protected by undo-only minimal logging. The concurrent accesses to the key-value store are synchronized using a lock. `heapo_kv_init()` creates the key-value store as the persistent object and maps it to the process' virtual address space. `heapo_kv_open()` maps the key-value store to the process's virtual address space. `heapo_kv_insert()` inserts the key-value pair in a transactional manner. `H-KVLib` adopts minimal logging. `heapo_kv_open()` checks if the respective key-value store is in the consistent state and triggers a recovery mechanism if necessary.

Figure 13 illustrates a sample code using HEAPO. This code creates the persistent object which has a linked list. The application creates a persistent object named "list," `heapo_create(''list'')`. If this object already exists, the application attaches it to

Table III. Average Name Space Lookup Latency

	HEAPO	EXT2	EXT4
Latency (sec)	0.9	4.4	6.5

its virtual address space, `heapo_map(''list'')`. The application obtains the starting address of the persistent object via `heapo_get_prime_node()`. If "list" is new, HEAPO reserves a fraction of the persistent heap for "list." At this phase, the persistent page has not been allocated yet. The persistent node is allocated to the persistent object by calling `heapo_malloc()`. Once the application is done with accessing the persistent object, it detaches the persistent object from its name space, `heapo_unmap()`. In this code, memory leak may occur if the system crashes after the call to `heapo_malloc` but before the memory is passed on `heapo_set_prime_node`. The cause for this problem is that it does not have any crash recovery handling feature in it. This problem can be resolved if we use `heapo_kv_insert()` of `H-KVLib`.

6. PERFORMANCE EXPERIMENTS

We implemented HEAPO on Linux 2.6.32 (64bit). All experiments were performed on an AMD Phenom X4 925 Processor (2.8GHz) and 12GB DDR3 DRAM. Berkeley DB (v. 5.1.29) [Olson et al. 1999], Memcachedb 1.2.0 [Chu 2008], `libevent` (v2.0.11), and `libmemcached` (v1.0.16) were used. For comparison with *mmap with ramdisk*, we formatted 10GByte ramdisk with EXT4. We enabled logging functionality of BDB to compare with minimal logging of HEAPO. All of the BDB file was stored in ramdisk.

The performance experiment encompasses six themes. First, we examined the lookup latency of name space. Second, we examined the overhead of creating, attaching, and expanding an object. Third, we examined, in detail, the overhead of metadata redundancy and of remapping involved in object expansion. Fourth, we developed a key-value library with HEAPO and compared the performance of database operations of the HEAPO-enabled key-value library with that of the Berkeley DB. Fifth, we examined the performance of Memcachedb to compare application-level performance. Sixth, we examined the lookup performance of the key-value library under absolute and relative address mode.

6.1. Name Space Lookup

We examined the name space lookup performance among HEAPO, EXT2, and EXT4. EXT2 and EXT4 adopt a linked list [Cao et al. 2005] and a B-Tree [Mathur et al. 2007], respectively, to organize the name space on disk. HEAPO adopts burst trie to organize in-memory structure of name space. We allocated 10GByte ramdisk and formatted it with the EXT2 and EXT4 to test the name space lookup performance. We inserted 1,000 entries into the directory and examined the time for directory lookup. Table III shows the average lookup latency of 1,000 name entries. HEAPO is 5.2× and 7.6× faster than EXT2 and EXT4, respectively, in searching the directory entry.

6.2. Metadata Operation

We examined the performance of metadata operation for different methods to realize persistent heap: HEAPO versus `mmap()` over `ramdisk`. We examined four operations: (i) create an object, (ii) attach an object (which corresponds to `opening` and `mapping` an object), (iii) expand an object, and (iv) shrink an object.

In HEAPO, creating a persistent object involves creating metadata and allocating page table entry. In `mmap()` over `ramdisk`, creating a persistent object entails creating a file and mmapping the file into the process' address space. File creation is a heavy process. It involves allocation and initialization of on-disk metadata, as well as its

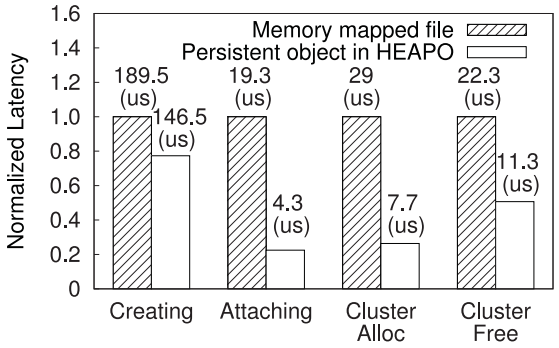

Fig. 14. Metadata operations: create, attach, expand (cluster allocation), and shrink (cluster free) in HEAPO and `mmap()` over `ramdisk`.

in-core counterpart, and updating the filesystem superblock and various bitmaps. For creating and attaching a persistent object, HEAPO is 1.3× and 4.5× faster than a memory-mapped file approach, respectively. In these tests, we assume that in-core metadata is already available in memory.

In `mmap()` over `ramdisk`, attaching a persistent object consists of opening the file that represents the persistent object and mapping the file into the process' virtual address space. Opening a file requires the operating system to read the file metadata from the filesystem and to initialize the in-core version of the metadata. In HEAPO, expansion of a persistent object involves allocating a virtual memory. In the memory-mapped file approach, expanding and shrinking an object eventually require the invocation of `write()` or `truncate()` system calls. A more serious issue is that in the memory-mapped file approach, a persistent object may fail to expand when the page frame that needs to be allocated for expansion has already been allocated.[1] If this happens, the respective persistent object needs to be unmapped and remapped to a different area in the virtual address space, and all the references need to be recomputed. Expanding and shrinking a persistent object are 3.8× and 2× faster, respectively, in HEAPO than in `mmap()` over `ramdisk`. Figure 14 summarizes the performance result of four operations: create, attach, expand, and shrink.

When an in-core copy of the metadata does not exist, the memory-mapped file approach first initializes the metadata and attaches it to the virtual address space. We examined the performance of attach and expand (allocate cluster) operations which accompany creating in-core metadata. Figure 15 illustrates the results. Attaching and expanding a persistent object are 12.7× and 4.8× faster, respectively, in HEAPO than in the memory-mapped file approach when in-core metadata needs to be created.

6.3. `H-KVLib` Versus Berkeley DB

We compared the performance of key-value operations of `H-KVLib` with that of BerkeleyDB. Berkeley DB uses files to manage the key-value store in a persistent manner. We inserted 1 million key-value pairs. The key and the value are 16 and 512 bytes, respectively. Figure 16 illustrates the results of the experiment. In a B-Tree-based key-value store, the `H-KVLib` is 1.4× to 3.6× faster than BerkeleyDB. In a hash table-based key-value store, the HEAPO-based key-value library is 5.9× to 18.1× faster than BerkeleyDB.

We examined the size of logs written for key-value operations in `KVlib` and BerkeleyDB. Table IV illustrates the log size of HEAPO and Berkeley DB for insert, delete,

[1]In `ramdisk`, a file needs to be allocated consecutive pages.

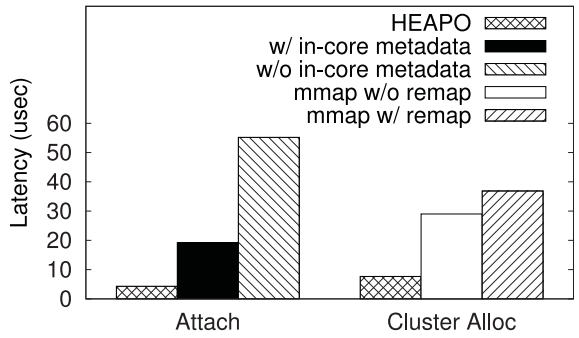

Fig. 15. Additional overhead of memory mapped file.

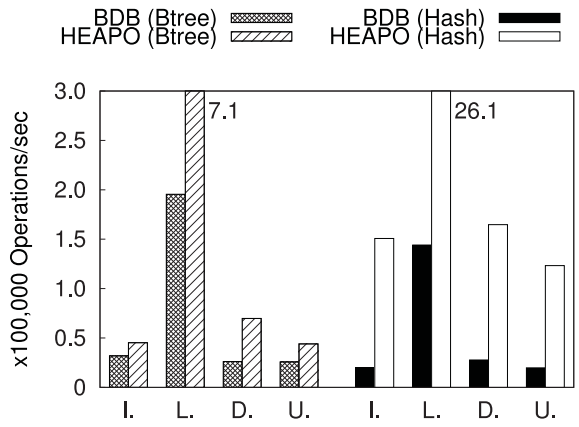

Fig. 16. Performance comparison between BDB and HEAPO. "I.," "L.," "D.," and "U." denote insert, lookup, delete, and update, respectively.

Table IV. Log Size (MByte) of BDB and HEAPO

	B-Tree		Hash Table	
	BDB	HEAPO	BDB	HEAPO
Insert	1,510	150.8	4,410	7.6
Delete	760	86.5	1,440	15.3
Update	1,500	150.4	3,960	11.4

and update operations. Berkeley DB records the state of the database before and after the update operation. Unlike Berkeley DB that records pre and post state, HEAPO records merely the old value of 8byte pointer fields. In B-Tree, Berkeley DB generates 8.8× to 10× more logs than HEAPO does. In the hash table, Berkeley DB generates 94.4× to 578× more logs than HEAPO does.

6.4. Memcachedb Performance

Memcachedb is a key-value store using BerkeleyDB (BDB). We modified Memcachedb to adopt `H-KVLib` and compared the performance of Memcachedb under two different key-value storage engines. Figure 17 illustrates the result. We inserted one million key value pairs in the key-value store and measured the performance. We inserted one delete operation for every four insert operations. We examined the BDB performance for both synchronous and asynchronous operations. Memcachedb server and client are

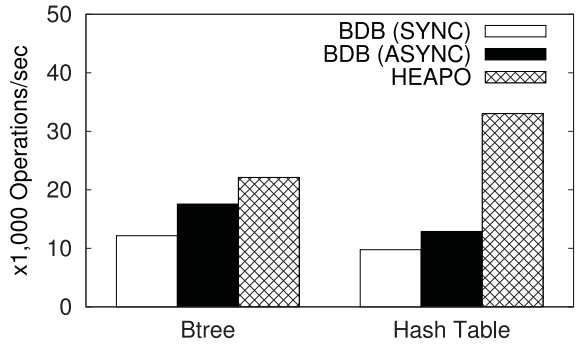

Fig. 17. Memcached with BDB versus Memcached with HEAPO.

on the same machine to minimize network interference. The sizes of the key and value are 16 and 512byte, respectively. In the B-Tree and hash table, HEAPO is 1.8× and 3.4× faster, respectively, than BDB. The performance gain of HEAPO becomes more significant in the hash table-based key-value store.

6.5. Static Binding versus Dynamic Binding

We examined the overhead of address computation in static binding and dynamic binding. HEAPO adopts static binding and, therefore, it does not entail the relocation and the pointer resolution overhead. Dynamic binding [Coburn et al. 2011] incurs address relocation overhead. We performed a lookup operation to B-tree and to a linked list with 5 million and 50,000 key-value pairs, respectively. In traversing the linked list, static binding brings 8.4% performance gain against dynamic binding.

7. RELATED WORK

The notion of persistency was proposed by Atkinson in 1981 [Dearle et al. 1992]. The idea of managing both primary storage (RAM) and secondary storage (disk) as a single unified storage component has been discussed for more than three decades under the term single-level store [Moss 1990; Shekita and Zwilling 1990].

NV-Heaps [Coburn et al. 2011] guarantees referential integrity between persistent and nonpersistent objects. Mnemosyne [Volos et al. 2011] provides consistency of data through software transactional memory. Mnemosyne supports word-based transactions, while NV-Heaps supports node-based transactions. Consistent and Durable Data Structures (CDDS) [Venkataraman et al. 2011] provides versioning instead of logging to support consistency of data structures such as B-Tree. HEAPO uses minimal undo logging for atomicity and durability and lock primitive for consistency. SoftPM [Guerra et al. 2012] constructs a persistent object store without including NVRAM in the system. It minimizes the number of lines of modified code required for persistency by allowing users to use a legacy programming interface. However, in SoftPM, explicit conversion entails extra memory copy overhead. Moraru et al. [2011] proposed protecting persistent data from wearout or corruption from NVRAM by using a virtual memory protection mechanism and a cache line counter. NVMalloc [Wang et al. 2012] proposes a library-level system for persistency of data that can operate on nonvolatile memory devices, such as solid-state disks (SSDs).

BPFS [Condit et al. 2009] provides atomic, fine-grained updates to NVRAM through short-circuit shadow paging. SCMFS [Wu and Reddy 2011] locates filesystem space into kernel space and then manages it by reusing the memory management module of the operating system for main memory. FRASH [Jung et al. 2010], Conquest [Wang et al. 2006], and LiFS [Ames et al. 2006] store metadata of the file system into NVRAM, thus

reducing the access time to in-memory metadata. Object-based FS [Kang et al. 2011] is a filesystem that accesses storage at the granularity of objects, not blocks. Volos et al. [Volos and Swift 2011] proposed a library-level filesystem that allows the user to access files on NVRAM in user mode. We carefully argue that the file system-based approach fits a relatively slower speed NVRAM device, for example, PCRAM owing to its system call overhead and redundant metadata. These approaches hardly fit our target device which exhibits DRAM-like access latency.

WSP [Narayanan and Hodson 2012] makes a system that facilitates NVRAM as main memory persistent by using residual energy through a flush-on-fail mechanism. With the special hardware used by WSP, the system does not require any consistency mechanism such as logging or transactional memory. HEAPO uses a software-based approach, `clflush` and `mfence`, for ordering guarantee.

8. CONCLUSION

The objective of this work is to develop a software layer that allows NVRAM to be seamlessly integrated with the existing computer system hierarchy and that can fully exploit the physical characteristics of the new memory device. One of the important constraints of the new software layer is that it should be *native*. HEAPO is designed to fully exploit the byte addressability and nonvolatility of the new memory device. The persistency mechanism of HEAPO does not rely on existing filesystem service. HEAPO defines its own persistent heap layout, persistent object, and name space structure, along with the comprehensive set of interfaces that manages them. We show that the HEAPO native persistent heap management module surpasses existing methods in performance. HEAPO offers an easy and versatile method for writing applications that fully exploit the physical characteristics of byte-addressable NVRAM.

REFERENCES

A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. 2011. Onyx: A prototype phase change memory storage array. In *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'11)*.

S. Ames, N. Bobb, K. M. Greenan, O. S. Hofmann, M. W. Storer, C. Maltzahn, E. L. Miller, and S. A. Brandt. 2006. LiFS: An attribute-rich file system for storage class memories. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST'06)*.

Apache. 2009. Cassandra. Retrieved from http://cassandra.apache.org/.

Anirudh Badam and Vivek S. Pai. 2011. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proc. of USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*.

M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. 2005. State of the art: Where we are with the ext3 filesystem. In *Proc. of the Ottawa Linux Symposium (OLS'05)*.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. (TOCS)* 26, 2 (2008).

Kristina Chodorow. 2010. Introduction to MongoDB. In *Free and Open Source Software Developers European Meeting (FOSDEM'10)*.

S. Chu. 2008. Memcachedb. Retrieved from http://memcachedb.org.

J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. 2011. 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'11)*.

J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'09)*.

A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas. 1992. An examination of operating system support for persistent object systems. In *Proc. of Hawaii International Conference on System Sciences (HICSS'02)*.

Diblotechnology. 2013. Memory Channel Storage. Retrieved from http://www.diablo-technologies.com/products/mcs.html.

R. F. Freitas and W. W. Wilcke. 2008. Storage-class memory: The next storage system technology. *IBM J. Res. Dev.* 52, 4.5 (2008), 439–447.

J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. 2012. Software persistent memory. In *Proc. of USENIX Annual Technical Conference (ATC)*.

T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2012. A file is not a file: Understanding the I/O behavior of Apple desktop applications. *ACM Trans. Comput. Syst. (TOCS)* 30, 3 (2012).

S. Heinz, J. Zobel, and H. E. Williams. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. (TOIS)* 20, 2 (2002), 192–223.

I. Jantunen, J. Hämäläinen, T. Korhonen, H. Kaaja, J. Jantunen, and S. Boldyrev. 2010. System architecture for mobile-phone-readable RF memory tags. In *Proc. of UBICOMM*.

J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. 2010. FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Trans. Storage (TOS)* 6, 1 (2010).

Y. M. Kang, H. J. Joo, J. H. Park, S. K. Kang, J. H. Kim, S. G. Oh, H. S. Kim, Y. J. Kang, J. Y. Jung, D. Y. Choi, and others. 2006. World smallest 0.34/spl mu/m~ COB cell 1T1C 64Mb FRAM with new sensing architecture and highly reliable MOCVD PZT integration technology. In *Proc. of Symposium on VLSI Technology*.

Y. Kang, J. Yang, and E. L. Miller. 2011. Object-based SCM: An efficient interface for storage class memories. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST'11)*.

S. Kim and C. H. Lam. 2012. Transition of memory technologies. In *Proc. 2012 International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA),* IEEE.

M. H. Kryder and C. S. Kim. 2009. After hard drives? What comes next? *IEEE Trans. Magn.* 45, 10 (2009), 3406–3413.

C. Lever and Sun-Netscape Alliance. 2000. *Linux Kernel Hash Table Behavior: Analysis and Improvements*. Technical Report 00-1. Center for Information Technology Integration, University of Michigan, Ann Arbor, MI (2000).

Z. Liu, B. Wang, P. Carpenter, D. Li, J. S. Vetter, and W. Yu. 2012. PCM-based durable write cache for fast disk I/O. In *Proc. of International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'12)*.

A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. 2007. The new ext4 file system: Current status and future plans. In *Proc. of Ottawa Linux Symposium (OLS'07)*.

I. Moraru, D. G. Andersen, M. Kaminsky, N. Binkert, N. Tolia, R. Munz, and P. Ranganathan. 2011. Persistent, protected and cached: Building blocks for main memory data stores. In *Proc. of Annual Parallel Data Lab Workshop and Retreat*.

J. E. B. Moss. 1990. Garbage collecting persistent object stores. In *Proc. Workshop on Garbage Collection (GC'90)*.

Microsoft Corp. msdn. 2010. Ramdisk. Retrieved from msdn.microsoft.com/en-us/library/ff544551(VS.85).aspx.

D. Narayanan and O. Hodson. 2012. Whole-system persistence. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*.

M. A. Olson, K. Bostic, and M. Seltzer. 1999. Berkeley DB. In *Proc. of the FREENIX Track: 1999 USENIX Annual Technical Conference*.

T. Perez. 2012. *Evaluation of System-Level Impacts of A Persistent Main Memory Architecture*. Ph.D. Dissertation. Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, Brazil.

M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *ACM SIGARCH Comput. Archit. News*, 37, 3 (June 2009). 24–33.

Objective Analysis Semiconductor Market Research. 2009. PCM Becomes a Reality: Memory Design will Never be the Same. Retrieved from http://www.objective-analysis.com/uploads/2009-08-03_Objective_Analysis_PCM_White_Paper.pdf.

E. Shekita and M. Zwilling. 1990. Cricket: A mapped, persistent object store. In *Proc. Workshop on the Persistent Object Systems*.

Sun Microsystems Inc. 2004. Solaris ZFS file storage solution. Solaris 10 Data Sheets.

A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. 1996. Scalability in the XFS file system. In *Proc. of the USENIX 1996 Technical Conference*.

S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of USENIX Conference on File and Storage Technologies (FAST'11)*.

Vikingtechnology. 2013. NV-DIMM. Retrieved from http://www.vikingtechnology.com/nvdimm-technology.

H. Volos and M. Swift. 2011. Storage systems for storage-class memory. In *Proc. of Annual Non-Volatile Memories Workshop (NVMW'11)*.

H. Volos, A. J. Tack, and M. M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

A. I. A. Wang, G. Kuenning, P. Reiher, and G. Popek. 2006. The conquest file system: Better performance through a disk/persistent-RAM hybrid design. *ACM Trans. Storage (TOS)* 2, 3 (Aug. 2006), 309–348.

C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. 2012. NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS'12)*.

K. L. Wang and P. Khalili Amiri. 2012. Nonvolatile spintronics: Perspectives on instant-on nonvolatile nanoelectronic systems. *SPIN*, 2, 2 (June 2012).

P. W. Y. Wong, B. Pulavarty, S. Nagar, J. Morgan, J. Lahr, B. Hartner, H. Franke, and S. Bhattacharya. 2002. Improving linux block I/O for enterprise workloads. In *Proc. of Ottawa Linux Symposium*.

X. Wu and A. L. N. Reddy. 2011. SCMFS: A file system for storage class memory. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*.

Jung H. Yoon, Hillery C. Hunter, and Gary A. Tressler. 2013. Flash & DRAM Si scaling challenges, emerging non-volatile memory technology enablement—Implications to enterprise storage and server compute systems. In *Flash Memory Summit.*

Received February 2014; accepted May 2014