PAPER

Fast Persistent Heap Based on Non-Volatile Memory

Wenzhe ZHANG[†], **Kai LU**[†a)], **Xiaoping WANG**[†], ***Nonmembers***, ***and*** **Jie JIAN**[††], ***Student Member***

SUMMARY New volatile memory (e.g. Phase Change Memroy) presents fast access, large capacity, byte-addressable, and non-volatility features. These features will bring impacts on the design of current software system. It has become a hot research topic of how to manage it and provide what kind of interface for upper application to use it. This paper proposes FP-Heap. FP-Heap supports direct access to non-volatile memory through a persistent heap interface. With FP-Heap, traditional persistent object systems can benefit directly from the byte-persistency of non-volatile memory. FP-Heap extends current virtual memory manager (VMM) to manage non-volatile memory and maintain a persistent mapping relationship. Also, FP-Heap offers a lightweight transaction mechanism to support atomic update of persistent data, a simple namespace to facilitate data indexing, and a basic access control mechanism to support data sharing. Compared with previous work Mnemosyne, FP-Heap achieves higher performance by its customized VMM and optimized transaction mechanism.

key words: *non-volatile memory, virtual memory manager, direct access*

1. Introduction

Non-volatile memory technologies, represented by Phase Change Memory (PCM), are maturing fast in recent years. Non-volatile memory delivers features such as fast access, large capacity, byte-addressable, and non-volatility. These appealing features will break the long premise of the two-level storage architecture that the main memory is small, fast, and volatile and the secondary storage is large, slow, and non-volatile. Hence, it has become a hot research topic of how to use it, i.e. how to manage it and provide what kind of interface for upper applications to access it.

In current operating system, the memory management system (Virtual Memory Manager or VMM) is designed for volatile DRAM and offers no support for non-volatility, while the file system is designed for block devices and cannot exploit the byte-addressability of non-volatile memory. Hence there are generally two trends to manage non-volatile memory [1]:

- Extending VMM to manage non-volatile memory. Exposing it directly to upper applications like DRAM and offering support for non-volatility.
- Optimizing traditional file system according to the byte-addressable feature of non-volatile memory.

We adopt the first strategy and expose non-volatile memory to upper applications through a persistent heap interface (e.g. FP_malloc). Traditional persistent data structures are backed-up by files through memory map (e.g. mmap) [2]–[4]. The persistent data needs to undergo several procedures (e.g. transformation and copy) to reside in non-volatile media. Now the byte persistency provided by non-volatile memory can allow persistent data to reside directly in non-volatile memory. Meanwhile, since the non-volatile memory has a large capacity, it can be regarded as an objects storage. For the data structures stored in it, the non-volatile memory may act as both working memory and residing storage. Previous work Mnemosyne [5] and NV-heaps [6] proposed similar concepts. However, they just offer direct access and other language level features (e.g. safe pointer) through user mode library. To manage non-volatile memory at a low level (i.e. operating system kernel), they rely on file system: Mnemosyne maps persistent regions to files to maintain and recover the memory mapping relationship; NV-heaps sets up a ramdisk on non-volatile memory and builds persistent heap based on it through memory map. As discussed before, file system is designed for block devices and may introduce unnecessary overhead when managing non-volatile memory.

This paper introduces FP-Heap: a fast persistent heap based on non-volatile memory. FP-Heap mainly focuses on the issues below:

- Extend current VMM to manage non-volatile memory and maintain a persistent mapping relationship.
- Offer an efficient transaction mechanism for atomic update of persistent data to ensure data consistency.
- Offer simple namespace and access control to facilitate data indexing and sharing.

FP-Heap offers familiar interfaces for upper applications to use non-volatile memory (e.g. FP_malloc, FP_free). A simple scenario is: an application can allocate a region of non-volatile memory and directly access it. The allocated non-volatile memory can be accessed again on next run as long as it has not been freed.

The rest of this paper is organized as follows: We introduce the background of non-volatile memory in Sect. 2.

Manuscript received October 19, 2016.
Manuscript revised December 27, 2016.
Manuscript publicized February 1, 2017.

[†]The authors are with Science and Technology on Parallel and Distributed Processing Laboratory, Collaborative, Innovation Center of High Performance Computing, State Key Laboratory of High-end Server & Storage Technology, College of Computer, National University of Defense Technology, ChangSha, 410072, China.

[††]The author is with College of Computer, National University of Defense Technology, ChangSha, 410072, China.

a) E-mail: lukainudt@163.com

DOI: 10.1587/transinf.2016EDP7429

Table 1 Comparison between PCM and DRAM.

	Read	Write	Endurance	Density
DRAM	60ns	60ns	10^{16}	$7F^2$
PCM	85ns	100-500ns	10^{7}	$4F^2$

Section 3 and Sect. 4 give our design and implementation of FP-Heap. We evaluate FP-Heap in Sect. 5. Related work is discussed in Sect. 6 and we conclude in Sect. 7.

2. Non-Volatile Memory

Non-volatile Memory represents a kind of memory technologies that offer fast access, byte-addressable, and non-volatile features. They can be connected to the memory bus and accessed by CPU through traditional load and store instructions. More importantly, they can hold data across system reboot naturally. Recent memory technologies include Phase Change Memory (PCM) [7], spin-torque-transfer RAM (STT-RAM) [8], and meristors [9]. Among them, Phase Change Memory (PCM) is the most developed and promising device that may act as an alternative to DRAM in the future.

Table 1 gives a comparison between PCM and DRAM on some key properties. The cell size of PCM is $4F^2$, about 60% of DRAM [10], meaning a better scalability than DRAM. Moreover, the Multi-level-cell (MLC) technology can enable a PCM cell to store more than 2 bits of data, further increases its capacity. PCM also has some drawbacks such as low write speed and poor endurance. There are many studies at architectural level to address these problems [11]–[14]. In this paper we assume the wear-leveling is done in the memory controller.

(1) Proposed Architecture

Like previous work [5], we assume non-volatile memory is placed beside DRAM to form a hybrid main memory system. Non-volatile memory shares the same physical address with DRAM and can be accessed directly by CPU through load and store instructions.

(2) Assumptions

As non-volatile memory is not available to us now, we make several assumptions like previous work [5]. Firstly, the hardware should support an atomic write of 64-bits. Thus a singly flying write may finish or have no affects when system crashes. Secondly, there should be a mechanism to block execution until previous writes reach non-volatile memory, such as fsync in file system.

3. Design of FP-Heap

The goal of FP-Heap is to offer familiar interface to access non-volatile memory along with achieving an efficient management of non-volatile memory.

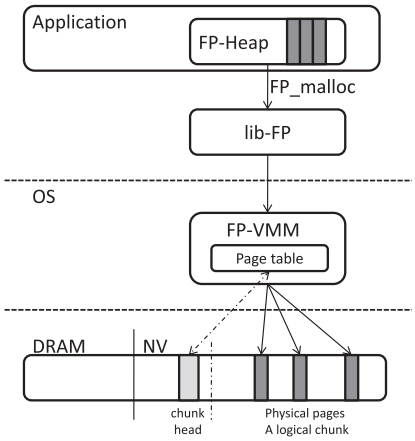


Fig. 1 Design overview of FP-Heap.

3.1 Design Overview

The design overview of FP-Heap is shown in Fig. 1. Since the dynamic allocation from heap is the most familiar concept to programmers to access memory, our work focuses on providing the FP_malloc and FP_free interfaces to upper applications. Every process which links to our lib-FP will have a persistent heap to serve the FP_malloc and FP_free.

Like other memory allocators [15], our heap is chuck (or superblock) based: any process or thread who wants to allocate memory will firstly get a fix sized virtual memory region (called chunk or superblock) and allocates memory from it. We organize the non-volatile memory in a chunk based way: we record the memory mapping at the granularity of a chunk (shown in Fig. 1). The information of mapping relationship and the occupied physical pages of a chunk is stored in the corresponding chunk head in non-volatile memory as shown in Fig. 1.

Moreover, we provide support for managing the non-volatility such as achieving data indexing, access control, and ensuring data consistency. For data indexing, every allocated data structure could be given a unique name to help locate it on next run or in other processes. For access control, every persistent heap will be tagged with the user name of who firstly creates the heap. Only this user can change the accessing mode of the heap. For avoiding data corruption, we provide a transaction mechanism to guarantee the consistency of data. We also rely on it to support parallel access to non-volatile memory.

3.2 Organization of Non-Volatile Memory

The non-volatile memory is organized into two parts: a data part and a meta-data part (shown in Fig. 2). The data part contains normal physical pages that are mapped to chunks. The meta-data part consists of a list of chunk head, a log zone used to protect the meta-data, and a global lock used to tune chunk allocations from different processes. Every chunk head mainly stores a page table for this chunk. The

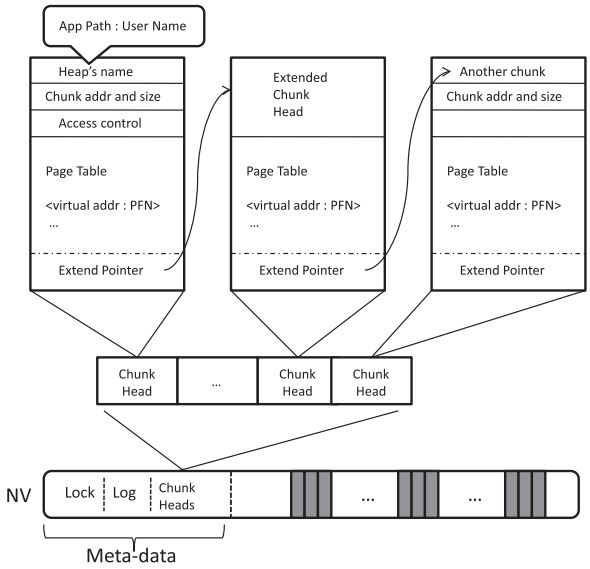


Fig. 2 Underlying organization of non-volatile memory.

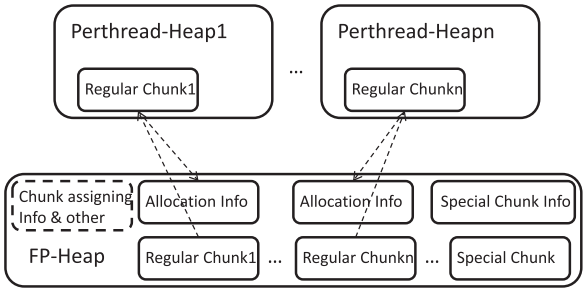


Fig. 3 Organization of persistent heap.

chunk address and size indicate the virtual memory region of the chunk. The chunk size is usually set fixed. If an upper application allocates an object that is more than a normal chunk, then it will get a larger special sized chunk. If the chunk is the first chunk of a heap, the chunk head will also store the information of the heap such as the name and the access control information. Every chunk head initially occupies one physical page and can be extended to another page if there are too many mapping information to record in the page table. If a heap contains several chunks, the chunk heads will be linked to facilitate searching (as shown in Fig. 2).

(3) Protection from stray writes

We set apart the data part and the meta-data part at the physical address level. The data part can be mapped into user processes' spaces to be accessed directly, while the meta-data part is invisible to users and could only be accessed by kernel code. In this way we can guarantee the meta-data will not be modified unintentionally. The chunk heads store important information of free and occupied physical pages. Any stray write will possibly cause a memory leak which is permanent.

3.3 Non-Volatile Memory Allocation

Every process links to Lib-FP will automatically have a persistent heap to serve request for non-volatile memory. Allocation for memory is chunk-based (describe below). The allocated chunk's information will be recorded in the chunk head. The allocation of the real physical pages and the establishment of the mapping relationship can be delayed to the first touch of every virtual page just like what the page fault mechanism operates DRAM.

We build our heap based on Hoard [15], a popular memory allocator. Hoard asks memory regions from operating system in the granularity of a superblock or chunk. Here we record the chunk's information in the non-volatile memory. These chunks are all fix-sized (65536) and are regarded as the regular chunk in FP-Heap. If a process asks for a large object, Hoard will directly ask for a special chunk from operating system which is exactly of that size. As shown in Fig. 3, the regular chunks can be assigned to each thread to facilitate multi-thread allocation for small objects while the special chunks are kept in global heap. Like Mnemosyne [5], we store the allocation information of each chunk in the global heap. The global heap itself (mainly the control information), is also regarded as a special chunk and is automatically created at process starts. The chunk head of this special chunk is always the first of this heap in the meta-data shown in Fig. 2.

Hoard also keeps a map storing which chunk is assigned to which thread. We do not store this information in non-volatile memory. We just re-assign the chunks to threads after system reboots in a simple way: after system reboots, if a thread firstly accesses a chunk or call free to a chunk, we assign the chunk to the thread. In other cases we just do random assigning.

(4) Serving chunk allocation from different processes

Process may conflict with each other when allocating chunks from non-volatile memory. We adopt a simple lock to tune the allocations. As shown in Fig. 2, the lock is stored in non-volatile memory and should be reset after system reboot. Meanwhile, a process may crash after getting the lock, which will block the other waiting processes permanently. To solve this problem, we introduce a simple timeout mechanism. Generally, the allocating of new chunk should be finished in a short time thus the timeout mechanism will work fine. This is mainly because we just need to find a new page in meta-data to store the chunk information and it only spends linear time to finish this.

3.4 Persistent Heap Management

Any process that links to lib-FP will automatically have its own persistent heap. It can allocate space from this persistent heap through the interface FP_malloc and then access it just like traditional DRAM. The heap will automatically be mapped into the program's space on next run and the objects which have not been freed can be directly re-accessed again.

This is done by scanning the chunk heads at the beginning of the program to find the heap and map it into the process's space. The whole mapping mechanism is transparent to programmers so that they can just focus on allocating and using non-volatile memory. This is the most common scenario of using persistent heap.

However, there may be some trouble of locating the objects on the next run if the programmers do not implement a root pointer to index all objects in the heap. In order to facilitate locating the objects in the persistent heap, we offer a mechanism in which programmers can give the object a name when allocating it using FP_malloc and can use this name to query the address of the object. The name occupies fix-sized region stored just before the allocated object. Compared with file system, our name space is very simple. Since FP-Heap mainly focuses on supporting direct access, our scheme is to offer programmers the flexibility and basic support to implement their own data indexing mechanism.

The new created persistent heap will automatically be assigned a unique name which consists two parts: the absolute path of the program and the user name who executes this program (as shown in Fig. 2). If another user executes this program or this user executes another program, a new heap with a different name will be created. We do not allow programs to create new heaps explicitly because (1) every process executed by a specific user will automatically get its own heap which is enough for it to access non-volatile memory and (2) explicit creating new heaps will lead us to a situation of managing complex namespace which is like files in file systems and is costly [16].

To achieve data sharing, we offer a mechanism to let a process access the heap of another process. Any process could use the name to explicitly map the heaps of other processes into its space. If a process explicitly maps another existing persistent heap, its current persistent heap will be unmapped automatically. This is for two reasons. (1) A process should only have one persistent heap so that it will not be confused when calling FP_malloc. (2) The virtual addresses of different persistent heaps may conflict with each other. Thus we just allow a process to have one persistent heap mapped in its space at any time.

(5) Access control

We achieved a basic access control mechanism based on the recorded user name. Only the user of a heap can change the accessing mode of the whole heap. There are two accessing modes which are read-only and writable. Changing the accessing mode is like what traditional mprotect do. The accessing mode is recorded in the chunk head as shown in Fig. 2. Every time when the heap is being mapped into a process's space, we will use the accessing mode in the chunk head to set the page table in operating system kernel. Moreover, if the heap is read-only, then other users can only map it into process space to read data from it. FP_malloc or FP_free on the heap is not allowed. If the heap is writable, then other users can modify the data in the heap and FP_malloc and FP_free on the heap.

```
Process 1:                                  Process 2:
…                                           …
Int * a =                                   PTM_begin();
FP_malloc(size_of(int), “name”);            Int * a =
…                                           FP_malloc(size_of(int), “name”);
*a = value;                                 PTM_write(a, value);
…                                           …
b = *a;                                     b = PTM_read(a);
...                                         PTM_end();
            (a)                             ...          (b)
```

Fig. 4 Sample code of using transaction interface.

Here we just designed a basic controlling mechanism. Sharing heaps among processes or users is like sharing files among users. There may be users deleting writable shared heap, which can cause data loss. It is like multiple users modifying a shared file. However, the only security problem is the loss of file content (heap data here). The meta data is guaranteed to be safe because they are modified by kernel code. More sophisticated mechanism will be our future work.

3.5 Data Consistency

Accidental system crashes or power cut would leave the data in non-volatile memory in an inconsistent state. We provide a transaction mechanism to avoid this problem. As shown in Fig. 4, a process can allocate a piece of non-volatile memory and access it directly as Fig. 4 (a) shows, while in this way it may risk leaving the data corrupted when system crashes. Alternatively, it can use a transactional interface to access the data like Fig. 4 (b) shows. In this way all the modifications will be logged and the unfinished transactions can be rolled back to help recover the persistent data after crashes.

We adopt redo logging where new values are written to the log and committed to the destinations on committing. Compared with undo logging where old values are copied to log and new values are written to memory, redo logging has several advantages. In undo logging, every write will result into two writes: copying old value to the log and writing new value. The two writes should be ordered using memory fence or write-through mechanism to avoid losing of the old value. While in redo logging, we just need to issue two memory fences at transaction commit.

Our transaction mechanism can be divided into two parts:

(1) For memory allocations (FP_malloc or FP_free or page fault, which may modify our persistent heap or the persistent mapping relationship), as we are based on Hoard which has already implemented fine-grained lock to tune parallel access, we just add logging to ensure data consistency. The corresponding data includes the mapping information in meta-data (e.g. the persistent page table) and the memory allocation information in global heap. The mapping information is logged in meta-data and the heap organization information is logged in heap. We only submit

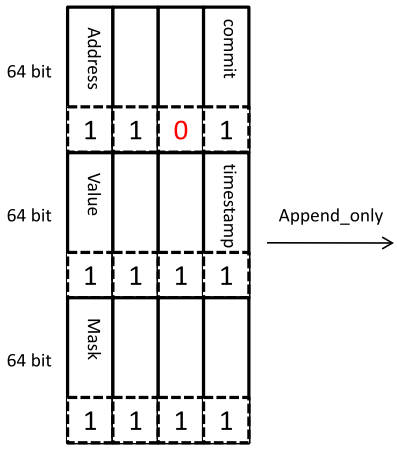


Fig. 5 Optimized log.

these two logs when we meet PTM_end. Thus in Fig. 4 (b), if system crashes, the FP_malloc will have no effects.

(2) For user accesses, we offers a transaction mechanism based on Tiny-STM [17], [18] to support atomic update.

We modify the Tiny-STM and put the write set of each thread in non-volatile memory as a log. Traditionally, every commit will need two fences: flush the write set to destinations and wait until it finishes, then write a commit flag and wait for it finishes. Here we borrow the idea of Mnemosyne's raw word log (RAWL) [5] to reduce a fence on every commit. We modify the write set in Tiny-STM to be append-only. Thus a later write to the same address in a transaction would appear later in the log instead of overwriting the former one. As shown in Fig. 5, every slot of the log contains 3*64 bits of address, value, and mask. The last bits of them are set to 1 to indicate this is a valid log slot. The last log slot contains commit flag and the timestamp. The timestamp is used to guarantee the redoing order of the committing during recovery. The mask field is set to 0 to indicate the last log slot as the mask cannot be 0 (As we use the mask to indicate the size of the writing value, it is impossible that in a log slot the mask field is 0. Because in this case we are not writing any data thus we do not need a log slot to record this.). In this way we can just issue one fence: if a system crash happens during commit, we can check whether all the writes reaches the log by testing whether all the log slots are tagged '1'. Figure shows that not all writes reaches the log and the transaction should be aborted.

For the truncated bits in address, value, and mask, we use other un-used bits in mask and address to recover them. For example, the mask will always be 0xFF00000000000000 (indicating one byte of written value), 0xFFFF000000000000, (2 bytes, 16 bits), 0xFFFFFFFF00000000 (4 bytes, 32 bits), and 0xFFFFFFFFFFFFFFFF (8 bytes, 64 bits). If the mask is 0xFF00000000000000, then it has many unused bit that we can use to store the truncated last bits of address and value. If the mask is 0xFFFFFFFFFFFFFFFF, then the last six bits in the address must be 0 and we can use them. (Normally any update to 64 bits memory is 64 bits address-aligned. Thus if we update 64 bits, the last six bits of the address is 0. As least in all STAMP benchmarks we find this to be true. We argue that this may because of their memory allocation pattern. Always doing 64 bits address-aligned access is good for performance and is adopted in much modern programs. Nevertheless, we have implemented a fall-back strategy: if ever we found any 64 bits update is not 64 bits address-aligned, we split it into two 64 bits address-aligned updates that cover it. This is done transparently in PTM_write(a, value) thus we can guarantee all 64 bits updates have their address to be 64 bits address-aligned.)

Discuss for the log fields. We adopted the Tiny-STM's write-ahead-log (we call redo log here) to support consistent update of any non-volatile data. That is, before updating any data, we need to write to the log first. The *PTM_write(a, value)* in Fig. 4 represents this process. In *PTM_write(a, value)*, the memory pointed by *a* is not updated. Instead, a log slot recording this information is written. Then at the *PTM_end()*, we write the value to the memory pointed by *a*. Any access to the memory (pointed by *a*) between *PTM_write(a, value)* and *PTM_end()* will be redirected to the log slot (e.g. *PTM_read(a)* as shown in Fig. 4). Here in order to record the *PTM_write* information, we need at least two fields in each log slot: (1) the address field recording which address is being written and (2) the value field recording the new value that should be written to the address at *PTM_end()*. Moreover, the most popular case is we just need to update 64 bits of a memory region in each *PTM_write*. But it is possible that we need to update less than 64bits in a single *PTM_write*. For example, what if we need to update only 8 bits (one byte) of the memory pointed by *a*? In order to support this, we leverage the mask field as shown in Fig. 5. In this situation, the parameter mask is passed to *PTM_write* explicitly and is recorded in the mask filed in each log slot. Later in *PTM_end()*, we need to use both the value and mask to determine what to write to the corresponding memory space.

(6) Recovering

If the system meets crashes or power cut, we should do recovery before the non-volatile memory is accessed again. The recovery is done after the system rebooting in two steps. Firstly a special kernel thread will scan the log of the meta-data of non-volatile memory. For the transactions which are at committing phase, it will re-commit all the modifications to the meta-data. For unfinished transactions, it just aborts them by discarding the log. Secondly, a special process will scan all the chunk heads and then map every heap into its space. Then it scans the logs in the heap to do the similar recovery. As we adopt per-thread log, the recovery should be done according to the timestamp of the log (shown in Fig. 5) After doing the recovery, the logs in meta-data part will be cleaned and the user logs in heaps will be freed.

Table 2 Interfaces provided by FP-Heap.

lib-FP interfaces		Description	Key arguments
Memory Interface	FP-malloc()	Allocate a piece of persistent memory.	The size of the allocated space. The name given to the new allocated object which helps retrieve its address.
	FP-free()	Free a piece of persistent memory.	The address of the space.
Support for non-volatility	FP-retrieve()	Find the address of a malloced object by its name.	The name of the object.
	FP-heap()	Map a heap into process space.	The name of the heap.
	FP-mprotect()	Change the access mode of a heap.	The name of the heap and the protect mode of the heap.
Transaction	PTM-begin()	Begin a transaction.	
	PTM-end()	End and commit a transaction.	
	PTM-read()	Transactional read a variable.	The address of the variable.
	PTM-write()	Transactional write a variable.	The address of the variable and the new value.

4. Implementation Details

FP-Heap is based on Linux (kernel version 3.11.0) and consists of 2 parts: (1) a kernel patch that manages physical pages of non-volatile memory and the mapping of chunks and (2) a library that exposes heap interfaces with support for non-volatility to programmers. Table 2 shows the main interfaces FP-Heap provides.

4.1 Emulating Non-Volatile Memory

As non-volatile memory is not yet available now, we use DRAM to emulate it. During the system rebooting, we set apart several GB of DRAM (currently 2 GB) to emulate non-volatile memory and we use a free-list to manage all the free non-volatile memory pages. In order to emulate the non-volatility of non-volatile memory, we dump the whole non-volatile memory pages to disk before system rebooting and copy it back after system rebooting. So it seems like the data saved in the emulated non-volatile memory are never lost.

For emulating the latency of non-volatile memory (in this paper we emulate PCM), we adopt the same method introduced in Mnemosyne [5]. The read speed of PCM is almost the same as DRAM thus we do not emulate the latency of read. Moreover, as most write to PCM will be cached by CPU, we only emulate the write latency at commit time before fence. At commit time, we firstly force all writes to non-volatile memory using clflush, and then insert a delay by using rdtsc to get and check the time stamp, and then issue a memory fence using mfence.

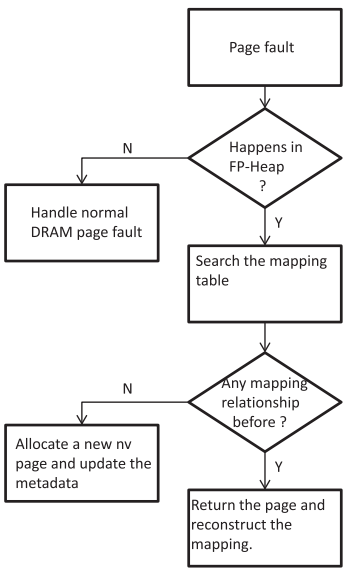


Fig. 6 Page fault handler.

4.2 Physical Memory Management

During the system rebooting, we will scan all the page tables in the chunk heads to see which physical page is occupied and which is free. We then reserve all the occupied pages and set up a free list to manage all the free physical pages. If an existing heap is mapped into a process space and accessed again, we will use the page table in the chunk head to recover the mapping relationship in operating system kernel. The whole process is similar like current page fault handler and is shown in Fig. 6.

Table 3 Benchmarks

Benchmarks		Description	Input
Stress Tests	Region_creation	Create n persistent regions. The size of each region is 1MB.	*n=64, 128, 256, 512, 1024*
	Page_fault_test	Create a persistent region of size n, then touch every page.	*n=64MB, 128MB, 256MB, 512MB, 1024MB*
	Access_test	Create a persistent region of size n. Then write access every byte.	*n=64MB, 128MB, 256MB, 512MB*
STAMP	bayes		*-e-1 -i1 -n4 -p10 -q1 -r4096 -s1 -t1 -v32*
	genome		*-g16384 -n4194304 -s64 -t1*
	intruder		*-a10 -l16 -n1048576 -s1 -t1*
	kmeans		*-m40 -n40 -t0.05 -i random2048-d16-c16*
	labyrinth		*-i random-x32-y32-z3-n96*
	ssca2		*-i0.5 -k2 -l3 -p3 -s17 -t1 -u0.1 -w0.6*
	vacation		*-n2 -q90 -u98 -r16384 -t4096*
	yada		*-a15 -i ttimeu100000.2*

5. Evaluation

We mainly evaluate the performance of FP-Heap compared with previous work on our own designed tests and STAMP transactional benchmarks.

We mainly evaluate two aspects as follows and they are shown in Sect. 5.1 and Sect. 5.2, respectively:

- The direct access overhead of FP-Heap compared with Mnemosyne and the optimized file system of PMFS [1].
- The performance of FP-Heap compared with the similar interface of Mnemosyne [5].

The experiments are all conducted on a platform with Intel Core 2 quad-core (2.2GHz) equipped with 8GB of physical DRAM running Linux kernel 3.11.0. We set apart 2GB of DRAM to emulate non-volatile memory.

5.1 Stress Tests for Direct Access

PMFS [1] is an optimized file system for non-volatile memory. It can offer the same functionality of supporting direct access through memory map. Mnemosyne [5] also offers similar interface pmap() and pmalloc(). In order to evaluate our FP-Heap compared with PMFS on supporting direct access, we introduce three simple stress tests as shown in Table 3. The three stress tests are Region_creation, Page_fault_test, and Access_test. In Region_creation, we use PMFS and Mnemosyne to create several persistent regions of 1MB through mmap() and pmap(). For FP-Heap, we set the chunk size to 1MB so that each time when we allocate a 1MB persistent region through FP_malloc(), it will try to get a new chunk. In Page_fault_test, we create a persistent region of size n and then touch every page to trigger page fault. This can test the underlying overhead of allocating physical pages and setting up mapping relationships. For FP-Heap, we set the chunk size to n. In Access_test, we create a persistent region and write every byte in the region to test if there is any other overhead after the mapping is set up. The above three stress tests can show the overhead in FP-Heap, PMFS, and Mnemosyne on supporting direct access.

5.1.1 Results

The results of stress tests are shown in Fig. 7. In all stress tests our FP-Heap introduces ignorable overhead compared with Base. We can see FP-Heap performs much better than PMFS and Mnemosyne in Region_creation and Page_fault_test. The main overhead of PMFS and Mnemosyne when creating a persistent region is to set up some mapping structures with a backed file. Moreover, every page fault will also result into a complex search for a proper physical page and setting up the mapping relationship. PMFS optimizes the file system for non-volatile memory and it performs better than Mnemosyne in Page_fault_test. While in FP-Heap, the chunk based organization of non-volatile memory makes it much easier to get a new chunk and map a new physical page. In Access_test, we can see there is not obvious difference among FP-Heap, PMFS, and Mnemosyne. This shows that the main overhead

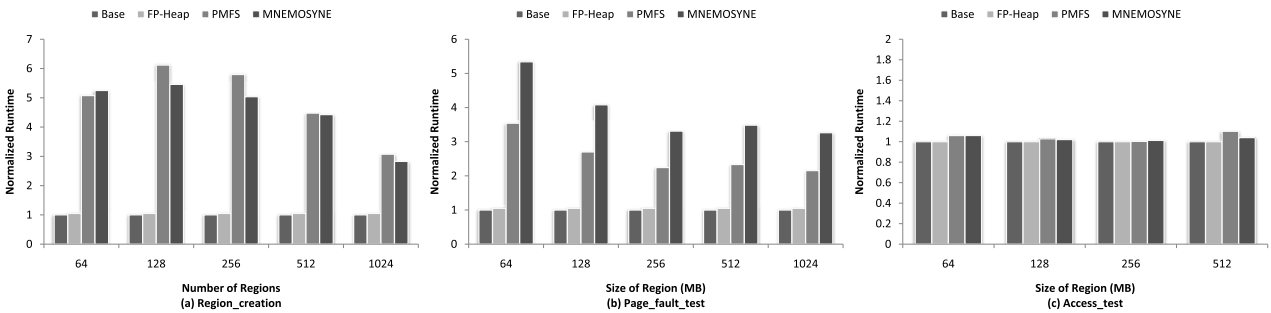


Fig. 7 Results of stress tests. (Base is normal execution with traditional DRAM management mechanism)

of supporting direct access is setting up persistent regions and handling page faults.

5.2 STAMP

On offering direct access, Mnemosyne is more like our work. It offers persistent heaps through pmalloc() and a transaction mechanism to support atomic update of data. We run the STAMP [19] benchmark to compare FP-Heap and Mnemosyne since STAMP is memory intensive and has already been well written in transactions. We do not compare FP-Heap with PMFS here because they focus on offering different interfaces and functionalities.

We firstly run STAMP in sequential mode without transaction mechanism to test the memory allocation overhead of FP-Heap compared with Mnemosyne. In the sequential test, we set every memory allocation from persistent heap as a single transaction. Secondly, we run STAMP with transaction mode for 1, 2, 4 threads to test the overall overhead of FP-Heap and Mnemosyne. We run STAMP with its input shown in Table 3. In ssca2, we set the problem scale to 17 instead of 20. This is because the parameter 20 will make it allocate very large amount of memory (more than 2 GB) and leave us no space to log.

5.2.1 Results

The results of sequential STAMP are shown in Fig. 8 and the running statistics are shown in Table 4. Both FP-Heap and Mnemosyne introduce some overhead compared with the base due to logging. FP-Heap can achieve 7%-20% speedup over Mnemosyne. FP-Heap outperforms Mnemosyne due to two reasons: (1) when serving every large object or new chunk allocation, FP-Heap will perform the allocation through the extended VMM, while Mnemosyne will create a new file and map the file through file system. The number of mapped regions is shown in Table 4. (2) The memory mapping information of every physical page is stored as <virtual memory address, physical page frame number>in FP-Heap, while in Mnemosyne it requires more space to store the <page frame number, file number, page offset in file>which leads to more writes to non-volatile memory at commit time. The total log size is shown in Table 4. According to these two factors, we can see FP-Heap performs better in the bayes, genome, intruder, ssca2, and yada. Furthermore, another importance factor which will affect the result is the computing/memory operation ratio. If the computation of a program accounts for the main part of executing time, then FP-Heap performs not so better than Mnemosyen (e.g. intruder). While if a program asks for a large amount of memory and just do little computation on it, FP-Heap can achieve a better result over Mnemosyne (ssca2 is a good example here). The computing/memory ratio can be seen from comparing FP-Heap to Base. In ssca2, the FP-Heap introduces large overhead due to logging thus it can be concluded that the computation time in ssca2 accounts for small parts.

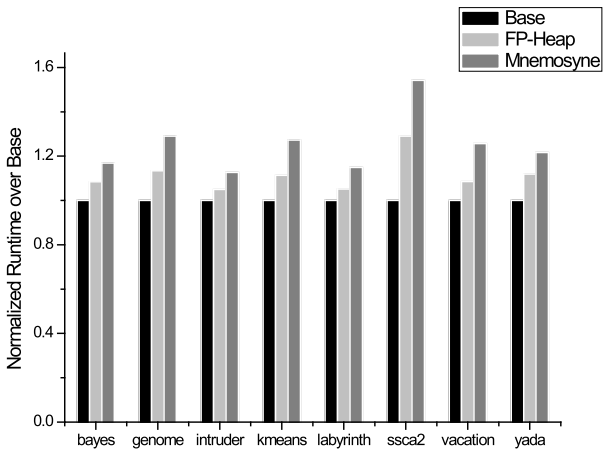


Fig. 8 Results of stamp seq. (The base is STAMP benchmarks running sequentially on traditional DRAM management mechanism and without transactional execution.)

The results of transactional STAMP are shown in Fig. 9. Both FP-Heap and Mnemosyne show scalability on almost all benchmarks except labyrinth and yada in which the four threads versions conflict more and limit the performance. The problem scale in bayes is small so that it does not show any scalability. The better performance in bayes of FP-Heap compared with Mnemosyne can just demonstrate our memory allocation is better than Mnemosyne. As we increase the scale of problem, FP-Heap and Mnemosyne performs closely because the memory allocation accounts for small part of the whole execution. Intruder shows a good ex-

Table 4 Statistics of STAMP.

	Number of object	Allocated memory (KB)	Log size FP-Heap (KB)	Log size Mne (KB)	File map in Mne
bayes	8828000	253410	9818	10313	3559
genome	2142295	400920	3709	4492	936
intruder	14990453	624282	17430	18649	6656
kmeans	33	1317	5	8	16
labyrinth	870	143	1.4	1.6	14
ssca2	72	343023	1340	2010	54
vacation	158187	6259	183	195	137
yada	1139329	43617	1310	1395	455

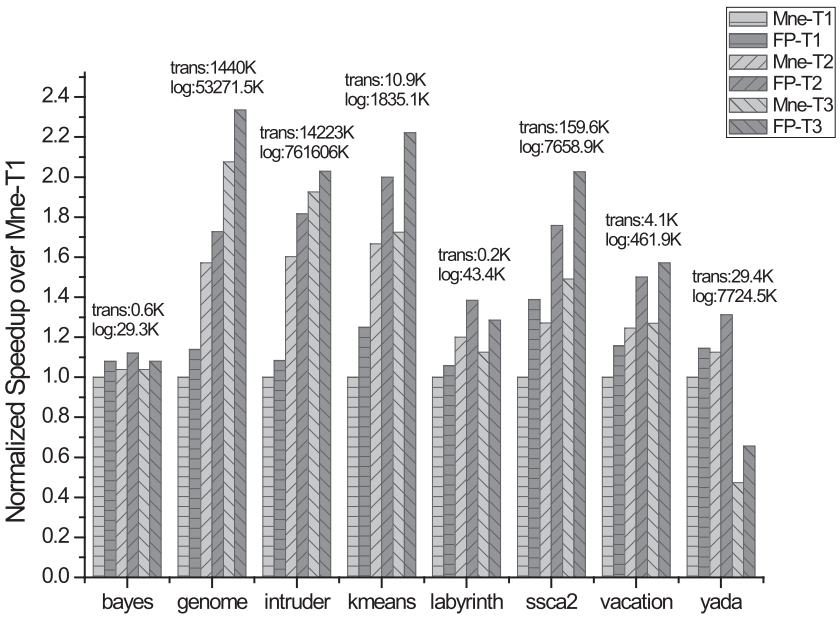


Fig. 9 Result of stamp scale.

ample of this. The total number of committed transactions and the total logs of the write set in TinySTM are shown above the bars in Fig. 9. On every commit, the write set log in Mnemosyne will firstly be flushed from DRAM into a persistent log in non-volatile memory and then flushed from the persistent log to destinations. While in FP-Heap, we store the write set just in non-volatile memory and rely on cache in CPU to accelerate accessing to the log. It performs a little better than Mnemosyne. In a whole, FP-Heap can achieve 4%-38% speedup over Mnemosyne.

6. Related Work

Mnemosyne [5] and NV-heaps [6] both expose non-volatile memory directly to upper applications. At operating system level, they rely on traditional file system to manage physical pages and mapping relationship. Different from them, FP-Heap proposes a lightweight chunk-based scheme to organize non-volatile memory and maintain the persistent mapping relationship. NV-heaps [6] provides some language level features that are not provided in FP-Heap such as safe point and garbage collection.

PMFS [1], Aerie [16], BPFS [20], and SCMFS [21] all propose using file system to manage non-volatile memory. Traditional file system works well on low-speed devices. But when it comes to fast non-volatile memory, the software overhead of file system will become a bottleneck [1], [16]. Thus the main work of the above studies is to optimize file system to fit the feature of non-volatile memory (e.g. removing block layer). Aerie [16] proposes flexible file system interface. It implements a basic management of non-volatile memory in operating system kernel and offers programmers the flexibility to implement their own customized file system interface. These studies resides on the other direction of research and do not conflict with our work.

Studies that can benefit directly from non-volatile memory are databases [22], [23] and persistent object systems [2]–[4]. They usually rely on secondary storage to make data persistent and focus on hiding the latency of writing files. Their performance can be greatly enhanced by FP-Heap. Other work relies on flash to make data persistent. However, they usually do not support direct access.

Rio vista [24] supports direct access to battery backed memory and relies on file mapping to manage persistent data. Unlike FP-Heap, it mainly focuses on the transaction mechanism. Other work RVM [25] also focuses on the

durable transaction mechanism.

There are some other studies that investigate how to make use of non-volatile memory. [26] discusses how to integrated non-volatile memory into current architecture and how operation system could adapt to it. [27] uses non-volatile memory to store the journal data of file system to reduce writes to disk.

7. Conclusion

This paper introduces FP-Heap, a persistent heap to support directly access to non-volatile memory. FP-Heap extends traditional VMM to manage non-volatile memory and maintain the mapping relationship. Also, FP-Heap offers a transaction mechanism to ensure data consistency, a simple namespace to support data indexing, and a basic access control to support data sharing. FP-Heap facilitates the design of persistent data structure. Compared with previous work Mnemosyne, FP-Heap performs better due to its customized VMM and optimized transaction mechanism.

Acknowledgements

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301, 2012AA010901, 2012AA010303, and 2015AA01A301, by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61272142, 61402492, 61402486, 61379146, 61272483, by the laboratory pre-research fund (9140C810106150C81001), and by the open project of State Key Laboratory of High-end Server & Storage Technology (2014HSSA01).

The authors will gratefully acknowledge the helpful suggestions of the reviewers, which have improved the presentation.

References

[1] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," Proc. Ninth European Conference on Computer Systems, pp.1–15, ACM, 2014.

[2] V. Singhal, S.V. Kakkad, and P.R. Wilson, "Texas: An efficient, portable persistent store," Persistent Object Systems, pp.11–33, Springer, 1993.

[3] M. Atkinson and R. Morrison, "Orthogonally persistent object systems," The VLDB Journal—The International Journal on Very Large Data Bases, vol.4, no.3, pp.319–401, 1995.

[4] L. DeMichiel and M. Keith, "Java persistence API," JSR, vol.220, 2006.

[5] H. Volos, A.J. Tack, and M.M. Swift, "Mnemosyne: Lightweight persistent memory," ACM SIGARCH Computer Architecture News, vol.39, no.1, pp.91–104, ACM, 2011.

[6] 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," ACM SIGARCH Computer Architecture News, vol.39, no.1, pp.105–118, ACM, 2011.

[7] B.G. Johnson and C.H. Dennison, "Phase change memory," US Patent 6,791,102, Sept. 14 2004.

[8] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of last-level on-chip cache using spin-torque transfer ram (stt ram)," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol.19, no.3, pp.483–493, 2011.

[9] M. Di Ventra, Y.V. Pershin, and L.O. Chua, "Circuit elements with memory: memristors, memcapacitors, and meminductors," Proc. IEEE, vol.97, no.10, pp.1717–1724, 2009.

[10] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M.G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong, "A 20nm 1.8V 8Gb pram with 40mb/s program bandwidth," Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International, pp.46–48, IEEE, 2012.

[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," ACM SIGARCH Computer Architecture News, vol.37, no.3, pp.14–23, ACM, 2009.

[12] S. Cho and H. Lee, "Flip-N-write: a simple deterministic technique to improve pram write performance, energy and endurance," Microarchitecture, 2009, MICRO-42, 42nd Annual IEEE/ACM International Symposium on, pp.347–357, IEEE, 2009.

[13] D.H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N.P. Jouppi, and M. Erez, "Free-p: Protecting non-volatile memory against both hard and soft errors," High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, pp.466–477, IEEE, 2011.

[14] M.K. Qureshi, M.M. Franceschini, A. Jagmohan, and L.A. Lastras, "PreSET: Improving performance of phase change memories by exploiting asymmetry in write times," ACM SIGARCH Computer Architecture News, vol.40, no.3, pp.380–391, 2012.

[15] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," ACM Sigplan Notices, vol.35, no.11, pp.117–128, 2000.

[16] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M.M. Swift, "Aerie: flexible file-system interfaces to storage-class memory," Proc. Ninth European Conference on Computer Systems, pp.1–14, ACM, 2014.

[17] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp.237–246, ACM, 2008.

[18] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," IEEE Trans. Parallel Distrib. Syst., vol.21, no.12, pp.1793–1807, 2010.

[19] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," Workload Characterization, 2008, IISWC 2008, IEEE International Symposium on, pp.35–46, IEEE, 2008.

[20] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," Proc. ACM SIGOPS 22nd symposium on Operating systems principles, pp.133–146, ACM, 2009.

[21] X. Wu and A.L.N. Reddy, "SCMFS: a file system for storage class memory," Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p.39, ACM, 2011.

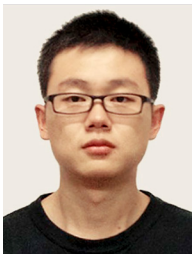
[22] M.A. Olson, K. Bostic, and M.I. Seltzer, "Berkeley db.," USENIX Annual Technical Conference, FREENIX Track, pp.183–191, 1999.

[23] A. Kumar, "The openldap proxy cache," IBM, India Research Lab, at least as early as May, 2003.

[24] D.E. Lowell and P.M. Chen, "Free transactions with rio vista," ACM SIGOPS Operating Systems Review, vol.31, no.5, pp.92–101, ACM, 1997.

[25] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler, "Lightweight recoverable virtual memory," ACM Trans.

Comput. Syst., vol.12, no.1, pp.33–57, 1994.
[26] K. Bailey, L. Ceze, S.D. Gribble, and H.M. Levy, "Operating system implications of fast, cheap, non-volatile memory," Proc. 13th USENIX conference on Hot topics in operating systems, p.2, USENIX Association, 2011.
[27] E. Lee, H. Bahn, and S.H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," FAST, pp.73–80, 2013.



Wenzhe Zhang was born in 1988. He received the B.S., M.S. degrees from the School of Computer, National University of Defense Technology in 2010 and 2012. Now he is a PhD candidate in School of Computer, National University of Defense Technology. His research interests include non-volatiel memory technologies, parallel programming and compiler optimization.

Kai Lu received the B.S., Ph.D. degrees from the School of Computer, National University of Defense Technology in 1995 and 1999. He is now a professor in School of Computer, National University of Defense Technology. His research interests include parallel programming, operating system and security.

Xiaoping Wang received the B.S., M.S., Ph.D. degrees from the School of Computer, National University of Defense Technology in 2003, 2006, and 2010. He is now an assistant professor in School of Computer, National University of Defense Technology. His research interests include operating system and sensor networking.

Jie Jian received his BS in computer science from Tsinghua University and MS in computer science from National University of Defense Technology (NUDT), China. He is currently pursuing his PhD degree at National University of Defense Technology. His research interests include high speed interconnect network, high radix router architecture and optical interconnection network.