Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 398369, 13 pages
http://dx.doi.org/10.1155/2015/398369

Research Article

Write-Combined Logging: An Optimized Logging for Consistency in NVRAM

Wenzhe Zhang,[1] Kai Lu,[1] Mikel Luján,[2] Xiaoping Wang,[1] and Xu Zhou[1]

[1] *Science and Technology on Parallel and Distributed Processing Laboratory, College of Computer, National University of Defense Technology, Changsha 410073, China*
[2] *School of Computer Science, The University of Manchester, Manchester M13 9PL, UK*

Correspondence should be addressed to Kai Lu; kailu@nudt.edu.cn

Received 31 July 2015; Revised 8 October 2015; Accepted 18 November 2015

Academic Editor: Wan Fokkink

Nonvolatile memory (e.g., Phase Change Memory) blurs the boundary between memory and storage and it could greatly facilitate the construction of in-memory durable data structures. Data structures can be processed and stored directly in NVRAM. To maintain the consistency of persistent data, logging is a widely adopted mechanism. However, logging introduces write-twice overhead. This paper introduces an optimized write-combined logging to reduce the writes to NVRAM log. By leveraging the fast-read and byte-addressable features of NVRAM, we can perform a read-and-compare operation before writes and thus issue writes in a finer-grained way. We tested our system on the benchmark suit STAMP which contains real-world applications. Experiment results show that our system can reduce the writes to NVRAM by 33%–34%, which can help extend the lifetime of NVRAM and improve performance. Averagely our system can improve performance by 7%–11%.

1. Introduction

Emerging nonvolatile memory (NVRAM) technologies [1] blur the boundary between memory and storage with its byte-addressability and fast access similar to DRAM and nonvolatility similar to disk. Systems with NVRAM attaching to memory bus have been widely advocated [2] which would greatly facilitate the construction of in-memory durable data structures [3]. In such systems, persistent data structures reside in NVRAM as they are created and modified rather than being operated in one format and transformed into another format to be durable [4]. Lots of performance benefits can be reaped from this uniform state, especially for the applications of database [4]. However, the consistency of persistent data is required to be maintained in case of software or hardware failure. Otherwise the data that persisted in NVRAM may be left in an invalid intermediate state after system reboot and is not reusable. This is also a classic problem in file systems and databases.

Figure 1 shows an example of data consistency, if we have a data structure of people which contains two fields: name and age. It is stored in NVRAM and its initial state is as Figure 1 shows. If we want to modify the structure to change it to store other people's information, such as name: XYZ and age: 28, we have to update the two fields separately (as step 1 and step 2 in the example) due to the hardware limitation. If the system crashes right after we finished step 1 and before step 2, although the information in the data structure is not lost after rebooting, it is wrong and is not reusable now (state shown in the dashed box). This is the problem of the data consistency for NVRAM.

Logging [5] is a widely adopted mechanism to guarantee the consistency of persistent data. With such mechanism, every consistent update to nonvolatile media will result in extra writes. For example, in Figure 1, in write-ahead logging, an update operation would (a) first write to the log, wait until the log is persistent, and then (b) update the correspondent locations according to the log. Thus if failure happens during the operations, we can recover the data structure to a consistent state. The write-twice problem [5] will lead to a degradation of performance and faster wearing out of NVRAM [6].

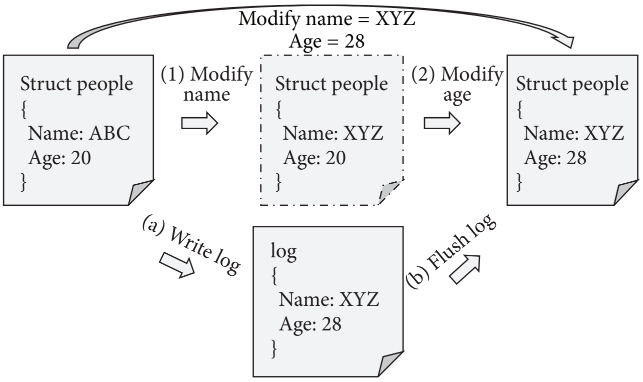


Figure 1: Example of data consistency and write-twice logging.

The write-twice problem has been in research [5] for long in traditional file systems and databases. Recently some work [9] proposed using nonvolatile memory as a combination of data buffer and logging (or journaling) area in front of the secondary nonvolatile media. This method separates the two writes to different destinations: one write to nonvolatile memory as logging and another write to secondary storage as true update. In this way it could improve performance by reducing the writes to secondary storage as normally the secondary storage is much slower than nonvolatile memory. However, this solution only applies when we store data mainly in secondary storage. As we introduced before, NVRAM allows data to be persistent directly in it without any transformation. In this new context the write-twice problem still exists and has never been studied before. Another study trend of tackling write-twice problem results in log-structured file system [10]. In such systems, all data appears in the form of log. It only does one write to the log for every update. However, this system may be trapped into large and complex data indexing problem and thus it fades out of mainstream use.

Above all, this paper proposes write-combination logging, a novel method to reduce the writes to nonvolatile memory while maintaining the consistency of data stored in it. By leveraging the byte-addressable and fast-read properties of NVRAM, we can perform a read-and-compare operation before every write and thus eliminate unnecessary writes. Also we can combine two 32 bits' modifications into a 64-bit write to reduce the number of writes to NVRAM. As nonvolatile memory is limited in terms of bandwidth and lifetime [11], reducing the number of writes to it would be very beneficial.

The main work of this paper is as follows:

(i) a nonvolatile heap based on NVRAM which offers transactional interface for upper applications to access nonvolatile data;

(ii) a novel write-combination mechanism to reduce the number of writes to NVRAM without sacrificing the data consistency. All the work is done at operating system and library level without any changes to the underlying hardware.

The rest of this paper is organized as follows: we give the background information and our motivation in Section 2. Sections 3 and 4 show our design and implementation in detail and the experiments results are shown in Section 5. Related work is discussed in Section 6 and Section 7 concludes.

2. Background and Motivation

2.1. Nonvolatile Memory. Nonvolatile memory (NVRAM), or storage class memory (SCM) [4], or persistent memory [1], has been developing fast recently. Phase Change Memory (PCM) is a representative one and is now available as prototype. It offers features as byte-addressablility, fast access, and being nonvolatile. Moreover, it is highly scalable on density. Along with the Multilevel Cell (MLC) technology [12], PCM can be very large in capacity. Table 1 shows a comparison between PCM and DRAM on some key features. As we can see from the table, the read speed of PCM is almost the same with DRAM while the write speed is much slower. This asymmetric access speed has been leveraged in many previous works [13] in architectural level to accelerate writes to PCM. Limited lifetime is another problem of PCM. Many wear-leveling mechanisms have been proposed to tackle this [14]. Also there are some studies [15, 16] at hardware level to reduce the number of writes to PCM to extend its lifetime. Above all, accelerating its write speed and extending its lifetime are essential for making better use of it.

2.2. Assumptions. In order to facilitate persistent data processing, we do our work based on the widely advocated architecture [4] in which the NVRAM is attached to the memory bus and forms a single physical address space with DRAM. In such architecture, NVRAM could be accessed directly by CPU via normal load and store instructions. The cache of CPU can accelerate accessing NVRAM as well. As we are tackling the data consistency problem of NVRAM, we make several basic assumptions like previous work [17]: (1) any 64-bit write to NVRAM is atomic, which means any 64-bit write to NVRAM either is persistent as a whole or has no effect at all; (2) a special memory fence should be provided by underlying hardware to stall execution until all previous writes reach NVRAM.

Table 1: Comparison between PCM and DRAM [4, 7].

	DRAM	PCM
Read	60 ns	100–300 ns (present) 50–85 ns (future)
Write	60 ns	10–150 us (present) 150–1000 ns (future)
Density	$7F^2$	$4F^2$
Endurance	10^{16}	10^7
Nonvolatility	N	Y
Byte-addressable	Y	Y

2.3. Observation and Motivation. NVRAM has appealing features of fast-read and byte-addressability, which allows us to perform a read-and-compare before writing to it. By doing this we may have some chances to issue less writes. For example, if we want to issue a 64-bit write to location A, we first check the old value in location A. If, luckily, we find the old value is the same as the new value, then we can eliminate the write. This lucky case would only account for small parts and a more common situation is when a 64-bit write modifies only 32 bits of the location. In this situation we can combine two 32 bits' modifications into a 64-bit write and store it to the log. In this way we reduce a write to the log (originally it would be two 64-bit writes). As the write operation of NVRAM is much more expensive than read, by doing this we can gain performance benefit and extend the lifetime of NVRAM.

We tested the modification ratio of the benchmark suit STAMP [18] to show the potential benefits we can reap. We choose the benchmark suit STAMP because it covers a wide range of applications domains (as described in Table 3) and can show real-world cases. More importantly, it is well written using transactional interface. The concept of transaction is widely used in database to achieve atomic update of data and in our system we will also adopt it to support consistent update of data stored in NVRAM. In transactional mechanism, every thread will first log all modifications in its own log and then update the memory locations at commit time. Here in the test we will show, among all the 64-bit writes, how many proportions are modifying nothing and how many are just modifying 32 bits.

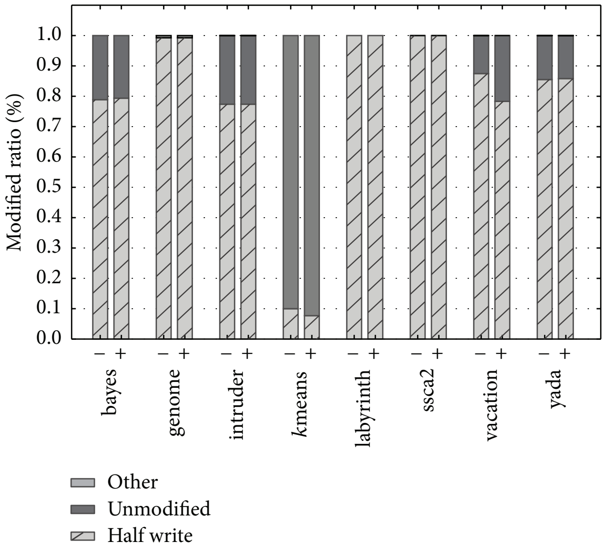


Figure 2: Modification ratio ("−" and "+" mean small and increased problem scales).

Figure 2 shows the modification ratio of benchmark STAMP with inputs shown in Table 3. We gained the modification ratio by comparing the new value and old value of every transactional write at transaction commit time. Like traditional database systems, we assume that the persistent data should be consistently updated at the commit time of every transaction. We recorded information of three types of update: (1) the new write writes the same value thus modifies nothing (shown in the figure as unmodified), (2) the new 64-bit write modifies only 32 bits of first half or second half (shown in the figure as half write); (3) other writes (shown as other). As we can see from the statistics, almost all benchmarks contain a dominating part of half write (except *k*means). Notice that all the transactional writes in these benchmarks are 64 bits. Thus this large proportion of half write shows great potential for optimization. Moreover, for the benchmarks bayes, intruder, vacation, and yada, we can see a considerable proportion (around 20%) of unmodified write. These writes are unnecessary and could be eliminated directly. Above all, for every benchmark we tested here, there are more than 99% writes (the unmodified part and half write part) that could be optimized (combined or eliminated) to tackle the write-twice problem. *k*means is an exception here because of its special memory-intensive accessing mode. Even so we still have near 10% of writes that could be optimized. Finally, as we increase the problem scale for every application, it shows the same trend.

3. Design

We first design a persistent heap based on NVRAM with transactional interface to facilitate allocating and modifying persistent data. Base on that, we give a baseline logging mechanism to maintain consistency and then introduce our optimized logging to reduce the number of writes.

3.1. Persistent Heap. To enable upper applications to access NVRAM directly, we extend current virtual memory manager (VMM) to offer a special system call nv_map(). Like mmap(), nv_map() is used to allocate a region of virtual memory but is mapped to NVRAM pages instead of DRAM pages. The mapping can be controlled in page fault handler simply. However, an important thing is that as NVRAM is nonvolatile, we should keep the virtual-physical mapping relationship nonvolatile too. This is done by storing a nonvolatile page table in NVRAM. We set the first several pages of NVRAM as reserved to be a metadata part. For every process, it mainly stores two kinds of information. (1) The process's nonvolatile virtual region: this is managed in a simple vector that describes all the nonvolatile virtual regions of the current process which are created by nv_map(). (2) The process's nonvolatile page table: this page table is not what controls the current mapping in the kernel. It is just for recovering purpose and is updated at the end of every transaction. When a new page in a nonvolatile virtual region is accessed, we will allocate a new NVRAM physical page and the information will be updated to the nonvolatile page table at the end of every transaction. To reduce the usage of metadata part, we just store the root page directory in the matedata part for every process. Other NVRAM pages that store the secondary level directories and tables are allocated and indexed in the normal way. (3) The process's id: this is the process's absolute path. We use the process's absolute path as the process's id. Thus if a program is reexecuted, we will recover all its previous virtual regions and mappings using the information in the metadata part.

When a program starts after system reboots, we recover the program's nonvolatile regions as follows. (1) Firstly at the start of the program we will recover the nonvolatile virtual regions using the information stores in the metadata part and by reexecuting nv_map() with specific starting addresses and lengths. (2) Then when it accesses some previous regions and triggers page fault, we search the nonvolatile table in the metadata part. If there has been a previous mapping, we will recover the mapping. If not, we will allocate a new NVRAM physical page and update the mapping. All the NVRAM physical pages which are in use will be set as reserved at system booting time according to the nonvolatile page tables in metadata part.

Based on the system call nv_map(), we can easily build a persistent heap for allocating persistent objects. We build our persistent heap based on previous memory allocator Hoard [19]. What a traditional memory allocator does is basically asking for large memory regions from operating system via mmap() and then retailing small objects to upper applications. Here we just modify Hoard to use nv_map() instead of traditional mmap() to ask for large persistent memory regions from operating system and offer memory allocation interface nv_malloc() and nv_free() for upper applications to allocate and deallocate persistent memory structures or objects.

3.2. Transaction System and Baseline Logging. The persistent heap introduced above enables upper applications to access NVRAM directly as they access traditional DRAM. They can allocate persistent objects through the persistent heap and read or store data in NVRAM. However, when they are writing NVRAM, they are not informed of nor assured about the order in which the writes would reach NVRAM. Furthermore, they would not know whether a write has reached NVRAM before a system crash. Thus there is no guarantee here that the data they stored in NVRAM is reusable after the system reboot. It is the classic consistency problem for persistent data.

Similar to the traditional database systems [20] and NVRAM-based work [3], we adopt the notion of transaction to support consistent update of data in NVRAM. We build our transaction system based on TinySTM [21, 22] and provide interface shown in Table 2. TinySTM is lightweight software transactional memory (STM) system. Compared with traditional database transactions which offer ACID (atomicity, consistency, isolation, and durability), software transactional memory systems usually only offer atomicity, consistency, and isolation but no durability. Here, with NVRAM, we can easily add durability to STM by storing data structures directly in NVRAM. On the other hand, we can easily achieve consistent update of data with the help of STM.

Table 2: Interface.

Interface	Description
nv_malloc (size, name)	Allocate a persistent object
nv_free (addr)	Free a persistent object
tm_begin ( )	Start a transaction
tm_end ( )	End and try to commit a transaction
tm_read ( )	Transactional read
tm_write ( )	Transactional write

Our transactional system is shown in Figure 3. The whole system acts as follows. (1) We first allocate persistent objects or data structures in NVRAM. Upper applications are required to access them using our transactional interface (or the persistent data can be accessed directly without using our transactional interface but in this case we do not guarantee the update to be consistent). During transaction running, TinySTM will keep a writing buffer to isolate the writes of different transactions (step 1 shown in the figure). In our system we put the writing buffer in DRAM to accelerate accessing NVRAM (as previous work [17] shows putting frequently accessed data in NVRAM would result in a performance degradation of around 25%). (2) When a transaction finishes and is ready to commit, we make it commit to a log stored in NVRAM instead of the original locations as traditional STM does. This step is shown as step 2 in Figure 3. The logging we added is the key to guarantee the update of persistent data to be consistent. Otherwise if a system failure happens during the committing to the original locations, we will lose the data stored in DRAM and the data stored in NVRAM are in a nonconsistent state. (3) After all data have reached the log in NVRAM, we can then do the logging flush to update the original locations (shown as step 3 in the figure).

In order to protect the metadata (objects allocation information) of our persistent heap, we also add log. The difference is that our memory allocator is based on Hoard which has been well written using fine-grained lock. Thus we just need to add log to buffer writes and do not have to detect transaction conflicts.

3.3. Write-Combined Logging. The logging we added is like journaling in file system which is important to maintain the consistency of the data stored in NVRAM. However, it comes with costs, which is the write-twice problem. Steps 2 and 3 in Figure 3 demonstrate the write-twice problem: every update of persistent data results in two writes to NVRAM (a write to log and another write to the original location).

In this section we design a mechanism called write-combined logging to reduce the writes to log (step 2 in Figure 3) without sacrificing information to maintain the

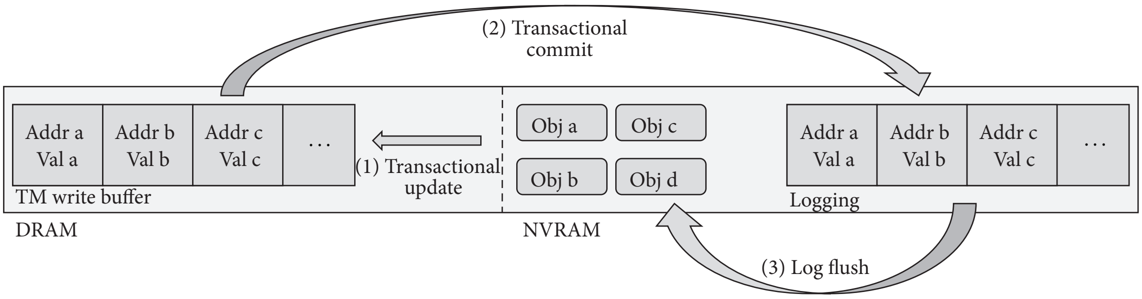


Figure 3: Transactional mechanism.

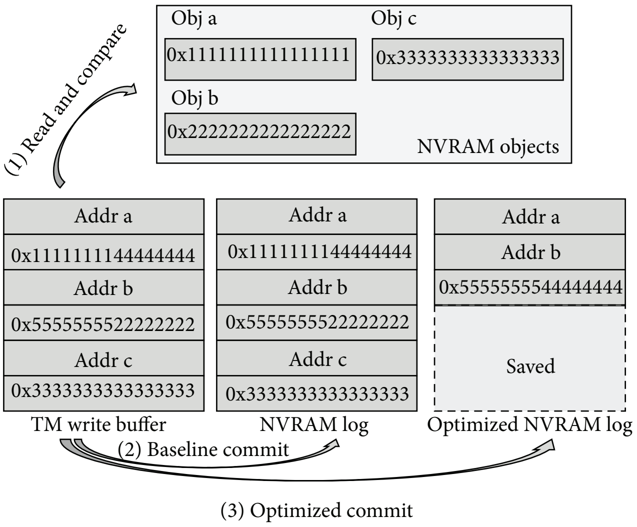


Figure 4: Transactional mechanism.

consistency of data. Our idea is based on the following key insights. (1) NVRAM is fast to serve read. Reading data from NVRAM is much cheaper than writing data to it. (2) Compared with traditional secondary storage, NVRAM is byte-addressable. Reading data from it is much more faster even when cache misses, while reading data from secondary storage may result into copying pages into main memory buffer. (3) Our tests (Figure 2) show that there is great potential to optimize the writing to log.

Based on the insights and observations introduced above, our write-combined logging mechanism is straightforward. We mainly modify step 2 in Figure 3 to optimize the writes to log at transaction commit time. Our write-combined logging is shown in Figure 4. Assuming, in this transaction, that Obj a and Obj b are both modified partially (Obj a is modified at last 32 bits and Obj b is modified at first 32 bits), Obj c is written in the same value as its old value. In the baseline design, the transaction commit will write $6 * 64$ bits of data (3 addresses and 3 new values) into the NVRAM log (shown as step 2 in Figure 4). However, if we first do a read-and-compare operation to check the modifications, we can eliminate and combine the writes. As step 3 shows in the figure, we combine the modifications of Obj a and Obj b into a single 64-bit write. Moreover, the update of Obj c is eliminated. In this case our write-combined logging reduces as much as 50% writes to NVRAM. Notice that when we combine two writes into one, we can reduce one write to NVRAM log. But if we eliminate one write, we reduce two writes (one for address and one for value) to NVRAM log. Thus although the unmodified write accounts only for small parts in a transaction, we can benefit greatly from it.

When we are combining the writes in the NVRAM log, we need to store extra information of the combination. We do not add extra space to achieve this. In the benchmarks we tested, all the writes are in the form of 64 bits, which leads to the addresses all having their last 4 bits to be zero. We can leverage the last 4 bits of every address to store the combination information. Moreover, for applications which need to do an update at arbitrary granularity less than 8 bytes, a mask field is often introduced in traditional STM [21] systems. In such situation we can reduce the writes to NVRAM more by combining the mask field with the value field. By comparing with the old value, we can store flipping bits in the new value field instead of the original new value. The flipping bits indicate which bit in the old value should be reversed in order to get the new value. In this way we can combine the mask field and value field in the NVRAM log efficiently. However, we argue that this less-than-64-bits write is rare in 64-bit systems; thus we do not implement this mask combination in our system. We tackle this problem by just aligning the pointer which has been passed to our interface tm_read() or tm_write() to the former 64-bit boundary and record 64 bits anyway. This may introduce some unnecessary records but we believe this situation will do no harm to the overall performance.

3.4. Discussion. The log in NVRAM can be flushed in ways of synchronous or asynchronous flushing. The asynchronous flushing can move the flushing out of critical path thus improving performance. However, it needs extra coding to indicate the flushing and we need to search and index data in the NVRAM log which introduces overhead, especially when the logging is getting large. Anyhow, both strategies need the log to be whole persistent at the time of transaction commit, which is in the critical path. This paper aims to reduce the logging data and thus could accelerate step 2 in Figure 3 no matter which flushing strategies we adopt. We do not discuss which strategy is better in this paper and just adopt synchronous flushing for simplicity.

4. Implementation

The whole system we implemented include a kernel patch to Linux kernel 3.11 to maintain the persistent mapping and a runtime library to manage the memory allocation and transactional accessing of persistent data structures. Our system needs no modification on underlying hardware.

4.1. Memory Management of NVRAM. As NVRAM is not widely available now, we add latency to DRAM to emulate NVRAM. We separate the whole physical address space into two parts (DRAM and NVRAM) by modifying the memory scanning process at system booting time. The DRAM part is used as usual and we run operating system on it. The NVRAM part will be managed by our kernel patch. Specifically, when an upper application asks for a region of virtual memory through nv_map(), it will firstly get a normal virtual memory region. Then if it accesses any virtual page of the region, our kernel patch would allocate one physical page from NVRAM and map the virtual page to that physical page in the page fault handler. Also the mapping relationship will be recorded into NVRAM. As in this scenario we just need one physical page at a time, we list all the free pages of NVRAM in a free list instead of using traditional body system to manage pages into large continuous blocks. At the system rebooting time, we scan the mapping information in NVRAM and set the already-in-mapping pages as reserved and other pages as free.

In order to emulate the nonvolatility of NVRAM, we dump all the physical pages of NVRAM part into disk before system rebooting and copy them back after system rebooting. Thus it seems that the data stored in NVRAM part are persistent. In order to emulate the access latency of NVRAM, we add latency after write to NVRAM which is like what the previous works [4, 17] do. The read speed of NVRAM is as fast as DRAM so we do not tackle reading specially. Notice that, as we adopt a DRAM buffer for updating NVRAM, the writes to NVRAM only happen at transaction commit time. At transaction commit time we firstly perform the writes to NVRAM log and then flush all the cache lines accordingly using clflush. The latency is added per cache line because the cache line is actually the granularity of updating main memory. After flushing all the cache lines and adding latency, we issue a memory fence using mfence.

4.2. Volatile Cache. Cache is volatile and will affect the consistency of persistent data in memory. In order to solve this problem, every time when we write to the persistent log (at transaction commit time), we firstly write to the log, then flush the corresponding cache lines and issue a memory fence, and then write a complete bit indicating that the log is good and consistent. Thus next time after system crash, we can check this complete bit and we will know if this transaction finishes successfully. The cache flushing and memory fence are used to rule out the faulty situation affected by volatile cache.

4.3. Atomic Heap Operations. We achieve the atomicity of heap operations (i.e., allocation and deallocation) by logging at two different levels. Our memory allocator runs in user level that asks for memory regions through system calls and then serves objects allocations. Here there are two kinds of information we should record and update consistently: (1) the page mapping relationship for NVRAM in the kernel and (2) the objects allocation information in the heap. We add logging to protect these two parts of information but at different privileged level. The logging of NVRAM page mapping is kept in the first several pages in NVRAM which could only be accessed by kernel code. Or otherwise the logging could be modified by user code to gain access to any physical memory region. The logging of object allocation is done by user code and is combined with the way we process the accesses of upper applications. At the point of view of our transactional system, our memory allocator can be regarded as a special upper application. We make our memory allocator run based on nv_map() and add logging to its codes.

The commit operation of the above two special logs can be delayed to the commit time of an upper application's transaction. This is because if we allocate an object and have not accessed it yet, the value of the object is invalid and thus the object could be discarded when system crashes.

4.4. Recovery. We do the recovery at two levels too. At system rebooting time, we first use the log written by kernel code to recover the mapping into a consistent state and then use the mapping table to locate all the reserved pages and free pages. At process starting, we then use the user level log to recover the process's persistent heap into a consistent state. The log will be always mapped at a fixed address to facilitate accessing.

Note that we are not using NVRAM to achieve a checkpoint. So the programs cannot recover from where they have been left off. What we are focusing on is the persistent data. We can guarantee that all the persistent data are in a consistent state at next time when the programs access these data again after system reboot. This goal is the same as the previous work of Mnemosyne [4] and NV-heap [3].

5. Experiments

This paper proposes an optimized write-combined logging mechanism to reduce the writes to NVRAM without sacrificing the information used to maintain the consistency of persistent data. In order to show how our work is able to tackle the write-twice problem, we mainly test and compare our work (referred to as WCL) with the baseline design (referred to as BSL) to show its advantages of reducing writes to NVRAM. Notice that the baseline design introduced in this paper is actually very similar to the method in previous work of Mnemosyne [4]. Mnemosyne focuses on providing lightweight programming interface for NVRAM and it does not show attention to tackling the write-twice problem. Moreover, our work differs from Mnemosyne in managing NVRAM where we extend virtual memory manager to manage it while Mnemosyne relies on traditional file system with memory mapping which has been proved to be heavy for managing NVRAM [23]. In this section we will give a small test to show the comparison of overhead between these two different memory management methods.

5.1. Methodology and Benchmarks. We choose STAMP benchmark suit [18] and STMBench7 [8] as our benchmark. STAMP benchmark suit contains applications that can cover different domains of algorithms (as described in Table 3). STMBench7 benchmark is derived from OO7 [24] and focus on simulating the CAD, CAM, and CASE programs. It contains a lot of data structures which are related to graphs and indexes and are often used in complex applications. These two benchmark suits are well-written using transactional interface; thus we do not have to modify the source code of the benchmarks. We set the input of the benchmarks as shown in Tables 3 and 4 with different problem scales to show that our mechanism could fit for different problem scales. We hook the memory allocation function to make all the benchmarks allocate objects from our memory allocator.

In the experiments, we first test and show the number of writes to NVRAM our WCL performs in every benchmark. Then we show the overall speedup our WCL can achieve over BSL by reducing writes. Finally we show the software overhead of the write-combine mechanism compared with the non-write-combining.

The experiments platform is an AMD server (2.2 GHz, 12-core CPU) with 16 GB DRAM running Linux kernel 3.11. We use 8 GB DRAM to emulate NVRAM and emulate the write latency of NVRAM to be 600 ns which is 10 times slower than its read as shown in Table 1.

5.2. Results

5.2.1. Results of Small Tests of Memory Management. As we manage nonvolatile memory and the mapping relationship in a different way from previous work [3, 4], here we did a simple test to show the comparison of corresponding overhead. WCL represents our mechanism while Mnemosyne represents the previous work [3, 4] which relies on file system to maintain the persistent mapping.

Table 3: STAMP benchmarks ("–" and "+" mean small and increased problem scales).

Benchmarks	Description	Input
bayes	Machine learning	(–) -e-1 -i1 -n4 -p10 -q1 -r128 -s1 -v32 (+) -e-1 -i1 -n4 -p10 -q1 -r8192 -s1 -v32
genome	Bioinformatics	(–) -g8192 -s64 -n32768 (+) -g16384 -s64 -n65536
intruder	Security	(–) -a10 -l16 -n524288 -s1 (+) -a10 -l16 -n1048576 -s1
*k*means	Data mining	(–) -i random-n16384-d24-c16 (+) -i random-n65536-d32-c16
labyrinth	Engineering	(–) -i random-x256-y256-z3-n256 (+) -i random-x512-y512-z3-n512
ssca2	Scientific	(–) -s18 -i1.0 -u1.0 -l3 -p3 (+) -s20 -i1.0 -u1.0 -l3 -p3
vacation	Transaction processing	(–) -n2 -q90 -u98 -r16384 -t1048576 (+) -n2 -q90 -u98 -r16384 -t4194304
yada	Scientific	(–) -a10 -i ttimeu1000000.2 (+) -a15 -i ttimeu1000000.2

We did three simple tests. (1) Region creation: we create several persistent regions. (2) Page fault: we create a region and then trigger all page faults in that region to set up the mapping. (3) Access after all regions have been created and all mapping have been set up. Figure 5 shows the first two situations.

As we can see, our mechanism performs much better in the first two cases. This is because (1) for region creation, previous work which relies on file system will create a file and map the file on every creation of a region and (2) for triggering page fault, previous work will look into the file system to find a proper page when serving the page fault. These two cases are the sources of main overhead. For the third case, our work and the previous work both introduce no overhead and thus we do not show it.

5.2.2. Results of STAMP. Firstly, Figures 6 and 7 show the total NVRAM writes our WCL performs compared with BSL. We can see that, for all the benchmarks except *k*means, our WCL reduces large amount of NVRAM writes. WCL behaves best on the benchmarks bayes, intruder, and yada, because there is considerable amount of unmodified writes in these three benchmarks. As we introduced before, by reducing every one of these unmodified writes, we can reduce two writes to NVRAM. Comparing Figures 6 and 7 we can see that WCL is able to behave well for both small or increased problem scales. The statistics are shown in Table 5. For small problem scale, WCL reduces less writes while the whole number of writes is small as well, resulting in the same proportion, which is compatible with our early observation shown in Figure 2. WCL does not reduce much writes in the

Table 4: STMBench7 benchmarks (workload type is read-write mixed; nonmentioned parameters are by default [8]).

Benchmarks	Description	Input
STMBench7-small	Small data structures and small scale	-w rw -s s (AtomicPartsPerComponent = 200, ComponentsPerModule = 500)
STMBench7-medium	Medium data structures and medium scale	-w rw -s m (AtomicPartsPerComponent = 500, ComponentsPerModule = 800)
STMBench7-big	Big data structures and big scale	-w rw -s b (AtomicPartsPerComponent = 800, ComponentsPerModule = 800)

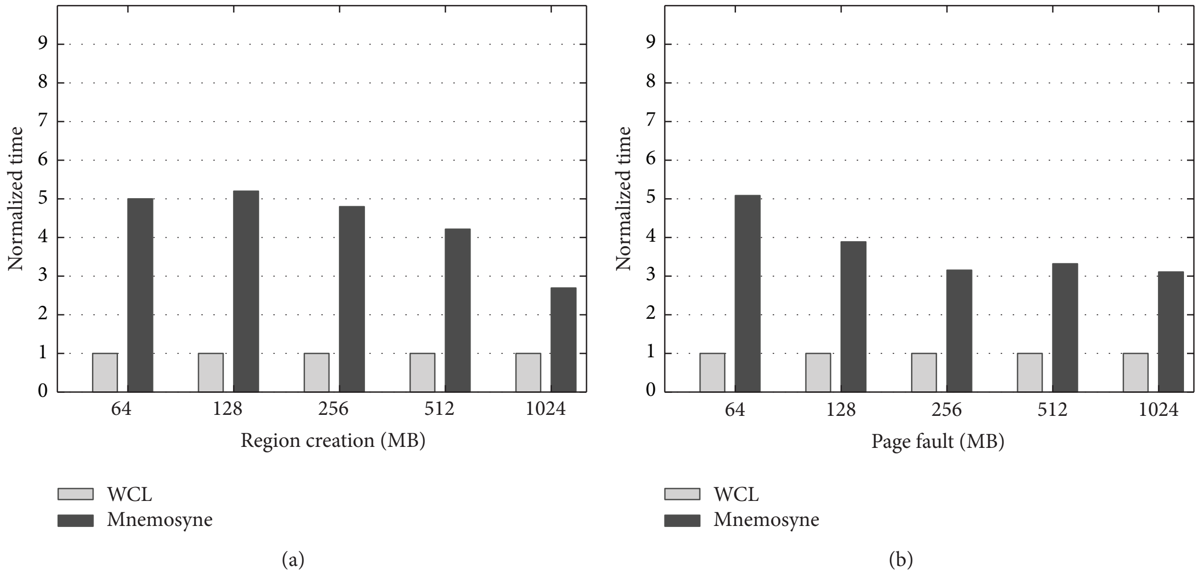


Figure 5: Tests of memory management.

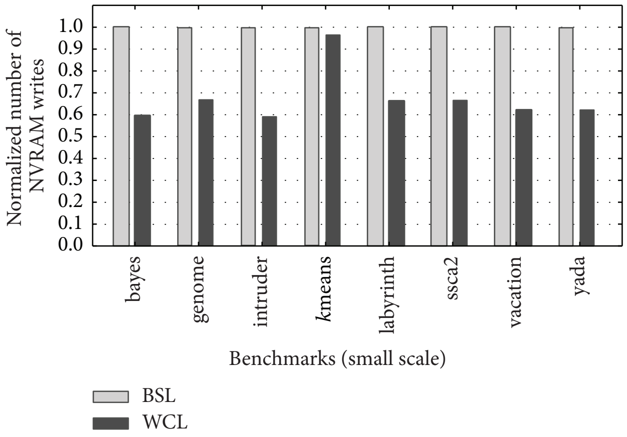


Figure 6: Comparison of total number of NVRAM writes (small scale).

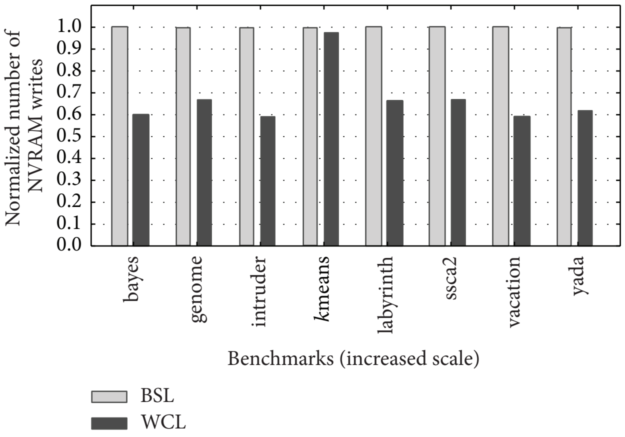


Figure 7: Comparison of total number of NVRAM writes (increased scale).

benchmark *k*means, which is also as expected according to Figure 2. Averagely, WCL can reduce the number of writes by 33%, which can greatly help extend the lifetime of NVRAM.

Figures 8 and 9 show the normalized runtime of WCL compared with BSL. WCL achieves better performance by reducing writes to NVRAM. Averagely, WCL achieves a speedup of 11% over BSL. How many performance benefits WCL can get mainly relies on how much proportion the writes occupy in the program. As shown in Figure 10, the benchmarks genome, ssca2, vacation, and yada spend considerable proportions of time on NVRAM writing. Thus we can see obvious improvements in these benchmarks after reducing writes. Moreover, we can see that WCL is able to improve performance on both small and increased problem

Table 5: Total NVRAM writes ("−" and "+" mean small and increased problem scales).

Benchmarks	BSL	WCL
bayes	(−) 4701	(−) 2803
	(+) 3795	(+) 2269
genome	(−) 3251313	(−) 2175511
	(+) 6552450	(+) 4384360
intruder	(−) 56129379	(−) 33188986
	(+) 112200870	(+) 66342596
*k*means	(−) 36044910	(−) 34843101
	(+) 337379526	(+) 328728706
labyrinth	(−) 136002	(−) 90668
	(+) 558789	(+) 372526
ssca2	(−) 33344904	(−) 22229910
	(+) 134173668	(+) 89449089
vacation	(−) 19708551	(−) 12316239
	(+) 104356923	(+) 62029785
yada	(−) 32814681	(−) 20292095
	(+) 78473364	(+) 48613975

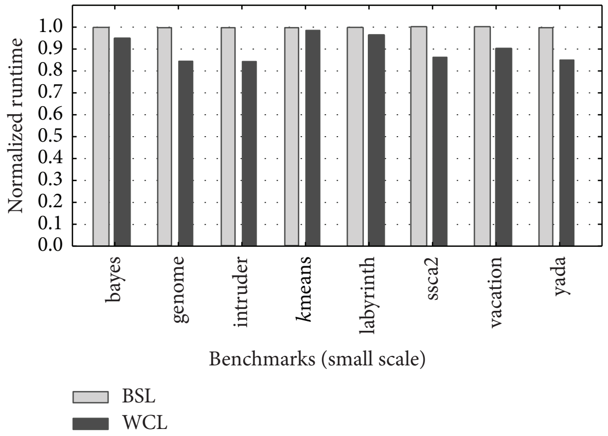


Figure 8: Comparison of runtime (small scale).

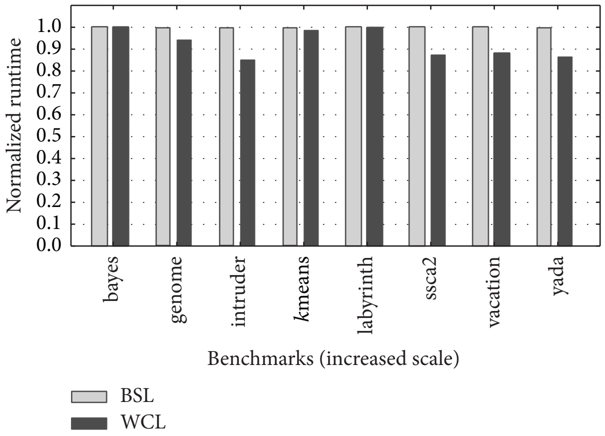


Figure 9: Comparison of runtime (increased scale).

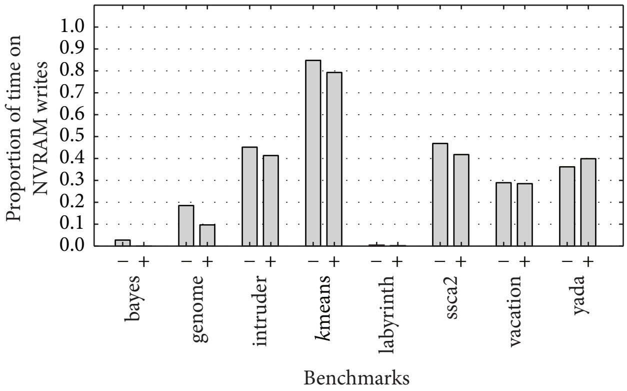


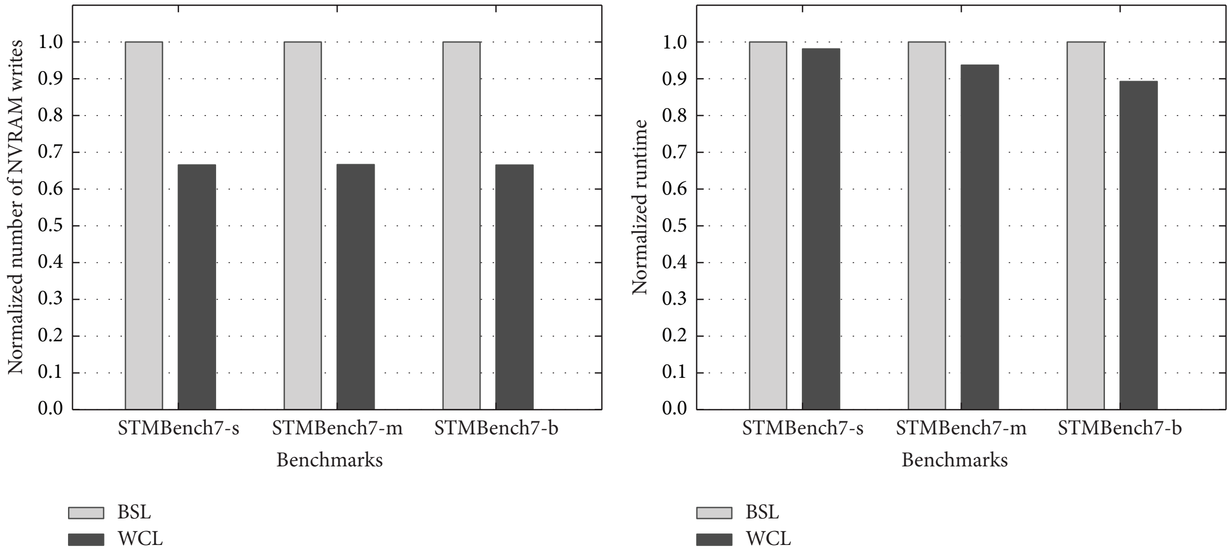
Figure 10: Proportion of time on NVRAM writes.

scales. For the benchmark genome, WCL behaves better on small problem scales than increased problem scales. This is because when we increase the problem scale in genome, it does 2 times more writes while the runtime increases by 4 times.

*k*means also spends large proportion of time on NVRAM writing. However, WCL cannot reduce much of the writing in *k*means as we discussed before. For the benchmarks bayes and labyrinth, although WCL reduces large proportion of writes, the writes only account for very small part of the execution.

5.2.3. Results of STMBench7. Figure 11 shows the comparison of total NVRAM writes and runtime. We can see that WCL can reduce a large proportion of NVRAM writes over BSL in different program scales and data structure sizes. The statistics are shown in Table 6. STMBench7 uses a lot of data structures for graph processing or data indexing. These data structures rely heavily on pointer operations. Usually for a 64-bit pointer, it is less likely to be modified more than 32 bits because most data are allocated nearby. Averagely, WCL performs 34% less than BSL and achieves 7% speedup.

Finally, the software overhead of write-combining is shown in Figure 12. Generally our mechanism incurs ignorable overhead. The overhead is mainly determined by the number of writes in every transaction, as our combining mechanism needs to perform a comparing operating for every writes in the write buffer. Generally, this is very simple and fast. Moreover, most read operations for comparing are cached by cache because they have been just accessed in this transaction. Compared with previous data compress work, our work actually leverages the fact that, for each value, we have two persistent copies of the value: (1) one resides at the original place which is waiting for being updated and (2) another one is in the persistent log. As these two versions of value are both persistent, we actually have data abundance here. Thus we can easily take advantage of these abundant persistent data to reduce the log, while, for previous data compress work, it has only one version of data and thus the problem is to achieve compression without losing any information, which would incur more overhead on discovering the abundance in data.



(a)

(b)

Figure 11: Comparison of total writes and runtime on STMBench7.

Table 6: Total NVRAM writes (Un: unmodified writes; Half: half writes).

Benchmarks	BSL	WCL	Un	Half
STMBench7-small	915573	609386	0	306187
STMBench7-medium	3634074	2422277	0	1211797
STMBench7-big	5803089	3860895	0	1942194

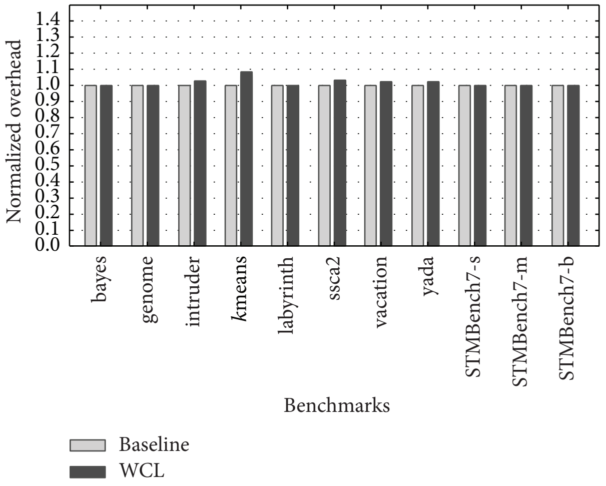


Figure 12: Normalized software overhead.

5.2.4. Discuss. How much NVRAM writes WCL can reduce lies on the lucky cases of unmodified write and half write. In these cases we can leverage the fast-read-slow-write feature of NVRAM to gain some benefit. However, it is not totally all lucky cases we are counting on.

Indeed, for the case of unmodified writes, it is lucky that we can have it in a program; the more, the better. However, as shown in Figure 2, only few applications perform only small parts of unmodified write. It is good but we are not counting on it to reduce large amount of NVRAM writes. What we are counting on is the case of half write. The half write mainly consists of two situations: (1) pointers; as discussed before, a 64-bit pointer is less likely to be modified more than 32 bits; this is because normally we allocate objects nearby; if a program does a lot of pointer operations such as using an iterator to traverse some data structures, the pointer is normally modified less than 32 bits; STMBench7 shows a good example as discussed before; (2) counters or indexes: a lot of programs heavily use counters or indexes as part of a loop or for statistics; these counters, although in 64 bits, are less likely to be modified much, leaving us the opportunities to reap the benefit.

Generally our mechanism incurs ignorable overhead. Compared with previous data compress work, our work actually leverages a natural fact that, for each value, we have two persistent copies of the value: (1) one resides at the original place which is waiting for being updated and (2) another one is in the persistent log. As these two versions of value are both persistent, we actually have data abundance here. Thus we can easily take advantage of these abundant persistent data to reduce the log, while, for previous data compress work, it has only one version of data and thus the problem is to achieve compression without losing any information, which would incur more overhead on discovering the abundance in data.

Above all, in the benchmark suit STAMP and STMBench7, WCL can effectively reduce NVRAM writes while maintaining the data consistency. WCL can help extend lifetime of NVRAM and improve performance by reducing writes to NVRAM.

6. Related Work

Data consistency in nonvolatile media has been in research for a long time. The maturing of NVRAM brings new opportunities to this. NVRAM has been used as a union of buffer cache and journaling to move the journal of file system from secondary storage to nonvolatile main memory [9], which could reduce the writes to secondary storage and improve performance. Similar research has been conducted for database systems [20, 25, 26]. However, these studies do not tackle the write-twice problem [27]. The write-twice problem has been noticed and addressed in log-structured file system [28, 29], which is not widely adopted due to large data indexing overhead.

In order to reduce writes to NVRAM to extend its lifetime, Flip-N-write [15] and Coset [16] also adopt a read-and-compare scheme and rely on special coding algorithms to reduce writes to NVRAM. Similarly, our work leverages NVRAM's feature of byte-addressable and asymmetric read-write to reduce the NVRAM writes. However, we are based on different layer. Flip-N-Write is at the hardware layer, which makes it adopt a general and blind way to just test and flip, while we are at the runtime system layer which directly serves the upper applications. Our idea is based on the fact that there must be a substantial proportion of half writes in all 64-bit updates in an application. This is because that the pointers, counters, and indexes are massively used in applications and they are less likely to be modified much. Moreover, Flip-N-write is at the architectural level which needs modifications on the underlying hardware. Finally, our work can be combined with the previous hardware work to achieve better performance.

The logging [30, 31] mechanism incurs overhead for both failure-free and failure cases. Some work [32, 33] introduces architectural mechanism to leverage residual energy in the system to flush volatile states into NVM and thus achieves zero overhead for failure-free operations. These studies are orthogonal to ours.

Previous work Shortcut-JFS [34] leverages the byte-addressability and adopts a shadow-update mechanism to reduce the write amount to PCM. It is another way of reducing write amount to PCM and it focuses on the block-based file system. Our work focuses on offering support for programming directly on PCM. The goal is similar, but the idea and the application are different. In our work, actually, we found some natural information of data abundance and we developed an easy and efficient way to leverage this information, as discussed in Section 5.2.4. For data compression work, a main challenge is to discover the abundant information. This point is new and different from previous work. While another data compress work [35] leverages the byte-addressability of NVRAM to store a compressed delta journal in it, the delta journal is generated by comparing the new block with the old one. For the data compression phase, it actually faces a 0-1 block with no semantics information. The overhead of its data compression is very likely to be more than us. Moreover, it is based on blocks. The block with modification ratio less than a threshold will be updated in the traditional way. Compared with it, our work is fine-grained.

Mnemosyne [4] and NV-Heaps [3] both introduce lightweight programming interfaces to support allocating and processing durable in-memory data structures based on NVRAM. They both adopt logging to maintain the consistency of persistent data. However, they do not show attention to the consistency-maintaining overhead of write-twice problem. Moreover, they rely on traditional file system and memory mapping to manage NVRAM pages, which may also incur overhead [36]. They also offer some features that we do not have. The NV-heap offers some higher language level support such as safe pointer. Mnemosyne offers a high-performance raw word log (RAWL) to reduce the memory fence at commit time.

For torn writes when some writes are failed to go to memory, traditional detection method [37] is to perform a read-after-write. It can be done in both hardware and software. In our system, it could easily be checked because all the writes to NVRAM happen just at the commit time of every transaction.

Traditional file system is heavy for NVRAM due to its block layer. Many studies tried to optimize file system to reduce the block layer (PMFS [23], Aerie [38], BPFS [36], and SCMFS [39]).

7. Conclusion

This paper proposed an optimized write-combined logging to reduce the writes to NVRAM log without sacrificing the information to maintain the data consistency in NVRAM. We leverage the fast-read and byte-addressable features of NVRAM to perform a read-and-compare before performing writes. This can help us reduce writes to NVRAM to extend its lifetime and improve performance. We tested our system on the benchmark suit STAMP and STMBench7. Experiments show that our system is able to reduce 33% of writes and improve performance by 11% in STAMP and reduce 34% NVRAM writes and improve performance by 7% in STMBench7.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61272142, 61402492, 61402486, and by the State Key Laboratory of High-end Server & Storage Technology (2014HSSA01). Mikel Luján is supported by a Royal Society University Research Fellowship and funded by UK EPSRC Grants DOME EP/J016330/1 and PAMELA EP/K008730/1.

References

[1] A. Badam, "How persistent memory will change software systems," *Computer*, vol. 46, no. 8, Article ID 6521316, pp. 45–51, 2013.

[2] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS '13)*, p. 2, USENIX Association, 2013.

[3] J. Coburn, A. M. Caulfield, A. Akel et al., "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, 2011.

[4] 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, 2011.

[5] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*, pp. 287–293, 2014.

[6] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[7] I. Koltsidas, R. Pletka, P. Mueller et al., "PSS: a prototype storage subsystem based on PCM," in *Proceedings of the 5th Annual Non-Volatile Memories Workshop (NVMW '14)*, La Jolla, Calif, USA, March 2014.

[8] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," Tech. Rep., 2006.

[9] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pp. 73–80, Amsterdam, The Netherlands, December 2013.

[10] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

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

[12] H. Yoon, N. Muralimanohar, J. Meza, O. Mutlu, and N. P. Jouppi, "Techniques for data mapping and buffering to exploit asymmetry in multi-level cell (phase change) memory," SAFARI Technical Report 2013-002, 2013.

[13] 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.

[14] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 14–23, ACM, June 2009.

[15] S. Cho and H. Lee, "Flip-N-write: a simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro '09)*, pp. 347–357, ACM, New York, NY, USA, December 2009.

[16] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, "Coset coding to extend the lifetime of memory," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA '13)*, pp. 222–233, Shenzhen, China, February 2013.

[17] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS '13)*, pp. 29–40, Boston, Mass, USA, May 2013.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: stanford transactional applications for multi-processing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '08)*, pp. 35–46, Seattle, Wash, USA, September 2008.

[19] 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.

[20] T. Wang and R. Johnson, "Transaction logging unleashed with NVRAM".

[21] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[22] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 237–245, ACM, February 2008.

[23] S. R. Dulloor, S. Kumar, A. Keshavamurthy et al., "System software for persistent memory," in *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[24] M. J. Carey, D. J. DeWitt, and J. F. Naughton, "The 007 benchmark," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 12–21, 1993.

[25] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[26] S. G. J. Xu, B. He, and B. C. H. Hu, "PCMLogging: reducing transaction logging overhead with PCM," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, October 2011.

[27] F. Douglis and J. Ousterhout, "Log-structured file systems," in *Proceedings of the 34th IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers (COMPCON Spring '89)*, pp. 124–129, 1989.

[28] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 102–107, 2006.

[29] J. Ousterhout and F. Douglis, "Beating the i/o bottleneck: a case for log-structured file systems," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[31] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE '11)*, pp. 1221–1231, IEEE, Hannover, Germany, April 2011.

[32] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey III, "Procrastination beats prevention," Tech. Rep. HPL-2014-70, HP Labs, 2014.

[33] D. Narayanan and O. Hodson, "Whole-system persistence," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 401–410, 2012.

[34] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-JFS: a write efficient journaling file system for phase change memory," in *Proceedings of the 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*, pp. 1–6, IEEE, San Diego, Calif, USA, April 2012.

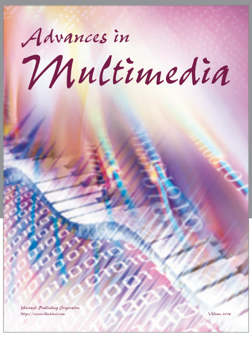
[35] J. Kim, C. Min, and Y. I. Eom, "Reducing excessive journaling overhead in mobile devices with small-sized NVRAM," in *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE '14)*, pp. 19–20, Las Vegas, Nev, USA, January 2014.

[36] J. Condit, E. B. Nightingale, C. Frost et al., "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pp. 133–146, ACM, October 2009.

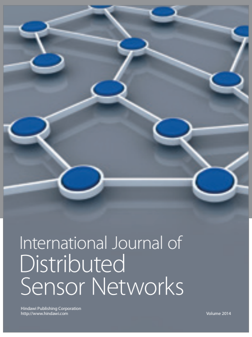
[37] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 3–14, 2010.

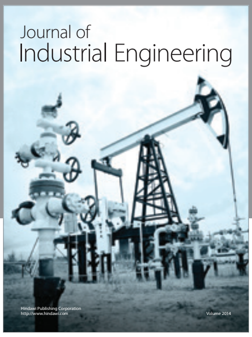
[38] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: flexible file-system interfaces to storage-class memory," in *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

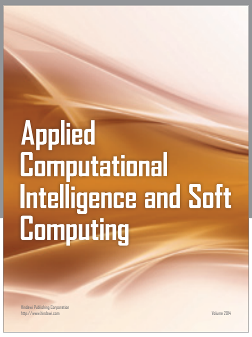
[39] X. Wu and A. L. N. Reddy, "SCMFS: a file system for storage class memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, p. 39, ACM, Seattle, Wash, USA, November 2011.

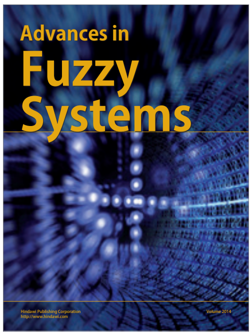

Advances in
Multimedia

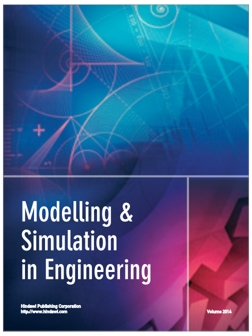
The Scientific
World Journal

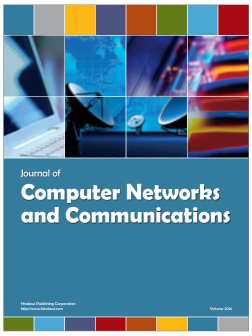

International Journal of
Distributed
Sensor Networks

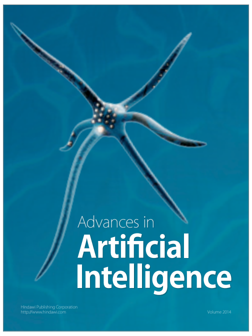

Journal of
Industrial Engineering

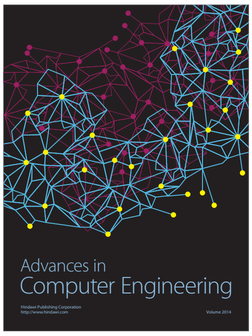

Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

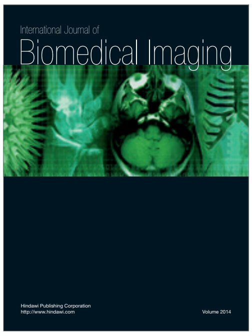

Modelling &
Simulation
in Engineering

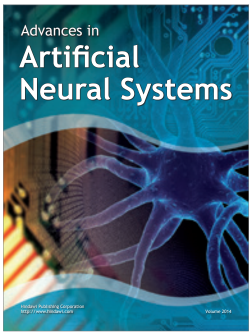

Journal of
Computer Networks
and Communications

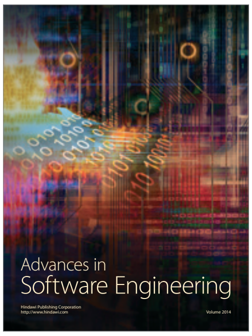

Advances in
Artificial
Intelligence


Advances in
Computer Engineering

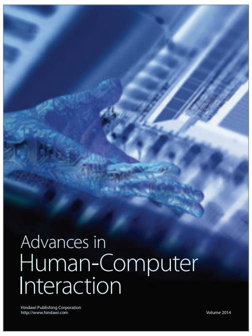
International Journal of
Computer Games
Technology

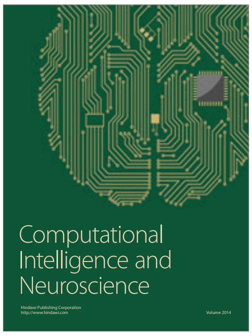

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering