SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection

Pradeep Kumar H. Howie Huang
The George Washington University
Email: {pradeepk, howie}@gwu.edu

***Abstract*—For many big data applications, non-volatile memory (NVM) can be utilized to store and process the data at faster rate due to its high-performance, scalable technology, DRAM-like interface and low energy requirement. NVMs such as phase change memory and memristor allow the applications to store persistent data directly in memory, and avoid data serialization and deserialization. However, NVM, like volatile memory, is susceptible to data corruption from software bugs. In this work, we present a paradigm shift from current process-based page-protection to a thread-based solution specifically designed for NVM. We have developed SafeNVM, a reliable NVM store to support application-specific data formats. SafeNVM will enable the NVM to provide strong data protection while delivering high performance access. We propose a simple hardware change in TLB and page table entry and exploit bound checking inherent to swizzled pointers. We show that SafeNVM is reliable against a collection of stray writes and the cost to achieve such protection is small.**

***Keywords*-Big Data; NVM; Data Reliability;**

I. Introduction

Big data is characterized by five Vs: volume, velocity, variety, veracity and value. To deal with the velocity, which is defined as faster arrival rate of big data, non-volatile random access memory (NVM) is well positioned due to its excellent scalability and DRAM like performance [1]–[4]. Prior works in big data [5] [6], HPC [7]–[9], and big data on HPC systems [10], [11] have identified NVM as an important hardware component, and envision that NVM will become a natural solution for achieving the performance required in such systems due to its memory like performance and lower energy requirement. Hence, big data technologies such as file-system [10], key-value store [12], graph processing [13], [14] and HPC technologies such as burst buffers [8] will get benefited from the NVM.

Examples of such memory technologies are Phase-Change Memory (PCM), STT-RAM [15], Memristor [16] etc. They can be integrated with the existing DRAM memory controllers [3] [4] [2] and thus can be addressed as load-store devices as the current DRAM memory. Hence, NVM can be used as RAM disk to make a drop-in replacement of disks [17] or as a kernel-mapped non-volatile memory to store the serialized data at kernel address space utilizing read/write system call interfaces in the existing file systems [18] [19] [20]. On the other hand, NVM can also be utilized to store persistent data in application-specific format directly at user address space [21] [22] to achieve better application performance. It provides better energy efficiency [23] than the kernel-mapped NVM. We refer them as non-volatile memory store (NVM store) in this paper.

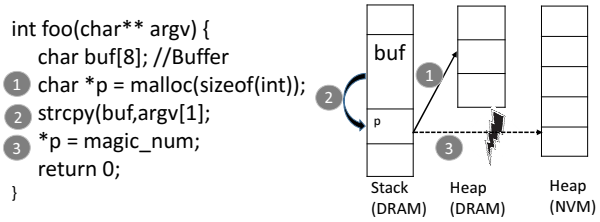


Figure 1: Motivation example on how NVM data could get corrupted due to buffer overflow in NVM store

Unfortunately, two NVM features, random access and byte addressability, introduce the risk of NVM corruption from a spectrum of application or device driver bugs such as buffer-overflow, dangling, uninitialized or manufactured pointer, etc. Such bugs are very common in the applications as well as device drivers and can potentially corrupt any memory location [24], [25]. For example, 17% of the application vulnerabilities in 2016 were due to buffer overflow errors alone according to United States Computer Emergency Readiness Team [26].

For NVM store, such bugs pose serious data reliability challenges when a stray write to NVM could corrupt persistent data, leading to loss of valuable data. Figure 1 illustrates that a volatile memory buffer *buf* depending upon the variable length argument *argv[1]* can corrupt the adjacent pointer *p* to make it point to NVM upon buffer overflow error. This pointer can corrupt the NVM data during the write.

Note that this example is very similar to prior works on cyber attacks due to memory corruption issues [27]. In such works, cyber attack is observed when pointer *p* is a control flow pointers such as function pointers, return address etc. In our case, pointer *p* is a data pointer which can point to an NVM address after the buffer-overflow or any other corruption. We re-visit all the existing mechanisms from NVM persistent data reliability perspective in Section II, however, we present a high-level summary in Table I.

Traditionally, memory corruption affects the volatile data in memory while the persistent data is safe in disk storage. Isolated from memory address space, disk-based storage

978-1-5386-1996-4/17 $31.00 © 2017 IEEE
DOI 10.1109/BigDataCongress.2017.18

Table I: Comparison of different Proposals

Proposal Name	Description	Issues
Linux/mprotect	NVM pages change from read-only to read-write	High overhead due to TLB-Shootdown
PMFS [19]	NVM pages change from read-only to read-write momentarily using CR0.WP	1.Interrupt and context-switching are disabled 2.Kernel-mapped only
PMBD [17]	NVM pages mapped privately during each read-write	1.Interrupt and context-switching are disabled 2.Kernel-mapped only 3.Write-window for many threads
Mnemosyne [21]	None	Reliability not covered
NV-heaps [22]	None	Only a subset are covered
WIT [28]	Allowing pointers to points-to-set	1.Memory/CPU overhead 2.No Safety against escaped dangling pointer.
SafeNVM (Proposed)	A thread momentarily gets write-permission to needed NVM pages	None

interacts with the OS and applications through block-based interfaces only. Hence, disk storage is less susceptible to application or device-driver memory corruption bugs. However, as NVM is addressed in the same way as DRAM, the isolation between the volatile temporary data and persistent data can no longer be ensured. The existing page-protection mechanisms do not provide the right granularity to safely write the data to NVM, e.g., the linux/mprotect technique only provides process-level page protection where all the threads of the process has the same page-permission, and causes severe application performance degradation due to TLB-shootdown [17].

Memory safety techniques like softbound [29], Memsafe [30], CETS [31], WIT [28] etc perform run-time check which introduce high overhead, hence they are deployed only in situations where safety is a primary concern [30]. However, unlike these works which concentrate on each memory pointers' bounds or points-to set, PMFS [19] and PMBD [17] propose time-window for data-access which effectively restricts the executing threads of all kernel-modules. Unfortunately, they need to disable interrupts and context-switching which affect the working of the system as applications and devices depend on such mechanisms.

Contribution: SafeNVM improves this time-window concept, i.e. allowing only the legitimate thread to write on a group of NVM pages within the interval. To this end, we separate the NVM and volatile memory, and propose a new thread-level page-protection for NVM that by default restricts the write on NVM by all the threads. Write access is gained using new hardware instructions. We then propose an application-specific persistent object store that takes advantage of inherent bound checking present in mandatory deswizzling of swizzled persistent pointer during NVM write, thus avoids additional overheads. Figure 2 presents an overview of SafeNVM architecture.

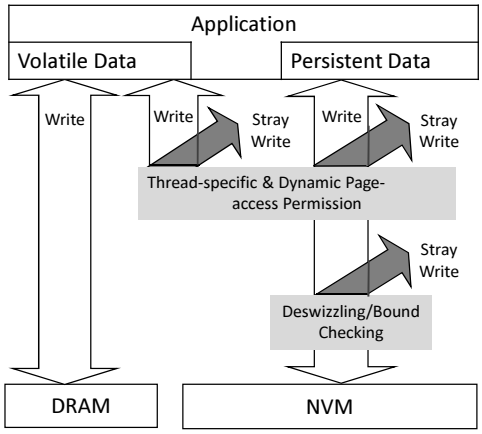


Figure 2: Data safety overview in SafeNVM. Modification in volatile data can not corrupt the persistent data due to thread-level page protection. In-built bound checker present in deswizzling process of persistent pointer saves a persistent object to be corrupted by another persistent object modification.

In SafeNVM, application do not need to check the bound of volatile pointers as hardware protection guarantees that the memory corruption bugs would never be propagated to NVM. While, the bound checker during swizzling protects one persistent object from other persistent object modification. We argue in Section III-A that the data proection provided by SafeNVM is equivalent to disk based data protection. We believe this approach is well-suited for multi-threaded applications running in a multi-core processors.

A kernel version of SafeNVM uses the proposed hardware changes along with NVM specific file system to provide an equivalent reliability that is easier to deploy as it does not involve application change. Note that the focus of this work is to protect persistent data stored in the NVM, rather than volatile data stored in DRAM or NVM. Consistency across threads and failures are not the focus of this work.

The rest of the paper is organized as follows: Section II presents motivation and related work. Section III presents the reliability model of SafeNVM and its architecture including hardware and software proposals. The evaluation is presented in Section IV, and the conclusion in Section V.

II. Related Work and Motivation

We discuss two types of related work here. Memory safety related works concentrate on the memory corruption, which motivates the need of hardware changes that we propose. While persistent object related works concentrate on avoiding the serialization/deserialization work between memory and disks, and inspire the software proposal that take advantage of swizzling/deswizzling technique as presented in these works.

A. Memory-safety

Type-safe languages like Java eliminate memory errors, but their run-time environment are written in type un-

safe language like c/c++, hence they can not be assumed memory corruption safe. Memory safety techniques like Memsafe [30], softbound [29] and CETS [31] does runtime check for every pointers. There have been proposals on control-flow integrity like CFI [32], code-pointer integrity [33], write-integrity testing [28], data-flow integrity [34]. WIT presents little less overhead compared to others, but it also has memory overhead apart from the performance overhead. The overhead is mainly because of runtime bounds check on pointer dereference to write only objects in its own approximate points-to set. Though, it is minimized using static analysis. However, WIT does not deal with temporal errors, the memory corruption could be possible through escaped dangling pointer [35].

Also, all of these technique do not distinguish between application's volatile and persistent data, and hence the technique, say WIT, would be applied on whole of the memory set instead of just NVM space. There is no way to restrict the technique for the safety of persistent data only. Also, with high capacity NVM added for application use, the number of memory objects would increase substantially, thus degrading the performance of these proposals.

Mnemosyne [21] and NV-heaps [22] provide high-performance, application-specific NVM store where reliability is not the main objective. Hence they do not address the issue of NVM data-corruption due to memory corruption bugs. Mapping the NVM pages as read-only in application's address space, and then selectively using mprotect system call to enable a small write-window can provide a reliable write method to store data in NVM, but it causes severe application performance degradation due to TLB-shootdown [17].

Alternately, NVM pages can be mapped in the kernel-space as read-only and writes can be enabled momentarily by using the CR0.WP bit as in PMFS [19]. However, CR0.WP flag is not saved during interrupt handling, so interrupts and context-switching are disabled in PMFS. This affects the performance of the overall system as context-switching is an important part of it, and the performance of devices degrade as they depend on interrupts. PMBD [17] suggests to use the private page-table mapping of NVM pages during each read and write call. Similar to PMFS, it disables the interrupts and context switching, and provides a write-window for many threads.

However, unlike prior works [28], [32], [34] which concentrate on each memory pointers' bounds or points-to set, the time-window concept from PMFS and PMBD for data-access effectively restricts the executing threads of all kernel-modules (say a device-driver) from write-access during the read window. Despite their shortcomings, the idea is well-suited for multi-core systems. Unfortunately it cannot be implemented for userspace-mapped NVM store which actually provides better performance and energy efficiency [23]. SafeNVM takes inspiration from the time-window concept.

B. Persistent Objects and Swizzling/Deswizzling

Persistent memory stores such as Objectstore [36], Thor [37], Texas [38], QuickStore [39], Persistent Java [40], SoftPM [41] allows application specific data-structure to be stored directly in block devices, hence pointer specific data structures can directly be stored in the disks. When using such systems, application works with a *persistent pointer* which can be think of as a specific object handler which behaves like a memory pointer, thus avoiding the need of serialization and deserialization when data moves to and fro memory and disks.

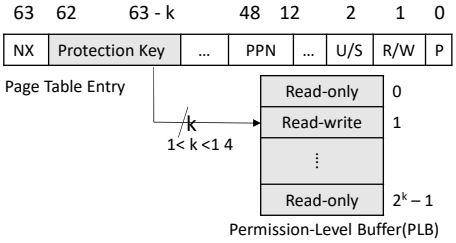
Since, objects could be loaded at any virtual memory address of the application, persistent pointers are created and stored as swizzled pointers. So, whenever a read or write is required deswizzling technique is used which converts the swizzled address to the actual address using few look ups and pointer arithmetic. Objectstore [36] utilizes page mapping and tag-table metadata to track all the persistent pages and objects in the page respectively, and uses these metadata for deswizzling the pointers present in the page.

Other systems [37]–[41] depend directly on such page mapping and tag-table structures or provide similar mechanisms. E.g. Texas [38] maintains a page mapping information to do the pointer swizzling. However, it also tracks all the pointer in the pages to reserve virtual pages in advance, that are pointed to by pointers present in the current page. Then, it relies on page-fault to load the actual page in the virtual memory.

Bound Checker in Deswizzling. Designed for disk-based storage, these solutions always maintain two copies of data: one in volatile memory and another in the disk during application run. For NVM, ideally one should only need to store one copy of data. Nonetheless, pointer swizzling/deswizzling is the necessary requirement to store the application-specific persistent data in NVM, without which data is tied to a fixed virtual address of one particular application. Hence sharing becomes a problem such as in [21]. It is the deswizzling process that resembles close enough to a bound checker, thus the arithmetic operation in the deswizzling is also used as bound checker in SafeNVM.

III. SafeNVM

SafeNVM is an object based NVM store as shown in Figure 4. It provides data reliability equivalent to disk-based system to support application-specific data-store in NVM such as databases, persistent key-value stores etc. It has two parts, a hardware proposal related to page-table and TLB structures that provides thread-level page protection. And, the persistent object store which involve design of persistent pointer, swizzling/deswizzling and bound checker, and memory management. At the end, we discuss kernel variant of SafeNVM which provides equivalent reliability



a) New Page-table and Permission-level Buffer

Instruction Name	PL bit	Action	Comment
set_write_access(Protection Key)	0=>1	Executing thread only gets NVM write-access	Permission stays during context switch
clear_write_access(Protection Key)	1=>0	Executing thread releases NVM Write-access	Read-access remains with all the threads

b) Proposed X86_64 Instruction

PL	R/W	Access Rights
R	R	read-only
W	R	Read-write

c) Page-access right calculation

Figure 3: Page table (and TLB) modification and permission level Buffer (PLB) proposal. Executing thread has to only change the PLB values using new hardware instruction to gain write permission on read-only pages. Changing PLB values does not necessitate any TLB-shootdown.

and is a stand-in replacement of current file-system based interfaces.

A. Reliability Model

We observe that with the advent of NVM store, the applications' address space or kernel space contain two types of data: volatile data in the DRAM and persistent data in the NVM. We compare the data safety of NVM store with disk-based storage system which is de-facto standard of persistent data-reliability, and list two major reliability differences:

First, isolated from memory address space, disk-based storage interacts with the OS and applications through block-based interfaces. Thus memory corruption bug can only corrupt the application's volatile data, not the persistent data in the disk. However, as NVM is addressed in the same way as DRAM, the isolation between the volatile temporary data and persistent data can no longer be ensured in the NVM store. The write to both data are through the CPU's load/store interfaces. Thus the same bug can corrupt the volatile as well as persistent data in an NVM store.

Second, File-system inode structure provides the required bound-checking to persistent data write to save one file modification from a stray-write in another file modification. E.g. in the write() system call, if user provides an out-of-bound offset, the inode data-structure find out the invalid offset, and protects the file. This bound-checking ensures that the persistent data modification remain isolated with other file modification. The same file-system inode technique could be employed to save a persistent NVM file data from another persistent file data modification bug in case of kernel-mapped NVM store. However userspace-maaped NVM store doesn't employ such technique and are prone to persistent data corruption by another unbounded persistent write.

SafeNVM Model. SafeNVM forces the update through an protected interface, thanks to the thread level page protection technique, which is equivalent to a block interface. Thus memory load-store interface cannot corrupt this data even in presence of memory corruption bugs. And, inherent bound checking present in deswizzling process is equivalent to bound check performed by file-system inode structure. Thus the reliability guarantee provided by SafeNVM is equivalent to a disk based system.

B. Protection Key Bits and Permission-Level Buffer

We present a lightweight page-protection method of NVM pages applied to each threads individually. We keep the isolation of NVM modification from volatile data modification so that threads modifying the volatile data should not be able to modify the persistent data. At the same time, applications can read the persistent data from NVM in the same way as they read it from DRAM.

To achieve this, we divide the NVM to groups of pages called *Protection Group* and map them as read-only which maximizes the protection against any memory corruption caused by threads modifying volatile data. For legitimate modification of the persistent data, only the modifying thread will gain the permission to do so using a new set of instructions within one protection group. This information should persist even after a context switch. Once the modification is completed, the permission of the modifying thread falls back to read-only. The access permission for volatile memory pages remains the same.

We propose a simple hardware change to achieve NVM page-protection model discussed above for SafeNVM. Specifically, we add *protection-key* (PK) bits in the page-table entry as shown Figure 3(a). If different NVM pages use the same key, they will be part of the same protection group. The corresponding TLB structure will also be modified to accommodate the PK bits. We then need to add one bit called *permission-level* (PL) corresponding to each PK entry in a new structure *permission-level buffer* (PLB). For example, if we use 6 bits for PK bits then the PLB size is 64 bits.

We propose two instructions to set and clear the PL bits to get and release write-permission respectively for group of pages having same PK bits (Figure 3(b)). The page permission will be calculated based on the user/supervisor (U/S) bit, the read/write (R/W) bit of the page table entry and the PL bit. The PL bit will be selected based on key stored in the PK bits. When the PK bits are clear, the PL bit is ignored and the existing technique is used to determine the page access rights. But when PK bits store some protection keys, PL bit is used for access rights calculation as shown in

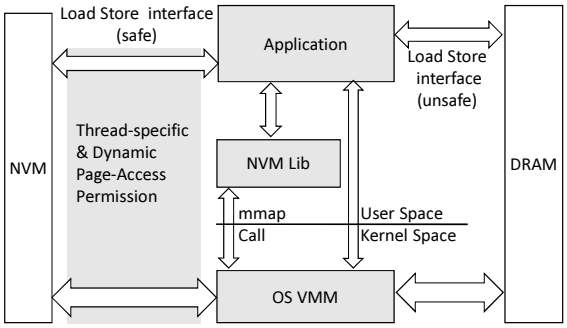


Figure 4: Userspace-mapped SafeNVM. Gray boxes show proposed changes.

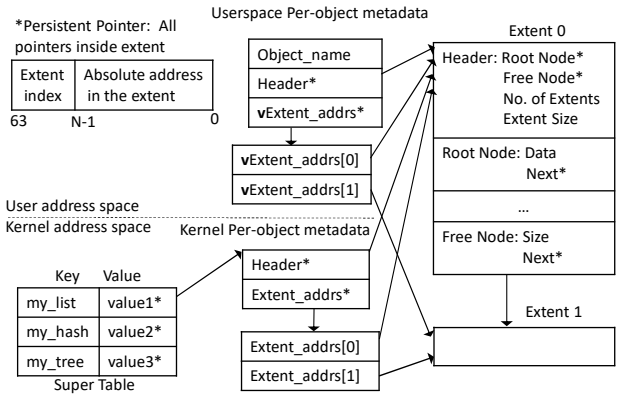


Figure 5: Persistent pointer (top left corner) and Object-layout in NVM as per userspace-mapped SafeNVM. Super Table is pointed by a fixed location in NVM (much like a super block in file system context) while per-object metadata is similar to a file inode structure.

Figure 3(c). Specifically, the thread can modify some read-only memory pages whose corresponding PL value is set.

PLB structure is part of the thread context-switch, so it needs to be saved during context switch, thus the thread gets the same write-permission when it resumes its work. Also, the modification of the PL bit of PLB does not necessitate a TLB-shootdown, while PK bits are set during page table entry creation of a virtual NVM page. Thus altering the permission of pages for a selected thread is very fast operation and we achieve a low-cost and high performing page protection mechanism.

Our design is similar to the protection-key mechanism in IA-64 processor architecture [42] that allows page permission to be altered by changing the protection-key register values. The key difference is that IA-64 allows only process-level page-permission change while our method allows thread-level page permission-modification.

Discussion. We believe that it is possible for the hardware vendors to incorporate the hardware changes in the future generation of multi-core processors. Intel recently incorporated memory protection extensions (MPX) in its processor and is committed to enhance the safety and security of the system [43]. Oracle has announced the silicon secured memory in SPARC M7 processor [44] that performs real-time check for each access. Vendors have modified the TLB and page-table structure in recent past such as process-context identifiers (PCID) or netsted page-tables to support virtualization. When the high capacity NVM will be used to store the persistent data, the reliability will become the main challenge. The vendors will be looking for new technique to provide reliability at lower performance overhead, and it will become the key distinguishing factor in the deployment. Our proposal would be another step in the right direction in this regard, and the proposed changes are also backward compatible.

C. Userspace-mapped SafeNVM

The hardware changes alone do not provide the equivalent data-reliability in SafeNVM that disk-based storage systems provide. Mapping the entire non-volatile memory to the application address-space may not be the wise thing to do because all the modules of the application get the access to the whole NVM address space. In case of any memory corruption bug in an application module, the corresponding thread that gains the write permission can corrupt other data-structures that belong to the same protection group.

To solve the issue, we utilize the inherent bound checking present in the pointer swizzling/deswizzling technique which is part of any object store. Figure 4 shows the high-level design of SafeNVM. It has two major components: the Kernel part and a userspace library (NVM Lib). Figure 5 shows the detailed design of persistent pointers and per-object metadata.

The kernel part resides inside OS virtual memory manager (VMM). It manages the NVM as a collection of extents (multiple of page size) using the NVM descriptor and handles the higher level of memory management such as extents allocation to the application data structures and mapping the allocated extents to the user address space. It also maintains a object table called *Super Table* where each object is maintained using a kernel specific *per-object metadata*. The super-table and the NVM descriptor are pointed from a fixed NVM location similar to super-block concept of file-system (not shown in Figure).

The NVM Lib, through its userspace per-object metadata, manages the memory allocation/deallocation within the extent. It interacts with the kernel using system call interfaces to create, load, destroy, extend and truncate the data-structure. It wraps those system call APIs in easy-to-use library APIs for application developers to hide the complexity of SafeNVM. Table II presents the proposed interfaces.

The format of the persistent pointer is shown in Figure 5. The N least significant bits of the pointer are stored as the address inside any extent with relative to the start of that extent, while the rest of most significant bits are reserved for index pointing to *vExtent_addrs* arrary. The value N is

Table II: API list of userpsace-mapped SafeNVM

Library API	
status	*create_object*(incore_pobj*, objname, flag)
status	*delete_object*(incore_pobj*, objname)
status	*load_object*(incore_pobj*, objname, flag)
void*	*deswizzle_ptr*(incore_pobj*, splptr_t)
splptr_t	*swizzle_ptr*(incore_pobj*, void*, extent_index)
splptr_t	*alloc*(incore_pobj*, size, void**)
void	*free*(incore_pobj*, splptr_t)
System Call	
status	*sys_create_object*(incore_pobj*, objname, flag)
status	*sys_delete_object*(incore_pobj*, objname, flag)
status	*sys_load_object*(incore_pobj*, objname, flag)
status	*sys_alloc_extent*(incore_pobj*, objname)
status	*sys_free_extent*(incore_pobj*, objname)

stored in *Extent Size* field of the header structure inside the *per-object metadata*. The value of *N* can be chosen independently for each object, and hence we can extend this design to support huge memory pages. For the current prototype, the extent size has been configured as 4KB only.

The allocated extents and objects are tracked using the per-object metadata. We maintain two slightly different copies of per-object metadata. Kernel-specific per-object metadata (pobj) is stored permanently in the NVM and contains the base physical address of the allocated extents, while application-specific userspace per-object metadata (incore_pobj) contains the base virtual-address of the allocated extents and is discarded when an application closes.

The advantage of using this per-object metadata is three-fold. First, the metadata is referred during each read-write of NVM to get deswizzled pointer, thus bound checking can be performed during each read-write to NVM free of cost. Second, we avoid the global page-mapping lookup each time a persistent pointer is deswizzled, as we have a local access to the base virtual address of each extent. Third, SafeNVM does not employ the tag-table and saves a lot of precious NVM, while page-mapping information is maintained at extent level, further removing memory pressure issues associated with page-mapping and tag-table.

D. Kernel-mapped SafeNVM

Kernel-mapped SafeNVM follows the ideas from file-system design. The file-system is written specifically for byte-addressable NVM, e.g., bypassing page-cache, sub-page write etc. The whole NVM is mapped into kernel-address space and data are stored in a format different than what an application uses. An example of such data serialization and deserialization can be seen in a linked list represented as a collection of data and pointers in memory that is saved on disk with the help of several metadata such as offset and index.

Userspace-mapped NVM store needs application changes to deploy. For quicker deployment, one can use kernel-mapped NVM store as it does not necessitate any application changes and existing file-system system call would be utilized.

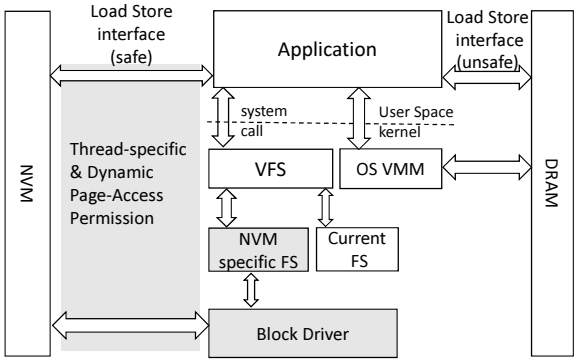


Figure 6: Kernel-mapped NVM store utilizes the hardware proposal for data reliability, while depends upon NVM specific file system design to serve files. Gray boxes show proposed changes.

Figure 6 shows the architecture of the kernel-space mapped NVM store. In this architecture, an NVM specific filesystem could be laid-out directly on the top of NVM [19]. It maps the entire persistent memory in the kernel-space and lays down a PM specific file-system on top of this. Application has to use the existing system call APIs to access the file-services. Alternately, NVM could be modeled as a block-based device and an existing filesystem could be laid-out on top of that [17].

We propose to utilize the proposed hardware changes to achieve the better reliability guarantee on top of the architecture proposed by [19] [17]. The proposed hardware changes are enough for the kernel-mapped NVM store to achieve the equivalent reliability of disk-based storage system along with either NVM specific or a general file-system.

IV. Experiments

The machine used for the experiments has a dual-socket of Intel Xeon CPU E5-2620 2.00 GHz with 6 cores each, and 32GB DRAM. We use the Ubuntu 12.10 server edition and the QEMU X86_64 emulator (version 0.14.1) to simulate all the hardware changes that we have proposed.

Our prototype has implemented one bit protection-key (PK) in the Linux kernel (version 2.6.38.8), and is installed on top of QEMU. We have implemented the proposed new page table structure to include the new PK bit and changes in the *mmap* system call to use this bit. The same changes are incorporated to the X86_64 TLB structure of QEMU. The permission-level (PL) bit is added to the EFLAGS register in the QEMU because the EFLAGS register is part of thread-context switch. The two newly proposed hardware instructions are simulated in QEMU.

We use RAMFS as well as DRAM based system to show the performance comparison with respect to SafeNVM. While we will show later that the worst performance is within an acceptable range, the QEMU-based simulation introduces some unnecessary overhead. We believe that a

hardware implementation will deliver a significantly better performance.

Protection Verification. To verify the effectiveness to catch stray writes, we have implemented a number of programs, shown in Table III, based on [45]. We have verified that SafeNVM is able to catch all the erroneous writes.

Table III: Cases of Stray Writes protected by SafeNVM

Case Number	Issues	Effect on NVM
CVE-2010-2160	Buffer Overflow	Data Corruption
CVE-2007-1211	Dangling Pointer	Data Corruption
CVE-2007-4000	Uninitialized Pointer	Data Corruption
CVE-2008-5187	Pointer Arithmetic	Data Corruption

Performance Verification. To measure the performance of userspace-mapped SafeNVM with respect to a RAMFS system, we choose to implement a linked-list based benchmark. This linked-list is implemented using the library API listed in Table II of *SafeNVM*. To compare the performance, the reference linked-list is implemented in DRAM and saved in a RAMDisk using RAMFS. Both implementations have the same memory allocation method so that the performance can be independent of memory allocation scheme. Each 4KB page is allocated using the mmap() system call and linked-list nodes are allocated from this page locally. The linked-list is a singly linked list of 131,072 nodes and each node has 128 byte data. Figure 7 shows the linked-list creation and traversal time. SafeNVM performs 3.6% better then RAMFS based system for linked-list creation, because in RAMFS we do serialization to RAMDisk.

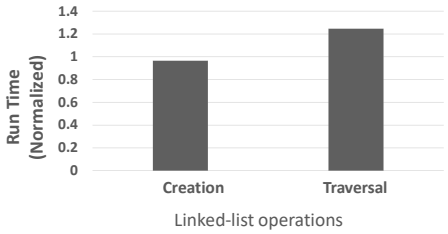


Figure 7: Normalized SafeNVM performance over RAMFS. For traversal, we ignore the deserialization cost for RAMFS and concentrate purely on traversal.

To measure traversal performance, we ignore the time taken by the RAMDisk to load the data in DRAM so that we can measure the overhead of deswizzling technique. The test shows that there is 24.5% performance degradation for SafeNVM, which is caused mostly by QEMU emulation. A real hardware implementation will minimize this overhead as those operations will be performed in hardware using fewer clock cycles.

Cost of Swizzling/Deswizzling. We also measured the swizzling/deswizzling prcoess overhead for a linked-list of 3.6 million nodes and for Redis [46] (a key-value store) operations. During linked-list creation all the pointers are swizzled, while during traversing the persistent pointers are deswizzled to get the actual addresses. Figure 8 shows this overhead for the linked-list. However, it should be noted that by using swizzling/deswizzling, we are avoiding the read and write from RAMFS and the serialization process. Hence, SafeNVM performs a lot better than the RAMFS based implementation where creation and traversal performs 48% and 73% better than RAMFS.

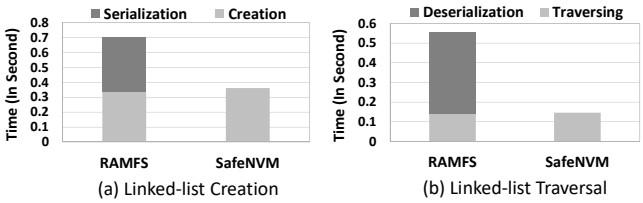


Figure 8: Swizzling/deswizzling technique and bound checker performance analysis on a linked List

If we ignore the serialization and deserialization of RAMFS, the resulting operations are purely DRAM operations. In this setup, creation and traversal performs 7.4% and 3.4% worst than DRAM based linked-list. We also compared the same operations performance difference for redis linked list using redis-benchmark provided by redis. Redis LPUSH operation is for creating the list of 10 million nodes, while redis LRANGE operation is like traversal, getting specified number of nodes (e.g. 100 in LRANGE_100) from the list. In this case, the performance difference is less than 1%, as shown in Figure 9. This is because redis-benchmark simulates redis key-value store in server mode, while the clients are running locally, sending those operations to server. Thus the swizzling/deswizzling overhead hardly affects the overall stack. Clearly, the software cost is well within an acceptable range.

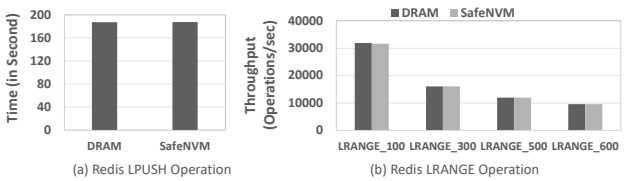


Figure 9: Swizzling/deswizzling technique and bound checker performance analysis on Redis key-value store (linked-list)

V. Conclusion

We presented SafeNVM, a highly reliable userspace-mapped NVM store to store the persistent data in application-specific format in the emerging NVMs, and shown that the cost to achieve the reliability is small. We also presented a highly reliable kernel-mapped NVM store as a replacement to block-interface based storage.

Acknowledgment

We thank the anonymous reviewers for their suggestions. This work was supported in part by National Science Foundation grants 1350766, 1320226, and 1618706.

REFERENCES

[1] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. c. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. h. Chen, H. l. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, 2008.

[2] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[3] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[4] 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 (ISCA)*, 2009.

[5] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[6] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561–2573, 2014.

[7] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *Computing in Science & Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[8] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[9] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[10] N. S. Islam, M. Wasi-ur Rahman, X. Lu, and D. K. Panda, "High performance design for hdfs with byte-addressability of nvm and rdma," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016.

[11] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "Can non-volatile memory benefit mapreduce applications on hpc clusters?" in *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, 2016.

[12] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems." in *FAST*, vol. 15, 2015, pp. 167–181.

[13] P. Kumar and H. H. Huang, "G-store: High-performance graph store for trillion-edge processing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[14] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[15] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. Ghosh, J. Lu, S. Poon, M. Stan, W. Butler, S. Gupta, C. Mewes, T. Mewes, and P. Visscher, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, June 2010.

[16] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, May 2008.

[17] F. Chen, M. P. Mesnier, and S. Hahn, "Flashy prefetching for high-performance flash drives," in *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST)*, June 2014.

[18] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[19] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the 9th European Conference on Computer Systems(EuroSys)*, 2014.

[20] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, 2011.

[21] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

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

[23] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A case for efficient hardware/software cooperative management of storage and memory," in *In Proceedings of Fifth Workshop on Energy Efficient Design, WEED*, 2013.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[25] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[26] "Us-cert vulnerability notes database," http://www.kb.cert.org/vuls/.

[27] K. Piromsopa and R. Enbody, "Survey of protections from buffer-overflow attacks," *Engineering Journal*, vol. 15, 2011.

[28] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP)*, 2008.

[29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[30] M. S. Simpson and R. K. Barua, "Memsafe: Ensuring the spatial and temporal memory safety of c at runtime," in *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2010.

[31] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, 2010.

[32] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009.

[33] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[34] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[36] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The objectstore database system," *Commun. ACM*, 1991.

[37] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira, "Safe and efficient sharing of persistent objects in thor," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1996.

[38] V. Singhal, S. V. Kakkad, and P. R. Wilson, "Texas: Good, fast, cheap persistence for c++," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 145–147, Dec. 1992.

[39] S. J. White and D. J. DeWitt, "Quickstore: A high performance mapped object store," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 1994.

[40] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence, "An orthogonally persistent java," *SIGMOD Rec.*, vol. 25, 1996.

[41] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[42] "Intel itanium architecture. http://www.intel.com."

[43] "Intel Memory protection Extensions," https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[44] Oracle, "Sparc m-7 press announcement," https://www.oracle.com/corporate/pressrelease/sparc-m7-102615.html.

[45] "Nist software assurance reference dataset. http://samate.nist.gov/sard/."

[46] S. Sanfilippo, "Redis key-value store," https://github.com/antirez/redis.