Hypervisor-based interposition framework for storage-class memory

Yushi Omote[1] Takahiro Shinagawa[2] Kazuhiko Kato[1]

Abstract: Recently persistent storage with near-DRAM performance termed as Storage-class memory (SCM) is emerging. Straightforwardly intercepting and manipulating access to SCM in the hypervisor layer can significantly degrade performance. Our goal is to design a hypervisor framework that intercepts access to SCM with minimizing overhead. In this paper, we present hypervisor-based storage encryption for SCM, which intercepts read/write access of the guest OS to SCM and transparently encrypts/decrypts the data. The hypervisor avoids overhead by encrypting data in page units. The hypervisor also keeps crash consistency with proper management of operation progress. We implemented the prototype system based on an open source thin hypervisor, BitVisor. The performance evaluation shows 4KB-record read/write throughput to encrypted SCM on the guest OS decrease only by 5%-50% compared to throughput on baremetal.

1. Introduction

Recently persistent storage with near-DRAM performance termed as *Storage-Class Memory* (SCM) is emerging in the market and expected to be a replacement of today's storage devices [1]. SCM is directly connected to the memory bus and accessed with load/store instructions, each of which costs extremely low latency (only nanoseconds-order). This implies that deep storage stacks of software e.g. device drivers can be a major contributor to access latency for SCM. To reduce access latency, recent researches are minimizing the storage stacks of operating systems (OSs) [2], [3].

In our previous research, we provide full storage encryption in the hypervisor layer for encforcing security [4], [5]. Some other systems also provides storage encryption features in the hypervisor layer [6] for the same purpose. Hypervisors conventionally perform storage encryption by intercepting I/O instructions and encrypting transferred data in block units. However, this conventional method of interception can be impractical on SCM because this is no longer controlled by I/O instructions. Furthermore, data in SCM is transferred via load/store instructions in byte units. Straightforwardly intercepting every load/store instruction can cause tragic overhead because hypervisor intervention occurs too frequently.

Our research goal is to design the framework for the hypervisor to interpose in storage access to SCM by avoiding overhead. In this paper, as a first step toward the framework, we present hypervisor-based encryption for SCM, which transparently intercepts read/write access to SCM and encrypts/decrypts the data. To intercept guests access to SCM without too much intervention, we use *page-granularity interception*; by which the hypervisor intercepts page access of guests and encrypts/decrypts page content in page granularity. To encrypt the page contents without compromising crash consistency, the hypervisor uses *persistent page buffer*, which is a pre-allocated buffer on SCM that properly holds the page content and the progress of the cryptographic operations.

We implemented the prototype system based on an open source thin hypervisor, BitVisor [4], although our encryption method is applicable to other hypervisors such as Xen [7] and KVM [8]. In an experimental environment for performance evaluation, we used *PMFS* [3], which is an open source in-memory file system dedicated to SCM on the guest OS and the hypervisor transparently encrypts the data stored in the file system. We here regarded DRAM as SCM because the performance of both device is untimately expected to be almost identical. Encrypted PMFS with our page-granularity approach made 4KB-record read/write throughput decrease only by 5%-50%.

2. Background

In this section, we briefly introduce the characteristics of storage-class memory and then clarify our assumption about how OSs use SCM.

SCM is a new type of persistent storage that achieves DRAM-like characteristics: low access latency (only nanoseconds-order) and byte-addressability. Because of the characteristics, one of the efficient usages of SCM is expected to be direct connection to the memory bus as shown in Figure 1. From a perspective of software, part or entire regions of physical address spaces become non-volatile as shown in Figure 2, and data stored on these regions remains after power down.

There are three possible ways for OSs to use these non-volatile regions. The first way is that OSs use the region as a replacement for conventional storage; that is, OSs store persistent data

[1] University of Tsukuba
[2] The University of Tokyo

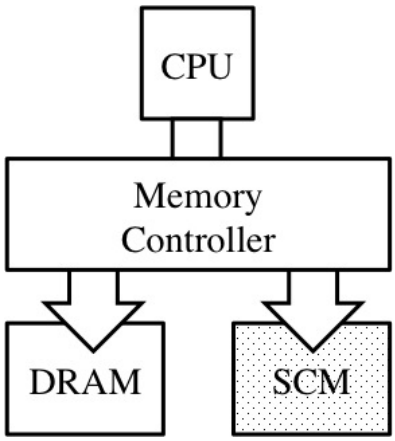


Fig. 1: **Hardware's perspective of a SCM-connected system**: SCM is directly connected to the memory bus.

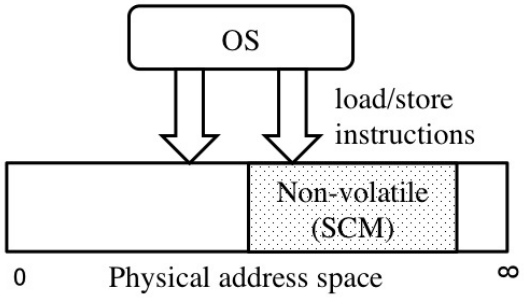


Fig. 2: **OS's perspective of a SCM-connected system**: SCM connected to the memory bus creates a non-volatile region in the physical address space.

such as files on these regions. The second way is that OSs use the region as a memory; that is, OSs store execution context of programs such as heaps and stacks, which is conventionally stored in a memory. The third way is that OSs store everything on non-volatile regions, using SCM as a single-level store.

In this paper, we assume the first way of usage (SCM as a replacement for conventional storage), which weighs compatibility with today's OS architecture, as some of other papers do [2], [3].

3. Difficulty on hypervisor-based interposition for SCM access

In this section, we mention basic operations the hypervisors need to perform in order to transparently encrypt conventional storage. Then, we describe the difficulty in encryption of SCM access in the hypervisor layer.

3.1 Hypervisor-based encryption for conventional storage

Basically, in order to perform storage encryption, hypervisors need to capture every read/write operation that transfers data between the guest OS and storage. For example, when the guest OS reads the content of storage, the hypervisor needs to capture the data being transferred from the storage to the buffer of guest OS. Then, it decrypts the content of data and copies the decrypted data to the guest OS instead of the original encrypted data on the storage. On the other hand, the hypervisor encrypts the data being written to the storage by the guest OS (see Figure 3).

OSs use I/O instructions to perform data transfer of the conventional storages such as ATA or SCSI devices. The hypervisor needs to intercept every I/O instructions to capture data transfer operations by the guest OS. Thanks to the support of direct memory access (DMA), the number of I/O instructions issued for a data transfer operation is not many. For example, in spite of transfer of a couple of megabytes of data, guest OS only needs to issue several I/Os to determine number and address of data blocks and to start the transfer operation. Even if the hypervisor intercepts all I/O instructions for encryption, this design limits the number of hypervisor intervention and its overhead.

3.2 Hypervisor-based encryption for SCM

On the other hand, when OSs access SCM, they transfer all data with load/store instructions (e.g. **mov** instruction) in byte units instead of I/O instructions. Therefore hypervisors cannot simply interpose in storage access as it does for conventional storage.

As a naive approach to storage encryption, it is possible for the hypervisor to intercept every load/store instruction and encrypt/decrypt data being transferred. However, this requires a lot of costly hypervisor intervention to a data transfer operation of the guest OS. For example, even if a kilobytes of data is being transferred, thousand times of cryptographic operation may be interrupted by the hypervisor. A simple preliminary evaluation (see Section 7) shows that this naive approach makes read-/write throughput of SCM one hundredth. Therefore, straightforwardly encrypting/decrypting data being transferred via load/store instructions is impractical due to the significant overhead.

4. Proposal

In this section, we present our approach for SCM encryption in the hypervisor layer.

4.1 Page granularity interception

The naive approach requires hundreds or thousands of hypervisor intervention with cryptographic operations for only a couple of kilo-bytes access. In our approach, in order to reduce the number of intervention, the hypervisor performs encryption in *page* units instead of byte units.

A possible approach is that the hypervisor intercepts *page mapping* and *page unmapping* operations of guest OSs. By intercepting this mapping operation, the hypervisor can find when the guest OS starts using the memory region. Hence, the hypervisor can perform decryption operations at the timing of mapping

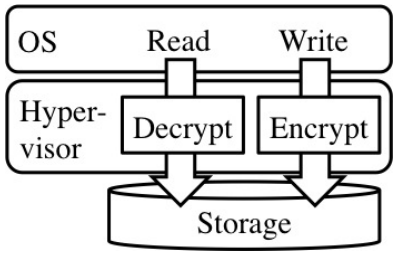


Fig. 3: Basic operations of hypervisor-based storage encryption

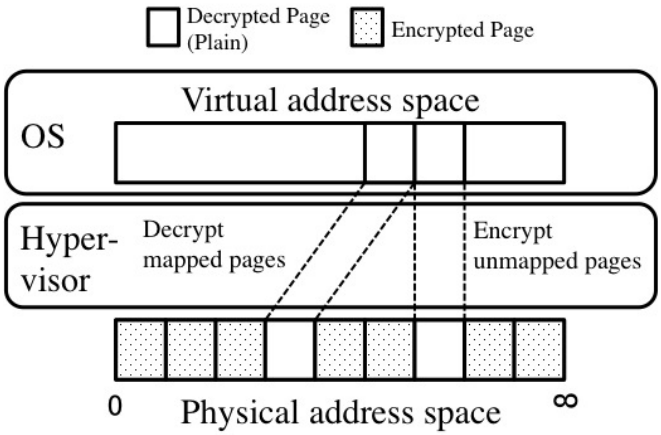


Fig. 4: SCM encryption

operations.

On the other hand, when an OS stops using a page, it unmaps the target page; it removes mapping information between virtual address and physical address. It is reasonable that the hypervisor performs encryption operations at the timing of unmapping operations (see Figure 4).

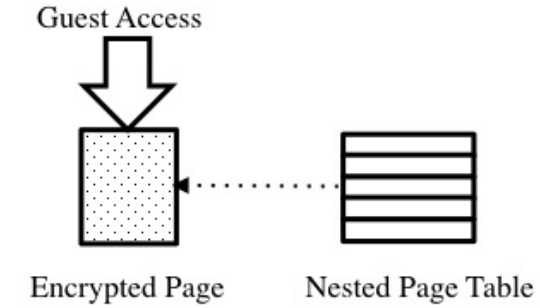
A possible way for the hypervisor to detect page mapping and unmapping operations done by guest OSs is that the hypervisor intercepts the update of page tables (by trapping the update of CR3 registers). By scanning and comparing the page table entries between before and after the update, the hypervisor can detect which page is mapped or unmapped. For example, if an page table entry exists in the page table before the update but does not exist after the update, it means the page is unmapped. On the other hand, a page table entry newly appears in the current page table, it means the page is mapped.

However, since the page table can be updated frequently during the usual operations of OSs, intercepting updates and scanning tables is costly and causes overhead. To avoid the interception of page table update, by use of nested page tables which is managed by the hypervisor (e.g. EPT on Intel CPU or NTP on AMD-V), we decrypt the target memory region on demand.

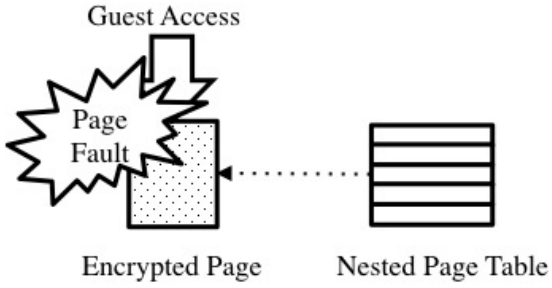
4.2 On-demand decryption

The hypervisor does not need to perform decryption operations until the target memory region is actually accessed by the guest OS. By use of nested paging, the hypervisor can cause a trap when the guest OS actually reads the memory region of the page.

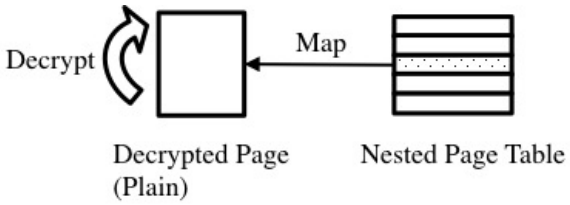
Figure 5 shows the operation flow of on-demand decryption. At the first time, the hypervisor keeps the encrypted regions unmapped in the nested page tables in order to trap any access to these regions by causing page faults (see Figure 5(a)). When the guest OS tries to access one of the encrypted regions, a page fault of nested page table occurs and the hypervisor can trap the access and gets the control of the CPU (see Figure 5(b)). Here, the hypervisor decrypts the region and overwrites the region with decrypted data. In addition, the hypervisor sets a page table entry for the decrypted page in the nested page table to avoid further traps for the access of the guest OS to the region. After the creation of the page table entry, the hypervisor gives the control of CPU back to the guest OS (see Figure 5(c)). After the decryption operation, further access from the guest OS to the decrypted page is allowed without being trapped by the hypervisor (see Figure 5(d)).



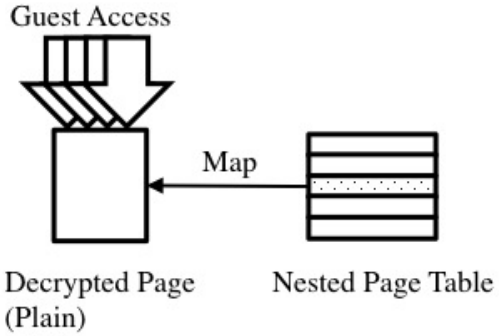
(a) **First access to encrypted page**: The guest OS tries to access the encrypted page, which is not mapped in the nested page table.



(b) **Page fault of nested page table**: The page fault of nested page table occurs because the page is not mapped on the page table.



(c) **Decryption and mapping**: The hypervisor decrypts the page and creates the page table entry for the decrypted page in the nested page table.



(d) **After decryption**: The guest OS can access the decrypted page without being trapped.

Fig. 5: On-demand decryption

Therefore the hypervisor does not have to precisely intercept the frequent page table updates of guest OSs. All encryption operations are done on-demand when the guest OS actually tries to access the encrypted pages.

4.3 Lazy encryption

The hypervisor does not also have to perform encryption operations immediately after the guest OS unmaps pages.

Scanning page tables is inefficient if only a single page of the page table is encrypted. In addition, only intercepting access to CR3 register is costly because it is frequently accessed during the normal OS operations.

In order to avoid CR3 interception as well as frequent scan of page tables, the hypervisor periodically checks the page tables of the guest OS less frequently and performs batch process of encryption.

5. Crash Consistency

If the system crashes or goes forcibly shutdown during operations of the hypervisor, the incomplete data may remain in SCM persistently. This can damage the system even if it is rebooted. To solve this problem, the hypervisor needs to keep crash consistency.

First of all, the hypervisor needs to preserve the information about which pages are encrypted and which are not in case of system crash. The hypervisor therefore has a bitmap that holds page states. For example, a set bit indicates a encrypted page and a cleared bit indicates a decrypted page. Since the bitmap should be preserved during shutdown or reboot, it should be located on non-volatile region of SCM.

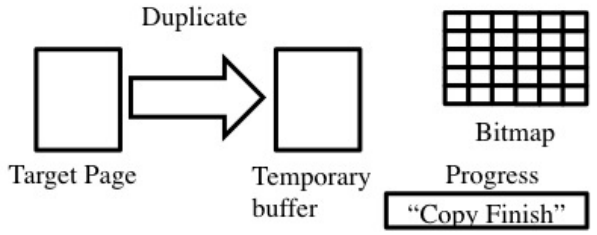
A system crash causes incomplete data when the hypervisor is encrypting or decrypting pages. This may leave some pages incomplete state. The hypervisor therefore needs to be able to recover from the wrong state. Figure 6 shows the steps of a cryptographic operation that takes crash consistency into account. After each step completed, the hypervisor atomically updates the progress; that is, which step has been completed. The progress information should be also in non-volatile region of SCM because it needs to be referred to state recovery. The initial progress state is "None".

To start over incomplete cryptographic operations in case of ill-timed crash, the hypervisor makes the copy of the target page in a buffer in SCM before starting encryption or decryption. After the copy operation completed, the hypervisor sets the progress state to "Copy Finish" that indicates the copy operation finishes (see Figure 6(a)).

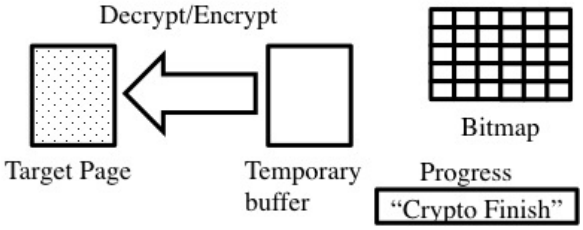
After that, the hypervisor actually performs encryption or decryption of the page. Using the copied page as the source of encryption or decryption operation, the result is overwritten to the original page. After this operation finishes, the hypervisor sets the progress state to "Crypto Finish" that indicates the cryptographic operation finishes (see Figure 6(b)).

The hypervisor checks the progress state when it starts next time. If the progress state is "Copy Finish" but not "Crypto Finish", the hypervisor again performs the cryptographic operation overwriting the original page because the previous cryptographic operation may be incomplete and the original page may have invalid content. If the progress state is "None", the hypervisor may have stopped copying before it finished last time. The hypervisor therefore discards the buffer and starts from the copy operation.

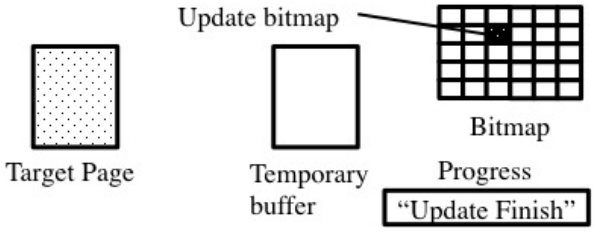
After encrypting or decrypting, the hypervisor needs to update the bitmap of page state. This update needs to be done consistently and must not indicate different state from the actual state of pages. If the system crashes while the hypervisor finishes the update of the bitmap after the cryptographic operation, bitmap has wrong information; although a page is ecrypted, the bitmap says it is not encrypted. To detect and fix this inconsistency, after the hypervisor finishes the update of bitmap, it sets the progress state to "Update Finish" (see Figure 6(c)). If the hypervisor finds the progress indicates "Crypto Finish" but not "Update Finish" when it starts up, the hypervisor ensures the update of bitmap.



(a) **Page duplication**: The hypervisor copies the target page to the temporary buffer and sets the progress to "Copy Finish" when it finishes.



(b) **Encryption/Decryption**: The hypervisor encrypts/decrypts the target page and sets the progress to "Crypto Finish" when it finishes.



(c) **Bitmap update**: The hypervisor updates the corresponding bit in the bitmap of page state and sets the progress state to "Update Finish".

Fig. 6: Three-step page modification for crash consistency

6. Implementation of prototype system

We implemented the prototype system that performs hypervisor-based encryption for SCM based on an open source thin hypervisor, BitVisor [4]. The prototype system supports Intel CPU and relies on the nested paging function, Intel EPT. Since SCM itself is under advanced development and we cannot have the real device, we implemented our system on DRAM, simulating a SCM.

6.1 Encryption and decryption

We assume all data on SCM is encrypted when the guest OS starts up first time. Then, in order for the hypervisor to intercept all first access to the region of SCM, the hypervisor does not create EPT page table entries. Therefore, when the guest OS reads

or writes to a part of the region, a fault occurs (VMExits) and the operation transits to the hypervisor mode. Then, the hypervisor decrypts the page and replaces its content to decrypted data.

After that, in order to avoid further faults to this page, the hypervisor creates the EPT page table entry that maps this region. Since BitVisor allows the guest OS to directly manage all physical address spaces, it uses identity mapping of pages; that is, the physical address of the guest OS is identical to the actual physical address which is seen from the hypervisor. Therefore, in the newly-created EPT page table entry, the target physical address is set to the same address that guest OS specifies. Other hypervisors such as KVM do not use identitiy mapping in order to allow multiple guest OSs to share a physical address spaces. Even in this case, difference is only the way of mapping and the decryption method of our approach is still applicable.

In addition to the decryption, the hypervisor has to do encryption of pages after the guest OS stops using the pages. Here, the hypervisor stops intercepting the update of page table to find which pages are unmapped and avoid causing VMExits for CR3. Instead, the hypervisor periodically causes the operation transition to the hypervisor mode and decrypts all unmapped pages. There are several ways for the hypervisor to achieve the control of CPU. For example, Intel CPU allows the hypervisor to cause VMExits on hardware interrupts. This allows the hypervisor mode is executed within a certain interval (at least the interval of hardware timers) and can get chances to periodically perform encryption operations of pages. However, the interrupt frequency grows drastically in I/O-intensive workloads (e.g. file or database servers). Instead, in order to use the workload-independent way to get the control of CPU periodically, the hypervisor uses preemption timer of Intel CPU. This timer forces the CPU to transit to the hypervisor mode unconditionally when the timer counter counts a certain ticks. By use of this timer, the hypervisor periodically checks the page table and encrypts the content of pages which is unmapped by the guest OS.

It is possible for the hypervisor to find which pages are actually unmapped by getting the value of CR3 registers of the guest OS and finds the pages which is on the EPT page table but not on the page table of the guest OS.

6.2 State management bitmap

As we mentioned in Section 5, the hypervisor needs to have the bitmap that manages which pages are encrypted and which is not encrypted. Each bit in the bitmap in our prototype indicates a single 4KB page. If the bit set to 1, the corresponding page is encrypted. We implemented a 256MB bitmap which can manage 1TB of SCM. Since this bitmap needs to survive after shutdown or reboot in case of system crash, the hypervisor needs to allocate it on the non-volatile region of SCM.

The hypervisor therefore needs to conceal a part of SCM from the guest OS and places the bitmap there because BitVisor shares physical address space with the guest OS. As the hypervisor already does to allocate physical memory regions to itself, it is possible to conceal the SCM region by returning fake results of BIOS calls that tell the guest OS the available address map. However, the way of concealing the region may depend on the actual specification of SCM devices. The information about which parts of physical address space can SCM also depends on the implementation of real SCM devices. Since we do not have real devices, we so far use hard coding, assuming a certain fixed region is SCM.

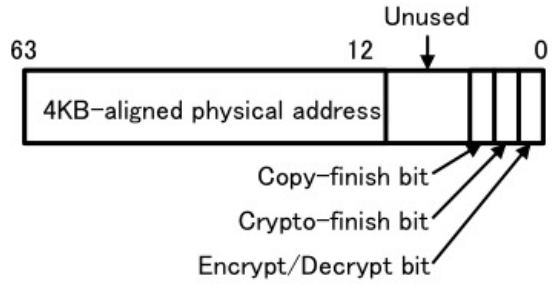


Fig. 7: **Cryptographic state field**: The 64-bit fields on SCM holds the current state of cryptographic operations.

6.3 Crash-consistent cryptographic operation

To allow recovery from incomplete state, the hypervisor has a 64-bit field to manage the progress of cryptographic operations and a 4KB page buffer to allow rollback to a complete state in case of crash. Both data structures are also located in a SCM region.

Figure 7 shows the format of 64-bit field. First of all, the hypervisor needs to remember which page it is currently handling in case of reboot on crash. Therefore the hypervisor writes the physical address of the page to this field. The lower 11 bits of physical address is not necessary information because the hypervisor only needs to know the base address of the target 4KB page (*physical page number*). Hence, only the upper 52 bits of this field are used to save the address.

In addition to the address, the hypervisor also needs to the type of operation; that is, which operation it is performing, encryption or decryption. This information is saved in the Bit 2 of the field. When the hypervisor is encrypting the page, the bit is set. Otherwise, the bit is cleared.

The hypervisor uses the least 2 bits to represent the progress state. Bit 0 of this field is *Copy-finish bit*, which represents whether duplication of the target page finishes or not. First, this bit is cleared to zero. As we mentioned in Section 6, prior to encryption or decryption operation, the hypervisor copies the target page for a possible rollback operation. After copy finishes, the hypervisor sets Copy-finish bit to say the progress state is “Copy Finish”.

Bit 1 of the 64-bit field is *Crypto-finish bit*, which shows whether a cryptographic operation is done or not. At first, the bit is cleared to zero. After setting Copy-finish bit, the hypervisor performs a cryptographic operation, reading the replica of the page saved in the 4KB buffer and writing the encrypted or decrypted page to the original place of the page. Then the hypervisor sets Crypto-finish bit. After the bit is set, the hypervisor updates the state management bitmap, and then it clears both Clear-finish and Crypto-finish bits atomically. This indicates the progress state is “Update Finish” (Here, the hypervisor can also regard this state as “None” and proceed to a next operation).

These two bits can determine the state recovery operation of

the hypervisor when the hypervisor starts. If both Copy-finish bit and Crypto-finish bit are cleared, the hypervisor does not need to perform any recovery operations. If Copy-finish bit is set but Crypto-finish bit is cleared, the hypervisor retries a cryptographic operation because the original page, which will be filled with the output of the encryption or decryption operation, may contain incomplete data. If both bits are set, the hypervisor again needs to update the bitmap because the hypervisor may have failed to update the bitmap due to the crash.

6.4 CPU cache effects

When the hypervisor updates this 64-bit field, some load/store instructions are insufficient in order to precisely update this field because the CPU cache may delay the actual update to SCM. For example, if the hypervisor sets Crypto-finish bit with a **mov** instruction, the actual bit on SCM is not set until the cache line is flushed. The system crash during this delay again causes inconsistency. To avoid this, the hypervisor uses a **clflush** instruction to flush the 64-bit field and **mfence** instruction to delay following instruction execution until the all data is really flushed to SCM.

The hypervisor also uses these instructions to ensure all updates are really applied to SCM before setting Copy-finish bit and Crypto-finish bit. For example, after the hypervisor performs encryption of the page, it uses **mfence** and **clflush** to ensure write of encrypted data on SCM.

6.5 Encryption on shutdown and startup

When the system goes shutdown, the some pages are being mapped on the page table of the guest OS, therefore the hypervisor leave the pages not encrypted. Hence, the hypervisor intercepts the shutdown signal (INIT signal) and encryptes all unencrypted pages.

When the system crashes without receiving INIT signal, the hypervisor may fail to encrypt some pages. In this case, when the hypervisor restarts, it encrypts all remaining unencrypted pages based on the information of the state management bitmap.

6.6 Multi-processor support

Since the page table exists per logical processor, the encryption and decryption operations previously mentioned are performed on each processor. Therefore, the 64-bit progress field and 4KB buffer should be allocated for each processor. On the other hand, the bitmap is shared by all processors to avoid encrypting or decrypting the same memory region twice. Therefore when a page fault of nested page table occurs in a processor, the hypervisor does not unconditionally decrypts the page. Instead, it checks the bitmap first. If the memory region is already decrypted, the hypervisor does not perform decryption and just maps the page. To avoid inconsistency, the update and check operations of the bitmap are mutually excluded by use of spinlock.

6.7 Huge paging

Since current BitVisor only supports 4KB paging and does not support huge paging such as 1MB paging and 1GB paging, our implementation only performs encryption in 4KB unit. However, implementation of huge paging shall be also possible. Since effects of 4KB paging may degrade the peformance of handling access to SCM in the hypervisor layer as described in Section 7, we plan to implement huge paging in the future.

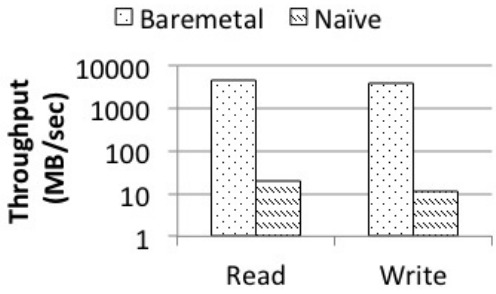


Fig. 9: Performance degradation of naive approach

7. Performance evaluation

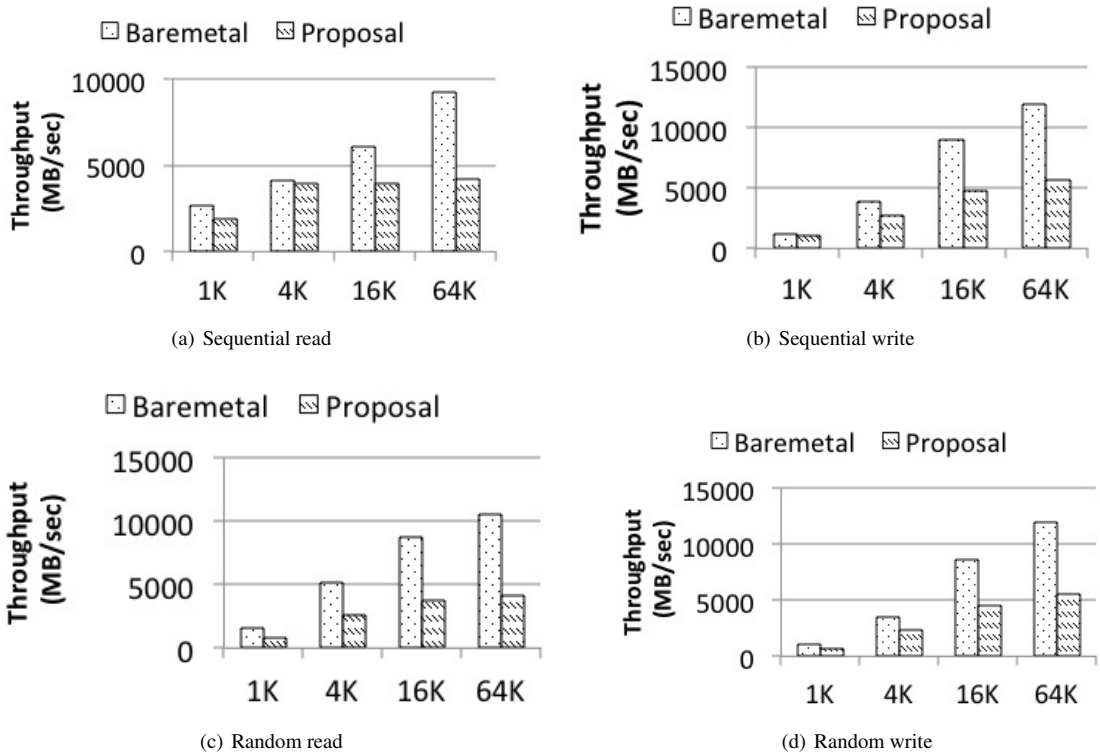
In this section, we present the results of I/O benchmark tests. We ran a persistent memory file system (PMFS) [3], which is an open source file system designed to store files directly on SCM bypassing block device drivers. The file system is implemented on Linux 3.11 and the source code is publicly available. Since we do not have real SCM devices, we use DRAM as a simulation of SCM. As an experimental environment, we used a machine with Inte Core i7-4771 CPU and 8-GB DDR3 SDRAM. In all tests, our system uses AES as a cryptographic algorithm and we just called the cryptographic functions of the original BitVisor with specifying a hard-coded key.

First, we performed preliminary experiment that tests performance effects of the naive approach to I/O performance of the guest OS. By use of PMFS, we can mount a region of physical memory instead of storage with **mount** command. On Linux, we mounted a 4-GB memory region starting from 0x100000000 (4-GB offset) with the following command. After the command finishes, any files created under the directory */mnt* will be created on the physical memory region.

```
mount -t pmfs -o \
physaddr=0x100000000,init=4G none /mnt
```

In that directory, we measured sequential write/read throughput, creating and reading a 2-GB file with 4K record using *dd* command. We ran this test on both baremetal and the implementation based on the naive approach on BitVisor that intercepts every load/store instruction for encryption. Figure 9 shows the results. Compared to baremtal, read and write throughput of naive approach decreased by 99.6%. The CPU utilization in both case easily reaches almost 100% and CPU becomes a bottleneck. Interception of every load/store instruction is extremely constly and impractical. The overhead is caused by intensive hypervisor intervention to load/store instructions.

Then, we evaluated I/O throughput on proposed system that uses page-granularity encryption and compared it to throughput on baremetal. We evaluated throughput by using *fio*, which is an open source benchmarking tool and measured both sequential and random access. We ran tests with some different record sizes from 1-KB record to 64-KB record. Figure 8 presents the bench-



(a) Sequential read
(b) Sequential write
(c) Random read
(d) Random write

Fig. 8: File I/O throughput

mark results.

Compared to throughput on baremetal, sequential read throughput decreased by about 5%-30% with 1-K and 4-K records (see Figure 8(a)). However, when the record size is bigger than 16 KB, CPU utilization reaches 100% and stops scaling. On the other hand, sequential read throughput on baremetal continues to scale. When the record size reaches 64 KB, the CPU utilization on baremetal reaches 100% and the throughput stops scaling. The same tendency is observed with the results of random read throughput (see 8(c)), while performance degradation due to proposed system was relatively large (throughput decreased 40%-50%). This higher CPU utilization is due to frequent page faults of the nested page table. Because the hypervisor uses 4KB paging, a single 16KB-record is expected to cause faults four times in the worst case. We believe the use of huge paging can reduce CPU utilization and achieve more scalability. In addition, recent CPU supports hardware cryptographic support such as AES-IN. Using this capability is also expected to improve the performance. Compared to throughput on baremetal, sequential write throughput decreased by 30%-50%. Unlike the case of read throughput, both tests on baremetal and proposed system does not saturate CPU utilization even when the record size is 64 KB. Though the performance improvement is still required, we reduce the performance degradation compared to all interception of load/store instructions.

8. Related work

SCMFS [9] and Aerie [2] remove the OS-level abstraction layer of storage stacks to improve performance of SCM access, allowing applications to directly access address spaces of SCM with byte addressing. Hence, the hypervisor that purposes the interposition in storage access also needs to find the efficient way of interposition that minimizes performance degradation.

To our best knowledge, this is the first attempt to hypervisor-based storage encryption for SCM. Although CloudVisor [10] supports encryption of page contents, the encryption is a trivial case and performed when illegal access occurs to prevent the information leak between guest OSs and hypervisors. It neither consider crash consistency nor perform lazy encryption to avoid leaving pages unencrypted.

BPFS [11], PMFS [3] and Mnemosyne [12] solve the challenge of crash consistency when software access to SCM. In our work, we also take crash consistency into consideration and applied the technique of these works in the hypervisor layer.

9. Summary and future work

Towards a hypervisor framework that manipulates access to SCM without significant overhead, we presented hypervisor-based storage encryption for SCM, which transparently intercepts read/write access to SCM and encrypts the data. Our prototype hypervisor transparently encrypts and decrypts access to SCM in page unit instead of byte unit of load/store instructions. We also support crash consistency to avoid data corruption during encryption. With I/O throughput benchmark for a file system on SCM, we found this was effective to reduce the performance degradation.

As future work, further performance improvement is desirable. First, we need to support huge paging of nested paging for more efficient encryption. Second, we also need to utilize hardware cryptographic support. After the all possible optimization, we

also need more performance evaluation. We compare the performance results of our hypervisor-level encryption and the OS-level encryption.

References

[1] Freitas, R. F. and Wilcke, W.: Storage-class memory: The next storage system technology, *IBM Journal of Research and Development*, Vol. 52, No. 4.5, pp. 439–447 (2008).

[2] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P. and Swift, M. M.: Aerie: Flexible File-system Interfaces to Storage-class Memory, *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, ACM, pp. 14:1–14:14 (online), DOI: 10.1145/2592798.2592810 (2014).

[3] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R. and Jackson, J.: System Software for Persistent Memory, *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, ACM, pp. 15:1–15:15 (online), DOI: 10.1145/2592798.2592814 (2014).

[4] Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, *Proc. 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pp. 121–130 (online), DOI: 10.1145/1508293.1508311 (2009).

[5] Omote, Y., Chubachi, Y., Shinagawa, T., Kitamura, T., Eiraku, H. and Matsubara, K.: Hypervisor-based Background Encryption, *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, New York, NY, USA, ACM, pp. 1829–1836 (online), DOI: 10.1145/2245276.2232073 (2012).

[6] Min, L. and Chaowen, C.: Full Disk Encryption based on Virtual Machine and Key Recovery Scheme, *Journal of Information and Computer Science*, Vol. 6, No. 3, pp. 163–172 (2010).

[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 164–177 (online), DOI: 10.1145/945445.945462 (2003).

[8] Kivity, A.: kvm: the Linux Virtual Machine Monitor, *Proc. 2007 Ottawa Linux Symposium*, pp. 225–230 (2007).

[9] Wu, X. and Reddy, A. L. N.: SCMFS: A File System for Storage Class Memory, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 39:1–39:11 (online), DOI: 10.1145/2063384.2063436 (2011).

[10] Zhang, F., Chen, J., Chen, H. and Zang, B.: CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization, *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 203–216 (online), DOI: 10.1145/2043556.2043576 (2011).

[11] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D. and Coetzee, D.: Better I/O Through Byte-addressable, Persistent Memory, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 133–146 (online), DOI: 10.1145/1629575.1629589 (2009).

[12] Volos, H., Tack, A. J. and Swift, M. M.: Mnemosyne: Lightweight Persistent Memory, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, New York, NY, USA, ACM, pp. 91–104 (online), DOI: 10.1145/1950365.1950379 (2011).