Lelantus: Fine-Granularity Copy-On-Write Operations for Secure Non-Volatile Memories

Jian Zhou Amro Awad Jun Wang
University of Central Florida
{jian.zhou, amro.awad, jun.wang}@ucf.edu

***Abstract*—Bulk operations, such as Copy-on-Write (CoW), have been heavily used in most operating systems. In particular, CoW brings in significant savings in memory space and improvement in performance. CoW mainly relies on the fact that many allocated virtual pages are not written immediately (if ever written). Thus, assigning them to a shared physical page can eliminate much of the copy/initialization overheads in addition to improving the memory space efficiency. By prohibiting writes to the shared page, and merely copying the page content to a new physical page at the first write, CoW achieves significant performance and memory space advantages.**

Unfortunately, with the limited write bandwidth and slow writes of emerging Non-Volatile Memories (NVMs), such bulk writes can throttle the memory system. Moreover, it can add significant delays on the first write access to each page due to the need to copy or initialize a new page. Ideally, we need to enable CoW at fine-granularity, and hence only the updated cache blocks within the page need to be copied. To do this, we propose Lelantus, a novel approach that leverages secure memory metadata to allow fine-granularity CoW operations. Lelantus relies on a novel hardware-software co-design to allow tracking updated blocks of copied pages and hence delay the copy of the rest of the blocks until written. The impact of Lelantus becomes more significant when huge pages are deployed, e.g., 2MB or 1GB, as expected with emerging NVMs.

I. Introduction

Emerging Non-Volatile Memories (NVMs), e.g., Intel's and Micron's 3D Xpoint, are entering the mainstream server market. With terabytes of capacity and ability to host persistent data, emerging NVMs are promising candidates to build future memory systems. However, emerging NVMs bring in unique challenges and aspects that require rethinking the current designs of computing systems. Protecting against data remanence attacks that are viable for NVMs is one of such challenges. Moreover, the large capacity of NVMs motivates more intelligent mechanisms for memory management, e.g., efficient use of huge pages. Additionally, systems that are equipped with NVMs are expected to take advantage of the data persistence feature through changes in system software and supporting libraries, e.g., Intel's PMDK.

The granularity of memory management in current operating systems is well-suited for DRAM capacities. However, with NVMs coming with much larger capacities, the trade-off between internal fragmentation (unused space within a page) and translation overhead needs to be revisited. Page size is an important factor affected by such trade-off, and the default page size for the current system is typically 4KB or 8KB. However, huge pages (2MB or 1GB) support is becoming common in spite of some challenges. When using huge pages, systems equipped with typical DRAM capacities, e.g., 32GB or 64GB per processor socket, would suffer from internal fragmentation and the limited number of pages, and hence limit concurrent applications. In contrast, for systems equipped with NVMs, of terabytes capacity per processor socket, it is practical to use huge pages to reduce the bookkeeping and translation overheads, especially when internal fragmentation is unlikely to burden applications.

Bulk operations such as page initialization and page copying have received lots of attention in the literature due to their excessive use in many operating system (OS) operations [3], [18], [31]. Such bulk operations become more expensive with larger page sizes. The limited write endurance and slow write operations of NVMs make bulk operations even more costly and can easily lead to throttling the memory system. For instance, when 1GB page sizes are used, at the first write operation of a page, the OS has to zero out the whole page, which can result in millions of write operations to the memory. Similarly, the common CoW optimization used to delay the instantiation of physical pages having content similar to existing pages would incur a significant number of writes at the first attempt to write a write-protected CoW page.

Enabling bulk operations to occur at fine granularity, and delay their actual writes until smaller granularity blocks are actually written/copied, can significantly improve performance and avoid expensive bulk operations in many OS operations and on-demand paging. Meanwhile, managing memory at a fine-granularity can add significant bookkeeping and translation overheads. To reduce the tension between using coarse-granularity and fast bulk operations, we propose *Lelantus*, a novel hardware/software co-design scheme that allows fine-granularity bulk operations while using coarse-granularity pages. Lelantus mainly relies on tracking initialized/copied cachelines on each page and do the actual copying/initializing of cachelines on-demand (lazily). Lelantus repurposes security metadata maintained per cacheline to enable the memory controller to recognize delayed copy/initialization operations. By doing so, the memory controller completes initialization/copying at cacheline granularity when written (or read). Lelantus preserves the software semantics and provides the same guarantees of data content as if initialization/copying has been done conventionally; however, in a more efficient and performance-friendly way.

Due to data remanence problem, emerging NVMs are always paired with encryption [3], [4], [9], [39], [41], [41]. Even commercial products, such as Intel's Apache Pass (DIMM form factor of 3D XPoint), are expected to have encryption built-in [12]. Meanwhile, current and future processors have support for processor-side encryption, e.g., AMD's Secure

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00056

Memory Encryption (SME) and Intel's Total Memory Encryption (TME) [17], [20]. Such available support will facilitate the adoption of NVMs. Unfortunately, memory encryption, due to its avalanche effect, exacerbates the write endurance problem and further amplifies the impact of NVM's slow writes on performance [3], [39]. Therefore, many recent works focus on reducing the impact of NVM writes on the presence of encryption [3], [39]. In this paper, we focus on improving the performance of the CoW operations for emerging NVMs in the presence of encryption.

To evaluate Lelantus, we modify the Linux kernel v5.0 and run our modified Linux on a cycle-level full-system simulator, Gem5 [6]. Our evaluation results show that Lelantus can improve the performance by up to 3.33x (2.25x on average) for regular pages and up to 67.53x (10.57x on average) for huge pages. Meanwhile, Lelantus can reduce the average number of NVM writes (and hence improve lifetime) to 42.78% for regular pages and 29.65% for huge pages.

We organize the rest of the paper as follows. First, we introduce the background and motivation in Section II. Second, we discuss the design and implementation details in Section III and Section IV. In Section V, we discuss our evaluation methodology and present our evaluation results. Section VI discusses the related work. Finally, we conclude our work in Section VII.

II. Background and Motivation

In this section, we discuss the background and present motivational data for the overhead of Copy-on-Write operations.

A. *Emerging NVMs*

Different vendors are developing a wide range of emerging Non-Volatile Memory (NVM) technologies. For instance, a recent announcement from Intel and Micron has revealed a new NVM technology, 3D XPoint [15]. Emerging NVMs promise an order of magnitude lower access latency than Flash, and an order of magnitude denser than DRAM. In that similar trend, HP Labs has been investigating Memristor as a promising technology to replace DRAM and make storage systems [34]. Such memory technologies are favorable to makeup storage systems for two key reasons [13], [37]. First, they are non-volatile, which is a strict requirement for storage systems. Second, they are orders of magnitude faster than current storage devices, such as NAND flash-based Solid-State Drives (SSDs) and Hard-Disk Drives (HDDs).

At the same time, NVM technologies are also promising to replace DRAM as primary memory devices due to several reasons. To begin with, they possess high densities, the area in which DRAM is struggling with time due to its reliability and manufacturing constraints. In addition to that, they are non-volatile, which promises near-zero idle power, hence an ideal candidate for systems built with energy efficiency in mind. As a result, major computing vendors envision emerging NVMs to play a promising role in replacing main memory and storage devices. Moreover, they are most likely to be merged into one system [13], [15].

Most of the emerging NVMs are based on phase-change or resistive material, in which the phase and other non-volatile and measurable levels can be used to represent the stored values. Passing high current or voltage can change such levels [22], [24]. In some cases, the direction of such current can be used to program the cells with different values [14]. Most of these technologies have limited write endurance. However, they feature an order of magnitude higher endurance than current SSDs [22], [24], [28], [37].

While many promising NVM technologies are compelling solutions for building future systems, they mostly share common features - high densities, ultra-low idle power, limited write-endurance, and access latencies that are comparable to DRAM.

B. *Secure NVMs*

Deploying emerging NVMs as main memory has always been paired with memory encryption [3], [9], [39]. Typically, counter-mode encryption is used due to its security and performance advantages. Figure 1 depicts the counter-mode encryption.

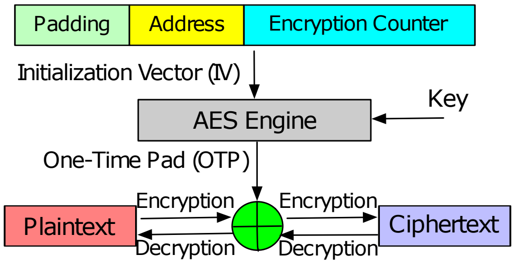


Fig. 1: Memory counter-mode encryption.

Counter-mode encryption can overlap the latency of fetching data and the encryption engine latency; an initialization vector (IV) is what gets encrypted to produce a one-time pad (OTP), which gets XOR'ed with ciphertext/plaintext to complete encryption/decryption. Therefore, such OTP could be pre-generated while the data is being fetched, and hence, the latency of the encryption algorithm (AES in this case) is hidden. From the security perspective, by associating an encryption counter with each memory block (e.g., 64 bytes), counter-mode encryption can ensure both temporal and spatial uniqueness of ciphertext for the same data. The temporal uniqueness is guaranteed through incrementing the counter associated with each data block after encryption, *i.e.*, writing the data block to memory. Meanwhile, the spatial uniqueness is ensured by including the block address in the IV, and thus the IV (and hence the OTP) will be unique for each block.

To reduce the storage overhead of encryption counters and improve counter cache locality, prior work proposed a split-counter scheme [36]. In the split-counter scheme, each 4KB memory region would have one major counter (64 bits) and 64 7-bit minor counters. Each minor counter corresponds to a specific 64-byte memory block within the 4KB memory region. The encryption counter (shown in Figure 1) of each 64-byte data block is composed of the concatenation of the corresponding minor counter and the region's major counter.

One major vulnerability for counter-mode encryption is tampering with counters or possibly replay their old values. Replaying counters enables known-plaintext attacks that compromise the security of the system. Therefore, the use of counter-mode encryption is typically paired with integrity protection of data and counters through Merkle Tree [29]. The overhead for protecting the integrity of counters and data is negligible (less than 2%) [29]. For the rest of the paper, we assume the split-counter scheme with Bonsai Merkle Tree similar to state-of-the-art work [4], [38], [41].

C. Copy-On-Write

Copy-on-write (CoW) is an efficient technique used in most modern operating systems (OSes) to enable efficient memory management. It delays an expensive copy operation until it is needed. If the data is copied but not modified, the copied data can exist on the source page. Only when the copied data is modified, the kernel performs an actual copy to create a private page. There are several use cases for the CoW technique.

The Fork System Call: When a process forks a new child process, the parent and child processes make a copy of all existing data. They then start from where the parent process is left off. By avoiding the expensive copy operation, the kernel memory management system renders both child and parent processes able to share the original pages and masks those pages as write-protected. Whenever either the child process or parent process updates those write-protected pages, a page fault is triggered, also, subsequently, the CoW technique is engaged to create a private copy of the page. The fork is widely used to create an isolated execution environment in many applications so that an error in the child process no longer affects the parent process. For example, the Apache HTTP server forks child processes as independent agents in response to network requests. Modern Web browsers fork child processes to render Web pages as well as create sandbox environments for plugins.

Virtual Memory Management: In modern OSes, when a process allocates a new memory page on the heap, its virtual address is mapped to a zero page in initialization. The zero page, which is filled with all zeros, is a read-only page reserved at system bootup time. Only when the process first writes to the virtual page, the CoW technique allocates a physical page and copies the zeros into the allocated page. The CoW technique enables the OSes to allocate virtual pages without performing expensive physical memory initialization immediately. For example, *malloc/remalloc* [19] implemented in *libc* use *brk* system call to resize the heap segment by default. Alternatively, when the allocated memory is larger than a threshold, the *libc* uses *mmap/mremap* to allocate large *mmap* blocks beyond the heap segment. The threshold is 128K by default and may be adjusted dynamically according to a specific implementation. Both allocation methods leverage the zero page and CoW techniques to delay the expensive tasks of zeroing out the allocated memory. It is also true for huge pages as all the newly allocated huge pages point to a particular huge zero page.

Data Deduplication/Virtual Machine Cloning: Data deduplication technologies search chunks or blocks containing the same data. It reduces the memory usage by removing the duplicates and then apply CoW to the remaining. For example, Kernel Same-page Merging (KSM) merges duplicated pages into a shared write-protected page [2], [8], [25], [32], [35]. Kernel-based Virtual Machine (KVM) first introduced KSM. However, in current systems, any application, that generates many instances of the same data, can advise those areas of memory space to KSM for deduplication by using the *madvise* system call. There is also Virtual Machine Cloning [21] technology proposed to significantly reduce the startup cost of VMs in a cloud computing server. Different from *fork*, VM Cloning makes a private copy of I/O resources and defines a set of identical virtual hardware configurations.

Snapshot/Checkpointing: A snapshot is a consistent copy of the whole dataset. By having a snapshot, we can restore the state of in-memory applications to the snapshot point. Alternatively, we can also recover to the current state if the latest transaction logs also exist [5], [33]. The CoW can be used to make a fast snapshot of in-memory applications. For example, Redis [7], one of the most popular in-memory key-value database, creates a snapshot by forking a child process. The parent process continuously serves any in-coming (read and write) requests. The child process makes a complete copy of the dataset at the snapshot point. By leveraging the CoW, the parent process handles the write requests in a private copy of memory. Hence, the dataset can be persisted to external storage by the child process in the background.

D. Motivation

As OSes implement the CoW at page granularity, they introduce two significant performance penalties. First, there exist substantial delay for the first write operation on a shared page. When we first update a CoW page, the whole page copying inevitably slows down the first write. Second, performing full page updates also results in duplicated write operations to the target page (first copy then write), especially for huge pages. For huge pages, the copy operation is typically implemented as non-temporal stores (to avoid cache pollution). Hence, any subsequent writes will be added to those incurred by the copy operation. For regular pages, the extra writes would result in pre-mature evictions from the cache due to cache pollution. In both cases, the number of physical writes can be significantly more than that of the logical writes for a CoW page, *i.e.*, CoW causes write amplification. The write amplification in CoW pages not only hurts the performance but can also reduce the lifetime of limited write-endurance memories such as NVMs. Figure 2 shows the number of physical writes performed in the memory when updating one byte per page and updating the whole page. The total size of the memory allocation to be modified is 16MB in all experiments. We can see the CoW has amplified the number of writes for both regular pages (4KB in size) and huge pages (2MB in size). For the first write, the average write amplification factor (number of physical memory writes versus logical writes to data blocks) is 7.07x for regular pages and 477.96x for huge pages. For the whole page write, the average write amplification factor is 1.87x for regular pages and 1.97x for huge pages.

III. Lelantus Design

In this section, we discuss the design choices of Lelantus. Before delving into the details of Lelantus, we first define our threat model.

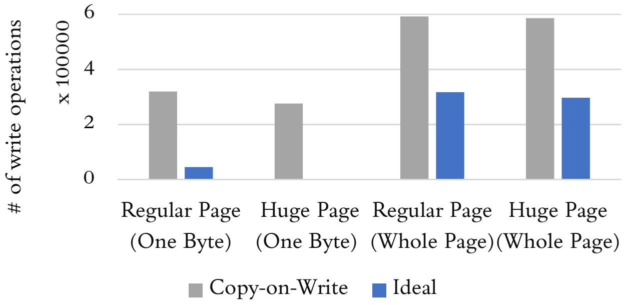


Fig. 2: Write amplification for CoW pages. The result is labeled in format: [size of pages (size of data been updated within each page)].

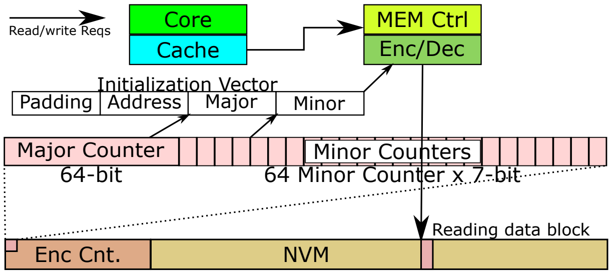


Fig. 3: Overview of secure memory.

Threat Model: In our paper, we assume secure NVM memory with the trusted boundary limited to the processor chip. Similar to the related prior work [3], [38], [41], we assume protection against physical access attacks, data and/or counter tampering, replay attacks, and bus-snoop attacks. The integrity of encryption counters and data is protected using Bonsai Merkle Tree (BMT) [29]. Access pattern leakage and side-channel attacks are beyond the scope of this paper.

Lelantus exploits the ability to track memory updates at fine-granularity in secure memory controllers. Such ability comes from the fact that secure memory controllers require metadata per cacheline to complete its encryption/decryption. Maintaining such security metadata at fine-granularity is essential to ensure temporal and spatial uniqueness of encryption pads, a critical security requirement [3], [39]. Lelantus aims at repurposing such metadata to additionally track uncopied cachelines within a page and allow reading them from the source page.

To better understand our scheme, let's first discuss how metadata in secure NVMs are handled. As discussed earlier (in Section II), in state-of-the-art secure NVM systems, each data block has a corresponding encryption counter in memory. In order to save memory space, the split-counter scheme is generally used to organize counters in a split fashion where the encryption counter is composed of a minor counter (per data block) and a major counter (shared across data blocks of the same page). Typically, the major counter is 64-bit, and the minor counter is 7-bit, and thus, for a 4KB memory page with 64B data blocks, the counter block (64 minor counters and one major counter) fits in 64B block [36].

Figure 3 shows how security metadata (encryption counters) are used in the context of secure NVM systems. As shown in the figure, for each memory operation (read or write), the counter block associated with the data block has to be read, and the initialization vector (IV) is established using minor counter, major counter, and others. The minor counter gets incremented on each write operation, *i.e.*, after each encryption, and the major counter is incremented only when minor counters overflow. However, the most related aspect to our work is that each read/writes from memory needs to check such security metadata.

The rest of the section discusses how we leverage such security metadata to track unmodified (not copied) data blocks efficiently and how to redirect them towards the source page while maintaining the semantic of CoW.

A. *Communicating CoW Information*

In Lelantus, we replace the actual copy operation at the first write of a CoW page with encoding the copy information in security metadata. Specifically, in the conventional process, first, a new physical page is created and mapped, and then the content of the source page is copied into the new page (destination). In Lelantus, this process is similar in that a new physical page is created. In contrast, instead of doing the actual copying to the new page, the security metadata of the new page would indicate that it is copied from another page and which cache blocks are not copied yet. Specifically, the kernel writes the following to memory-mapped IO registers within the memory controller: ① The physical address of the source page. ② The physical address of the new page (the copy). The modified kernel code adds memory fence after such write operations to ensure the memory controller has received them.

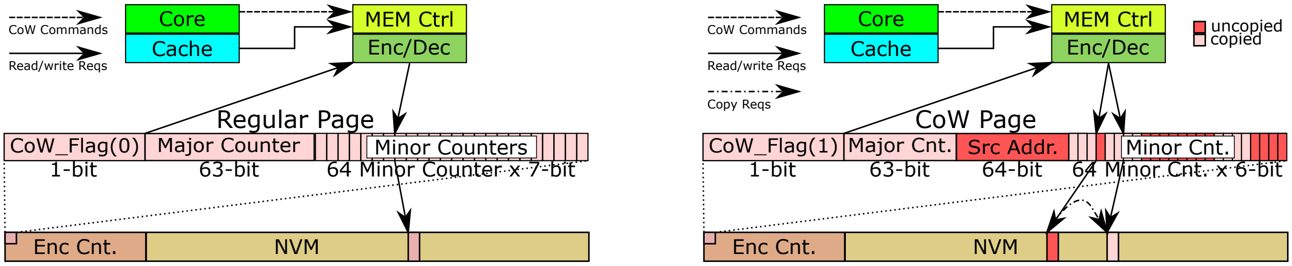
B. *Encoding CoW Mappings*

One critical question is how to leverage security metadata to maintain the aforementioned source page address and track not-copied-yet cache blocks within the destination page. To this purpose, we propose two distinct strategies. The first one is to reduce the size of encryption counters, thus increasing the chances of getting counter overflows. The second one is to add supplementary CoW metadata; therefore, it has a higher space overhead. Table I shows a summary of both schemes, which we will discuss in more detail later.

Encoding Scheme	**Minor Counter Overflow**	**Space Overhead**	**Extra RW Traffic**
Resizing Counter Blocks	200%	*none*	*low*
Supplementary CoW Metadata	0.07%	0.02%	*medium*

TABLE I: Comparison of two CoW encoding schemes.

Solution 1: Resizing Counter Blocks: To encode information about CoW pages in security metadata, we reserve one bit in each 64B counter block for a `CoW_Flag`. If the `CoW_Flag` is 0, then the counter block covers a regular page, *i.e.*, not a CoW page. Meanwhile, if `CoW_Flag` is 1, the counter block covers a CoW page. Based on whether the counter block covers a regular page or a CoW page, its format/anatomy is different, as shown in Figure 4. When used as a CoW page, minor counters size becomes 6-bit instead of 7-bit, and a 64-bit source page



(a) Counter Block for Regular Page.

(b) Counter Block for CoW Page.

Fig. 4: Our Solution 1: Lelantus: Resizing Counter Blocks.

address field is created using the 1-bit saved from each of the 64 minor counters. The source page address field, as its name indicates, holds the physical address of the source page from which the corresponding page is copied. Moreover, for counter blocks of CoW pages, a value of zero in minor counter indicates that the corresponding 64B cache block has not been copied yet to the corresponding page, *i.e.*, we need to read it from the source page. Meanwhile, if the counter block covers a CoW page and the minor counter of interest has a non-zero value, this indicates its corresponding block has been modified earlier. Hence, the requests to the corresponding block should no longer be forwarded to the source page.

Solution 2: Supplementing CoW Metadata: The major drawback of resizing counter blocks lies in the fact that a smaller minor counter for CoW pages increases the counter overflow rate, which in turn leads to more re-encryption cost. To resolve such a problem, in this solution, we employ supplementary CoW metadata to track the source page address. As shown in Figure 5, we create only one field in the CoW metadata to save the source page address, and store the CoW metadata in the same way as security metadata is stored in NVM. This scheme incurs minor space overhead, 8B per page. For example, given 4KB pages, the space overhead is 2B for each KB. However, we no longer need to reduce 1-bit in all 64 minor counters in order to track a 64-bit CoW source page address. Instead, we only reserve one value in the minor counter. A minor counter with the reserved value, zero as an example, indicates that the corresponding cache block belongs to a CoW page and has not been copied yet. A minor counter with any other values indicates that the corresponding cache block is a regular one. If a minor counter has a value of zero, we then issue another request to retrieve the source page address in the CoW metadata. To overcome the delay of querying such CoW metadata, we employ part of the counter cache as a small cache for CoW metadata. More specifically, we enable one counter cache slot, which is 8x8B in size, to host up to eight CoW mappings.

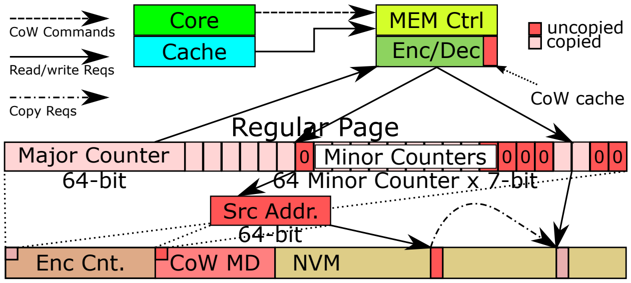


Fig. 5: Our Solution 2: Lelantus-CoW: Supplementing CoW metadata.

C. Decryption of CoW Blocks

In Lelantus, within a CoW page, there could be blocks that have been updated and others that are still not copied yet. One issue that arises here is deciding which major counter and minor counters to use for not-copied-yet blocks. For not copied yet blocks, the requested data blocks need to be fetched from the source page and thus needs to be decrypted using an initialization vector that consists of the major counter, minor counter, and address of the physical source page to ensure correct decryption. Re-fetching the counters of the source page adds delays to access CoW blocks, as depicted by Figure 6. Secure memory controllers employ a counter cache to reduce such delay. Since many CoW pages share the same source page, the counter block of the source page has an even higher locality. As a result, the delay of such re-fetching should usually be acceptable.

D. Handling Early Reclamation of Source Page

Usually, when we fork a child process to perform subtasks, the parent process continuously runs in the background. In this case, at least the parent process is owning the shared (source) pages. The child process can get the original data from the source pages without a problem. However, it is also possible for the OSes to reclaim the shared pages before the child processes release the copied pages. For example, the parent process may exist before the child process ends, and the parent process may make a private copy of the shared pages as well. Under both circumstances, the map count (the number of processes referring) of the source page decreases by one. When the map count reaches one, the system assumes the source page starts to be owned exclusively by one process. The kernel will then call *wp_page_reuse* to mark the source page as writable, and call *page_move_anon_rmap* to tell reverse lookup (rmap) code to not search parent or siblings. As a result, unless restricted by the application, there's no guarantee that the source page is continuously write-protected after the copy. We have observed extra writes to source pages in the early stage of this research. If the counter block of the copied pages still references to the source page, we could get modified

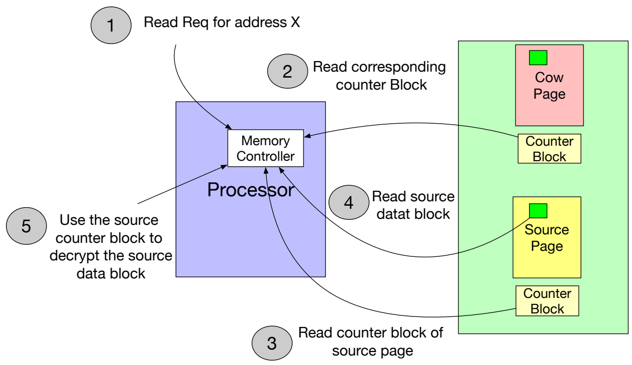

1
Read Req for address X
2
Read corresponding counter Block
Cow Page
Memory Controller
Counter Block
Processor
4
Read source datat block
5
Use the source counter block to decrypt the source data block
Source Page
Counter Block
3
Read counter block of source page

Fig. 6: Decryption of a CoW block.

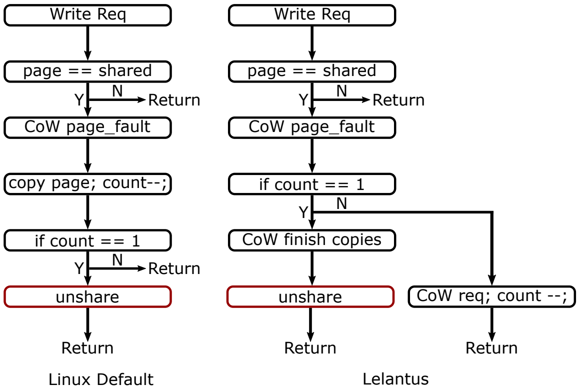


Fig. 8: Early reclamation of source page

data afterward instead of the original copy. One way to solve this problem is to capture the writing and releasing events of the source page, then perform a reverse lookup to find the copied pages. Moreover, we need to apply real copies for those uncopied cachelines before writing or releasing the source page.

Fortunately, since the physical to virtual page reverse lookup (rmap) code has already been implemented in the Linux kernel, we can easily trace the provenance of such copied pages. As shown in Figure 7, an *anon_vma* structure is created in the kernel for each process and linked with a structure called *anon_vma_chain*. A list of *vm_area_struct* structures from the same process are then linked to the *anon_vma_chain* via *same_vma* pointers. The *anon_vma_chain* from all the forked processes are connected in light of reverse lookup as well. From the *page* structure, which points to a related *anon_vma*, we can transverse the forked processes to trace all the possible copied pages.

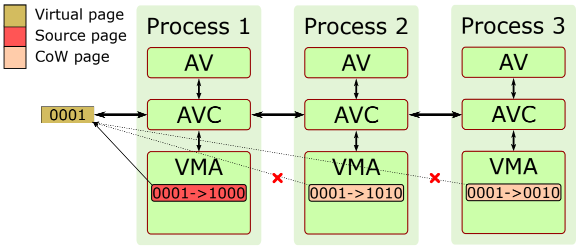


Fig. 7: Reverse lookup for source pages. "AV" is the *anon_vma* structure, and "AVC" is the *anon_vma_chain* structure. Each "VMA" represents a chunk of continuous virtual memory space, such as the stack or heap segments.

To implement this, as shown in Figure 8, as long as the page's map count reaches one, we pause the kernel's invocations to functions *wp_page_reuse* and *page_move_anon_rmap*. Hence, upon writing the page later, another page fault will kick in. The page fault handler checks whether the page's map count equals to one or not. If so, we handle the early reclamation of the source page before invoking the two above-mentioned delayed function calls. More specifically, we first perform a reverse lookup in the virtual memory area (VMA), which initiates the fault, to find the corresponding virtual address mapped to the source page. Second, given any of the page's parents or siblings, if their identical virtual addresses map to another physical page, this associated physical page could be a copied page. Finally, we send a physical page copy command to the memory controller to copy the uncopied cachelines. Note, it is possible to find a destination page that is not still pointing to the source page. Hence, we need to check again in the memory controller to see if the destination page still refers to the source page. Also, before releasing a (shared) source page, we check if the page is write-protected. If true, we perform the same operations to handle early reclamation.

Although we delay the function calls of *wp_page_reuse* and *page_move_anon_rmap* to unmask a "previously" shared page as write-protected until we actually perform an in-place update, we are not delaying page free operations. Upon freeing such a shared page, the early reclamation handler kicks in instantly to perform the remaining cacheline copies.

E. Handling Recursive Copy Chains

Since a forked process can also fork its child process, the OSes could copy a copied page again. Generally, when the secure memory controller observes a copy operation, the copied page should have at least one modified cacheline. Otherwise, the CoW technology delays the copy operation in the kernel. However, for huge pages, one modification results in multiple physical page copies. Moreover, part of the physical pages in a huge page could leave unmodified.

For example, suppose Lelantus first copies page A to page B. Later it copies page B to page C. If page B is unmodified, as in huge pages, we record page A's address for page C. Upon the decryption of page C's uncopied cachelines, we directly load page A's counter block. While reclaiming page A, we would schedule real copy for both page B and C. These real copies can be safely done in parallel to leverage row buffers and achieve maximum memory bandwidth.

On the contrary, if we already have some cachelines modified on page B, while copy it to page C, we save page B's address on page C's metadata. As for the decryption of page C's uncopied cachelines, we need to load page B's counter block. Again, if the corresponding cacheline on page B is uncopied, we need to bring page A's counter block. While reclaiming page A, we only need to perform a real copy for page A's direct sibling, page B, without impact page C.

Note that we do not delay the releasing of page A in any case. Thus, there is no garbage collection issue and no space overhead. Handling early reclamation and recursive copy mainly aims to guarantee correctness to avoid the potential crash. Therefore we have not evaluated related performance impact.

F. Security Discussion

Since encryption counter blocks have their integrity protected using Merkle Tree, any tampering with CoW metadata in counter blocks will be detected as in typical secure memory systems. Moreover, for the supplementary CoW metadata blocks scheme, the CoW metadata blocks can be treated as normal data blocks, *i.e.*, encrypted, and integrity protected using counter-mode encryption and integrity tree, respectively. Since such CoW blocks are encrypted, only if the CoW metadata is stored plainly in counter blocks, then some information can be exposed to the attackers. While such information is limited to indicate if the accessed block is copied from another address or not, such a relationship can be used to learn access patterns and leveraged in access pattern attacks. While access pattern leakage is beyond the scope of this work, such new information can be hidden through encryption, *i.e.*, the CoW metadata is encrypted within the counter block.

G. Applicability to Non-Secure Memory

While Lelantus leverages security metadata to encode the CoW information, it can be applied to unencrypted memories with slight modifications. The only requirement is that the CoW data are associated with each cache block and visible to the memory controller. We choose to alter security metadata, which is available in platforms use Intels TME [17], Intels SGX [11], AMDs SME [20], as they are maintained per cacheline and visible to the memory controller. Any other metadata that satisfies our requirements, such as memory tagging [27], [30], can be applied to implement Lelantus with minor modifications. Moreover, maintaining and fetching encryption blocks has minimal overheads, as shown in prior studies [4], [38], [41]. Thus, even if memory is not encrypted, Lelantus can use similar counter blocks that incur minimal storage overheads ($\approx 1.5\%$) and negligible performance overheads. Note that if Lelantus is used for non-secure NVM, counter-like blocks do not need to be protected by Merkle Tree, and hence Lelantus only incurs the overheads of retrieving and updating the counters.

IV. Implementation

In this section, we discuss the implementation of Lelantus.

A. Memory Controller Support

As described in Table II, we introduce three CoW commands in the memory controller: *page_copy*, *page_phyc* and *page_init*. These commands enable the software to offload the page copy and allocate operations to the hardware. Similarly, bulk instructions, such as *rep movsd*, *rep stosb*, *ermsb* in x86 [16] and *mvcl* in IBM S/390 [10], are already presented in several modern processors. However, those CPU instructions would require significant effort to extend ISA, change the microarchitecture, and add new decoding units in CPUs. Our idea is to add a memory-mapped IO register in the memory controller for communication. Specifically, the *page_copy* command tells the memory controller to save the source page address on the metadata of the destination page. The *page_phyc* command checks if the source page address is equal to the address saved on the metadata of the destination page. If true, the memory controller then performs a real physical copy for the uncopied cachelines. The *page_free* command tells the memory controller to drop the source page address on the metadata of the destination page even if there are uncopied cachelines.

Command	Parameters	Semantics
page_copy	src, dst	Logical page copy.
page_phyc	src, dst	Physical (real) page copy.
page_free	dst	Clear CoW metadata for dst.

TABLE II: Semantics of the memory controller commands.

There are three crucial aspects of the execution semantics of our CoW commands. First, all commands rely on the software to take care of the alignment and size constraints. Page alignment enables Lelantus to leverage the security metadata to track the copying/initializing. It is also feasible to let the microarchitecture to determine if Lelantus can fully/partially accelerate a particular instance of copying/initializing. However, since the modern operating system manages current physical main memory in the unit of page granularity, it is natural to re-implement existing page manipulation functions to max out the benefits from Lelantus. Second, the microarchitectural implementation ensures that any data in the on-chip caches are kept consistent during the execution of these operations. Third, for processors that perform out-of-order execution, the activities to the destination pages need to be stalled. In the meanwhile, the processor can still accept interrupts and handle any page faults during the execution of these operations.

B. Processor Microarchitecture Support

To guarantee the cache coherence, upon the submission of the *page_copy* command, the processors need to conduct 1) flushing any dirty caches of the source page, and 2) invalidating any caches (clean or dirty) of the destination page. Otherwise, applications can get stalled data while reading the destination page. The good news is, the processors already implements such flush and invalidate instructions, and the operating systems can conduct such cache flushing/invalidating operations. First, before marking the source page as write-protected, its on-chip cache should be flushed by the OSes. Second, while allocating a new physical page, the OSes should invalid its on-chip cache. Note, although Lelantus take care of cache coherence issues, there are no cache consistency issues.

C. Operating System Support

In the Linux kernel, we mainly perform three modifications to the memory management functions to utilize the above mentioned CoW commands. The kernel implements CoW pages in the unit of page granularity. Whenever an application writes to a CoW page, the processor generates a CoW fault. The kernel then handles the CoW fault and copy the data from the CoW page to a freshly allocated new page by using *copy_page* kernel function. After the copying, the kernel then updates the page mapping and point the virtual addresses to the freshly allocated physical page. Finally, the processor resumes the write operations. As this *copy_page* function is already page size aligned, we only need to replace the corresponding function by sending the source and destination pages in a *page_copy* command. Other kernel features, such as fork, kernel same-page merging, and huge pages, which directly or indirectly rely on the *copy_page* kernel functions, can then benefit from Lelantus transparently. For example, the kernel translates the copy of a huge page into a set of physical page copy operations.

Another necessary kernel modification is to perform additional reverse lookup before the releasing or writing a shared page. As discussed in Section III-D, we do not add additional overhead to trace where we copied a shared page. While writing or releasing a shared page, it's map count should be equal to one, which means only one process is owning the shared page. We need to transverse the *anon_vma_chain* to find the new "destination" page where the same virtual address is mapped to in other forked processes. We then send the source and destination pages in a *page_phyc* command. The memory controller then checks if the destination page is already copied from the source before performing the physical copy. By proposing *page_phyc* instructions, we delay the copy operations further compared to the CoW implemented in the kernel. This delay enables the memory controller to merge more writes and copies in the request queue.

The last kernel modification is to send a *page_free* command when releasing a copied page. When we free a copied page, all the unperformed cacheline copies would be no longer necessary. Thus, we remove the related metadata if it still exists to get rid of the unwanted copy.

V. Evaluation

In this section, we conduct performance evaluations.

A. Methodology

We model Lelantus in Gem5 [6] with the system configuration presented in the Table III. To perform full-system simulation, we modify the Linux kernel v5.0 to mainly re-implement the *copy_user_page*, *do_wp_page* and *put_page* functions. We extended gem5's memory controller to include a memory-mapped I/O register and a counter cache. The memory-mapped I/O register enables the above mentioned three CoW commands: *page_copy*, *page_phyc* and *page_init*. We assume all CoW commands have the same transfer latency as write operation from the processor to the memory controller. We implement a 256KB, 16-way set associative counter cache, with a total number of 4K counters and employ the battery-backed write-back scheme as the counter cache management scheme by default. The counter blocks are randomly initialized to model the counter overflow. Similar to prior work, we assume the overall AES encryption latency to be 24 cycles, and we overlap fetching data with encryption pad generation. We compare four CoW schemes in our evaluation:

① **Baseline** that is implemented in default Linux kernel.
② **Lelantus** that resizes the counter blocks to save the source page address.
③ **Lelantus-CoW** that employs supplementary CoW metadata. 32KB of a 256KB-sized counter cache is reserved for CoW metadata.
④ **Slient Shredder** [3] that only avoids zero initialization. An encryption counter has a value of zero indicates the corresponding cacheline is filled with all zeros.

Component	Parameters
Processor	8-core, 1GHz, out-of-order x86-64
L1 Cache	2 cycles, 64KB size, 8-way, LRU, 64B block
L2 Cache	8 cycles, 512KB size, 8-way, LRU, 64B block
L3 Cache	25 cycles, 8M size, 8-way, LRU, 64B block
Main Memory	16GB, 2 ranks, 8 banks
PM Latency	60ns read, 150ns write
Page Size	4KB, 2MB
Counter Cache	256KB, 16 way, LRU, 64B block

TABLE III: Configuration of the simulated system.

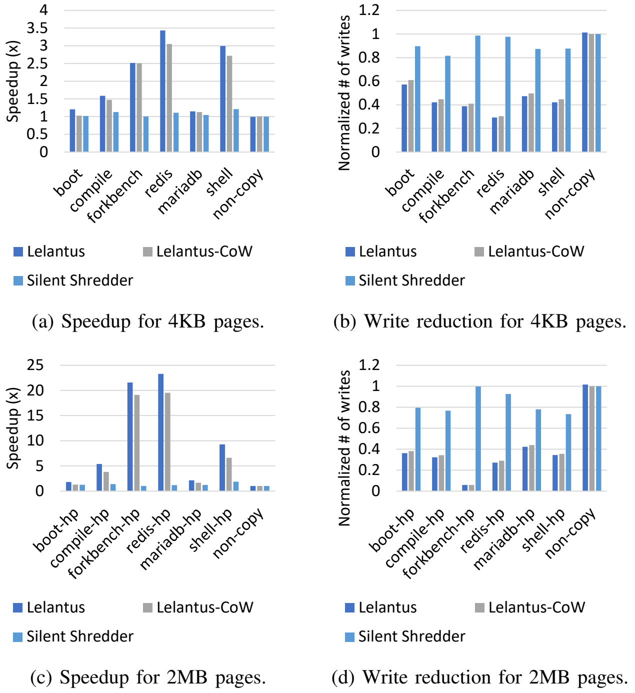
B. Copy/initialization-intensive Applications

To stress-test our design, we select six representative copy/initialization-intensive real-world applications in the Table IV. The goal is to evaluate the performance and the write overhead of our design. In all experiments, the full-system simulation is fast-forwarded to skip the initialization phase and then followed by the execution of the specific phase of the benchmark applications. To study the impact of huge pages, we use *libhugetlbfs* [1] to enable huge pages for particular applications, except that in *boot* benchmark, we use huge pages for the entire system.

Name	Description
boot	A phase booting up the Buildroot embedded Linux system (while starting processes by reading information from the /etc/inittab file).
compile	A phase the GNU C compiler performing the compilation by running cc1.
forkbench	A phase of running the forkbench described in Section V-D.
Redis	An in-memory key-value database, a phase inserting many key-value pairs while forking a background persist instance.
mariadb	An on-disk database system, a phase loading the sample employeedb database.
shell	A Unix shell script running "find" on a directory tree with "ls" on each sub-directory (involves filesystem accesses and spawning new processes).

TABLE IV: Copy/initialization-intensive benchmarks.

The forkbench and Redis (in a phase of doing bgsave) are highly copy/initialization-intensive applications. The fork is one of the most expensive yet frequently-used system calls in modern systems. When updating the shared pages from the parent or child process after the fork, the CoW technique triggers a large number of page copy operations. By performing CoW in the cacheline granularity and delaying the copy operations further, Lelantus can significantly improve the performance of a forked process. To validate this idea, we develop a micro-benchmark, *forkbench*. In the forkbench, we first initialize a number of memories (16K in this experiment), we then fork a child process to update those initialized memories. The performance results are taken between the child process finish updating. In the experiments, we uniformly update 32 cachelines per page for regular pages and 512 cachelines per page for huge pages. The Redis [7] is an in-memory database that relay on point-in-time snapshots to perform data persistence. Whenever Redis needs to dump the dataset, it forks a child process and a parent process. The child then starts to persist the dataset to a temporary RDB file; the parent process continuously serves the client requests. This method allows Redis to benefit from CoW semantics. In the evaluation, we collect the insert performance of Redis while the child process is persisting the data. Although it is common to use Redis as a cache that is read dominant, it also widely deployed for statistics and memorization, which is both read and write dominant. We initialize Redis with 100K key-value pairs then perform 10K set and get operations. Besides, we also select the boot, mariadb, compile, and shell workloads as the moderately copy/initialization-intensive applications. These workloads also evolve extensive I/O requests through direct memory access (DMA) after the forking.



(a) Speedup for 4KB pages.

(b) Write reduction for 4KB pages.

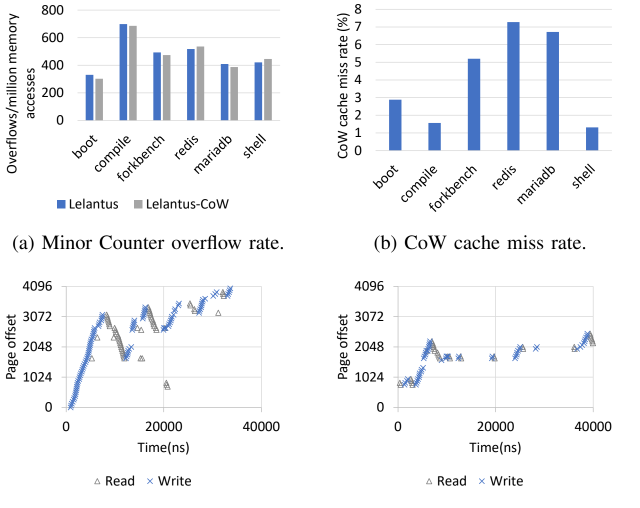
(c) Speedup for 2MB pages.

(d) Write reduction for 2MB pages.

Fig. 9: Lelantus on copy/initialization-intense applications.

C. Experimental Results and Analyses

To validate the effectiveness of Lelantus, we first collected end-to-end results of two Lelantus schemes and the baseline. Figure 9a and Figure 9c compares the performance of the baseline with that of Lelantus. As seen from the figure, Lelantus improves the performance of all applications. Specifically, the performance of boot and mariadb, which consists of a negligible number of copy operations, are marginally improved by the Lelantus for regular pages by 20% and 14.7%. For huge pages, the performance improvement of boot and mariadb is 57.1% and 47.4%, respectively. Regarding the two copy intensive applications forkbench and Redis, performance improvement for regular pages is as much as 2.24 and 3.43 times. The speedup is much more significant at a huge-page setting. For forkbench, Lelantus is 30.57 times faster. For Redis, Lelantus is 23.28 times faster. Lelantus speed ups the compile benchmark by 1.58 times for regular pages and 5.39 times for huge pages. The speedup of the shell benchmark is 2.99 times for regular pages and 9.27 times for huge pages. Silent Shredder speed ups the performance by 1.20 times on average, which is much less than Lelantus, especially for forkbench and redis. This is because Silent Shredder only avoids zero initializing pages, which is a small percentage of CoW operations.



(a) Minor Counter overflow rate.

(b) CoW cache miss rate.

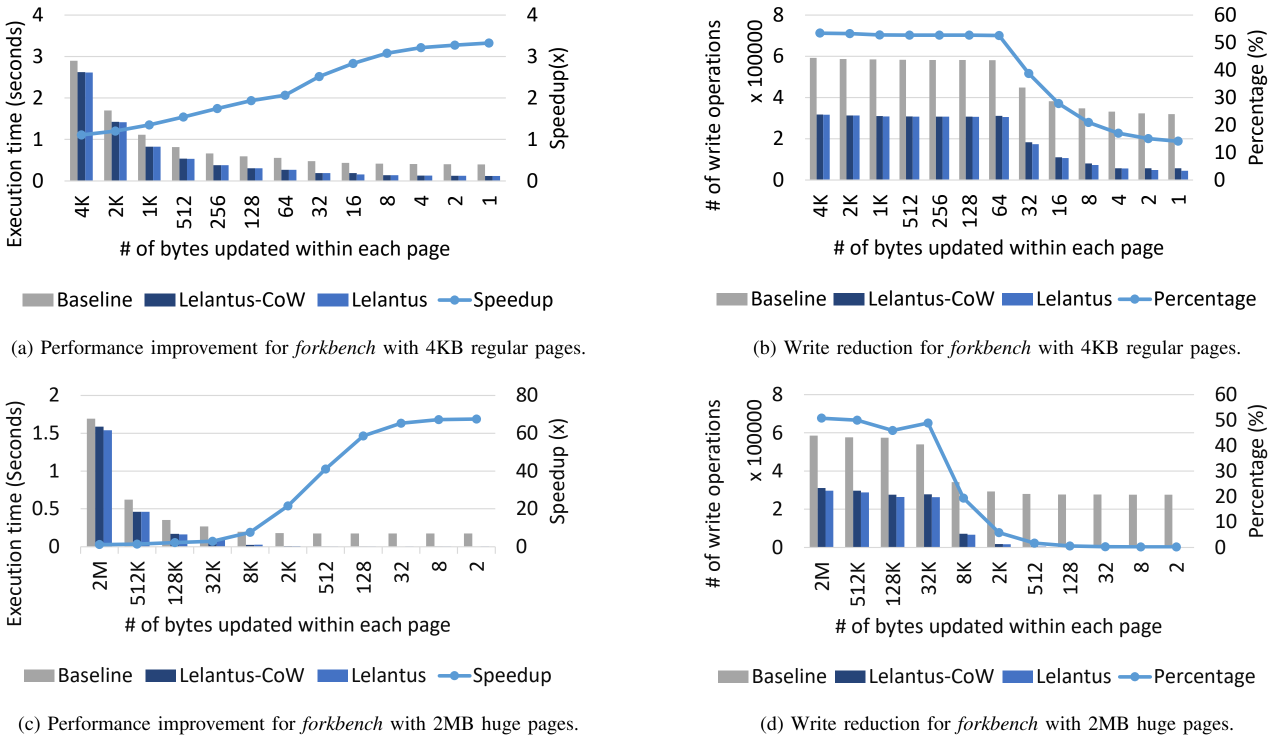
(c) Page access for Baseline.

(d) Page access for Lelantus.

Fig. 10: Comparison of Lelantus design choices.

The performance improvement mainly results from fewer writes in Lelantus. As illustrated in Figure 9b and Figure 9d, Lelantus reduces the number of writes for all the applications mentioned above. More specifically, the number of writes in boot and mariadb is cut to 57.12% and 47.36% of baseline system for regular pages. For forkbench and Redis, the average number of write requests is reduced to 38.76% and 29.20% respectively for regular pages. This explains why Lelantus speeds up the most in both cases, as echoed in Figure 9a and Figure 9c. The number for compile and shell workloads is 42.12% and 42.15%. The reduction of write requests when using huge pages is much more significant. The boot and mariadb have reduced the number to 36.10% and 42.15% for regular pages. For forkbench and Redis, the average number is 20.32% and 27.19%, respectively. For compile and shell workloads, they are 32.28% and 34.39%. Comparing to Lelantus, Comparing to Lelantus, Lelantus-CoW has introduced 5% extra writes on average. This is because Lelantus-CoW needs to perform additional writes to update the CoW metadata in each *page_copy* command. For comparison, Silent Shredder reduces the number of write requests by 13% on average.

To study the overhead, we develop a non-copying intensive workload, which is shown as non-copy in Figure 9. In non-copy, we skip the initialization phase then launch the same workload as forkbench to modify all allocated memory without spawning a child process. As seen from the figure, both Lelantus and Lelantus-CoW have no impact on the performance of



(a) Performance improvement for *forkbench* with 4KB regular pages.

(b) Write reduction for *forkbench* with 4KB regular pages.

(c) Performance improvement for *forkbench* with 2MB huge pages.

(d) Write reduction for *forkbench* with 2MB huge pages.

Fig. 11: Effect of Lelantus on *forkbench*.

the regular page read/write. This is because Lelantus/Lelantus-CoW does not change the read/write request workflow for a regular page. Also, because the size of the minor counters for regular pages is untouched in both design, their counter overflow ratios are comparable.

To gauge the efficiency of Lelantus, we collect encryption counter overflow overhead, CoW cache miss rate, and page access footprint, as illustrated in Figure 10. The overhead of Lelantus is a function of the number of counter overflow. As shown in Figure 10a, both Lelantus schemes bear an extremely low overflow possibility, as low as one ten-thousandth. This probably derives from the fact that the update frequency of each cacheline is not high. For a CoW page, 6-bit minor counter is able to accommodate up to 62 writes before it overflows. For a regular page, 7-bit minor counter holds up to 127 writes before it overflows. However, assuming the minor counters for regular pages are randomly initialized, the average number of writes before overflow is equal to 63. Our experiments indicate that it is unusual to update one cacheline more than 60 times. The Lelantus-CoW overhead is a function of CoW cache misses. In Figure 10c and Figure 10d, we plot the memory access footprint of CoW pages with writes engaged. In the baseline system, the page copy function initializes the whole page before any other operations are issued. While in Lelantus, we successfully avoid such copy operation, as evidenced by the fact that only a few scattered cachelines are accessed. Finally, in Table V, we present the percentage of copy operations. The greater percentage of copy traffic of an application, the higher the speedup of both Lelantus solutions.

boot	compile	forkbench	redis	mariadb	shell
51.96%	46.32%	82.77%	71.57%	48.11%	59.1%

TABLE V: Percentage of copy and initialization traffic.

D. The fork System Call

We conduct a sensitivity study on how both Lelantus schemes work with different numbers of update bytes per page at a wide range of page sizes and the number of cachelines. This is to validate two key designs in Lelantus: 1) integrating the cacheline copy with its first write, and 2) avoiding copying unmodified cachelines as much as possible.

In the previous evaluation, we run the forkbench under specific settings. However, the performance speedup of the forked process depends on two above-mentioned parameters: 1) the size of the page used by the OSes - which determines how much data may be supposed to be copied by the Baseline, also 2) the number of cachelines updated within each page - which determines how much data are really copied by Lelantus. In order to exercise both parameters, we first vary the number of bytes to be updated in both regular and huge pages. We then make all the writes in the child process evenly distributed. Suppose we use 4K page size and 64 bytes block size, while uniformly update 64 bytes in the page, we write one byte for each cacheline. The parent process waits until the child process to complete updating the pages. Finally, we collect the execution time and number of write operations issued during the execution of the child process.

Figure 11 shows the evaluation results for Lelantus on forkbench. We study both the 4KB regular page size and the

2MB huge page size. To understand the performance impact of Lelantus, we update a different number of bytes within the page while we fix the total number of pages updated to 16KB. Due to the high locality in the given workload, the cache hit ratio in CoW metadata and encryption counters are both very high. Thus, the performance difference between Lelantus-CoW and Lelantus is neglectable for all experiments. Overall, both Lelantus schemes excel the baseline due to the aforementioned two core designs. Specifically, for the regular pages, Lelantus is 1.11 times faster than the baseline when the whole page is updated, and 3.33 times faster when only one byte is updated. For the huge pages, Lelantus speeds up the performance by 1.10 times when the whole page is updated and by 67.53 times when only one byte is updated. In the meantime, Lelantus can reduce the number of writes to the memory to 53.45%-14.14% for regular pages and 50.76%-0.20% for huge pages.

It may be noted that there is a knee point when the number of bytes updated is less than 64 in regular pages, and 32K in huge pages, respectively. This lies in a fact that, when the number of updated bytes is less than the number of cachelines, and hence some of the cachelines are not modified, Lelantus enables unmodified cachelines no need to be copied. Namely, Lelantus enables the physical copy to operate in fine block granularity. Note we can observe the average performance speedup for the first write (update one byte for each page) to a CoW page is 3.33 times for regular pages and 67.53 times for huge pages.

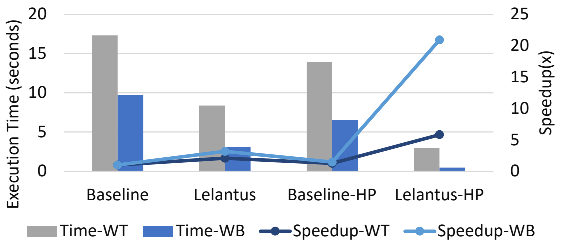


Fig. 12: Impact of write-through (WT) and write-back (WB) schemes for encryption counter.

E. Impact of Write Strategy in Counter Cache

Every CPU write leads to one write to data page and one write to encryption counter in secure memory. Lelantus emphasizes at former writes while it is intriguing to study how latter writes effect Lelantus. We vary the updating scheme for the encryption counter while running the Redis workload. In the write-through scheme, all updates to the encryption counter update are flushed to memory immediately. In the write-back scheme, the updates to the encryption counter are returned when the write hit the cache, and the data is flushed to memory by cache evict policy. Figure 12 plots the evaluation results. The bars show the average execution time for inserting key-value pairs while performing background checkpointing, and the lines show the performance speedup. We calculate the speedup by comparing it with the baseline using write-through or write-back schemes, respectively. We can see that by switching to the huge pages, the write-through scheme is 1.24 times faster, and the write-back scheme is 1.47 times faster. Lelantus can further improve performance. While using regular pages, Lelantus can speed up the performance by 2.07 times and 3.16 times for write-through and write-back schemes, respectively. While using huge pages, Lelantus can speed up up the performance by 5.83 times and 20.94 times for write-through and write-back schemes, respectively.

VI. Related Work

Offloading Copy/Initialization Operations. Prior works [18], [31], [40] have proposed to offload bulk data copy operations to a separate engine or memory controllers. Seshadri et. al. [31] aims to eliminate the data transfer over the memory channel by performing bulk data copy and initialization using row buffers in DRAM. Jiang *et. al.* [18] propose to use a dedicated DMA engine close to the memory controller to complete bulk data copy and initialization. This design can reduce pipeline stalls (that waiting for the entire copy operation to complete) and avoid cache pollutions. As all prior design works, the copy and initialization still occur at page granularity in a conventional DRAM setting. To take advantage of newly released NVM as the main memory, we have already shown the effectiveness of Lelantus in performing the copy operations at cacheline granularity. The techniques to bypass memory channel data transfer, reduce pipeline stalls, and avoid cache pollution can be combined with Lelantus in an orthogonal fashion to improve performance further.

Bulk Data Initialization. Lewis *et. al.* [23] track the uninitialized memory at a cacheline granularity. They then avoid fetching uninitialized blocks on initializing store misses. While their work and Lelantus are similar in terms of performing cacheline granularity memory tracking, Lelantus further reduce the unnecessary cacheline copy and initialization. It is also feasible to combine their work to conduct copy or initializations in on-chip caches directly to minimize the reads further and improve performance.

Reducing Write Overhead in Secure Memory. Due to data remanence attack, NVMs are paired with encryption. However, it exacerbates the write endurance of NVM. To improve write endurance in secure NVM, Chhabra *et. al.* [9] firstly proposed to encrypt different parts of the main memory at a different time based on the prediction that data in a particular part is no longer used by the processor to enhance memory performance. Young *et. al.* [39] further suggested to re-encrypt the words with actual change instead of whole cacheline encryption for each write operation to reduce unnecessary writes. Later, Awad *et. al.* [3] repurposed initialization vectors used in standard counter mode encryption to indicate zeroed cachelines without physically writing zeros to them in data shredding stage. Likewise, Lelantus repurposes split counters to improve write endurance and gain performance by gradually modifying cachelines in a newly copied page during the page initialization stage in NVM. We believe this approach is orthogonal to all methods mentioned above.

Fine Granularity Cacheline Tracking. Ni *et. al.* [26] propose to add a bitmap in TLB to track updated cachelines for shadow sub-paging. Their primary goal is to solve the data consistency issue as compared with logging, while Lelantus mainly targets fork related CoW operations. Lelantus achieves fine granularity cacheline tracking by leveraging existing split counters in secure memory. In addition, Lelantus imposes no changes to the TLB, and mainly modifies the memory controller and its interactions with OSes. Hence, it is easier to be deployed into production systems incrementally. Last, Lelantus could be further extended to support shadow sub-paging.

VII. Conclusion

Limited write endurance and data remanence vulnerability have been the main challenges that hinder the adoption of NVM as the main memory. Secure NVM controllers resolve the data remanence issue by introducing encryption counters at cacheline granularity. However, due to the limited write endurance and slow write operations of NVMs, bulk operations, especially CoW operations in most modern OSes, are still extremely expensive and can easily lead to throttling the memory system. In this paper, we repropose encryption counters in Lelantus to enable fine-grained CoW operations. We implement Lelantus based on the Gem5 simulator and Linux kernel v5.0. The evaluation results for six copy/initialization-intensive real-world applications show an average 2.25x speedup when using regular pages and 10.57x speedup when using huge pages. Meanwhile, the average number of writes has reduced to 42.78% for regular pages and 29.65% for huge pages.

ACKNOWLEDGEMENT

This project is supported in part by the US National Science Foundation Grant CCF-1527249, CCF-1717388 and CCF-1907765, CNS-1814417.

References

[1] "Libhugetlbfs," 2010. [Online]. Available: https://github.com/libhugetlbfs/libhugetlbfs

[2] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the linux symposium*. Citeseer, 2009, pp. 19–28.

[3] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 263–276, 2016.

[4] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 104–115.

[5] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 21.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[7] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[8] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "Cmd: classification-based memory deduplication through page access characteristics," in *ACM SIGPLAN Notices*, vol. 49, no. 7. ACM, 2014, pp. 65–76.

[9] S. Chhabra and Y. Solihin, "i-nvmm: a secure non-volatile main memory system with incremental encryption," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 177–188.

[10] I. Corporation, "Enterprise systems architecture/390 principles of operation," 2001.

[11] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[12] I. Cutress and B. Tallis, "Intel launches optane dimms up to 512gb: Apache pass is here!" 2016. [Online]. Available: https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here

[13] HP, "The Machine: A new kind of computer," http://www.hpl.hp.com/research/systems-research/themachine/.

[14] Y. Huai, F. Albert, P. Nguyen, M. Pakala, and T. Valet, "Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions," *Applied Physics Letters*, vol. 84, no. 16, pp. 3118–3120, 2004.

[15] Intel, "Intel 3D XPoint," IEEE, 2012.

[16] ——, "Intel 64 and ia-32 architectures optimization reference manual," 2012. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[17] ——, "Intel architecture memory encryption technology specification," 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf

[18] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, "Architecture support for improving bulk memory copying and initialization performance," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 169–180.

[19] P.-H. Kamp, "Malloc (3) revisited." in *USENIX Annual Technical Conference*, 1998, p. 45.

[20] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[21] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 1–12.

[22] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture*, 2009.

[23] J. A. Lewis, B. Black, and M. H. Lipasti, "Avoiding initialization misses to the heap," in *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 183–194.

[24] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, 2013.

[25] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "Xlh: More effective memory deduplication scanners through cross-layer hints," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC13)*, 2013, pp. 279–290.

[26] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 836–848.

[27] Oracle, "Hardware-assisted checking using silicon secured memory (ssm)," https://docs.oracle.com/cd/E60778_01/html/E60755/gphwb.html, 2016.

[28] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 14–23.

[29] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 183–196.

[30] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves c/c++ memory safety," *arXiv preprint arXiv:1802.09517*, 2018.

[31] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "Rowclone: fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 185–197.

[32] P. Sharma and P. Kulkarni, "Singleton: system-wide page deduplication in virtual environments," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, 2012, pp. 15–26.

[33] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou *et al.*, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 2004, pp. 29–44.

[34] R. Tetzlaff, *Memristors and memristive systems*. Springer, 2013.

[35] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2013, pp. 1–12.

[36] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 179–190.

[37] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[38] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 403–415.

[39] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 33–44, 2015.

[40] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell, "Hardware support for bulk data movement in server platforms," in *2005 International Conference on Computer Design*. IEEE, 2005, pp. 53–60.

[41] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322252