CoWLight: Hardware Assisted Copy-on-Write Fault Handling for Secure Deduplication

Santhosh Kumar T
Indian Institute of Technology Kanpur, India
santkum@cse.iitk.ac.in

Debadatta Mishra
Indian Institute of Technology Kanpur, India
deba@cse.iitk.ac.in

Biswabandan Panda
Indian Institute of Technology Kanpur, India
biswap@cse.iitk.ac.in

Nayan Deshmukh*
Indian Institute of Technology Kanpur, India
nayan26deshmukh@gmail.com

ABSTRACT

Memory deduplication in virtualized systems is shown to be a very useful memory optimization as it is simple to use and provides memory efficient cloud hosting. However, memory deduplication based side channel attacks—information disclosure attacks and covert channel construction across virtual machines—can be mounted using the timing information available because of Copy-on-Write (CoW) fault handling semantics. The CoW semantic has been a necessary-evil with regard to deduplication as it plays a vital role in supporting guest OS transparent deduplication but enables a timing channel for exploitation. Thus to decimate the huge access time difference between a normal write and a write to a shared page, we propose CoWLight, a combination of hardware and software techniques for handling the CoW page faults in an efficient manner. In this work, we propose to address the security issues at its genesis as opposed to mitigate the side-effects by offloading the CoW fault handling to the hardware itself. Further, we show that CoWLight can reduce the access latency differences significantly (by up to 30x) which is within the noise thresholds in a moderately busy system.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; • **Computer systems organization → Architectures**.

KEYWORDS

secure deduplication, copy on write, page fault handling

ACM Reference Format:
Santhosh Kumar T, Debadatta Mishra, Biswabandan Panda, and Nayan Deshmukh. 2019. CoWLight: Hardware Assisted Copy-on-Write Fault Handling for Secure Deduplication. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3337167.3337170

*The author contributed when he was a student at IIT Kanpur.

HASP '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-7226-8/19/06…$15.00
https://doi.org/10.1145/3337167.3337170

1 INTRODUCTION

Cloud computing provides support for on-demand usage of computing resources and enables services with pay-on-use business models. Virtualization solutions (a.k.a. hypervisors), the core enablers of cloud computing, play an important role in realizing cloud computing platforms by efficiently virtualizing the physical resources across multiple virtual machines (VMs) and managing the resources in an efficient manner. Memory is a precious resource in a multi-hosting setup and most hypervisor designers target efficient management of memory as it determines the degree of multi-hosting in a given physical machine.

For efficient memory management, most hypervisors implement content based memory deduplication where memory pages with same content are deduplicated by maintaining a single copy [2, 12, 15]. One of the approaches of memory deduplication is to scan all memory pages used by the VMs periodically from within the hypervisor, find and remove the duplicates by changing the memory mapping of VMs (using the pages containing duplicate content) to point to a single copy with read-only permissions [2, 15]. On a write access from any of the VMs to a shared page, *a page fault* trap is generated by the hardware. The hypervisor page fault handler provides an exclusive copy of the shared page to the VM, a technique commonly known as Copy-on-Write (CoW) [2, 14]. Periodic scan and merge based memory deduplication techniques are common in both open source (e.g., KSM in Linux KVM) and commercial hypervisors (e.g., VMWare ESX). Several research works have shown the benefits of deduplication in terms of memory efficiency at the cost of some additional CPU resource [3, 14, 16]. While memory deduplication improves the memory efficiency of virtualized systems, it also creates serious security vulnerabilities [18]. In this paper, we attempt to address *the security issues* related to memory deduplication.

At the core of all security vulnerabilities arising due to deduplication lies the *timing information leaked on write to shared pages* i.e., write to a normal (not shared) page is orders of magnitude faster than write to a shared page (See §2.1). For example, covert channel construction [8, 17] relies on this timing differences to realize a signalling mechanism (Refer §2.2 for details). Similarly, information disclosure attacks try to exploit deduplication and the timing channel to gather the application profile of other VMs [8, 18]. Existing approaches to mitigate the security issues [13, 18] either disable the deduplication completely or employ a very cautious approach to select the memory to deduplicate on explicit administrator instructions or implicit memory characterization. To the best of our

knowledge, this work presents the first approach to address the issues at its origin by augmenting the CoW fault handling at the architecture and the OS layers.

In the current system, the overheads of CoW faults can be attributed to three major components. First, on a write to a shared page, the hardware page table walker generates a fault causing a pipeline flush to ensure a precise state before jumping to the OS page fault handler routine. Second, the OS fault handler handles the fault by performing a full page copy irrespective of the memory operand size of the faulting instruction. Third, on a return from fault, the pipeline is flushed again impacting the instruction level parallelism significantly. In this paper, *we propose a prevention technique at the root of the problem rather than mitigating the side-effects resulting from the core issue.* We address the first and the third issues by avoiding a page fault in the first place. To achieve no page faults on writes to shared pages, we propose a specialized hardware page fault handling with minimal changes to page table translation logic coupled with OS assistance. To address the second issue, we design a solution that incurs write latency proportional to the operand size (or a cache line) using minimal hardware enhancements at the L2 cache controller.

We have implemented and evaluated our prototype on Gem5 [4] full system architectural simulator and Linux kernel (§4). The evaluation results show that, our approach can reduce the effectiveness of the timing channel by reducing the CoW fault handling latency by up to 30x compared to state-of-the-art CoW handling latency and is comparable to normal write latency (write to a page with write access) in a noisy system. Moreover, efficient detection techniques can be designed by leveraging the historical information provided by the hardware without any additional overheads.

Our main contributions are,

- We establish the root-causes of the timing channel that exposes various vulnerabilities in memory deduplication, an essential optimization for efficient memory management in virtualized systems (§2).
- We, design and implement CoWLight, a hardware-assisted light-weight page fault handling technique to nullify the timing information to a great extent otherwise available and exploited through orchestrated Copy-on-Write exploitations (§3 and §4).
- Experimentally validate the efficacy of CoWLight and its effectiveness towards achieving secure deduplication (§5).

2 BACKGROUND AND MOTIVATION

In this section, we provide a brief overview of the memory deduplication process, significance of Copy-On-Write mechanism in realizing memory deduplication. We also show covert channel construction on a cloud setup as an illustration of exploiting the timing side-channel available as a side-effect of memory deduplication. Specifically, the significant access latency difference between a write to a non-shared page and a shared page where a write to shared page results in a CoW fault. Further, we motivate CoWLight by highlighting the noise-free and reliable timing information available in the systems employing memory deduplication. We also present an analysis of the root-causes of the timing information and lay down the basis of our proposed design.

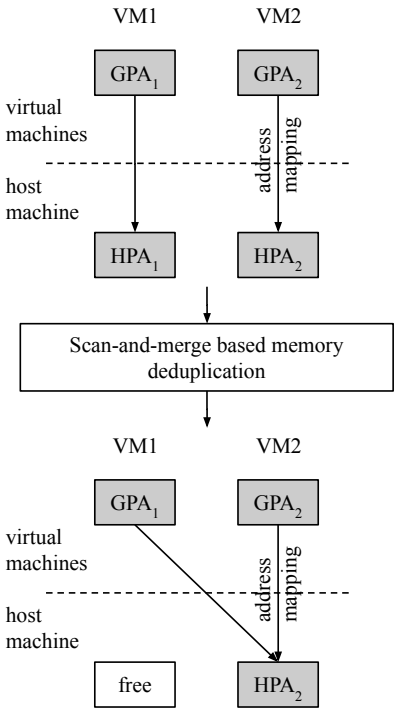

Figure 1: Scan-and-Merge based memory deduplication in virtualized systems. GPA: Guest Physical Address and HPA: Host Physical Address

2.1 Out-of-band scan and merge based memory deduplication

In virtualized systems, a commonly used memory deduplication technique is based on out-of-band scan and merge. A scanner daemon, executing in the hypervisor or host OS, periodically scans all the memory pages used by the VMs and locates the memory pages with similar content. When a duplicate memory page is found, the guest physical address (GPA) to host physical address (HPA) mapping is updated to the unique copy and the old HPA is freed (Figure 1). As the HPA is shared by multiple VMs after deduplication, write to the shared page (a.k.a. sharing break) is required to be handled in manner such that the correctness is ensured. Therefore, the GPA to HPA mapping of all VMs are marked as read-only such that whenever a VM tries to modify the shared page, a page fault is generated. The hypervisor handles the page fault by creating an exclusive copy which is commonly known as Copy-on-Write (CoW). Note that, in virtualized systems, there are two different page table levels managed independently by the guest OS and the hypervisor. Therefore, the page table updates and CoW fault handling to realize the benefits of deduplication is transparent to the guest OS.

Hypervisors like VMWare ESX [15] and KVM [10] provide page deduplication feature as a memory optimization in virtualized systems. The benefits of memory deduplication and the performance overheads are well studied and it is widely accepted that the benefits can not be ignored because of the guest OS transparent design and non-trivial memory savings in real systems. The CPU overheads

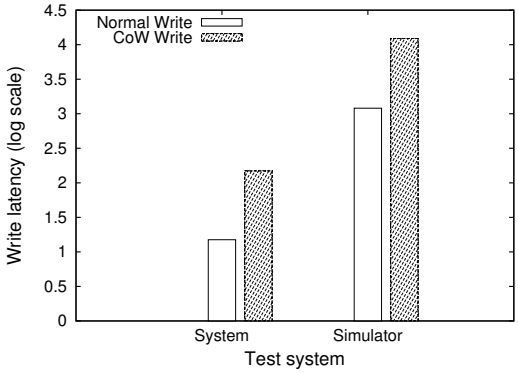

Figure 2: Write latency of a shared page vs Write latency of a normal (non-shared) page.

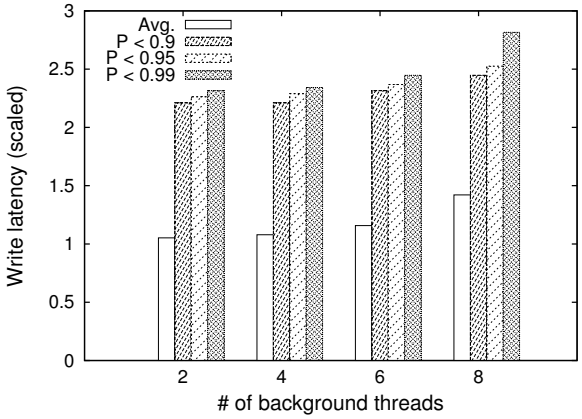

Figure 3: Noise in write latency under different load conditions.

due to CoW faults because of sharing breaks is manageable from the performance point of view by employing specialized amortizations. For example, frequently updated memory pages may not be considered for deduplication to avoid the sharing break overheads [2, 14]. However, the timing information available from within a VM when a shared page is written can be used by the attackers to perform several attacks.

To study the nature and magnitude of timing difference between normal writes and write to shared pages (resulting in a CoW fault underneath), we performed the following experiment. We measured the time taken to write to a set of normal and shared pages in an idle Linux system with a Intel-i7 processor. We also performed the same experiment in Gem5 simulator. As shown in Figure 2, write to a shared page results in ~70x and ~ 100x more latency compared to a normal write, in the simulated system and the physical system, respectively. To study the extent of noise introduced because of other system activities, we measured the write latency for a normal page under varying load conditions. We created a number of background threads which performed synthetic memory accesses. As shown in Figure 3, 99^{th} percentile write latency for a normal

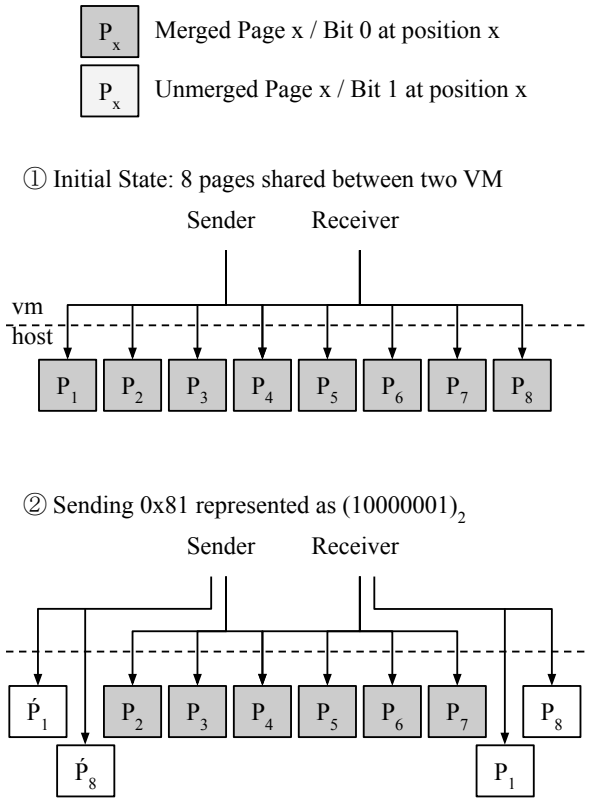

Figure 4: Covert channel construction by exploiting memory deduplication.

page under moderate load conditions (8 threads) is 2.8 times of the average latency in an idle system. From the above experiments, it is clear that there is a *noise free* timing channel exposed because of memory deduplication. We demonstrate construction of a covert channel [17] between two VMs exploiting the timing side-channels.

2.2 Covert channel construction

To create a covert channel, two collaborating processes in two separate VMs on the same physical host try to communicate with each other (without being detected) using the timing based side channel available as a consequence of memory deduplication. As an attacker can detect the write-time differences between a merged page and a normal page (shown previously), the attacker could utilize this behavior as a signalling mechanism (See Figure 4). First both the communicating entities in two different VMs fill-up memory pages with unique pre-negotiated signatures. Depending on the message, the sender VM can construct a message by writing to the page numbers corresponding to the bit positions of the message. For example, as shown in the Figure 4, the sender VM writes to P_1 and P_8 in order to encode a byte value `0x81`. This will result in a CoW fault for P_1 and P_8, exclusive copies ($P_1^{'}$ and $P_8^{'}$) will be provided to the sender while all other pages (P_2 to P_7) remain shared. The receiver writes to all the pages and interprets values 0 or 1 based on

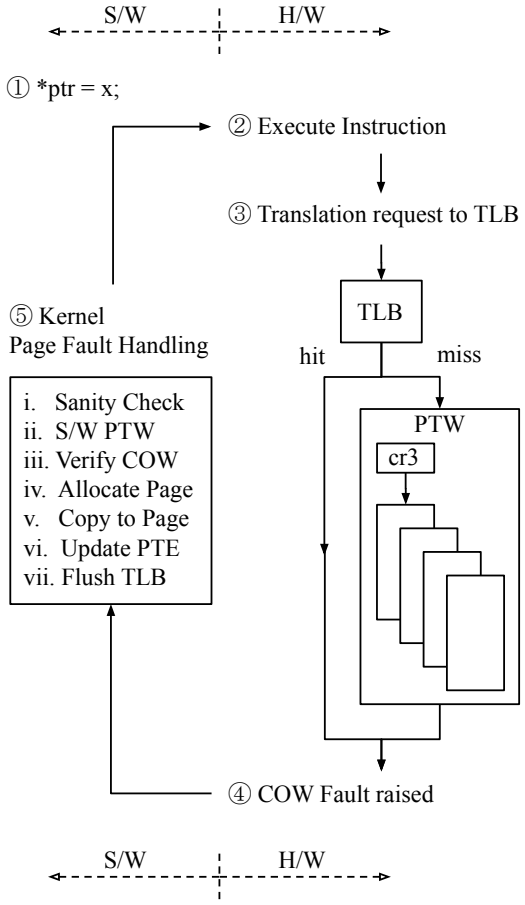

Figure 5: Role of hardware and OS in CoW fault handling.

the time taken to write the particular page. In this example, write to P_1 and P_8 will take less time compared to write to other pages.

There are several other attacks like information disclosure [9, 18], DirtyCow exploits [1] etc. based on the CoW fault based timing channels have been proposed. Next, we present our analysis related to the sources of latency for handling a CoW page-fault in the current systems.

2.3 Analysis of CoW fault handling in X86_64 systems

In paging enabled systems, when a memory request to a virtual address is issued from the processor, the virtual address (VA) needs to be translated to its corresponding physical address (PA) before the request can be passed down to the memory hierarchy. The Translation Look-aside Buffer (TLB) performs this translation and caches the recently used translations. On a TLB miss, the Page Table Walker (PTW) traverses the page tables (4-levels in x86_64) to determine the physical address at the last level of page table (a.k.a. the Page Table Entry or PTE). The translated entry is inserted into the TLB after performing access permissions and privileges before initiating the memory access.

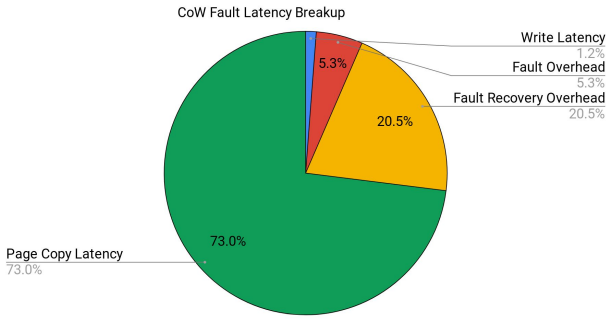

Figure 6: Break down of Copy-on-Write latency.

A schematic diagram of CoW fault handling process is shown in Figure 5. When a shared page (with read-only permission) is written (Step 1), a page fault is triggered (Step 2 to 4) as there is an access permission violation. The page fault handler, registered by the OS during virtual memory subsystem initialization, ascertains if the fault is recoverable by validating that the mapping was previously made read-only to realize deduplication (Step 5). In such a scenario, a new page is allocated, the contents of the accessed page are copied to the new page and the PTE is updated to point to the new page with write access (Step 5). The OS also performs other house keeping tasks like updating the reverse page map (rmap) etc. before returning from the fault. After returning from the fault, the faulting instruction is re-executed by the processor. Therefore, compared to a normal write (i.e., write to a non-shared page), there are three additional processing steps—a fault raised by the hardware, the OS handles the page fault by performing a full page copy irrespective of the write operand size, and, return from the fault—each contribute to the increased latency for a write to a shared page as shown earlier.

To isolate the sources of the write latency, we characterize and quantify the operations involved in serving the write request to a shared page. To isolate the page fault entry and exit overheads, we augmented the page fault handler to skip over the faulting instruction by updating the instruction pointer before returning from the fault handler. We calculated the page to page copy overhead and the total CoW fault handling time separately. The break-up of overheads measured on Gem5 architectural simulator is shown in Figure 6. Most of the overheads is due to page copy (73%), page fault validation and book keeping overheads (shown as fault recovery overheads in the Figure 6). Further, switching to and from the OS page fault handler and the hardware consumes more than 5x time compared to the time taken for a normal write access. In short, the fault generation followed by software fault handling which includes a full page copy incurs the maximum overheads and are potentials for optimizations. In an ideal scenario, if these overheads are removed, the write access latency will come down significantly, even below the noise thresholds shown before.

We try to ask the following design questions and seek answers in our proposed system. First, can the page fault be avoided in the first place? This will save the context switching overheads and OS processing delays in the critical path. Second, as the write operation

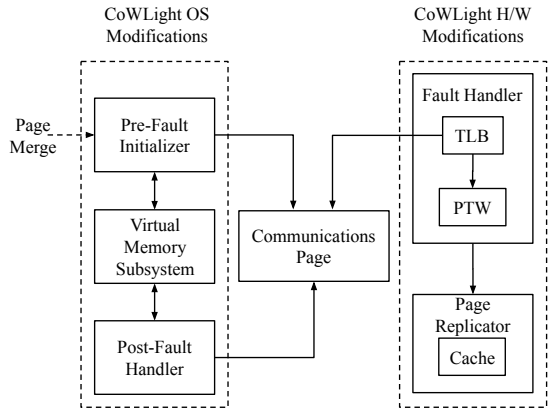

Figure 7: Overview of CoWLight architecture.

requires a cache line at best, can there be an asynchronous copy operation not impacting the critical path? In short, we intend to offload the page fault handling to the hardware and try to reduce the overheads in the critical path.

3 DESIGN

Our approach to reduce the CoW fault handling latency is based on masking and avoiding some of the major bottlenecks in the existing systems. An efficient handling of page fault resulting because of writes to shared pages should perform only the bare-minimum activities synchronously (by the hardware) to realize fast critical path processing. To avoid the page fault entry/exit overheads and other software processing in the critical path, the hardware should handle the page faults. Further, to hide the latency of page copy, the hardware must serve only the demand request and perform the rest of the copy operation in an asynchronous manner, as much as possible. Towards this objective, we present the high-level design of CoWLight (Figure 7). The major components of CoWLight spanning across the hardware and the OS are described below.

3.1 Architectural modifications

We propose hardware extensions to add a per-core *communication buffer* which is used to communicate between the hardware and the OS, and, exchange the necessary details for CoWLight page fault handling. The OS accesses the communication buffer to provide information about the free pages which is used by the hardware during CoW fault handling. The OS also accesses the communication buffer to perform several book keeping procedures after a page is consumed. The size of the communication buffer is proposed to be 4KB, implemented using a SRAM based buffer or a set of registers.

CoWLight hardware fault handler modifies the functionalities of per-core TLB and PTW to handle the CoW faults in hardware by avoiding a OS trap. To avoid OS page fault trap, the modified page table walker logic uses one of the free pages preallocated by the OS through the communication buffer. Then, it maps the free page as the destination for the faulting virtual address by updating the corresponding PTE. The CoWLight fault handler also copies the demand cache line from the old target page (read-only and shared) into the newly allocated page. For example, a translation request to serve a write to virtual address V with a read-only (and shared) mapping to page P does not trigger a page fault. Rather, a new page Q is used from the free list (communication buffer) and the corresponding cache line in Q is filled by reading same cache line offset of P. This step is necessary to avoid full page copy in the critical path and avoid multiple rounds of DRAM access latency in the worst case. Therefore, the thread of execution waits for completion of the above steps and resumes. Next, the hardware fault handler initiates the hardware page copy mechanism to replicate the remaining cache lines of the page in the background. Before finishing the virtual to physical translation process, the communication buffer is populated with information needed by the CoWLight post-fault handler.

The *CoWLight page replicator* logic (part of L2 cache) performs asynchronous copy to complete the CoW fault handling in a deferred manner. The page replication mechanism keeps track the state of all the CoW faults handled by the hardware and fills up the remaining cache lines of the new page (Q from the last example). There are subtle design considerations regarding the aggressiveness of the page copy mechanism and handling subsequent demand requests to yet to be filled memory addresses. To address the first issue, we propose a conservative approach where the number of outstanding memory requests depend on miss status holding register (MSHR) occupancy. For the second issue, we modify the read target to the old page address (P from the last example) and use the read response to fill the corresponding cache line of Q.

3.2 CoWLight OS modifications

The *CoWLight pre-fault initializer* is part of the OS that performs two major functions. First, it initializes a per-core communication buffer provided by the hardware and prepares the communication buffer by providing a list of free physical memory pages. The free pages provided by the initializer is used to serve CoW faults from the hardware. Second, it demarcates the shared pages by setting a reserved bit in the PTE (referred as Cow-bit) so that, the hardware fault handler can distinguish and apply the optimization during the virtual to physical translation. Note that, read-only permission in a PTE can not be used as an indicator as there can be other read-only mappings that are not shared because of deduplication (e.g., text area of a process)

CoWLight Post-Fault Handler is a per-core OS thread which is responsible to perform the house-keeping activities carried out by a normal page fault handler during CoW faults. In most of the OSs, a reverse mapping from the physical page (PFN) to the virtual address is maintained to implement functionalities like swapping and deduplication. During a normal CoW fault handling by the OS, the old page reverse mappings are updated to remove the faulting process virtual address and reverse mapping to the new page is added. In our design, as the hardware handles the fault and it is impractical for the hardware to update the software states like reverse mapping, CoWLight post fault handler perform these steps in a deferred manner. The per-core OS threads periodically scan all the consumed pages and the consumer information from the core's communication buffer and update the reverse mappings. Further, they also refill the communication buffer with a fresh set

of free pages. Note that, compared to the memory savings through deduplication, the memory pages provided to hardware is negligible and the rate of consumption depends on the number of sharing breaks. Therefore, in the worst case, if the free list size is set to be 128, then ($\#of cores \times 128$) pages will be reserved.

To handle corner cases, several minor modifications to the OS virtual memory subsystem is required. First, the OS unmap logic—for a virtual address which resulted in a recent CoW event where the post fault handler is yet to perform the necessary book keeping—is required to be modified. Second, in a scenario when the hardware page fault handler does not find a free page, it should raise a fault (like a normal system) which should be used as an indication to refill the communication buffer. Therefore, the scheduling interval of the post processing thread plays a vital role for efficient hardware CoW fault handling.

4 IMPLEMENTATION

As the CoWLight hardware fault handler and page replicator require additional hardware functionalities to the existing x86 architecture, we have used the Gem5 full system simulator platform to implement and test the proposed modifications to the TLB + PTW logic and L2 cache. We have implemented the OS modification in Linux Kernel.

We have used a normal DRAM page as the communication buffer in our implementation which is referred to as the *communication page* from now on. The communication page is allocated during boot time by the CoWLight pre-fault initializer and filled up with a list of three tuples—free page physical address, faulting virtual address and shared page physical address. The OS provides the free page physical address, which is consumed by the hardware fault handler. The hardware fault handler fills up the faulting virtual address and the shared physical frame number i.e., the PFN used by the faulting page before sharing break. The OS post processing thread consumes the hardware filled information to update the reverse mappings as explained before. For cache efficiency, we have aligned every entry in the communication page to 32-bytes. The communication page is mapped to a fixed kernel virtual address so that it can be accessed from any software context without performing any (temporary) virtual to physical mapping.

4.1 CoWLight hardware fault handler

The CoWLight hardware fault handler logic is integrated into the Gem5 page table walker module to handle write to shared pages without generating faults and trigger asynchronous copy using the CoWLight replicator (Figure 8).

Whenever the PTW encounters a CoW mapping (set by OS pre-fault initializer) for a write access, the hardware fault handling logic is triggered. The free list manager checks the communication page to find a free page for further processing. If the free list is empty, the free list manager triggers a page fault trap to the OS (not shown in Figure). A cache line copy request corresponding to the demand request is generated. This copy request packet from the PTW is marked with a special bit for signaling the CoWLight page replicator to initialize the state for deferred copying. Finally, after the copy operation corresponding to the demand request finishes and a response is received from the L2 cache (not shown in the figure), the PTE of the faulting VA is updated with the new page.

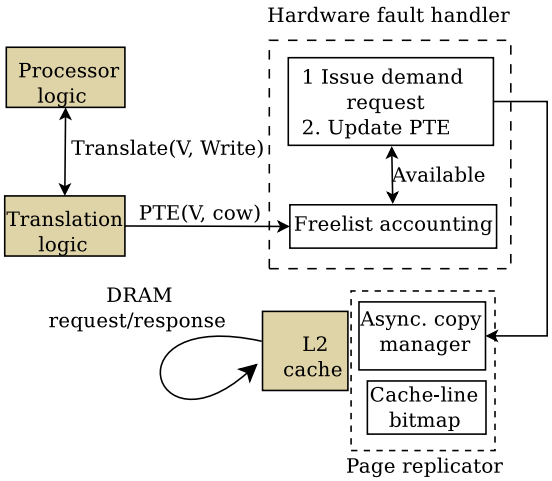

Figure 8: CoWLight architectural modifications

Therefore, in addition to setting the accessed bit and the dirty bit in the PTE, we also update the page mapping, write permission bit and unset the CoW bit.

4.2 CoWLight hardware page replicator

We have implemented the asynchronous copy mechanism in the L2 Cache controller because in Gem5 simulation of the X86 system, L2 cache is directly connected to the page table walker ports. When the page replicator receives a request with the special bit set by the fault handler, it initializes a state tracking through a bitmap. The cache line bitmap represents the cache lines in the new page which are yet to be replicated. The demand request is forwarded to the cache after setting the corresponding bit position and the rest of the cache line copy is performed asynchronously. The page copy logic keeps on sending new copy requests starting from the demand request point till all the cache lines are copied.

To handle demand requests for the same page originating from the processor while executing subsequent instructions, the page replicator checks those requests and forwards them after setting the corresponding bit in the cache line bitmap. In the current implementation, we change the order of asynchronous copy when a demand request is encountered which is yet to be copied. The space required to store the cache line bitmap depends on the number of incomplete asynchronous copy operations. Based on our observations, we have chosen the threshold value to be 16. If all the cache line bitmaps are busy, we notify the CoWLight fault handler to raise a page fault to the OS by piggybacking an indicator bit in the demand response packet.

4.3 Linux kernel modifications

We have implemented the pre-processing logic by modifying the boot-time initialization code of Linux kernel to allocate and map the communication page to a fixed kernel virtual address. Note that, due to the monolithic nature of the kernel, all processes in the system share the kernel mappings and therefore, the communication page can be accessed from any context in privileged mode. To demarcate

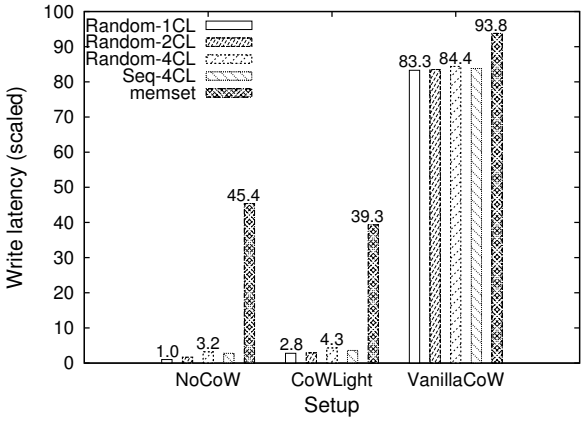

Figure 9: Comparative write access latency for different settings. The results are scaled to the base line idle system write latency without CoW.

Operation	Percentage Cycles
Access free page from list	70
Copy demand cache line	29
Update PTE	1

Table 1: Operation-level breakdown of different CoWLight hardware operations in the critical path.

Write Pattern	Copy time (nanosecs)	Demand request overtake (perc.)
memset	5429	100
Random-4CL	1332	100
Random-8CL-Comp	1812	57
Random-16CL-Comp	1987	46
Seq-16CL-Comp	2095	6.7

Table 2: Demand request overtake of page replicator.

the special CoW bit in the PTE when a copy-on-write mapping is created, we hooked into two call paths in Linux kernel. They are—(i) when KSM (deduplication thread) finds a duplicate and tries to merge by modifying the mappings, (ii) On a `fork()` system call where the child and parent mappings are converted to read-only.

To implement CoWLight post processing, we have implemented a kernel thread which is scheduled on two conditions. First, when a thread interval timer (configurable) expires. Second, when a CoW page fault is triggered by the processor to notify either empty free list or to indicate overwhelming number of CoW faults resulting in crossing the cache line bitmap threshold. We have modified the kernel page fault handler to find out the fault reason from the error code (with new error codes for CoWLight) and notify the post processing thread to perform necessary actions.

5 EVALUATION

Setup: For the experimental evaluation, we have used Gem5 full system architectural simulator to simulate a 64-bit X86 bit system in full system simulation mode with a single core TimingSimpleCPU. The simulated machine was configured with 2GB DRAM, 64KB L1 dcache, 32KB of L1 icache and 2MB of L2 cache. We have used Linux Kernel version 3.4.112 with CoWLight modifications. Note that, deduplication depends on memory footprint and can not be deterministically controlled for arbitrary workloads. Therefore, we have used micro benchmarks to emulate sharing (by setting the CoW bit) and performing write to the shared pages.

5.1 CoWLight efficiency

To analyze the effectiveness of CoWLight, we performed an experiment where we write 1000 different memory pages with three scenarios—(i) write to writable pages (NoCoW), (ii) write to shared pages with CoWLight fault handling (CoWLight) and, (iii) write to shared pages with normal CoW fault handling (VanillaCoW). We present the average write performance relative to normal write latency in Figure 9. To compare the implications of different write behavior (within a page) we have used five different kinds of write operations (Figure 9). They are: (i) write to a random cache line (*Random-1CL*), (ii) write to two random cache lines (*Random-2CL*), (iii) Write to four random cache lines (*Random-4CL*), (iv) write to four sequential cache lines (*Seq-4CL*) and, (v) write to the full page using memset (*memset*). Note that, for CoWLight and VanillaCoW, these different write patterns trigger a single CoW fault.

For a single write to a random cache line (*Random-1CL*), CoWLight improves the write performance by ~ 30x compared to VanillaCoW and is 2.8x slower than a normal write (NoCoW), respectively. For *Random-4CL*, CoWLight results in 1.34x slower write performance compared to NoCoW. By considering the latency variance we had previously measured for motivation experiment, where in a moderately loaded system the write access latency was 2.8x times the idle system latency, we can see from the evaluation results that CoWLight fault handler is capable of handling a CoW with latencies of 2.8x to 4.3x depending on the access pattern of the cache lines. In case of memset, owing to the inherent prefetching-like mechanism of CoWLight hardware page replicator, the latency is faster that that of a vanilla memset operation on Gem5 (by 1.15x). Due to the simulation environment, we found very little variance in the results. A more accurate extrapolation to a real system could be made by implementing sophisticated prefetching mechanisms that are CoWLight aware.

To evaluate the effectiveness of CoWLight hardware fault handler in mitigating the critical path write latency, we profiled the execution time to deduce operation-level breakdown. As shown in Table 1, significant amount of time (70%) is spent to access the communication page and locate a unused free page. This is primarily because we need multiple memory access cycles which if avoided can result in further optimizations of CoWLight.

To evaluate the efficiency of CoWLight asynchronous page replicator, we profiled the number of demand request (referred to as demand request overtakes) whose replication are yet to be completed by the CoWLight asynchronous page replicator. Note that, the demand request overtake also counts the outstanding memory requests already initiated from the L2 cache by CoWLight. Along

with *memset* and *Random-4CL*, we introduced three new access patterns where minimal computation (decrementing a random number to zero) is performed between accesses to random cache lines. *Random-8CL-Comp*, *Random-8CL-Comp* and *Seq-16CL-Comp* represent access to eight random cache lines, access to sixteen random cache lines and access to sixteen sequential cache lines, respectively, where computation is performed between each consecutive accesses. The percentage demand requests which could not be immediately fulfilled as the CoWLight asynchronous page replicator had not copied the corresponding cache lines is presented in Table 2. Copy time represents the time between the first demand request (originated in critical path) and the last cache line copy. For write patterns interleaved with compute, the effectiveness of asynchronous copy is comparatively better than burst memory operations (e.g., memset). Further, CoWLight replicator acts like a prefetcher for sequential write patterns interleaved with computation (only 6.7% overtakes).

To determine the ideal scheduling interval for CoWLight post-fault handler thread, we performed an experiment where the CoW fault rate is varied from 10/sec to 250/sec. As discussed earlier, if all free pages are consumed when the hardware CoW handler requires one, a page fault is raised. The objective of finding a right scheduling interval is to avoid any such faults. We found that 20ms scheduling interval results in no page faults for all CoW rates.

6 RELATED WORK

Memory deduplication in virtualized systems was proposed by VMware [15] latter implemented in open source hypervisors like KVM-Linux [2, 10] as a OS-level daemon (KSM). Several research works show the benefits of deduplication in terms of memory savings [6, 14, 16] and the cost of scan-and-merge along with CoW overheads [14]. XLH [11] proposes optimizations to vanilla KSM by providing hints to the KSM scan for efficient deduplication. Similarly, Catalyst [7] offloads the hash computation (required to find the duplicates) to the GPU and provides scanning hints to KSM.

Security attacks like `FLUSH + RELOAD` [19] use the memory deduplication side-channels as a noise removal measure while other attacks are used for information disclosure [5, 9] and covert channel construction [17]. There are a very few attempts to mitigate the vulnerability. As a consequence, memory deduplication is disabled resulting in memory wastage. Vusion [13] proposed an alternate deduplication technique designed considering the security aspects of deduplication along with deduplication efficiency. Vusion compromises the deduplication efficiency in favor of security and therefore, yet to gain wide acceptance. Moreover, because of additional software logic, Vusion incurs more CPU overheads compared to KSM. On the other hand, CoWLight tries to address the issue at its origin and provides secure and efficient deduplication.

7 CONCLUSION AND FUTURE WORK

Memory deduplication in virtualized systems is shown to be a very useful memory optimization as it is simple to use and provides memory efficient VM packing. However, many side channel attacks exploiting the CoW semantic of realizing memory deduplication are possible. At the core, the timing difference between write access to a normal page and a shared page remains the gateway for attackers.

To bridge the huge access time difference between a normal write and a write to a shared page, we proposed CoWLight, a combination of hardware and software techniques for handling the CoW page faults in an efficient manner. We discussed the design considerations and presented a prototype implementation of CoWLight using Linux kernel and the Gem5 architectural simulator. We evaluated CoWLight with different write patterns and showed that CoWLight can improve the write latency to shared pages by a factor of 30x. Further, effectiveness of different CoWLight features were experimentally analyzed. We plan to extend the CoWLight implementation for multi-core systems and explore several possible optimizations. For example, as shown in §5, the performance overhead of memory page based communication mechanism is significant and we plan to design alternate approaches in the future. Further, we also plan to characterize the fault-handling performance improvements of CoWLight in real-world scenarios.

ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers for their helpful comments. This work is supported in part by IITK initiation grant IITK/CS/2017477 and SRC grant SRC-2853.001.

REFERENCES

[1] Kernel local privilege escalation dirty cow - cve-2016-5195. https://access.redhat.com/security/vulnerabilities/DirtyCow.
[2] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using ksm. In *Proceedings of the Linux Symposium* (2009), pp. 19–28.
[3] Barker, S., et al. An empirical study of memory sharing in virtual machines. In *Proceedings of the USENIX ATC* (2012), pp. 273–284.
[4] Binkert, N., et al. The gem5 simulator. *SIGARCH Computer Architecture News* (2011), 1–7.
[5] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy (SP)* (2016), pp. 987–1004.
[6] Chang, C.-R., Wu, J.-J., and Liu, P. An empirical study on memory sharing of virtual machines for server consolidation. In *Proceedings of ISPA* (2011), pp. 244–249.
[7] Garg, A., et al. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *Proceedings of VEE* (2017), pp. 44–59.
[8] Gruss, D., Bidner, D., and Mangard, S. Practical memory deduplication attacks in sandboxed javascript. In *Proceedings of ESORICS* (2015), pp. 108–122.
[9] Irazoqui, G., Sinan Inci, M., Eisenbarth, a., and Sunar, B. Know thy neighbor: Crypto library detection in cloud. PETS 2015.
[10] Kivity, A. kvm: the linux virtual machine monitor. In *OLS '07: The Ottawa Linux Symposium* (2007), pp. 225–230.
[11] Konrad, M., et al. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the USENIX ATC* (2013), pp. 279–290.
[12] Miłós, G., Murray, D. G., Hand, S., and Fetterman, M. A. Satori: enlightened page sharing. In *Proceedings of the USENIX ATC* (2009), pp. 1–14.
[13] Oliverio, M., et al. Secure page fusion with vusion: https://www.vusec.net/projects/vusion. In *Proceedings of SOSP* (2017), pp. 531–545.
[14] Rachamalla, S., Mishra, D., and Kulkarni, P. Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems. In *Proceeding of HiPC* (2013), pp. 59–68.
[15] Waldspurger, C. A. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review 36*, SI (2002), 181–194.
[16] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of VEE* (2009), pp. 31–40.
[17] Xiao, J., Xu, Z., Huang, H., and Wang, H. A covert channel construction in a virtualized environment. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12, pp. 1040–1042.
[18] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of DSN* (2013), DSN '13, pp. 1–12.
[19] Yarom, Y., and Falkner, K. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (2014), SEC'14, pp. 719–732.