Copy-on-Pin: The Missing Piece for Correct Copy-on-Write

David Hildenbrand
Technical University of Munich
Munich, Germany
Red Hat GmbH
Grasbrunn, Germany
hildenbr@in.tum.de
david@redhat.com

Martin Schulz
Technical University of Munich
Munich, Germany
schulzm@in.tum.de

Nadav Amit
VMware Research
Palo Alto, USA
namit@vmware.com

ABSTRACT

Operating systems utilize Copy-on-Write (COW) to conserve memory and improve performance. During the last two decades, a series of COW-related bugs—which compromised security, corrupted memory and degraded performance—was found. The majority of these bugs are related to page "pinning", which operating systems employ to access process memory efficiently and to perform direct I/O. Unfortunately, the true cause of these bugs is not well understood, resulting in incomplete bug fixes. We show this by: (1) surveying previously reported pinning-related COW bugs; (2) uncovering new such bugs in Linux, FreeBSD, and NetBSD; and (3) showing that they occur because the COW logic does not consider page pinnings correctly, resulting in incorrect behavior (e.g., I/O of stale data). We then address the underlying problem by deriving when/how shared pages must be copied and under which conditions pinned pages can be shared to maintain correctness. Based on this assessment, we introduce the "Copy-on-Pin (COP)" scheme, an extension of the COW mechanism that handles pinned pages correctly by ensuring pinned pages and shared pages are mutually exclusive. However, we find that a naive implementation of this scheme hampers performance and increases complexity if pages are copied only when strictly necessary. To compensate, we introduce a relaxed-COP design, which does not require precise tracking of page sharing, maintains correctness without increasing complexity, and (while potentially needlessly copying pages in some corner cases) marginally improves performance. Our relaxed-COP solution has been integrated into Linux 5.19.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; • **Security and privacy** → *Software and application security*; • **Computer systems organization** → Architectures.

KEYWORDS

COW; copy-on-write; page pinning; page sharing; virtual memory; fork; memory deduplication

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575716

ACM Reference Format:
David Hildenbrand, Martin Schulz, and Nadav Amit. 2023. Copy-on-Pin: The Missing Piece for Correct Copy-on-Write. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575716

1 INTRODUCTION

Copy-on-Write (COW) is an efficient and ubiquitous technique that Operating System (OS) memory managers use to save memory, enhance performance and create memory snapshots [13, 16, 33, 46, 49, 52, 68]. Using COW, a memory page can be shared and mapped multiple times as long as there are no modifications, while modifications trigger a separate mapping not visible through the other mappings. To implement this, all shared page mappings are write-protected, thereby permitting read accesses from the page, but are set to trap write accesses. When the OS traps a write access to the page, if the page is still shared, it performs *COW-unsharing*, whereby a writable copy of the page is mapped instead of the shared page (Figure 1). Otherwise, if the page is no longer shared, the OS write-unprotects the page, i.e., *reuses* the page without copying it first.

The most well known use-case of COW is as an optimization of the POSIX `fork()` system call (syscall). When a parent process forks a child process, the OS needs to ensure that future memory modifications of private mappings—in either the parent or the child—are not visible to the other process. To do so, the OS can copy the parent's memory during fork and map the copy in the child's address space, but doing so would cause performance and memory overheads. Therefore, OSes refrain from copying memory, and instead use COW to safely share memory between the two processes.

Despite COW ubiquity, bugs that caused data leaks, memory corruption and performance degradation were often found in commodity OS implementations of COW. Most of these bugs were caused by wrong interactions between COW and *page pinning*, an OS mechanism that prevents memory pages that the OS or I/O devices need to access directly from moving or being paged-out. Once a page is pinned, the OS can access the memory through *linear kernel page-table mappings*—a privileged memory alias that is mapped in every address space—and I/O devices can use Direct Memory Access (DMA) to application buffers.

While page pinning simplifies OS design and improves performance, it can interfere with COW operations. The OS and I/O devices access the memory through *non-remappable references*, which

unlike common userspace memory references, cannot be remapped or write-protected. As remapping and write-protection are required for COW, OSes need to cautiously handle the interactions between COW and page pinning. However, until now, these interactions have not been properly formalized.

To illustrate one possible wrong interaction of COW and page pinning, consider a scenario in which a page is shared between two processes, P_1 and P_2. P_2 initiates an I/O write operation, performs a DMA to read from that page, and shortly after, before the I/O operation completes, P_2 unmaps the page using `munmap()`. Then, when P_1 writes to the page, the OS might wrongly consider the page not to be shared, as it is only mapped once, and allow P_1 to "reuse" the page. As a result, writes of P_1 to the page, which should be private, would potentially propagate to the file through DMA. This scenario, which is depicted in Figure 2b, resembles a recent security issue in Linux [21].

This aforementioned issue is not an isolated case. As we survey OSes, we find a long list of related bugs as well as inconsistent and faulty fix attempts. Such bugs are not specific to a certain OS and some of the bugs have never been properly fixed to this day. Moreover, we analyze the key code sections of several common OSes for *previously unknown bugs*, and find correctness issues in three commodity OSes: security issues in FreeBSD 13.0 and Linux 5.17 and a memory corruption in NetBSD 9.2.

Following our survey, we analyze the possible wrong interactions between COW and page pinning. We find five fundamental classes of wrong interactions of COW and page pinning: (1) *wrong COW-sharing*, (2) *wrong COW-unsharing*, (3) *missed COW-unsharing*, (4) *impossible COW-unsharing* and (5) *unnecessary copies*. Our analysis leads us to the conclusion that under the assumption of most OS designs, COW-shared pages and pinned pages must be mutually exclusive, even if the pinning was used to obtain a non-remappable read-only reference.

Preventing COW-shared pages from being pinned and preventing pinned pages from being COW-shared might sound trivial to implement. Yet, tracking whether a page is pinned or COW-shared can introduce complexity and overheads. For example, as we detail in §5, in Linux, tracking whether a page is pinned can consume 1.5% of the system memory due to cache-line alignment considerations, and tracking whether a page is COW-shared was found to be error-prone. We therefore need to find a simpler way to prevent pinned pages from being shared.

We build on the insight that to properly handle COW and page pinning, precise tracking of page sharing and pinning is not necessary. As long as the OS can correctly identify COW-shared and pinned pages, it may incorrectly consider other pages as COW-shared or pinned, and still handle COW and page pinning correctly. Based on this insight, we design a general scheme for handling the interactions between COW and page-pinning, which we name Copy-on-Pin (COP). COP allows the OS developers to tradeoff memory sharing opportunities for simplicity and relaxed tracking.

Using our general scheme, we design a relaxed COP system for Linux, which addresses the known issues in recent Linux versions. Our solution prioritizes simplicity and robustness and introduces only negligible memory overhead to track page pinning and sharing. We implement our solution, which was integrated into Linux 5.19.

Although the main motivation of our relaxed COP design is robustness, we also measure the performance of our relaxed COP implementation as well as the current and the previous COW implementations in Linux. We show that our relaxed COP solution performs in common scenarios as well as or better than previous implementations, as it effectively performs memoization whether pages may be shared or are not shared, thereby reducing the overhead of determining whether pages should be copied or reused.

Specifically, we make the following contributions:

- We identify bugs in the way COW is handled in three commodity OSes, and show that such bugs are prevalent.
- We analyze the different classes of correctness and performance issues that may be caused when COW is used alongside page pinning.
- We describe the possible design options for the correct handling of COW of pinned pages and introduce the Copy-on-Pin scheme.
- We design, implement, and upstream a simple and efficient COP handling scheme for Linux. Our solution resolves COW problems, does not increase complexity and even improves COW operation performance.

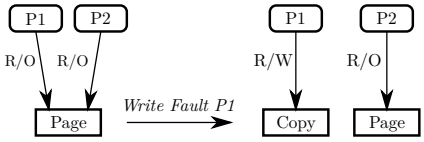


Figure 1: *COW-unsharing on write fault*: replacing a COW-shared page in a private mapping P_1 by a private copy on write fault.

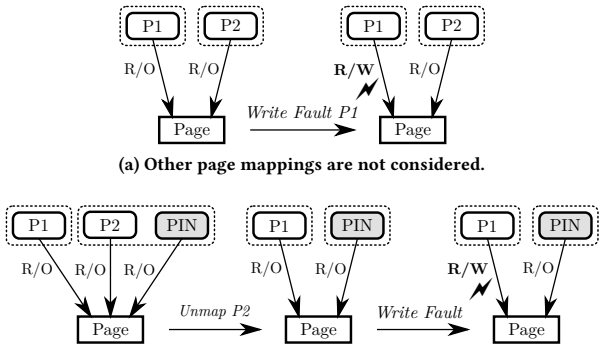


Figure 2: *Missed COW-unsharing*: reusing a page in a private mapping P_1 even though COW-unsharing is required results in private modifications through P_1 being observed from mapping P_2.

2 BACKGROUND

COW is one of the basic techniques that OS memory managers implement to improve system performance. Introduced in TENEX [16] during the 1960s, COW saves memory and enhance performance by deferring and potentially avoiding memory copies until memory is actually written to. COW was originally designed to prevent executables' memory duplication, and its use was later extended for efficient implementations of the `fork()` syscall [49], for efficient communication between processes via message-passing [30, 31, 63] and for memory deduplication [11, 17, 68].

The usefulness and the operation of COW can be exemplified by the benefit that it provides to the `fork()` syscall, which creates a new process by duplicating the calling process (parent) into a new process (child). The fork operation requires memory to be copied logically, but duplicating the memory physically is wasteful since it consumes CPU time and memory and since the child process might unmap the entire memory shortly after, e.g., by executing a new program using the `exec()` syscall. The use of COW can save time and memory: the OS write-protects memory pages and shares them between the parent and the child. Only when a memory page is written to, the OS traps the write operation, copies the original page content into a new page, and changes the page table entry (PTE) in the writing process's page-table to point to the new page, which is set as writable.

A common optimization that COW mechanisms employ is copy avoidance when a page is no longer shared: COW, if done frequently and unnecessarily, might introduce significant performance overheads, as it requires a page copy operation and invalidation of the hardware virtual memory translation cache, which is known as Translation Lookaside Buffer (TLB). Therefore, if a page is no longer shared, a copy operation can be avoided and instead the page write-protection can simply be removed. Such a situation occurs, for instance, if a memory page is shared between a parent and a child, the child exits, and then the parent writes to the page. To detect these cases, OSes often track the number of times each page is mapped, and avoid copying the page if it is only mapped once.

Although the basic COW logic is rather simple, its interaction with pinned pages, as we show later, is not well understood. Pinned pages are memory pages that are (or were) mapped to processes, and cannot be moved or paged-out to allow the OS or I/O devices to access them directly and irrespective of the active address space. There are various cases in which pages are pinned, for instance, to establish shared I/O rings between applications and the OS (I/O-uring [12]), for direct uncacheable file access (`O_DIRECT`) and Remote Direct Memory Access (RDMA).

Page pinning is necessary to enable the OS and I/O devices to access user memory efficiently. When pages are pinned, the OS can access the pages using *OS linear mappings*, a static virtual memory range in which the entire physical memory of the system is mapped. Accessing memory through the OS linear mapping is faster than using address-space pointers since (1) the linear mapping is always mapped and does not require address-space switching; and (2) the OS linear mappings use huge-pages and therefore using it causes fewer TLB misses. Beside its performance benefit, page pinning also simplifies the OS implementation, since it eliminates page-faults in complicated scenarios.

Similarly, page pinning allows I/O devices to use DMA to access I/O buffers of applications directly. Without page-pinning, bounce buffers had to be used for I/O and additional copy operations were required, which would have slowed down I/O accesses. It might seem that the use of an I/O Memory Management Unit (IOMMU) would render page pinning unnecessary; however, most I/O devices are incapable of safely restarting I/O transactions following an I/O page-fault [45], and the use of an IOMMU can introduce additional overhead [9].

When the OS pins a page, it effectively obtains a *non-remappable reference* to a page. We distinguish these *pinning references* from userspace pointers in application address-spaces, which we define as *remappable references*. Note that pinning references are always instantiated from remappable references (i.e., application mappings) and have an implicit protection (i.e., read-only/read-write), which is defined at the time of their instantiation. Read-only protection, however, is not enforced by the paging mechanism, as the OS linear mappings are not modified. Moreover, the protection of pinning references can be different from the protection of the remappable reference from which they were created.

These characteristics of pinning references violate the basic hidden assumption of COW: if a page is pinned, some references of the page cannot be write-protected and cannot be remapped. In the next section we will see that this violation is the root-cause of a variety of bugs.

3 PINNING-RELATED COW PROBLEMS IN OSES

COW-related bugs can have serious implications on the system stability, security and performance. At the same time, some of the bugs are timing- and workload-dependant, and are therefore hard to identify, reproduce and debug. During our research, we not only uncover a long list of past pinning-related COW bugs in OSes, but also identify previously unknown bugs in three commodity OSes, which emphasizes the significance of the problem.

3.1 Survey of Pinning-related COW Bugs

Over the years a variety of pinning-related COW bugs has been found in Linux. In 2005, the developers of Linux found that memory corruption in Linux can occur because of incorrect COW handling with pinning references [27]. The next year, correctness issues due to COW arose when the `fork()` syscall was initiated after memory was used for RDMA [62]. In 2014, bugs were found when COW-unsharing was forced on write-protected shared memory [28]. Then, in 2020, Jann Horn revealed a security issue, in which a child process could use the `vmsplice()` syscall to read private modifications of the parent's private memory because COW-unsharing was missed [38].

The fix [59] for this security issue, however, caused new bugs, including userspace page-fault handlers that hung because read-only pinning was treated like write access [70]. The following fix [60] caused memory corruption [7, 34] and performance degradation [8] due to excessive page copies when handling write faults.

Other OSes also encountered issues due to COW-related operations. The developers of FreeBSD, for instance, found in 2009 that COW-sharing of wired (pinned) pages during `fork()` has never

worked correctly [23]. A bug in handling COW that prevented debuggers from setting breakpoints correctly on wired pages, was fixed in 1997 [29], and that fix was found to be insufficient 12 years later [24].

This long list of bugs, which does not include additional bugs for brevity, shows that latent and severe COW-related bugs can lurk for years in OS kernels. In addition, the chaotic nature of the fixes shows that a full understanding of the problem is missing. The fixes themselves are also inconsistent, further emphasizing the confusion about the correct solution. For example, the logic in Linux that decides whether COW-unsharing is required for an anonymous page has changed along the years from originally using the page reference counter to the number of page table mappings of a page [27] and then back to the reference counter [60]. Some solutions required applications to explicitly notify the OS which memory might be pinned if they intend to fork [62]. Yet, in practice bug reports show that this interface was not used correctly [34].

3.2 Testing and Uncovering COW Bugs

While we review the code of Linux, FreeBSD and NetBSD looking for pinning-related COW bugs and find such bugs in Linux and FreeBSD, we also observe that finding such bugs reliably is hard. This is particularly true, as they frequently involve a lot of code complexity and only trigger in specific situations. We therefore wish to identify such problems across OSes more easily, for example, to prevent adversarial programs from degrading system stability or leaking secrets, and non-adversarial programs from failing due to corrupted memory. However, most OS interfaces that rely on page pinning differ heavily between OSes making an automated or even semi-automatic approach almost impossible. We, therefore, build on an indirect method, namely direct file I/O reads and writes via `O_DIRECT`, which are supported by most OSes.

Following this idea, we develop three multi-threaded, OS agnostic test cases [4] that use combinations of `O_DIRECT`, `fork` and write faults. Test case 1 and 3 call `fork()` with concurrent direct file I/O, while further modifying the page that is under DMA; such a scenario could similarly happen in non-adversarial programs, for example, when snapshotting a Virtual Machines (VMs) via `fork()` [10]. Test case 2 triggers direct file I/O from the child process, while concurrently modifying the page that is under DMA from the parent and the child process; we suspect that only adversarial programs will trigger this behavior in practice. Table 1 describes our test cases in more detail.

NetBSD. Test cases 1 and 3 (see Table 1) fail under NetBSD 9.2 causing both memory corruption and resource leakage: in addition to failure of our test, the in-memory file-system `tmpfs` runs out of available memory. Reviewing the code indeed shows that NetBSD 9.2 might share pinned pages during `fork()`, regardless of whether the pages are pinned for read or write. When the pages are later written, the OS performs COW-unsharing, replacing the mappings of the pinned pages with mappings to new page copies. This leads to memory corruption, as I/O writes are written to a different page than the mapped one. In addition, we noticed accounting mistakes that lead to the resource leakage.

FreeBSD. While none of our test cases (see Table 1) fail under FreeBSD 13.0, we identify a security issue[1] during manual code inspection, which is similar to the `vmsplice()` security issue [21] in Linux. We create a specific test case that uses *direct pipe writes* [14], whereby a `write()` to a pipe will not be buffered in a kernel buffer, but instead read direct from the source page using page pinning. Similarly to the `vmsplice()` security issue, this security issue is the result of COW-unsharing logic deciding to reuse a COW-shared page in the parent process on write-access that, although no longer mapped by the child process, can still be read using a pinning reference (held by the pipe) by reading from the pipe. This scenario is depicted in Figure 2b.

The root cause of this issue is that FreeBSD, although it reuses pages only under very restrictive conditions [49], relies on the number of mappings to detect page sharing, which does not consider pinning references from other processes.

Linux. All our test cases (see Table 1) fail under Linux 5.7 and only Test case 3 fails under Linux 5.17. However, this is not a bug in the COW logic, but a bug in the `O_DIRECT` implementation under Linux, which uses the wrong interface for pinning pages. Pages referenced via the wrong interface are not identified as pinned during `fork()` and will get shared using COW. The Linux community is actively working on adjusting the `O_DIRECT` implementation to use the correct interface.

As discussed in §3.1, the COW logic in Linux 5.17 still suffers from memory corruptions and performance problems. During our code inspection, we identified another unfixed instance of the `vmsplice()` security issue [21]: in case a page gets remapped into a process from the swapcache on write access, the ordinary (fixed) write-fault handler is bypassed and the wrong COW logic will reuse a page in the parent process during a write-fault instead of creating a private copy, even though the page is still pinned by a child process. The root cause for this issue is an inconsistent and incorrect COW logic.

4 COW CORRECTNESS WITH PAGE PINNING

The logic behind COW is often perceived as simple: the OS can share identical private memory pages by mapping them as read-only; when a write-access is trapped, the OS replaces the read-only mapped page by a writable copy. This simple approach, however, always results in a copy, even if the original page is not shared, causing an *unnecessary copy*, resulting in performance overheads (see Figure 3). As an optimization, OSes usually try to avoid such copy operations by reusing pages if they are no longer shared.

However, allowing COW logic to reuse pages unwittingly might violate *correctness*. As COW is a transparent *optimization*, the use of COW by the OS should not affect application execution, excluding timing and memory consumption. Specifically, when pages are COW-shared, for instance following a `fork()` syscall invocation, write operations through private mappings of one process—the parent or the child—must not be observable by the other process. Consequently, a page must be copied and cannot be reused if a page is shared. Otherwise, as depicted in Figure 2a, reusing the page can result in data leakage or memory corruption.

[1] We reported this security issue to the maintainers and they are looking into the problem.

Table 1: Our generic test cases for testing COW and page pinning via O_DIRECT read()/write() in combination with fork() and write accesses.

Case	Description	Checked Symptom
1	Trigger `write()` from Part A of a page. Concurrently fork a child and modify Part B of the page in the parent and Part A in the child.	Wrong file content after `write()`.
2	Fork a child and trigger a `write()` from Part A of a page in the child. Concurrently modify Part B of the page in the child and Part A in the parent.	Wrong file content after `write()`.
3	Trigger `read()` to Part A of a page. Concurrently fork a child and then modify Part B of the page in the parent.	Wrong page content after `read()`.

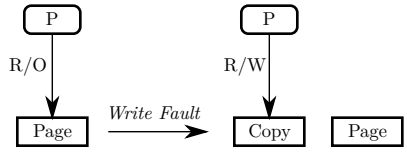


Figure 3: *Unnecessary copy*: during a write-fault on mapping P, the mapped page is replaced by a copy instead of reusing the read-only physical page that is not shared.

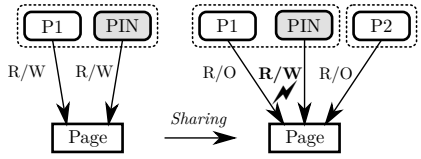


Figure 4: *Wrong COW-sharing*: sharing a writable pinned page that is associated with mapping P_1 causes private modifications to be observed from mapping P_2.

COW Correctness Without Pinning. COW can be described as an invariant that is maintained for *private mappings*. We define private mappings as mappings of pages whose initial content is identical, but whose updates must not be visible through other private mappings. Note that this definition encapsulates both POSIX private anonymous mappings and deduplicated memory. For a given physical page frame f, we denote P^f_{RO} and P^f_{RW} as sets of read-only and writable private mappings, respectively, which map the page-frame. We define P as their union: $P^f = P^f_{RW} \cup P^f_{RO}$. To maintain correctness by preventing "missed COW-unsharing" as depicted in Figure 2a, any COW mechanism must therefore maintain the following invariant for each physical page frame, which indicates that, if there is any private writable mapping, it must be a single exclusive mapping:

$$|P^f_{RW}| > 0 \rightarrow |P^f| = 1 \tag{1}$$

However, this invariant is over-simplistic as it does not consider the interaction between COW and some other prevalent OS mechanisms. First, in POSIX-compliant OSes, file-backed memory may also be accessed through *shared mappings*, and memory modifications through private mappings should not be observable through shared mappings either. As a side-note, the specifications do allow for write operations that are performed through shared mappings to be visible through private mappings [58].

Second, the OS might hold *caching references* to the page and use them later to create new mappings of the page or access the page. Memory modifications using private mappings should still not be observable through any other mapping or through caching references. For example, a file-backed memory page can be cached in the page-cache and later read using the `read()` syscall. Writes to private mappings that mapped the page should not be reflected in the read buffer.

Third, considering page mappings only as "direct" architectural virtual-memory mappings—i.e., pages that are mapped in the page-tables and accessible using pointers—is inadequate. Page mappings can also be logical-only references of pages that are not mapped architecturally in the page-tables. Specifically, when a page is about to be swapped out, the OS might replace the page-table entry that maps a page with a special page-table entry that references a swap slot instead. At this point, the page might still reside in memory. The definition of mappings should therefore include all page mappings, including logical-only references.

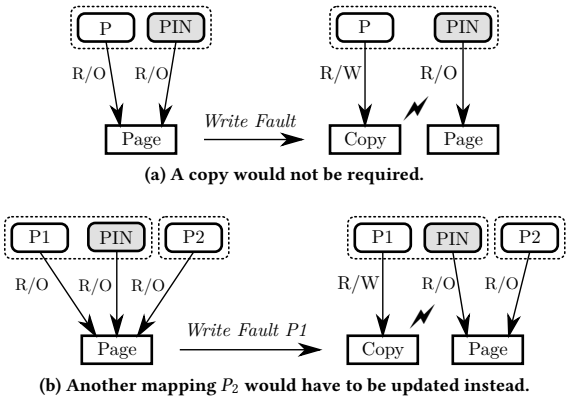
Based on these OS characteristics we update the previous invariant. We denote for the page frame f the set of shared mappings as S^f, and the set of caching references as C^f. This leads us to the following refined invariant:

$$|P^f_{RW}| > 0 \rightarrow |P^f| + |S^f| + |C^f| = 1 \tag{2}$$

This logic actually resembles the logic that Linux used in the past to evaluate whether COW-unsharing of write-protected anonymous pages is needed.

COW Correctness with Page Pinning. So far we did not consider the impact of page pinning. Applications can implicitly pin pages using various interfaces, for instance by performing direct I/O operations (e.g., using `O_DIRECT`) that use DMA. By pinning the page we say that the application obtained a *pinning reference* to the page, which is non-remappable. Pinning references must originate from shared or private mappings, but they may outlive the mapping from which they were derived, for instance, following the `munmap()` syscall. The protection of pinning references cannot be easily changed because it is enforced by software (without synchronization) and not by hardware.

As COW is a transparent optimization, it cannot change the semantics of page pinnings. Write operations through pinning references must be observable through the memory mapping they are associated with and vice versa. Write operations through pinning references that originated from private mappings must not be observable through other mappings. Mishandling pinned pages



(a) A copy would not be required.

(b) Another mapping P_2 would have to be updated instead.

Figure 5: *Wrong COW-unsharing*: a wrong decision whether/how COW should be unshared can result in a mapping getting disconnected from its derived page pinnings.

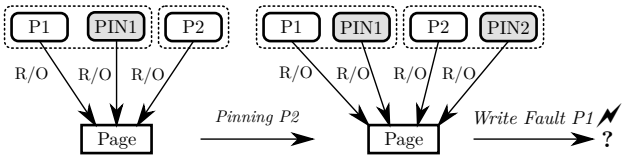


Figure 6: *Impossible COW-unsharing*: having multiple mappings and pinning the page in each mapping without copying the page, cannot be resolved following a write-fault: the mapping reference and the pinned reference point to different pages.

can lead to various problems, which we classify in the following categories:

(1) *Missed COW-unsharing* (Figure 2b): an application causes a COW-shared page to be pinned and then unmaps the mapping that was used for pinning, for instance, using the `munmap()` syscall. When the page is written, if the OS considers the page as unshared since it is only mapped once, it would not unshare COW. As a result, writes to the page would be visible through the pinning reference, thereby leaking data. If the pinning reference can be used to write to the page, memory corruption is possible. Examples for such bugs include the recent `vmsplice()` vulnerability of Linux [38] and the bug that we found in FreeBSD (see §3.2).

(2) *Wrong COW-sharing* (Figure 4): an application causes a page to be pinned. The page is then shared by write-protecting the mapping. However, the pinning reference cannot be write-protected. If the page is written through the pinning reference, the modifications, which should be private, are visible through all the mappings. This can cause memory corruption and leak information into other processes. Such a scenario was possible in Linux [62], when RDMA pinned a page for write-access and then invoked `fork()` to create a child process.

(3) *Wrong COW-unsharing* (Figure 5): a page mapping and an pinning reference that should point to the same page, point to two different pages. Such a scenario might occur since an OS might not be able to determine whether a page is shared, and as a result might copy the page, considering an unnecessary copy operation to be benign. Later, when the application writes to the page, the modifications will not be visible through the pinning reference. A similar scenario might occur when the page is shared. The OS, noticing that the page is shared, might copy the page upon write without updating the pinning reference that cannot be remapped. Both of the scenarios might result in memory corruption. Such bugs were found in Linux [27, 34], and we found a similar bug in NetBSD as well (see §3.2).

(4) *Impossible COW-unsharing* (Figure 6): a situation whereby COW-unsharing cannot be resolved without removing mappings or pinning references. The OS might allow multiple COW-shared mappings to be pinned if the corresponding pinning references are read-only. However, once a write operation occurs, the OS has no valid course of action: it cannot copy the page, since this would prevent modifications from being visible through the matching pinning references, and it cannot allow the page to keep being shared, since this would make modifications of the page to be visible through the other mapping. The only possible resolution for such situations is removing mappings or the pinning references, which requires aborting processes in practice.

Considering page pinning, we first define an invariant to prevent "wrong COW-unsharing", and to ensure OS references that originated from certain mappings point to the same page frame as the mapping. We denote by PIN^f the set of mappings from which pinning references to page frame f were established.

$$\forall f.\forall f'.\forall p \in P^f.\forall i \in PIN^{f'}.p = i \rightarrow f = f' \tag{3}$$

Next, we define invariants to prevent the "wrong COW-sharing" and "missed COW-unsharing" scenarios. As soon as there is one writable reference (mapping or pinning reference) from a private mapping, no additional references from other mappings may exist. We denote the subset of private mappings in PIN^f as $PPIN^f$ and the shared ones as $SPIN^f$. We denote by $PPIN^f_{RW}$ the subset of mappings in $PPIN^f$ that were used to create a writable pinning reference. Note that, since a pinning reference can outlive the mapping it was created from, PIN^f, $PPIN^f$, and $PPIN^f_{RW}$ might include torn down page mappings, for instance mappings that were unmapped using the `munmap()` syscall. In other words, P^f is not a superset of $PPIN^f$. We extend the invariant 2 to regard pinned pages, i.e., pinning references, as follows:

$$\begin{aligned} |P^f_{RW} \cup PPIN^f_{RW}| > 0 \rightarrow \\ |P^f \cup PPIN^f| + |S^f \cup SPIN^f| + |C^f| = 1 \end{aligned} \tag{4}$$

To prevent the "impossible COW" issue, we must not have a page frame that is referenced by a pinning reference that originated from a private mapping, and any pinning reference that originated from another mapping. We, therefore, must not have a page pinned through a private mapping, while it is also pinned through another

mapping. This leads us to a slightly more generic invariant:

$$|PPIN^f| > 0 \rightarrow |PIN^f| = 1 \quad (5)$$

Practical COW. While these invariants are correct in theory, they do not provide a practical solution for OSes. Consider a situation in which a page has multiple private mappings and a single pinning reference that is associated with one of the mappings. Based on the aforementioned invariants, such a scenario is valid, however in practice cannot be handled. When the OS traps a write operation through the mapping that is associated with the pinning reference, the OS must copy the page, but at the same time cannot remap the mapping that triggered the fault to maintain invariant 5.

Instead, the OS has to remap all the other mappings. However, remapping the other mappings is impractical since it is complicated to implement and might introduce significant latency since the number of the mappings is unbounded.

Therefore, a practical solution requires that sharing and pinning of private mappings would be mutually exclusive. To be more precise: (1) if a page with a private mapping is shared—COW-shared, mapped through shared mapping, or has a caching reference—it would not be pinned; and (2) if a page has a private pinning reference, it would not be shared in any of the aforementioned manners. A page with no private mapping is allowed, of course, to be shared and pinned.

We would note that since POSIX leaves the impact of memory modifications through shared mappings unspecified, some relaxation of this solution is possible. Specifically, a page with a shared pinning reference can have a private mapping without causing correctness issues. For simplicity, we disregard optimizations that rely on this unspecified scenario.

5 DESIGN

Proper support for page pinning in an OS that uses COW requires modifications of three components: page pinning, COW-sharing decisions (e.g., during `fork()`) and COW-unsharing decisions (when a write operation to a write-protected page is trapped). Based on the condition that pinned and COW-shared pages must be mutually exclusive, we introduce the Copy-on-Pin (COP) scheme, which extends the COW scheme to properly handle pinned pages. In general, handling COP should be performed in the following manner:

(1) *Pinning*: Whenever a COW-shared page is pinned, for either read or write, the page should be unshared.
(2) *Sharing*: Pages cannot be COW-shared if they are pinned.
(3) *Prevent COW on pinned pages*: Pinned pages should not be replaced with a copy: they would never be COW-shared and replacing them with a copy would lead to a "wrong COW-unsharing" scenario.

The main challenge in supporting page pinning with COW is in determining (1) whether a page is COW-shared and therefore should be copied on write or pinning; and (2) whether a page is pinned and therefore cannot be COW-shared and cannot get replaced by a copy. It might appear easy to accurately track and determine whether each page is COW-shared or pinned. Determining certain types of sharing—by shared mappings and caching references—is indeed simple, as tracking this information does not require exact counters and the OS already tracks it for other purposes. However, tracking COW-sharing or pinning references is not always possible or performant. As we show in §5.2, Linux intentionally does not track this information accurately, and adding such tracking was deemed unacceptable. We focus on this type of sharing in the rest of our analysis.

There are various reasons that can lead OSes developers to decide not to track accurately whether pages are pinned or shared. Tracking whether each page is pinned can increase memory consumption by page metadata dramatically. Tracking whether each page is COW-shared is complex, fragile and might require acquiring locks, which would introduce unacceptable overheads. One major reason for this limitation is that OSes did not need to determine accurately whether pages are COW-shared before page pinning was introduced. Without page pinning, the OSes can do best effort checks whether a page is COW-shared, and if the checks cannot determine whether it is COW-shared, the OSes can simply replace the page with no negative impact on correctness. In contrast, if the OS considers a pinned page as COW-shared, this would lead to "wrong COW-unsharing".

Our insight is that accurate tracking of page pinning and sharing is actually not required for proper support of COW with page pinning. In fact, there is no single correct solution for COW mechanisms to support page pinning, but a variety of solutions that introduce different trade-offs. Next, we describe the possible design options, and based on these options, we choose the solution that is suitable for Linux (§5.2).

5.1 Design Options

In general, correct support for page pinning with COW requires two primitives: (1) a check whether a page *might* be COW-shared, which we denote as $maybe\_shared(page)$; and (2) a check whether a page *might* be pinned, which we denote as $maybe\_pinned(page)$. Two invariants must be maintained for pages that might need to be copied on write:

(1) If a page is COW-shared:
$maybe\_shared(page) \rightarrow true \wedge$
$maybe\_pinned(page) \rightarrow false.$
(2) If a page is pinned:
$maybe\_shared(page) \rightarrow false \wedge$
$maybe\_pinned(page) \rightarrow true.$

In other words, the OS needs to be able to identify COW-shared and pinned pages reliably and ensure they are mutually exclusive. By definition, using *maybe_pinned* and *maybe_shared* primitives is more restrictive than determining precisely whether a page is pinned or shared, as consequently using these primitives ensures correct detection of pinned and shared pages.

Once a page is pinned or a write-access to a write-protected page is trapped, the OS needs to decide whether the page should be reused (i.e., write-unprotected on write-accesses) or should be copied (COW-unshared). Each case should be properly treated: (1) COW-shared pages must be copied; (2) pinned pages (i.e., not COW-shared) must be reused; (3) unshared and unpinned pages can be either copied or reused. For performance and memory savings, it is better to reuse pages when possible. Accordingly, on pinning event

or trapped write-access, the OS would COW-unshare the page if *maybe_shared*(*page*) is true, and otherwise would reuse the page.

COW-sharing of pinned pages, for instance during `fork()` cannot be performed, as we show in §4. The OS therefore checks whether the page is pinned using the *maybe_pinned*(*page*) primitive if the page might be pinned, and prevents COW-sharing of these pages.

Various implementations of *maybe_pinned*(*page*) are possible. As one example, the OS can decide to consider all pages as potentially pinned, which would effectively prevent any COW sharing. On the other extreme, the OS can track the exact number of times each page is pinned and consider a page as potentially pinned if the counter is non-zero; doing so would allow the OS to maximize the page sharing opportunities, but would increase memory consumption.

Similarly, *maybe_shared*(*page*) can be implemented in different ways. One option, if page pinning is tracked accurately, is to consider any non-pinned page as potentially shared. This would be correct, but would lead to unnecessary copying of each page that is not pinned. At the other extreme, the OS can track all the page mappings and provide an accurate answer whether the page is shared; this would minimize the number of COW-unsharing operations at the cost of additional complexity and performance overheads. Another option is for the OS to consider a page as potentially COW-shared only if it is known not to be pinned and some lightweight best-effort checks cannot determine that it is not shared.

5.2 Linux Design

Following the analysis of the design options, we consider how they can be applied in Linux. We first explore whether Linux already provides proper primitives of *maybe_pinned*(*page*) and *maybe_shared*(*page*).

As it turns out, Linux provides a suitable basis for the *maybe_pinned*(*page*) primitive, which can be used to determine whether a page might be pinned. Indeed, Linux does not track the exact number of times that each page is pinned, since adding such a counter would have required an increase in the size of each page's metadata (`struct page`). As the metadata is aligned to the cache-line size in most architectures (64 bytes) for performance reasons, adding such counter would have doubled the page metadata memory consumption and reduced the amount of available memory by 1.5%. Linux tracks pinning imprecisely, by adding a large constant (1024) to the page's reference counter when a page is pinned, and reducing the counter by that constant when the page is unpinned.

This check might indicate pages that are COW-shared by many mappings (>1024) as pinned, which violates our requirement that *maybe_pinned*(*page*) $\rightarrow$ *false* for COW-shared pages. Conceptually, we could consider a page as potentially pinned only if it is not also considered as potentially COW-shared, as we describe next. However, as long as *maybe_pinned*(*page*) is not used for making COW-unsharing and pinning decisions, but only for COW-sharing decisions, there will not be correctness issues (for example "missed unsharing" as shown in Figure 2b). Consequently, we use the *maybe_shared*(*page*) primitive for making COW-unsharing and pinning decisions and restrict the use of the *maybe_pinned*(*page*) primitive to COW-sharing decisions.

Tracking whether a page is potentially COW-shared requires a new tracking scheme. Recent Linux version, which our implementation is based on, did not provide reliable primitives to determine whether a page is potentially COW-shared. Linux 5.8 and prior versions attempted to definitively determine whether a page is COW-shared based on number of mapping references (*map_count* and *swap_count*). This approach, however, introduced many correctness and security bugs [20–22] as well as performance overheads. It was consequently abandoned.

Due to the complexity of determining whether a page is COW-shared, Linux versions 5.9–5.18 use a simpler, yet inaccurate, logic to determine whether a page is COW-shared. However, this logic might effectively consider pinned pages as COW-shared, which can lead to memory corruptions. For example, if the lock of the page metadata is contended, a pinned page might be mistakenly considered as COW-shared.

To address this issue and to properly implement the *maybe_shared*(*page*) primitive, we introduce the Relaxed Copy-on-Pin (RelCOP) design. In RelCOP, we add to each page's metadata a single *exclusive* flag, which can be implemented without expanding the metadata size. When the flag is set it indicates that the page is guaranteed not to be COW-shared. We set the exclusive flag when a private page is allocated and following COW-unsharing events. We clear the flag when the page is COW-shared. This flag can therefore always provide—as required for our approach—a correct answer whether a page is potentially COW-shared without false-positive indications that a pinned page is COW-shared. To avoid unnecessary copies when deciding whether COW-unsharing is needed, if the exclusive flag is clear, we perform several lightweight checks to see if it is possible to quickly determine that the page is not COW-shared. If the page is determined not to be COW-shared, we set the exclusive flag.

Unlike Linux 5.8—the last version that attempted to determine accurately whether a page is COW-shared—our solution might cause unnecessary copies once a page was COW-shared and the *exclusive* flag was cleared. Such operations do not affect correctness, but might introduce small performance and transient memory overheads in certain cases. In practice, our solution performs better than Linux 5.8, as checking whether COW-unsharing is needed is faster. More importantly, our solution is more robust and has lower complexity, since it does not require accurately determining whether a page is COW-shared.

6 IMPLEMENTATION

We implemented Copy-on-Pin (COP) in Linux via our Relaxed Copy-on-Pin (RelCOP) design, as described in §5.2, for anonymous memory. We focus on anonymous memory because it is the most common one, and based on bug reports and our tests, the most problematic one. Our implementation comprises 747 added and 340 removed lines of code; it was integrated into upstream Linux 5.19 [61].

Exclusivity Detection. We implement the *exclusive* flag using a page flag in the page metadata, reusing an existing flag that does not have semantics for anonymous pages. We adjust COW and page pinning code to handle the *exclusive* flag as described in §5.2.

Prior to our work, the COW handling of anonymous memory used inconsistent and incorrect logic to determine whether COW-unsharing is needed. In some cases, the presence of more than a single page mapping was used to determine that the page is shared and should be copied. In others, the page was copied if the page had additional references, i.e., the reference counter was greater than one. We change the COW logic to determine that a page is exclusive, i.e., not shared, based on the *exclusive* flag. As mentioned above, this flag is always set for pinned pages, for which COW-unsharing must be avoided. If the flag is clear, we use the absence of additional references as definitive and robust indication that a page is not shared, and the page is reused without copying. Otherwise, the page may be shared. We perform COW-unsharing in this case, as we have already confirmed that the page is not pinned and, therefore, copying and remapping the page would not cause any correctness issues.

Exclusivity Detection Refinements. In practice, even pages that have additional references, and, therefore, their reference counter is greater than one, are frequently not shared, and copying these pages might introduce performance overheads. To avoid unnecessary copies, if the page has additional references we attempt to release these page references and recheck its reference counter. Specifically, we try to release the page from the swapcache and the Least Recently Used (LRU) cache, which is used to add pages efficiently to the LRU lists that are used for memory reclamation decisions.

Handling Swapping and Page Migration. If a page is swapped out or migrated, it is no longer associated with a page, and the OS cannot determine whether it is exclusive based on the *exclusive* flag residing in the page metadata. Losing track of the flag value would lead to unnecessary copies and can cause performance overheads. Therefore, we retain the flag value in such cases.

When pages are swapped out or migrated, Linux stores an indirect mapping—"swap entry" or "migration entry" respectively—in the non-present Page Table Entrys (PTEs) that mapped the page. The format of these entries needs to be revised to reflect the value of the *exclusive* flag. As one example, we introduce a new flag in the "swap entry" to store the *exclusive* flag while a page is swapped out. When `fork()` duplicates an indirect mapping, it updates swap entries and migration entries to indicate the page is not exclusive, similarly to the way the *exclusive* bit is cleared for present pages.

Handling Read-only Pinning. When pinning a COW-shared page for read-only accesses, it needs to first be copied for correctness. Treating read-only pinning as a write event is semantically incorrect and introduces various bugs that are not related to COW-unsharing logic [70]. Instead, we implement COW-unsharing during read-only pinning by leaving the copy mapped read-only, depicted in Figure 7. We reuse the write-fault handling code that handles traps on write-protected pages in order to perform COW-unsharing when necessary, before the page is pinned. Invoking this code is done by introducing a new flag to distinguish COW and read-only COP events.

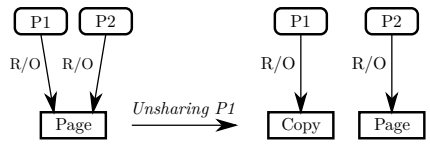


Figure 7: *COW-unsharing before read-only pinning*: replacing a COW-shared page in a private mapping P_1 by a private copy. In contrast to COW-unsharing during a write fault, the copy is mapped read-only.

Handling Transparent Huge Pages. Linux supports Transparent Huge Pages (THPs) [19], which are large pages (e.g., 2 MiB) that Linux maps transparently and opportunistically into a process address spaces instead of small (base) pages. As applications are unaware when large pages are used instead of base pages, they might remove part of the mapping, e.g., using the `munmap()` syscall. In such a case, Linux would first break the large page into base pages and replace the mapping of the large page with mappings of the base pages that the large page is comprised from. We need to ensure that the *exclusive* flag of each base page metadata is correct when the page is broken. Therefore, as long as a THP is mapped as a large page, we store the *exclusive* flag in the metadata of the large page. If and when a THP is broken up, we propagate this flag into the metadata of each base page that the large page is comprised of.

7 EVALUATION

While the motivation for our work and the major benefit of *Relaxed Copy-on-Pin (RelCOP)* is in providing simple, robust and correct COW handling scheme for pinned pages, we also wish that our solution would be performant and would not introduce noticeable overheads. We therefore compare the performance and overheads of our solution to those of the two prior COW handling schemes of Linux.

The first COW handling scheme, which we name *Precise Copy-on-Pin (PreCOP)*, was used in Linux 5.8 and prior versions. This scheme handles page pinning by determining *precisely* whether a page is shared based on the number of page mappings. As we discussed in §5.2, this approach has been abandoned as it was found to be complex and error-prone.

The second scheme, which we name *No Copy-on-Pin (NoCOP)*, is used in Linux 5.9–5.18. This scheme simplifies the COW logic by relaxing the conditions that lead to COW-unsharing. As we describe in §5.2, this scheme is incorrect. Nevertheless, we present the performance of *NoCOP*, as this scheme is the base for our implementation and has been used in Linux deployments. We compare these schemes to our scheme, *RelCOP*, which was integrated in Linux 5.19.

In order to ensure that our evaluation only measures the impact of the different COW schemes on the performance, and is not affected by other unrelated code changes of Linux, we use a single Linux kernel version—5.18—as a base for three custom Linux kernels that only differ in the way that they handle COW. By reverting COW-related patches we create kernels that use *PreCOP* and *NoCOP* schemes; by applying our patches we create a kernel that uses the *RelCOP* scheme. Note that we revert some patches to create a *NoCOP* kernel, because Linux 5.18 already contains some of our preparatory changes. Table 2 shows the custom kernels that were used for evaluation.

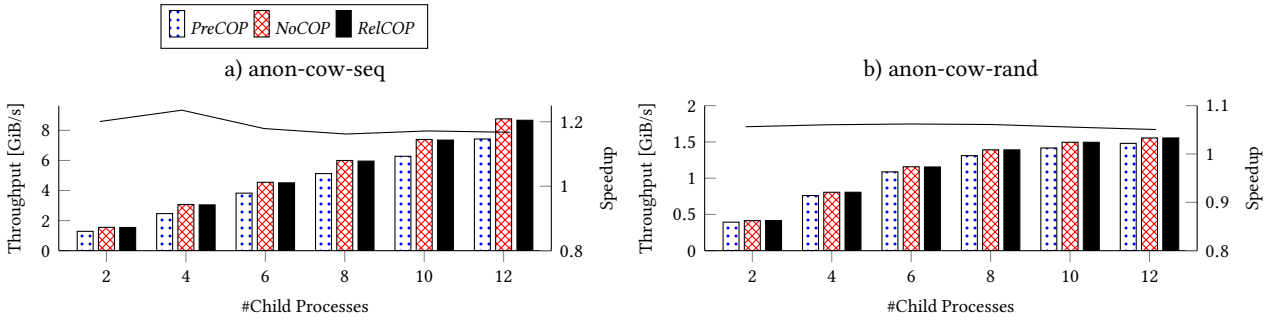


Figure 8: Average throughput when running the vm-scalability COW-handling benchmarks with THP enabled 50 times for the given number of child processes. The line shows the speedup of *RelCOP* relative to *PreCOP*.

Table 2: Custom Linux kernels we evaluate based on Linux 5.18, re-creating the COW and page pinning logic for anonymous memory in different Linux versions.

Approach	Linux	Known inaccuracy	Precise	Source
PreCOP	5.8	Some bugs	Yes	[1]
NoCOP	5.9–5.18	Yes	No	[3]
RelCOP	5.19-rc1	No	No	[2]

We perform our measurements on a server with a 12-core Intel Xeon Gold 6126 CPU, 2.6 GHz and 96 GiB of DDR4 memory. We run Fedora 36 with custom Linux kernels compiled using a Linux kernel configuration based on 5.18.5-200.fc36.x86_64, which is supplied by Fedora. We disable Intel Turbo Boost and Hyper-Threading to improve reproducibility. Further, we configure disk-based swap space to enable the swapcache, as required for one of the micro-benchmarks.

7.1 Micro-Benchmarks

We run micro-benchmarks to measure the direct impact of the different COW handling schemes on execution time.

Impossible Reuse. We measure the performance of COW-unsharing events using the `anon-cow-seq` and `anon-cow-rand` micro-benchmarks. These benchmarks are part of the vm-scalability test-suite [5] and are intended to measure the performance of COW handling of writes to anonymous memory. The benchmarks allocate private memory in a parent process and then fork several child processes, and as a result the OS performs COW-sharing of the allocated memory. The child processes write to the private memory sequentially and randomly respectively, thereby triggering page faults as the shared pages are write-protected. The kernel then checks whether the pages are shared, and since they are indeed shared, unshares them.

We run the benchmarks with different number of child processes (2–12), and with THP both enabled and disabled. We set the benchmark `unit size` to 1 GiB per child process, and run each benchmark 50 times. The average memory bandwidth result as measured by the benchmark is shown in Figure 8. Based on the results we calculate the speedup of *RelCOP* relatively to *PreCOP*, and present it as a line that corresponds to the secondary y-axis.

While the benchmark results without THP are practically identical in all schemes (data not shown), the results of the experiment with THP show that *RelCOP* is faster by up to 23% than *PreCOP* in the sequential access benchmark and by 6% in the random access benchmark. Accurate checks whether a THP is shared in the *PreCOP* scheme require to acquire a lock and access metadata of multiple base-pages, thereby inducing notable overhead. *NoCOP* avoids this overhead by using a simple check to determine that the page is potentially shared based on its reference count, and therefore should be copied. *RelCOP* performance is effectively the same as *NoCOP*'s and better than *PreCOP*'s, while being correct, simpler and more robust than the alternatives.

Possible Reuse. The main limitation of our solution, *RelCOP*, is that it can in certain corner cases copy pages unnecessarily, and as a result introduce performance and momentary memory overheads. To check the impact and extend of this limitation, we run multiple benchmarks that might cause *RelCOP* to unnecessarily copy pages that are in fact not shared. In these benchmarks, page reuse is always safe.

We run the STREAM [48] memory bandwidth benchmark, and configure it to run on a single core and access 1 GiB. We make minor changes to the benchmark in order to initiate, before each iteration, different types of memory management operations that might mislead *RelCOP* into considering pages that are not shared as shared. These operations can cause short-term sharing of pages and/or elevate page reference counters that *RelCOP* uses as a secondary indicator whether pages are shared. The different operations that are carried are shown in Table 3.

In addition, to stress the COW systems when used alongside Kernel Samepage Merging (KSM)—Linux's memory deduplication engine—we make further modifications to the benchmark. The impact of KSM on the COW page-sharing checks is the greatest when pages are scanned for duplicates most frequently. In contrast, actual memory deduplication introduces performance overheads that would dominate the results. We therefore change the benchmark to set different content on each page and configure KSM to scan pages for duplicates as frequently as possible.

We report the average memory bandwidth and the average number of page copies per second in the write-fault handler. Note that these page copies are unwarranted as pages are not COW-shared in these tests. The results are depicted in Figure 9.

Table 3: Operations we initiate before each iteration in our modified STREAM benchmark that all result in the pages getting mapped read-only, triggering the write-fault handler on next write-access.

Operation	Description	COW-shared	OS References
mprotect	Write-protect+unprotect the pages using `mprotect()`.	No	unlikely
swap	Swapout pages via `madvise()`; read all pages.	No	swapcache
mprotect+ksm	Operation "mprotect"; trigger KSM run.	No	KSM
fork	Fork a child process and wait; let the child exit.	Briefly	unlikely
swap+fork	Operation "swap" follow by action "fork".	Briefly	swapcache
fork+ksm	Operation "fork"; trigger a KSM run.	Briefly	KSM

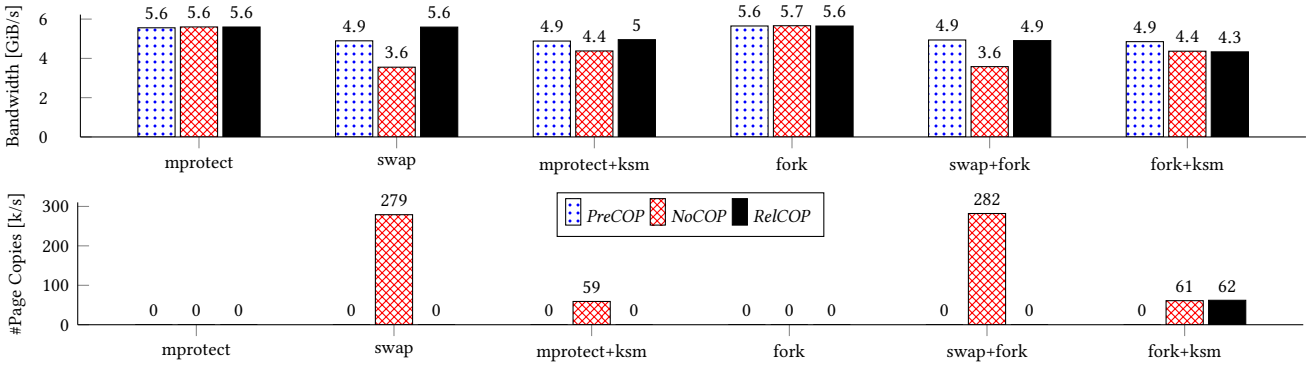


Figure 9: Average memory bandwidth and average number of page copies per second in the write-fault handler when running our modified STREAM benchmark, whereby we initiate different operations (Table 3) before each iteration.

As shown, *RelCOP* copies pages unnecessarily only in 1 of the 6 tests. In contrast, *NoCOP* unnecessarily copies pages in 4 of the 6 test, and in two of the tests—"swap" and "swap+fork"—copies essentially each one of the pages, although they are not shared. *NoCOP* copies pages unnecessarily quite often since it cannot determine whether a page is shared using its simplified checks, and therefore copies pages as a fallback. *RelCOP* not only addresses *NoCOP* correctness issues, but also, by tracking page exclusivity, eliminates most of the unnecessary page copies.

The memory bandwidth results, which reflect the benchmark performance, further show the benefits of *RelCOP*. *RelCOP* performs better than *NoCOP* in the two tests that swap out pages by up to 57%. *RelCOP* performance is as good or better than *PreCOP* in all tests excluding "fork+ksm", in which *RelCOP* creates unnecessary page copies.

In general, the maximum memory bandwidth in the KSM cases is lower compared to the other cases. KSM walks process page tables and scans pages to identify duplicates while the benchmark is running. As this concurrent KSM activity consumes memory bandwidth, it reduces the memory bandwidth that is available for the benchmark.

Unnecessary page copy operations during COW-unsharing reduce the memory bandwidth that is measured by the benchmark. Copying each page requires to retrieve the target page from the memory to the caches, and to perform several synchronization and accounting operations. During the time of these operations the benchmark does not run. As a result, the benchmark reports considerably lower memory bandwidth when unnecessary page copy operations are carried by the page fault handler, for instance when NoCOP is used in the "swap" and the "swap+fork" cases.

Interestingly, *RelCOP* performance is higher by 14% than the performance of *PreCOP* when memory is first swapped out. Our analysis shows that *RelCOP* performs better since it does not remove pages frequently from the swapcache—as done by *PreCOP*—which can introduce significant overheads. Unlike *RelCOP*, *PreCOP* cannot easily realize whether a page that resides in the swapcache is shared and needs to perform expensive synchronization to check whether the page is shared or should be removed from the swapcache. In contrast, *RelCOP* only removes pages from the swapcache when the *exclusive* marker is not set.

Write-fault Performance without COW. While our solution is intended to improve COW-related operations, it also affects other operations, and specifically write page-faults in which no COW-unsharing is necessary. Usually, regardless of the COW handling scheme, such page-faults do not trigger page copying. However, determining whether COW-unsharing is necessary is done differently in each scheme. For write page-faults that can be quickly resolved by the OS, i.e., without expensive I/O operations, memory copies or TLB shootdowns, the overhead of determining whether COW-unsharing is needed can be observable.

We therefore evaluate the overhead of write page-fault handling when no COW-sharing takes place. We allocate 1 GiB of memory, initialize the memory with zeroes, and then map the pages write-protected into the page table such that the next write access will trigger a write-fault to map the page writable again. We then perform one write operation to each page. We ensure that the OS does

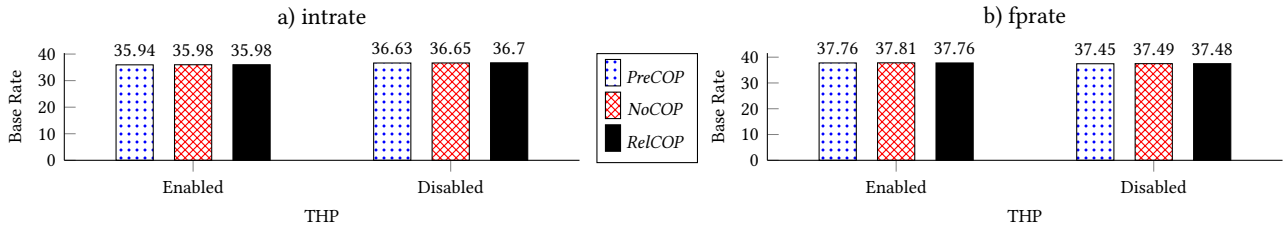


Figure 10: Final results when performing a reportable execution of the `intrate` and `fprate` benchmarks part of SPEC CPU 2017 suite with 12 copies and three iterations. We run the benchmarks once with THP enabled and once with THP disabled.

not perform unnecessary-COW operations during the tests. By measuring the overall time of the test, we calculate the average duration of a single write access—including write fault overhead.

Our results show that write-fault handling in *RelCOP* (1092 ns) is 0.8% faster than *NoCOP* (1101 ns) and 2.2% faster than *PreCOP* (1116 ns). These results highlight an additional benefit of *RelCOP*: it effectively acts as a memoization mechanism to save whether a page may be shared or is not shared. As a result, the page-fault handler can handle write-faults that do not require COW-unsharing faster.

7.2 Macro-Benchmarks

We assume that the COW handling scheme would not have observable performance impact on common workloads. First, common workloads do not make heavy use of `fork()` such that a differing COW strategy would notably affect overall application performance. Second, the usage of KSM is usually restricted to very specific applications, such as VMs, and the extreme KSM scanning as configured in the micro-benchmark is not used in practice due to the notable scanning overhead. Third, swapping already negatively affects performance of common workloads, for example, due to expensive disk access. Common workloads that are performance-sensitive neither rely on swapping nor KSM. Consequently, we expect that the differences revealed by our micro-benchmarks do not affect common workloads.

To validate our assumption, we run the SPECrate2017 [56] test-suite, which executes a variety of different workloads derived from real user applications. The output of this benchmark is a "base rate" score, which correlates with the benchmark throughput; a higher "base rate" corresponds to a better benchmark result. We run two reportable benchmark executions with 12 copies and three iterations, once with THP enabled and once with THP disabled. The results are depicted in Figure 10. As expected, the `fprate` and `intrate` results are practically identical: the difference between the lowest and the highest "base rate" is in all cases at most 0.7, corresponding to at most 0.2% of the lowest "base rate". Consequently, the SPECrate2017 results do not indicate a practical performance impact or benefit for any approach.

7.3 Discussion

As our evaluation shows, unnecessary copies in the *RelCOP* approach are in general not an issue once optimizing for the important cases (e.g., swapcache) and do not affect performance of common applications in any noticeable way. *RelCOP* effectively acts as a memoization mechanism to whether a page may be shared or is not shared, which avoids unnecessary copies in the important cases and achieves the fastest write-fault performance in the common case. Excluding one corner case, *RelCOP* performs the same or better than *PreCOP* due to more efficient write-fault handling.

8 RELATED WORKS

COW is used in various domains (e.g., storage), yet resource pinning appears to be unique to memory management. We therefore limit our related work survey to memory management.

Interestingly, the COW algorithm itself does not receive a lot of attention. Fábrega et al. formulate the correctness of COW, in the context of the Mach micro-kernel [30]. Their work is specific to Mach and does not consider pinning. Accetta et al. describe that COW implementation in Mach [6]. Cranor et al. similarly describe COW in UVM (Universal Virtual Machine) [26], and compare it to the traditional BSD VM [49].

COW Optimizations. On-Demand-Fork uses COW for the deduplicate page-tables during `fork` to reduce application invocation latency [73] . Similarly, sharing of data structures during fork() is proposed by Mitosis [25].

CoWLight offloads COW-unsharing to the hardware for reduced copy latency [54]; the hardware has access to a list of free pages and the OS uses a newly define architectural PTE flag to perform hardware assisted COW. *Coverage-based Copy-on-Write (CCoW)* optimizes COW for write-intensive workloads, by initiating COW-unsharing of adjacent pages during page faults, which reduces the number of page faults [36].

Seshadri et al. [55] use *page overlays* to extend the concept of COW to *Overlay-on-Write*: instead of unsharing COW on write access, the original page remains mapped and modifications are recorded in a page overlay. The Difference Engine [35] proposed a similar software solution for memory deduplication via *patches* to shared pages.

Memory Deduplication. COW is commonly used for memory deduplication. Memory deduplication for virtual machines was first described for VMware ESX [68] and later similarly adapted under Linux via KSM [11] and under Windows [17]. It is an active research field, with a focus on making memory deduplication more efficient [18, 32, 35, 40, 42, 50, 51, 66, 72] and identifying security implications [17, 43, 47, 64, 65, 69].

Additional COW Use Cases. WINNIE [41] uses COW for performant fuzzing of Windows applications by providing an efficient

`fork()` implementation under Windows. Xu et al. [71] describe a new `snapshot()` syscall that employs COW for efficient capturing of the state of a process, to revert to that state later for fuzzing purposes. Bittau [15] similarly introduces `checkpoint()` and `resume()` syscalls that rely on COW for saving and restoring process state to avoiding expensive `fork()`.

Several studies use COW for fast live-cloning of VMs, for example, for cloning short-lived honeypots [44, 57, 67]. z-READ [53] remaps user pages by COW-sharing pagecache pages during the `read()` syscall, avoiding copying page content. Chu [39] describes zero-copy for TCP in Solaris using COW when writing data to a socket: the source page is marked COW-shared in the mapping such that it gets replaced by a copy on modification attempts before the transmission is done.

9 CONCLUSIONS

The interaction of COW with page pinning has not been well-defined for a long time, resulting in various OS bugs that affected security, correctness and performance. In this work, we analyzed these bugs and the desired behavior of COW mechanisms. Based on this analysis we showed how COW can be performed correctly without introducing performance overheads or complexity. Our solution is robust and has been included in Linux 5.19, resolving a variety of bugs that were lurking for years.

While our work addresses the interaction of page pinning and COW, the need for page pinning requires further thought. Page pinning has undesired side effects, such as memory fragmentation and increased memory consumption, but it is currently considered a "necessary evil" as it provides considerable performance gains over the alternatives. The question whether a performant alternative to page pinning can be developed remains open.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers and shepherd as well as Dr. David Alan Gilbert for their valuable feedback. Further, we would like to thank the Linux community—most notably Linus Torvalds, Peter Xu, John Hubbard, Jason Gunthorpe, Hugh Dickins, Vlastimil Babka and Matthew Wilcox—for their previous work on Linux' handling of COW with page pinning and for their invaluable feedback and guidance. Last but not least, we would like to sincerely thank Andrea Arcangeli for all his prior work on discovering pinning-related COW bugs in Linux and development of possible fixes, including the idea of not treating read-only pinning as a write event.

A ARTIFACT APPENDIX

A.1 Abstract

The artifacts include patches for three custom Linux kernels based on Linux 5.18, one of these kernels corresponding to our RelCOP implementation. In addition, the artifacts contain our modified STREAM benchmark, a copy of the vm-scalability benchmark, a simple write-fault-duration benchmark, a SPECrate2017 config file, some helper scripts, and a copy of the O_DIRECT+fork test cases.

A.2 Artifact Check-List (Meta-Information)

- **Program:** Three micro-benchmarks; SPECrate2017 as macro-benchmark.
- **Compilation:** GCC as included in Fedora 36.
- **Run-time environment:** Fedora 36.
- **Hardware:** x86-64 server with at least 12 cores and 32 GiB of RAM.
- **Execution:** Sole user with sudo (root) permissions; multiple reboots required.
- **Metrics:** The vm-scalability benchmark measures throughput. The modified STREAM benchmark measures memory bandwidth and the number of page copies per second. The write-fault-duration benchmark measures the duration in ns. The output of the SPECrate2017 benchmark is a "base rate".
- **Output:** CSV files.
- **Experiments:** Three custom Linux kernels are evaluated; reboots are required to switch between these kernels. OS scripts are provided to run the benchmarks.
- **How much disk space required (approximately)?:** 5 GiB, excluding SPECrate2017 (~50 GiB)
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 5 hours, excluding SPECrate2017 runtime (~24 hours per kernel).
- **Publicly available?:** Yes, our implementation was included into upstream Linux and the artifacts are publicly available.
- **Code licenses:** The Linux Kernel is provided under the terms of the GNU General Public License version 2 only (GPL-2.0).
- **Archived:** 10.5281/zenodo.7333207 [37]

A.3 Description

A.3.1 How to Access. The artifacts are available on Gitlab (https://gitlab.com/cop_paper/ae) and include a `README.md` file with further information on installation, configuration and evaluation.

A.3.2 Hardware Dependencies. x86-64 machine with root access, $\geq$ 12 cores and $\geq$ 32 GiB of memory.

A.3.3 Software Dependencies. Some Linux packages have to be installed from the distributions' package manager in order to compile the benchmarks and the custom Linux kernels. The included `install.sh` script will perform this step automatically.

A.3.4 Data Sets. The experiments include running the proprietary SPECrate2017 benchmark, part of SPEC CPU 2017; this benchmark is not included. Note that alternative benchmarks might be used to show that the different approaches have no effect on the performance of common workloads.

A.4 Installation

We only provide a short overview of installation and configuration steps. More information can be found in the included `README.md` file.

A.4.1 Installing Custom Linux Kernels. The `kernels` folder contains an `install.sh` script that will install package dependencies, download Linux 5.18, compile the three custom Linux kernel and install them automatically.

Table 4: Possible relationship between the custom Linux kernels and the boot indexes.

Index	Kernel Name	Approach
0	5.18.0-relcop	RelCOP
1	5.18.0-nocop	NoCOP
2	5.18.0-precop	PreCOP

A.4.2 Installing SPECrate2017. Please follow the official SPEC CPU 2017 instructions to install SPEC CPU 2017, and to check whether the installation is working as expected.

A.4.3 SWAP Configuration. The modified STREAM benchmark requires configuration of proper disk-based swap space, otherwise the swapcache is not used as intended.

A.5 Switching Between Custom Linux Kernels

In order to run the experiments under the three custom Linux kernels, Linux has to be re-configured to boot with a specific kernel. This can either be achieved by selecting the desired Linux kernel in the boot manager, or by using the `grubby` command.

See the `README.md` file for details on how to switch between Linux kernels. The relationship between the custom Linux kernels and the boot indexes can be observed via the `grubby` command and might be as shown in Table 4.

A.6 Basic Test

Being able to successfully boot Linux with the custom Linux kernel that corresponds to our RelCOP implementation is sufficient to show that the kernel was installed correctly and that the basics are working—Linux makes heavy use of `fork()` and, therefore, COW during boot.

Boot the installed RelCOP custom Linux kernel (**5.18.0-relcop**) as described in A.5 and validate that the correct kernel was booted:

```
$ uname –a
5.18.0–relcop
```

A.7 Experiment Workflow

The overall experiment workflow is as follows:

(1) Compile and install the three custom Linux kernels.
(2) Install SPEC CPU 2017.
(3) Boot into RelCOP kernel and run benchmark experiments.
(4) Boot into NoCOP kernel and run benchmark experiments.
(5) Boot into PreCOP kernel and run benchmark experiments.

The benchmark experiments for each custom Linux kernel are as follows:

(1) Run vm-scalability benchmark experiment.
(2) Run modified STREAM benchmark experiment.
(3) Run write-fault-duration benchmark experiment.
(4) Run reportable SPECrate2017 experiment.

A.8 Evaluation and Expected Results

The `README.md` file contains details on how to prepare, how to run the benchmarks, and which CSV files are generated. We only shortly summarize the evaluation in this section.

A.8.1 Vm-Scalability Benchmark. To run the two *anon-cow* vm-scalability benchmarks with THP enabled and with THP disabled, use the `run-vm-scalability.sh` script in the `benchmarks` directory. Use the helper script `eval-vm-scalability.sh` to compute the average throughput across all benchmark iterations, and to output the results to new CSV files.

Figure 8 depicts our results from this experiment with THP enabled. With THP disabled, the results should be practically identical. Note that the speedup of RelCOP relative to PreCOP has to be calculated manually.

A.8.2 Modified STREAM Benchmark. To run the modified STREAM benchmark that executes selected actions before each benchmark iteration, use the `run-stream.sh` script in the `benchmarks` directory. Use the helper script `eval-stream.sh` to compute the average memory bandwidth and average number of page copies per second across all benchmark iterations, and to output the results to new CSV files.

The results from this experiment are expected to be similar to the results depicted in Figure 9.

A.8.3 Write-Fault-Duration Benchmark. To run the write-fault-duration benchmark, use the `run-write-fault-duration.sh` script in the `benchmarks` directory. To compute the average duration across all benchmark iterations and to output the results to a new CSV file, use the helper script `eval-write-fault-duration.sh`.

The results from this experiment are expected to be similar to the results presented in §7.1.

A.8.4 SPECrate2017 Benchmark. Follow the SPEC CPU 2017 installation instructions and our instructions to properly setup and configure the SPECrate2017 experiment in the `README.md` file.

Use the `run-spec.sh` script in the `benchmarks` directory to run the intrate and fprate benchmarks with 12 tasks and 3 iterations, once with THP enabled and once with THP disabled. As discussed in §7.2 and depicted in Figure 10, the benchmark results for each of the custom Linux kernels should be practically identical.

REFERENCES

[1] 2022. Soruce code of custom Linux kernel based on 5.18 that implements the COW logic from 5.8. https://gitlab.com/cop_paper/linux/-/tree/precop.
[2] 2022. Source code of custom Linux kernel based on 5.18 that implements the COW logic from 5.19. https://gitlab.com/cop_paper/linux/-/tree/relcop.
[3] 2022. Source code of custom Linux kernel based on 5.18 that implements the COW logic from 5.9. https://gitlab.com/cop_paper/linux/-/tree/nocop.
[4] 2022. Source code of generic O_DIRECT and fork() test cases. https://gitlab.com/cop_paper/o_direct_fork_tests/-/tree/cop_paper.
[5] 2022. Source code of vm-scalability benchmark. https://git.kernel.org/pub/scm/linux/kernel/git/wfg/vm-scalability.git.
[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young, and Robert Baron Mike Accetta. 1986. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 Usenix Conference*. USENIX Association, San Diego, CA, USA, 93–112.
[7] Nadav Amit. 2020. mm/userfaultfd: fix memory corruption due to writeprotect. https://lore.kernel.org/all/20201219043006.2206347-1-namit@vmware.com/.
[8] Nadav Amit. 2021. mm: unnecessary COW phenomenon. https://lore.kernel.org/all/FFA0057D-1A17-4DF4-9550-A8CDEE9E0CE0@gmail.com/.
[9] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *International Symposium on Computer Architecture*. Springer, Springer-Verlag, Berlin, Heidelberg, 256–274. https://doi.org/10.1007/978-3-642-24322-6_22
[10] Andrea Arcangeli. 2014. Re: [Qemu-devel] [PATCH 00/17] RFC: userfault v2. https://lists.gnu.org/archive/html/qemu-devel/2014-11/msg03088.html.

[11] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Ottawa Linux Symposium (OLS)*. Montreal, Quebec, Canada, 19–28.

[12] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[13] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A fork() in the road. In *ACM Workshop on Hot Topics in Operating Systems (HOTOS)*. Association for Computing Machinery, New York, NY, USA, 14–22. https://doi.org/10.1145/3317550.3321435

[14] A. H. Bell-Thomas. 2020. Interprocess Communication in FreeBSD 11: Performance Analysis. https://arxiv.org/abs/2008.02145. https://doi.org/10.48550/ARXIV.2008.02145

[15] Andrea Bittau. 2009. *Toward Least-Privilege Isolation for Software*. Ph. D. Dissertation. University College London.

[16] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. 1972. TENEX, a Paged Time Sharing System for the PDP - 10. *Communications of the ACM (CACM)* 15, 3 (1972), 135–143. https://doi.org/10.1145/361268.361271

[17] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 987–1004. https://doi.org/10.1109/SP.2016.63

[18] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. 2014. CMD: Classification-based Memory Deduplication through page access characteristics. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2576195.2576204

[19] Jonathan Corbet. 2011. Transparent huge pages in 2.6.38. https://lwn.net/Articles/423584/.

[20] The MITRE Corporation. 2020. CVE-2020-29368. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-29368.

[21] The MITRE Corporation. 2020. CVE-2020-29374. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-29374.

[22] The MITRE Corporation. 2021. CVE-2021-39802. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39802.

[23] Alan Cox. 2009. Correct an error in vm_fault_copy_entry(). https://github.com/freebsd/freebsd-src/commit/e4ed417a355e2cfcb7ee5b9caa6be9c2ed239fae.

[24] Alax Cox. 2009. Simplify both the invocation and the implementation of vm_fault() for wiring. https://github.com/freebsd/freebsd-src/commit/2db65ab46e54af2f56b711c9049e5321bab88a17.

[25] Alan Cox and Juan Navarro. 2001. *Mitosis: A High Performance, Scalable Virtual Memory System*. Technical Report. Rice University, Houston, Texas, USA.

[26] Charles D. Cranor and Gurudatta M. Parulkar. 1999. The UVM virtual memory system. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, San Diego, CA, USA.

[27] Hugh Dickins. 2005. can_share_swap_page: use page_mapcount. https://lore.kernel.org/all/Pine.LNX.4.61.0506062058090.5000@goblin.wat.veritas.com/.

[28] Hugh Dickins. 2014. mm: get_user_pages(write,force) refuse to COW in shared areas. https://lore.kernel.org/all/alpine.LSU.2.11.1404040120110.6880@eggly.anvils/.

[29] John Dyson. 1997. Fix the gdb executable modify problem. https://github.com/freebsd/freebsd-src/commit/a04c970a7aa272333bfa26014f64f461006db115.

[30] Francisco Javier Thayer Fábrega, Francisco Javier, and Joshua D. Guttman. 1995. Copy on Write. (1995).

[31] Robert Fitzgerald and Richard F. Rashid. 1986. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems (TOCS)* 4, 2 (1986), 147–177. https://doi.org/10.1145/214419.214422

[32] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. 2017. Catalyst: GPU-assisted rapid memory deduplication in virtualization environments. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. Association for Computing Machinery, New York, NY, USA, 44–59. https://doi.org/10.1145/3050748.3050760

[33] Google. 2021. Android Developer Documentation: Overview ofmemory management. https://developer.android.com/topic/performance/memory-overview.

[34] Jason Gunthorpe. 2020. Re: mm: Trial do_wp_page() simplification. https://lore.kernel.org/all/20200914143829.GA1424636@nvidia.com/.

[35] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2010. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM (CACM)* 53, 10 (2010), 85–93. https://doi.org/10.1145/1831407.1831429

[36] Minjong Ha and Sang Hoon Kim. 2022. CCoW: Optimizing Copy-on-Write Considering the Spatial Locality in Workloads. *Electronics (Switzerland)* 11, 3 (2022). https://doi.org/10.3390/electronics11030461

[37] David Hildenbrand, Martin Schulz, and Nadav Amit. 2022. *Software artifacts for the paper "Copy-on-Pin: The Missing Piece for Correct Copy-on-Write"*. https://doi.org/10.5281/zenodo.7333207

[38] Jann Horn. 2020. Linux: CoW can wrongly grant write access. https://bugs.chromium.org/p/project-zero/issues/detail?id=2045.

[39] Hsiao Keng Jerry Chu. 1996. Zero-copy TCP in Solaris. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, San Diego, CA, USA.

[40] Shuaijie Jia, Chentao Wu, and Jie Li. 2017. Loc-K: A spatial locality-based memory deduplication scheme with prediction on k-step locations. In *IASTED International Conference on Parallel and Distributed Computing and Systems (ICPDCS)*. IEEE, 310–317. https://doi.org/10.1109/ICPADS.2017.00049

[41] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE : Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Proceedings 2021 Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2021.24334

[42] Sung Hun Kim, Jinkyu Jeong, and Joonwon Lee. 2014. Selective memory deduplication for cost efficiency in mobile smart devices. *IEEE Transactions on Consumer Electronics* 60, 2 (2014), 276–284. https://doi.org/10.1109/TCE.2014.6852004

[43] Taehun Kim, Taehyun Kim, and Youngjoo Shin. 2021. Breaking kaslr using memory deduplication in virtualized environments. *Electronics (Switzerland)* 10, 17 (2021). https://doi.org/10.3390/electronics10172174

[44] Denis Lavrov, Véronique Blanchet, Shaoning Pang, Muyang He, and Abdolhossein Sarrafzadeh. 2017. COR-Honeypot: Copy-On-Risk, virtual machine as Honeypot in the cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 908–912. https://doi.org/10.1109/CLOUD.2016.0134

[45] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page Fault Support for Network Controllers. *ACM SIGARCH Computer Architecture News (CAN)* 45, 1 (2017), 449–466. https://doi.org/10.1145/3093337.3037710

[46] Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. 2021. A Comparative Study of Consistent Snapshot Algorithms for Main-Memory Database Systems. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2021), 316–330. https://doi.org/10.1109/TKDE.2019.2930987

[47] Jens Lindemann and Mathias Fischer. 2019. On the detection of applications in co-resident virtual machines via a memory deduplication side-channel. *ACM SIGAPP Applied Computing Review* 18, 4 (2019), 31–46. https://doi.org/10.1145/3307624.3307628

[48] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995), 19–25.

[49] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Longman Publishing Co., Inc., USA.

[50] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. 2012. KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*.

[51] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. 2013. XLH: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, San Jose, CA, USA, 279–290.

[52] Jiwoong Park, Yunjae Lee, Heon Young Yeom, and Yongseok Son. 2020. Memory efficient fork-based checkpointing mechanism for in-memory database systems. In *ACM Symposium on Applied Computing (SAC)*. IEEE, 420–427. https://doi.org/10.1145/3341105.3375782

[53] Jiwoong Park, Cheolgi Min, Heon Young Yeom, and Yongseok Son. 2019. Z-READ: Towards efficient and transparent zero-copy read. In *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 367–371. https://doi.org/10.1109/CLOUD.2019.00066

[54] T. Santhosh Kumar, 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*. Association for Computing Machinery, New York, NY, USA, 8 pages. https://doi.org/10.1145/3337167.3337170

[55] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA, 79–91. https://doi.org/10.1145/2749469.2750379

[56] Standard Performance Evaluation Corporation. 2020. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[57] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. 2009. Fast live cloning of virtual machine based on xen. In *2009 11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 392–399. https://doi.org/10.1109/HPCC.2009.97

[58] The Open Group. 2008. Base Specifications Issue 7. IEEE Std 1003.1-2008.

[59] Linus Torvalds. 2020. gup: document and work around "COW can break either way" issue. https://patchwork.kernel.org/project/linux-mm/patch/20210421225750.60668-1-surenb@google.com/.

[60] Linus Torvalds. 2020. mm: do_wp_page() simplification. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=09854ba94c6a.

[61] Linus Torvalds. 2022. Merge tag 'mm-stable-2022-05-25' of git://git.kernel.org/pub/scm/linux/kernel/git/akpm/mm. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=98931dd95fd4.

[62] Michael S. Tsirkin. 2006. madvise MADV_DONTFORK/MADV_DOFORK. https://lore.kernel.org/all/20060213233517.GG13603@mellanox.co.il/.

[63] Shin-Yuan -Y Tzou and David P. Anderson. 1991. The performance of message-passing using restricted virtual memory remapping. *Software: Practice and Experience* 21, 3 (1991), 251–267. https://doi.org/10.1002/spe.4380210303

[64] Fernando Vano-Garcia and Hector Marco-Gisbert. 2020. An Info-Leak Resistant Kernel Randomization for Virtualized Systems. *IEEE Access* 8 (2020), 161612–161629. https://doi.org/10.1109/ACCESS.2020.3019774

[65] Fernando Vano-Garcia and Hector Marco-Gisbert. 2020. KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant cloud systems. *J. Parallel and Distrib. Comput.* 137 (2020), 77–90. https://doi.org/10.1016/j.jpdc.2019.11.008

[66] T. Veni and S. Mary Saira Bhanu. 2014. MDedup++: Exploiting Temporal and Spatial Page-Sharing Behaviors for Memory Deduplication Enhancement. *Comput. J.* 59, 3 (2014), 353–370. https://doi.org/10.1093/comjnl/bxu149

[67] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review (OSR)* 39, 5 (2005), 148–162. https://doi.org/10.1145/1095810.1095825

[68] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review (OSR)* 36, Special Issue (2002), 181–194. https://doi.org/10.1145/844128.844146

[69] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2013. Security implications of memory deduplication in a virtualized environment. In *IEEE International Conference on Dependable Systems & Networks (DSN)*. IEEE, 1–12. https://doi.org/10.1109/DSN.2013.6575349

[70] Peter Xu. 2020. mm/gup: Allow real explicit breaking of COW. https://lore.kernel.org/all/20200808223802.11451-1-peterx@redhat.com/.

[71] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 2313–2328. https://doi.org/10.1145/3133956.3134046

[72] Lingjing You, Yongkun Li, Fan Guo, Yinlong Xu, Jinzhong Chen, and Liu Yuan. 2019. Leveraging Array Mapped Tries in KSM for Lightweight Memory Deduplication. In *2019 IEEE International Conference on Networking, Architecture and Storage, NAS 2019 - Proceedings*. IEEE, 1–8. https://doi.org/10.1109/NAS.2019.8834730

[73] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *EuroSys 2021 - Proceedings of the 16th European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258

Received 2022-07-07; accepted 2022-09-22