Don’t shoot down TLB shootdowns!

Nadav Amit
VMware Research

Amy Tai
VMware Research

Michael Wei
VMware Research

Abstract

Translation Lookaside Buffers (TLBs) are critical for building performant virtual memory systems. Because most processors do not provide coherence for TLB mappings, TLB shootdowns provide a software mechanism that invokes inter-processor interrupts (IPIs) to synchronize TLBs. TLB shootdowns are expensive, so recent work has aimed to avoid the frequency of shootdowns through techniques such as batching. We show that aggressive batching can cause correctness issues and addressing them can obviate the benefits of batching. Instead, our work takes a different approach which focuses on both improving the performance of TLB shootdowns and carefully selecting where to avoid shootdowns. We introduce four general techniques to improve shootdown performance: (1) concurrently flush initiator and remote TLBs, (2) early acknowledgement from remote cores, (3) cacheline consolidation of kernel data structures to reduce cacheline contention, and (4) in-context flushing of userspace entries to address the overheads introduced by Spectre and Meltdown mitigations. We also identify that TLB flushing can be avoiding when handling copy-on-write (CoW) faults and some TLB shootdowns can be batched in certain system calls. Overall, we show that our approach results in significant speedups without sacrificing safety and correctness in both microbenchmarks and real-world applications.

ACM Reference Format:
Nadav Amit, Amy Tai, and Michael Wei. 2020. Don’t shoot down TLB shootdowns!. In *Fifteenth European Conference on Computer Systems (EuroSys ’20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3342195.3387518

1 Introduction

Translation lookaside buffers (TLBs) are per-core caches which store virtual to physical memory mappings. TLBs are vital for ensuring the performance of the virtual memory system which enables the modern multitasking operating system. If a processor cannot find the mapping for a virtual memory address in the TLB, a costly TLB miss occurs which results in a time-consuming page walk to calculate the correct physical address from the page tables.

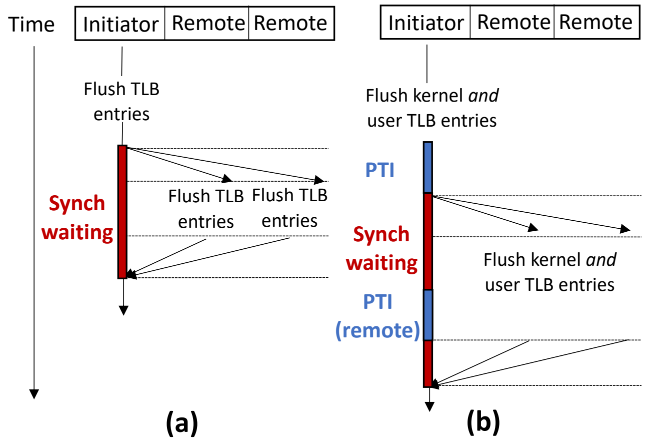


Figure 1. (a) In the baseline Linux TLB shootdown protocol, much of the shootdown is spent synchronously waiting for remote flushes. (b) Kernel page table isolation (PTI), a mitigation for Meltdown, exacerbates this wait time.

At the same time, it is critical for the mappings cached in the TLB to reflect the actual state of the memory resident page tables, providing what is known as TLB coherence. Otherwise, an application could access a stale mapping which could cause correctness or security issues. To update the TLB state, the operating system (OS) performs a TLB flush, which can either selectively remove or drop all mappings stored in the TLB. In most processors, however, TLBs are not coherent caches, so in a multiprocessor system, the OS must perform an explicit *TLB shootdown* [8], which sends an inter-processor interrupt (IPI) to remote cores to flush their TLBs. TLB shootdowns are costly, as they invoke a complex protocol which burdens all processors in the system and must wait for the acknowledgement of remote cores, requiring several thousand cycles to complete. The Meltdown hardware vulnerability [22] and the kernel Page Table Isolation (PTI) mitigation [30] used to defend against it has increased the requirement for synchronizing the TLBs for security. Figure 1 shows why TLB shootdowns have high overhead.

State-of-the-art work tries to avoid TLB shootdowns in order to improve the overall performance of virtual memory. These approaches can be broadly categorized into two

EuroSys ’20, April 27–30, 2020, Heraklion, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387518

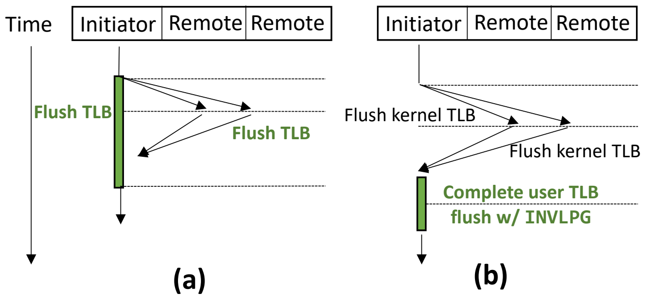


Figure 2. (a) In our modified protocol, we avoid synchronous waiting by concurrently flushing the local and remote TLBs and early acknowledgement from remote cores. (b) To deal with PTI, we defer user-level TLB flushes until kernel exit, in order to use the more efficient INVLPG instruction on user-level entries.

categories: *batching*, which can be performed either synchronously or asynchronously [2, 15, 31], and *interface changes*, which relax the requirements of traditional POSIX system calls, alleviating the OS from the burden of synchronizing TLBs through shootdowns [21]. While both approaches have been shown to provide significant speedup, we argue that aggressively eliminating TLB shootdowns can be error-prone and sometimes result in performance degradation and even safety violations. Interface extensions have the potential to provide performance improvements, yet adapting new OS interfaces usually takes considerable time.

In this work, we take a bottom-up approach instead, focusing first on improving the performance of the TLB shootdown and then carefully selecting TLB shootdowns to batch or eliminate. We argue that we do not have to resort to new microarchitectures or interface-breaking shootdown protocols. Instead, by spending our efforts making careful improvements to existing shootdown protocols, we not only achieve performance gains comparable to other redesigns, but also identify improvements that apply to TLB shootdowns triggered by any type of virtual memory operation. For example, we introduce two optimizations to the Linux TLB shootdown protocol: *concurrent flushing*, which flushes initiating and remote TLBs concurrently, and *early acknowledgement*, where remote cores acknowledge a shootdown as soon as they enter the interrupt handler, instead of after their local TLB flush. These two powerful optimizations eliminate much of the synchronous waiting present in existing TLB shootdown protocols.

We also implement *cacheline consolidation*, which consolidates the shared state necessary for a shootdown, thereby reducing cacheline contention. To alleviate the additional overhead of PTI on TLB flushing, we introduce *in-context flushes*, which defers flushing userspace entries to when the kernel switches the address space back to the user context. This allows us to take advantage of the INVLPG instruction, which is faster than the INVLPCID instruction, which is used to invalidate pages in a different context and must be used if flushing userspace entries from the kernel address space.

General techniques	1. Concurrent Flushing (§3.1)
	2. Early Acknowledgement (§3.2)
	3. Cacheline Consolidation (§3.3)
	4. In-Context Flushes (§3.4)
Use-case specific	5. CoW Fault (§4.1)
	6. Userspace-safe Batching (§4.2)

Table 1. Optimizations introduced in this work.

These four techniques effectively improve performance for any triggered shootdown. We further identify specific cases in which TLB flushes can be eliminated and shootdowns can be batched. For copy-on-write (CoW) pages, we avoid the local TLB flush by leveraging the page fault handler. We also introduce *userspace-safe batching*, which batches the flushing of mappings that will only be accessed in userspace. This applies to system calls such as msync and munmap. A summary of our optimizations can be found in Table 1.

We apply our optimizations to Linux 5.2.8 and show that we obtain significant performance improvements in both microbenchmarks and real-world workloads such as Sysbench and Apache. This paper makes the following contributions:

- We provide background on TLB shootdowns and how current approaches which focus on avoiding shootdowns can introduce more problems than the performance improvements they provide (§2).
- We describe our bottom-up approach, which identifies techniques for improving the performance of shootdowns and selectively reducing shootdowns (§3-§4).
- We evaluate the performance of our optimizations on Linux 5.2.8 and show that our approach extracts significant performance improvement without sacrificing safety or usability (§5).
- We discuss other sources of inefficiencies around TLBs, such as undocumented behavior which we leave as potential sources of optimizations in future work (§7).

2 Background

TLB flushes and shootdowns are mechanisms used by operating systems to synchronize page table entries (PTEs) cached in TLBs with the underlying page tables. Our work focuses on the Intel x86 architecture, though most other architectures also require software to provide TLB coherence. In the following sections, we first describe how the Linux kernel currently performs TLB flushes and shootdowns. Then we describe related work, which comes in two main categories: adding new hardware features and software changes, which primarily focus on avoiding shootdowns.

2.1 TLB Flushes

The x86 architecture provides several mechanisms to control the contents of the TLB. The coarsest mechanism is a full flush, which invalidates all PTEs not marked with a global bit (G). Kernel pages have the G bit set, as kernel PTEs do not typically change across context switches, when PTEs are typically flushed. On the x86, a full flush is achieved by writing to a control register known as CR3 [19]. The INVLPG instruction, introduced in the 80486 (1989), provides fine grain control by taking a virtual address to be invalidated [19]. Multiple pages can be invalidated only by calling the INVLPG instruction multiple times, so software has to weigh the costs of multiple INVLPG calls against just flushing the entire TLB by writing to CR3 (both mechanisms have similar costs [17]). FreeBSD, for example, performs a CR3 write when more than 4096 PTEs need to be invalidated, whereas Linux places the ceiling at 33 [17].

Starting with Westmere (2010), Intel introduced support for multiple address spaces IDs (ASID), with a feature known as process-context identifier (PCID). This feature allowed TLBs to cache mappings of multiple address spaces and associate each mapping with its address-space ID to perform address translations correctly and selectively avoid flushing the full TLB upon context switch. Later, Haswell (2013) extended its support for PCIDs, introducing the INVPCID instruction that enables selectively flushing TLB entries of inactive address spaces [19]. However, the adaptation of PCID by operating systems was slow for two reasons. First, it did not seem that there were many use-cases in which performance was degraded by frequent TLB misses caused by frequent context switches. Second, using PCIDs was nontrivial for operating systems writers, as the number of address spaces was limited to 1024, which prevented operating systems from simply associating process and address space based on the process ID.

The Meltdown security vulnerability [1, 22] took the community by surprise, and it turned out PCIDs were essential for restoring lost performance. Meltdown showed that it is possible to bypass privilege checks and leak data through speculative execution on vulnerable processors. As a result, an attacker may be able to access privileged kernel pages, which are kept in the TLB through the use of the G bit. The mitigation, kernel page table isolation (PTI) [9, 16], turns the G bit off for kernel data pages and introduces the requirement of flushing the TLB when exiting the kernel to ensure that kernel data is inaccessible, even speculatively, to the user. This requirement increased the number of TLB flushes that must occur and also reduced the utility of the TLB by increasing TLB misses. In CPUs that supported PCIDs, these TLB flushes could be eliminated by associating each process with two address spaces instead of one—a user address space, which only holds mappings of user accessible pages, and a kernel address space that holds both the mappings of the user and the kernel. Consequently, on CPUs that suffer from the Meltdown vulnerability, every TLB flush needs to be performed twice, once on the kernel address space and once on the user address space.

The operating system may perform a TLB flush for a number of reasons, for example: memory deduplication [33], reclamation, huge page compaction [12] and NUMA node memory migration [5]. Applications may trigger a flush by calling system calls which modify PTEs such as mprotect, mmap, munmap and msync, as well as writing to copy-on-write (CoW) pages.

2.2 TLB Shootdowns

When the OS needs to update or remove a PTE in another core's TLB, the processor must do more than simply flush the TLB (a local operation) since x86 hardware does not provide TLB coherence. Instead, the OS performs a TLB shootdown, which allows a core (known as *the initiator*) to flush PTEs from the TLB of other cores (known as *the remote cores*). In order to perform the shootdown, the initiator sends an IPI with *work*, a data structure which indicates which entries need to be flushed, to remote cores, which perform flushes locally and send an acknowledgement back to the initiator by clearing a bit that the initiator spin-waits on. IPIs can be sent to a single target CPU or multiple ones using a multicast IPI. On Intel CPUs, when the number of cores is greater than 8, the CPUs are broken into clusters of up to 16 CPUs and each multicast IPI can only target a subset of one of the clusters [18, 19].

While sending an IPI and invoking the interrupt handler can take considerable time, software overheads are not negligible either. When the kernel modifies a PTE, it needs to determine on which CPUs to initiate TLB flushes. For each address space the kernel tracks in which CPUs an address space is currently active. Flushing PTEs from an address space that is inactive on a remote core does not require an IPI. Instead, the kernel tracks the "generation" of each address space, which it increments whenever PTEs are modified. Before the kernel loads an address space, whose stale mappings might be cached in the core's TLB, it uses the address space generation to determine whether PTEs have changed while the address space was inactive. If so, the kernel flushes the address space.

Overall, the current TLB shootdown protocol in Linux is quite expensive, costing thousands of cycles compared to the ≈200 cycles for a local INVLPG instruction [7, 17]. The real cost is often higher because a shootdown may cause a full flush, resulting in increased TLB misses in subsequent execution. The kernel also typically holds locks during flush, increasing contention [10, 13], and if the remote cores have interrupts disabled, for example, while in device driver code [11], the latency to handle and acknowledge the IPI may be even higher.

2.3 Related Work

Since TLB shootdowns have a high cost, work from the Linux kernel community, academia, and industry have sought to reduce their impact. Some of this work includes hardware changes, while others focus on changes in software only.

2.3.1 Hardware Changes

New hardware features have been proposed to reduce the cost of TLB shootdowns or eliminate the need for the OS to initiate them altogether. This includes implementing TLB coherence itself by adding new [29] or augmenting existing cache coherence protocols [34], adding remote cache invalidation instructions [3], adding directories so caches can be invalidated selectively [32], adding microcode support for hardware IPI handling [25], adding bloom filters to reduce invalidations [26], or adding time-based invalidation to TLB entries so that shootdowns can be avoided [4].

While new hardware can greatly improve the performance of shootdowns, they come at the cost of additional hardware complexity, and the OS may also have to be modified to take advantage of the changes. In addition, new hardware cannot improve existing processors.

2.3.2 Software Changes

Software based approaches primarily focus on avoiding shootdowns in order to avoid paying the penalty of the IPIs that initiate a shootdown. ABIS [2] introduces several techniques to Linux which exploit page access tracking to reduce the number of unnecessary TLB flushes and shootdowns. Barrelfish [7] attempts to avoid IPIs by using message passing instead of IPIs. LATR [21] avoids IPIs and handles shootdowns lazily, deferring the shootdown to be performed asynchronously through a message passing mechanism. RadixVM [10] attempts to avoid shootdowns by tracking page mappings through a cache-optimized radix tree.

Notably, some of these works were evaluated without multicast IPIs [10, 21], under the assumption that IPIs have to be issued synchronously, and may not yield the same, or any, improvement with now prevalent APICs which support multicast IPIs. For instance, RadixVM [10] is evaluated on a system where shootdowns take ≈500,000 cycles, whereas a shootdown in Linux with a x2APIC in cluster mode takes on the order of several thousand cycles.

In addition, many of these systems change the interface for the programmer. Barrelfish does not provide full POSIX compatibility. RadixVM runs on a research kernel developed on xv6 [14] and cannot run standard POSIX applications. Lazy flushing in LATR changes the semantics of POSIX memory operations by not freeing pages immediately, causing applications which expect page faults when accessing unmapped pages (used commonly by userfaultfd) to fail. Furthermore, lazy flushing comes with a cost: if a flush is typically done with a lock, the bookkeeping required to defer the complete context of the flush, including the locks that need to be reacquired and state that must be recalculated, is non-trivial. As a result, lazy flushing can be error-prone when combined with code which take locks, resulting in correctness or potentially even safety issues[1]. Finally, a practical implementation of ABIS would require hardware page sharing tracking.

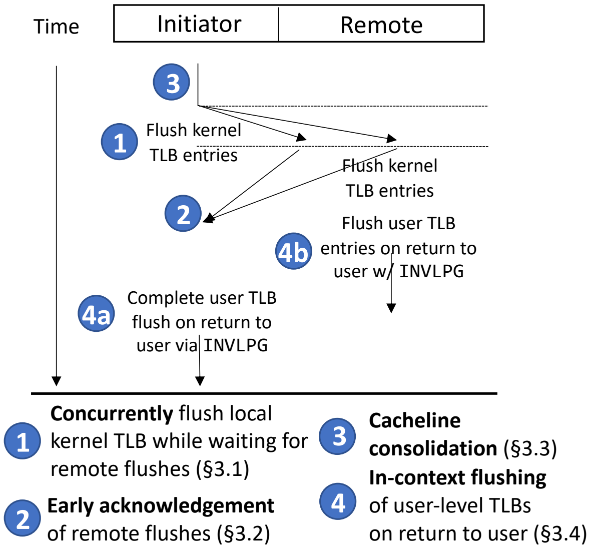

Time
Initiator
Remote
3
1
Flush kernel TLB entries
Flush kernel TLB entries
2
Flush user TLB entries on return to user w/ INVLPG
4b
4a
Complete user TLB flush on return to user via INVLPG
1
Concurrently flush local kernel TLB while waiting for remote flushes (§3.1)
2
Early acknowledgement of remote flushes (§3.2)
3
Cacheline consolidation (§3.3)
4
In-context flushing of user-level TLBs on return to user (§3.4)

Figure 3. We identify four opportunities to improve the current TLB shootdown protocol.

2.4 This Work

In this work, instead of avoiding shootdowns, we take a principled bottom-up approach to dealing with TLB synchronization. The goals of our system are to first improve the performance of a single shootdown, then to reduce the number of overall shootdowns by examining the remaining bottlenecks. By speeding up the existing shootdown mechanism and reducing overuse, we are able to show significant performance improvement without resorting to interface changes which lead to increased complexity and burden on developers. Our work focuses exclusively on software changes, but takes advantage of hardware changes such as the PCID feature available on newer x86 processors. The following section describes the architecture of our system.

3 Improving TLB Shootdown

Our approach begins by modifying the baseline Linux TLB shootdown protocol (Figure 1). We find that there are several

[1] One such issue can be found in LATR's implementation where NUMA migration should take place. mmap_sem should be taken in task_numa_work in order to flush. Because mmap_sem is not taken, the VMA may be invalid by the time change_prot_numa() is called again.

opportunities for improving the protocol, which include *concurrent flushing*, *early acknowledgement*, *cacheline consolidation* and *in-context TLB flushes*. We discuss each optimization in the following sections, and Figure 3 presents the final protocol combining all techniques.

3.1 Concurrent Flushes

Within a TLB shootdown, there are no constraints on the ordering of local and remote flushes. However, both Linux and FreeBSD sequentially flush local then remote TLBs, which means the initiating core is spinning for an extended time while waiting for remote acknowledgements. Instead, we observe that the initiating core can use this waiting period to flush its local TLB. Flushing the TLB entry of a single PTE can take over 100ns. In Linux, up to 33 entries can be flushed during a single TLB shootdown operation, which means a local TLB flush can take over 3μs [17]. If the initiating core does this local flush while waiting for remote TLBs, we directly eliminate this latency from the critical path.

We therefore modify the shootdown algorithm so that the initiator flushes its local TLB while waiting for IPI acknowledgement from remote cores (see Figure 3).

3.2 Early Acknowledgement of Remote Shootdowns

Currently, responder cores do not communicate shootdown success to the initiator until they have finished local invalidation. Performing TLB shootdowns in a fully asynchronous manner—sending the IPI and continuing execution immediately after—is unsafe. There is no guarantee that the responding core would receive the IPI immediately, as interrupts might be masked on the responding core. There are also no architectural guarantees on how long the IPI delivery would take. In the meanwhile, the sending core might assume that the flush was completed, which can lead to data corruption or security issues.

Instead, we propose that responder cores acknowledge success as soon as it is *safe*. In other words, responders can send acknowledgements as soon as they can ensure that no TLB entries in the shootdown are accessible. In particular, once a responding core enters the TLB shootdown interrupt handler, it does not use any userspace mappings in the page-tables. Therefore, it is safe for the remote core to send acknowledgement to the initiator as soon as it enters the interrupt handler, eliminating TLB invalidation on the remote core from the critical path.

There are two exceptions. First, if page-tables are released, speculative page-walks can cause machine-check exceptions, as noted in Section 2. This optimization therefore cannot be used if page-tables are released. Page-tables are mainly released during the munmap system call. In Linux, there is already a flag in the *work* data structure that indicates whether page tables are released, so the initiator decides whether to use early acknowledgment based on this flag and instructs the responders accordingly.

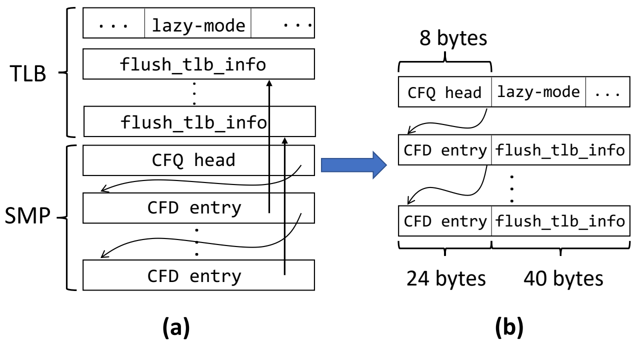


Figure 4. (a) Linux separates all cachelines required during a TLB shootdown based on whether they contain TLB or SMP information. (b) We observe that we can cleanly inline these in-memory variables in order to reduce cache contention.

Second, if another higher priority interrupt is delivered after the TLB shootdown was acknowledged but before flushing took place, the interrupt handler might access userspace memory using the inconsistent TLB. In Linux, only the non-maskable interrupt (NMI) can be delivered during TLB shootdown. A similar issue might occur if a probe that was set by the user (kprobe in Linux) accesses the userspace memory. Both the NMI handler and the probing mechanisms in Linux already incorporate code to ensure, before accessing userspace memory, that the userspace mappings are those of the running process. This is already required to avoid inconsistencies that might occur when an NMI is delivered in the middle of a context switch (see nmi_uaccess_okay()) We extend this check to ensure all TLB flushes have been completed. Because the NMI handler is already expensive, this check adds minimal overhead.

Note that early acknowledgement complements concurrent flushes as described in Section 3.1. IPI delivery often takes more time (potentially over 1000 cycles) than TLB flushing (~ 200 cycles per entry), so when concurrent flushes are used with early acknowledged remote shootdowns, the initiator's local TLB flush still overlaps (roundtrip) IPI delivery.

3.3 Cacheline Consolidation

The Linux TLB shootdown protocol generates unnecessary cache contention due to the way the TLB layer is abstracted from the SMP (symmetric multi-processing) layer, which maintains per-core queues of function and data to be invoked, sends IPIs, and waits for acknowledgments.

We do not consider the FreeBSD TLB shootdown scheme, because FreeBSD's smp_ipi_mtx mutex only allows a single TLB shootdown to be delivered and served at a time. In contrast, Linux's shootdown protocol allows for concurrent remote shootdowns.

We highlight four types of cacheline access that are highly contended during a TLB shootdown (see Figure 4(a)).

1. *Lazy mode indication.* Before sending a shootdown, the initiator checks whether a remote core is in "lazy-mode", which means that although it uses the affected address-space, it is currently running a kernel thread. If so, an IPI does not have to be sent to the target core, as the core would check whether a TLB flush is needed to synchronize the TLB before resuming the user thread [28]. Cores do not enter lazy-mode frequently, but the cacheline that holds the lazy-mode indication is shared with other frequently changing data holding per-core TLB information, which causes false sharing.
2. *TLB flushing information.* This data structure stores which TLB entries should be flushed and, prior to our work, was kept on the stack of the initiator core. Accessing this information can potentially cause TLB misses which would not occur if the data resided in a global variable, as the stack is mapped using 4KB pages and the global variables use 2MB pages, which are likely to hold other frequently accessed data.
3. *Call Function Data (CFD).* Used by the SMP layer to provide the called function and data, as well as to acknowledge that the function execution completed.
4. *Call Single Queue (CSQ).* A per-core lockless list of CFD that are pending execution on a core.

We reduce the contention caused by these accesses by inlining related information. We colocate the lazy-mode indication with the head of the CFQ as they are likely to be accessed one after the other. Next, Linux stores each TLB flush info as a pointer in each CFD, but we observe that we can inline this info and still fit the CFD within a single cacheline. Figure 4(b) shows how sharing cachelines across the TLB and SMP abstractions greatly reduces the number of contended cachelines during TLB shootdown.

3.4 In-Context Page Flushes

When PTI is turned on, TLB flushes need to be performed on two address spaces: the kernel's, which is the active one, and the user's. In Linux, if a full TLB flush is needed, for example since many PTEs are changed, the kernel address space is flushed immediately, and a deferred flush indication is set for the user address space. Once the OS returns to user-mode, if this indication is set, the kernel flushes the user page tables while it reloads the user address space. The CPU loads the page tables and performs the TLB flush of the user address space atomically.

However, when PTEs need to be flushed selectively, the kernel invalidates the PTEs of both kernel and user address spaces eagerly. To do so, the kernel uses the instruction provided by Intel CPUs to selectively flush PTEs: `INVLPG`, which invalidates a certain PTE in the current address space, and `INVPCID`, which invalidates a PTE in any address space (note that it can also perform additional types of TLB flushes, which are outside of the scope of our discussion). Previously performed measurements indicated that `INVPCID` is slower at flushing a single PTE than `INVLPG`, as tested on the Skylake microarchitecture [23].

Accordingly, the kernel flushes the PTEs in its address space, which is active, using the `INVLPG` instruction, but flushes the user PTEs, which are in an inactive address space and out of context, using `INVPCID` since the kernel PCID, which is active, is different than the user PCID. This introduces significant overhead due to `INVPCID`'s suboptimal performance.

Using `INVLPG` to eagerly flush the TLB, however, can induce overheads instead of improving performance, as it would require loading the user address space, performing the required TLB flushes, and then reloading the kernel address space to resume execution. The overhead of switching address spaces can exceed the benefit of using `INVLPG`, especially if only few PTEs need to be flushed.

Instead, we decide to defer the flushing of user PTEs until the user address space becomes the active address space. Then we can flush PTEs using the more efficient `INVLPG` instruction. As this address space is not used on the given core until the kernel returns to userspace, this deferring does not introduce correctness issues. This scheme is shown as (4b) in Figure 3, which we refer to as an *in-context page flush*.

To enable in-context page flushes, the kernel records the user address-space flush range (start and end) as well as the stride shift (i.e., page size) in a per-core data structure. If multiple flushes are required before returning to userspace, the kernel tries to merge all pending flushes into a single range. If the resulting range size exceeds a fixed threshold (we use Linux's default of 33 entries), a full flush is performed upon return to userspace.

There are a few caveats. First, flushing TLB entries immediately before returning to userspace requires a stack in order to preserve the userspace registers that have just been reloaded. In rare cases, such as returning to 32-bit compatibility code with `IRET` instead of `SYSRET`, no such stack is available. In this case, we perform a full TLB flush. Furthermore, we do not defer TLB flushes that remove entire page tables from the page table hierarchy, since in this case we must flush the TLB before switching to the userspace address space and context.

Second, since there is no serializing instruction after the userspace address space is loaded and before the actual return to userspace, there is the potential for some TLB flushes to be skipped speculatively. This might occur if the conditional branch that loops through the TLB flushes is mispredicted and introduces a security vulnerability. Without addressing this issue, a malicious user might exploit the Spectre-v1 CPU vulnerability, to speculatively skip flushes and leak data from the pages whose PTEs should be flushed. This is problematic, since at the time of the deferred flushing the pages that are mapped through these PTEs might already be recycled and used for another purpose. To prevent such an attack, we use the `lfence` instruction when the loop that invalidates the user page table PTEs is done.

In-context page flushing creates a subtle interaction with concurrent flushing (§ 3.1) and early acknowledgement (§ 3.2) on the initiator core, see (4a) in Figure 3. With concurrent flushing, the initiator has spare cycles while it waits for acknowledgment from the remote cores (this is true even with early acknowledgement). These spare cycles can be used to flush user-space PTEs, instead of deferring them. Therefore, we keep flushing user PTEs until the first remote acknowledgment is received, whereby we defer the rest until the switch to userspace.

4 Use-case Specific Improvements

The previous section presented techniques to reduce the overhead of any TLB shootdown. In this section we introduce techniques for avoiding TLB flush for copy-on-write mappings and reducing the frequency of TLB shootdowns in some cases. Avoiding TLB flush reduces the amount of time it takes the initiator to complete its local flush, thereby improving the overall latency of the shootdown. Reducing the frequency of shootdowns naturally reduces the amount of time the kernel spends in this costly operation.

4.1 Avoiding TLB flush for CoW

Copy-on-Write (CoW) is a common memory sharing technique in which memory pages are write-protected and copied only when they are modified. The first modification invokes the page-fault handler, which copies the accessed page, updates the PTE to point to the new copy, and sets write permissions on the PTE.

Changing the PTE requires a TLB flush, as the PTE now points to a new target. However, while a TLB shootdown is necessary if other threads use the same mapping, we argue that a local TLB flush can be avoided by writing to an address in the modified page after the PTE is updated. On CoW events, since the modified PTE was previously write-protected, the CPU cannot use the old PTE for translation and instead should walk the page-tables and cache the new PTE. This avoids the overhead of a PTE flush and its side-effects (invalidation of the page-walk caches on x86), and caches the updated PTE, which is about to be used.

This write access might not appear necessary, as the Intel manual explicitly states a faulting PTE is invalidated during a page-fault [19]. However, in practice the stale PTE might be cached in the CPU for two reasons. First, the CPU is free to cache PTEs speculatively after the page fault is triggered and before the PTE is updated. Second, the page-fault handler might be preempted and later be scheduled to run on a different core than the one that triggered the page-fault. Hence the explicit kernel memory access ensures that the stale PTE is removed.

Another challenge comes from the fact that the explicit memory write access does not affect mappings that are potentially cached in the instruction TLB (ITLB). Therefore, we avoid using this optimization if the PTE is executable to prevent the unlikely case in which the PTE is cached in the instruction TLB.

Finally, to avoid any potential races, we do not want the memory write to corrupt data if it is concurrently written from another core. We therefore perform an atomic operation that does not modify the data at the faulting address.

4.2 Userspace-safe Batching

As noted in Section 2, TLB batching, such as the lazy flushing proposed by LATR, can cause correctness issues if done too aggressively. However, if the kernel can guarantee that TLB flushes complete before userspace mappings are accessed, batching can be safely done. To achieve this guarantee, there would need to be a memory barrier to check for TLB flushes every time the kernel prepares to leave kernel mode *and* when it needs to access userspace data, for example during the `read` system call.

Hence, we only implement TLB batching for suitable system calls such as `msync`, `munmap`, `madvise(MADV_DONTNEED)`, which require write-protecting and cleaning PTEs that map dirty (i.e., recently-written) pages of memory mapped files. These system calls are ideal candidates for batching because the memory barrier can be piggy-backed on the release of the `mmap` semaphore (`mm->mmap_sem`), and there are no accesses to userspace during the system call.

We implement batching by adding a new `batched_mode` variable that indicates whether to batch a TLB flush. We also allocate 4 entries to keep track of the deferred flushes. For this we use the existing data structure `flush_tlb_info`.

5 Evaluation

We implement our changes on Linux 5.2.8 and use it as the baseline in our measurements. Table 2 summarizes the number of lines of code required for each change. We conduct our experiments on a Dell R630 server with 2 Intel Skylake Xeon E5-2660v4 CPUs, each having 14 physical cores and 28 logical cores (SMT threads) and 256GB of memory. We use the maximum performance governor during our evaluation to reduce jitter and run each test 5 times.

Every benchmark is also run in two setups, "safe", which is Linux's default mode, and "unsafe", where we disable kernel mitigations against recent security vulnerabilities such as Spectre and Meltdown. This "unsafe" setup helps determine the effectiveness of our proposed techniques in existing and future architectures that may no longer require these patches. The most relevant mitigation that causes a performance difference between the two setups is page-table isolation (PTI), which requires in the "safe" setup to perform each TLB flush

Optimization	Lines of code
Concurrent flushes	103
Early ack + Cacheline consolidation	73
In-context page flushing (deferring)	353
CoW	35
Userspace-safe Batching	221

Table 2. Number of lines of code required to implement each optimization.

	Safe Mode	Unsafe Mode
1 PTE	39% / 13%	39% / 18%
10 PTEs	58% / 22%	54% / 14%

Table 3. [Initiator / Responder] Overall latency reduction when initiator and responder are on different sockets, after applying all four techniques in Section 3.

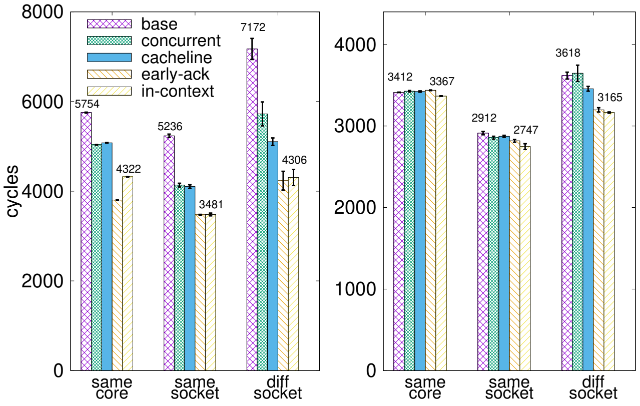
twice: once for the kernel page tables and once for the user page tables.

5.1 Microbenchmarks

We measure the performance impact of each of the techniques presented in Section 3 with a microbenchmark that uses mmap to create an anonymous mapping, touches a number of pages to trigger their allocation, and then runs the madvise(DONTNEED) system call, telling the OS these pages are not needed, thereby causing them to be reclaimed and the page-table mappings to be removed. This page-table update requires a TLB flush. The benchmark also spawns an additional thread that runs a busy-wait loop during the test. This thread acts as a "responder" thread during the TLB shootdown.

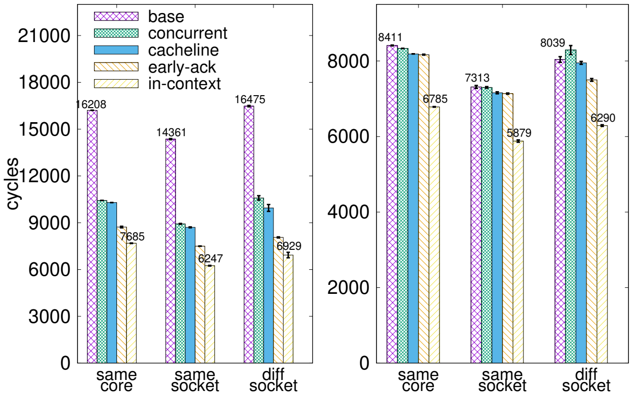
We separately report the number of cycles for the shootdown on both the initiator and responder threads. On the initiator thread, we report the number of cycles that the madvise system call took. On the responder thread, we report the number of cycles that the thread was interrupted due to handling the shootdown. Each test runs 100k madvise system-calls. We run the test 5 times, and report the average and standard deviation of these executions.

Each experiment is also run on three different core configurations: the initiator and responder are either on the same core, the same socket, or different (NUMA) sockets. We also present results for when 1 PTE and 10 PTEs are flushed during the shootdown. Figures 5 - 8 present the results for all experiments. In each figure, we report the latencies as we iteratively activate the optimizations, in the order in which they appear in each figure's legend. Finally, in unsafe mode there is no PTI, so for those experiments we do not show the in-context flush optimization.



(a) Initiator (b) Responder

Figure 5. Safe mode, flush 1 PTE.



(a) Initiator (b) Responder

Figure 6. Safe mode, flush 10 PTEs.

Overall speedup. For the initiator core, we observe that concurrent flushes (§ 3.1) and early acknowledgement (§ 3.2) consistently have the largest impact, reducing shootdown latency by 10%-20% in both safe and unsafe mode.

For the responder cores, concurrent flushes and early-return have little effect, because these two techniques enable the *initiator* core to reduce the end-to-end latency of a shootdown. Instead, deferred PTI flushing (§ 3.4) benefits responder cores, and this effect is more clearly highlighted when there are more PTE entries to flush, as seen in Figure 6. Cacheline consolidation also helps reduce latency in responder threads by up to 5% and 10% in safe and unsafe mode, respectively.

Table 3 summarizes the overall latency reduction for a shootdown after employing all four techniques.

Concurrent flushes. Performing flushes concurrently provides the greatest benefit for the initiator. The speedup is proportional to the number of entries that need to be flushed and hence greater when multiple PTEs are flushed (Figures 6(a),8(a)). If only a single PTE is flushed, the benefit of concurrent flushes is lower (Figures 5(a),7(a)). In particular, when the initiator and responder are on the same core, concurrent flushes in safe mode provide around 38% latency reduction on the initiator when 10 PTEs are flushed, and only around 10% latency reduction when 1 PTE is flushed.

Furthermore, if the responder resides on a different NUMA node than the initiator, sending the TLB shootdown request before performing the local TLB flush hides the responder code invocation latency, which is greater because the IPI and flush information must traverse the interconnect. Hence even when flushing 1 PTE, concurrent flushing results in 20% latency reduction on the initiator (Figure 7(a)).

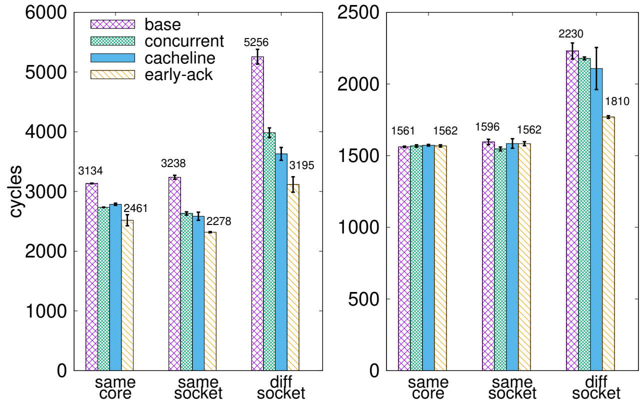
The benefit of concurrent flushes is greater in safe mode than in unsafe mode for two related reasons. First, due to PTI, each PTE needs to be flushed twice, which allows concurrent flushes to eliminate greater overhead. Second, the security mitigation techniques against hardware vulnerabilities, specifically PTI, cause the latency of entering the kernel from userspace to be higher. Concurrent flushes hide part of this latency.

Cacheline Consolidation. While consolidating cachelines improves the performance of both the initiator and the responder, its benefit is relatively small. Reducing the number of cachelines that need to traverse between the initiator and the responder has the greatest impact when the initiator and responder reside on different NUMA nodes. In this case, the cache-lines cross the interconnect, inducing higher overheads. In particular, in Figures 5 and 7, cacheline consolidation has a negligible improvement when the initiator and responder are on the same core. However, this improvement jumps to 5-11% latency reduction for either thread in both safe and unsafe mode when the threads are on different sockets. These improvements become greater in Figures 6 and 8, because more PTEs are flushed.

Early Acknowledgment. Figures 5- 8 show that the speedup from early acknowledgment is greater when the initiator and the responder are further away. Although concurrent flushes can hide some of the flush latency, the actual TLB flush on each core will take time, which means the initiator will finish its flushes some time before the responder.

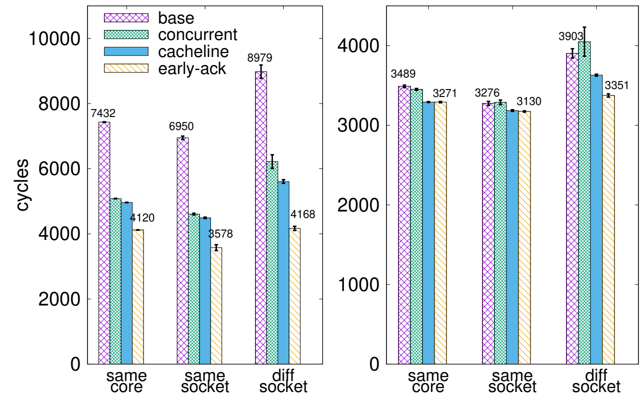
Therefore, without early acknowledgment, after the initiator invokes the responder flush, it must wait for both TLB flushes and for the completion indication. With early acknowledgment this time is effectively shortened to the time it takes to flush the local TLB and wait for the completion indication, because the remote TLB flush is removed from the critical path.

Accordingly, the overhead savings from early acknowledgment are greatest when both TLB flushes take a long time — when multiple PTEs are flushed or when PTI is on (safe mode) — and when the responder resides on a different socket (Figure 8a).



(a) Initiator **(b) Responder**

Figure 7. Unsafe mode, flush 1 PTE.



(a) Initiator **(b) Responder**

Figure 8. Unsafe mode, flush 10 PTEs.

While we designed this mechanism purely to improve the initiator performance, we were surprised to see that early acknowledgment also improves, although to a lesser extent, responder performance, specifically when the initiator and responder are on different sockets (Figure 7b). Based on the code and the MESI protocol, this should not be due to fewer cache lines that are passed between the caches. Nevertheless, the fact that speedup is not proportional to the number of flushed PTEs hints that this optimization indirectly improves intercore communication.

In-context flushing. In the safe setup, flushing PTEs in the user page-tables can have a big impact on performance: when 10 PTEs are flushed, in-context flushing reduces the initiator and responder execution time by ~ 1100 cycles in all the configurations, as shown in Figure 6.

Note that in-context flushing not only improves performance, but also safety, as INVPCID, in individual-address invalidation mode, does not flush entries in the page-walk cache that do not belong to the address being invalidated. On the other hand, INVLPG flushes the entire page-structure cache. In practice, this does not appear to pose a problem in Linux, but might in operating systems that rely on the selective TLB flush to flush the page-walk cache as well.

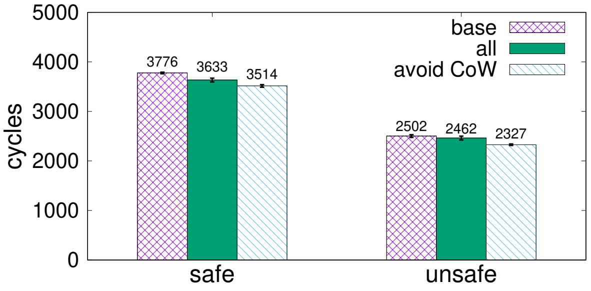


Figure 9. Impact of avoiding TLB flush during copy-on-write page-fault. We first measure the impact of the previous optimizations (all), and then measure the addition of the CoW technique.

Avoiding CoW Flush Recall that we present a technique for avoiding a local TLB flush while handling a CoW page-fault (§ 4.1). Instead, we explicitly remove the PTE by making a kernel memory access. This optimization applies exclusively to the local (initiator) core and does not affect responder cores, so we only present latency measurements for the local core.

We create a micro-benchmark that causes a copy-on-write event as it writes to a private memory-mapped file. We measure the visible time in cycles that the memory access, including the page-fault has taken. The results are shown in Figure 9 for both safe and unsafe mode. Note that the effect of the previous optimizations (all) is small, because they are mostly intended for TLB shootdowns rather than for a local TLB flush. Avoiding the TLB flush on copy-on-write by accessing the page further reduces the event time by about 130 cycles in both modes, about 3% and 5% respectively.

5.2 Sysbench

We run Sysbench [20], a multi-threaded benchmark tool for database systems. We use the benchmark to measure the overhead of random write to a memory mapped files. The benchmark periodically calls the fsyncdata system call to ensure the data persists in the file system. These operations trigger a TLB flush on a single thread, and TLB shootdown when multiple threads are used.

We use emulated "persistent" memory, i.e., DRAM which is reported as persistent memory, as the backing storage in which the tested filesystem is set, which allows us to benchmark the system behavior with high speed storage. We use a 3GB file and leave other parameters in their default value. We allow the OS to schedule the threads, i.e., we do not set the affinity of each thread to a certain core. However, to reduce the variance of the results we set all the threads to run on CPUs that are associated with a certain NUMA node.

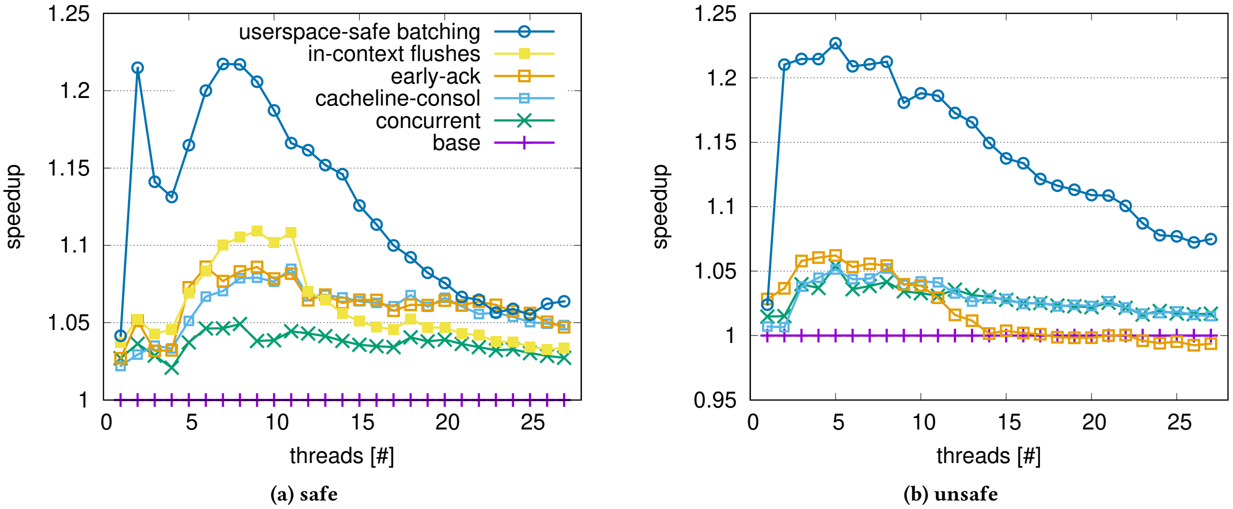
The results are shown in Figure 10, for safe and unsafe mode. Note that when the number of threads is smaller than 12, each optimization provides an added performance gain in safe setup, with up to 1.22× speedup.

However, when the number of threads increases above 10, "in-context flushing" in the safe configuration and "early acknowledgement" in unsafe mode, degrade performance. Profiling shows that as the number of threads increase above 10, the experienced TLB flush storm causes the existing TLB generation tracking logic to frequently detect that there are additional pending TLB flushes. In such cases, Linux's existing TLB flushing logic performs a full TLB flush and updates its internal accounting data structures to indicate that the outstanding TLB flushes can be later skipped.

This affects both optimizations. If a TLB flushing request can be skipped, as it was already fulfilled ahead of time, the TLB flushing function completes very fast and therefore there is no benefit in "early response". Similarly, in the safe setup, when a full TLB flush is needed, the baseline system already flushes the user page-tables efficiently when it switches back to the user page-tables. The optimization of "in-context flushing" addresses partial TLB flushes, which are performed infrequently during such TLB flush storms.

The greatest benefit is provided by "userspace-safe batching". In this benchmark each core spends the majority of its time within the fdatasync syscall, which is why this optimization can provide up to 1.18× performance gain. However, the benefit of "userspace-safe batching" diminishes with more threads. This is due to an inherent limitation of this optimization: it provides its greatest benefit when all the threads that use the address space are running a system call, as it avoids sending and waiting on the delivery of an IPI. In safe mode more time is spent in the trampoline code that performs entry to the kernel and in the IPI handlers, which causes the benefit of this optimization to be less than in the unsafe setup.

An interesting phenomenon occurs in the safe setup, when the number of threads is between 3 and 5. As shown, the benefit of "concurrent TLB flushes" and "userspace-safe batching" is lower in these cases. Our analysis indicates that in these cases, the optimizations cause a higher portion of the TLB shootdown IPIs to be delivered while user-code is running. It appears that the slowdown is caused since in the safe setup dispatching the interrupt handler while userspace code runs is considerably slower than dispatching it while kernel code runs. It is not entirely clear why more IPIs are delivered while userspace code runs, and whether this also occurs in other workloads.



(a) safe

(b) unsafe

Figure 10. Sysbench random write benchmark using memory mapped memory, and issuing periodic synchronizations (fdatasync). The file is set on emulated persistent memory. We present the speedup of the benchmark as we sequentially add each optimization, starting with concurrent flushes.

We also observe that "early acknowledgement", "userspace-safe batching", and "in-context flushing" in the safe scenario gradually provide lower speedups as the number of threads increases. This is due to the fact that as the number of threads increases , the portion of time that the CPU spends performing tasks that we did not optimize, such as determining the cores to which the shootdown IPI should be sent, is greater.

5.3 Apache Webserver

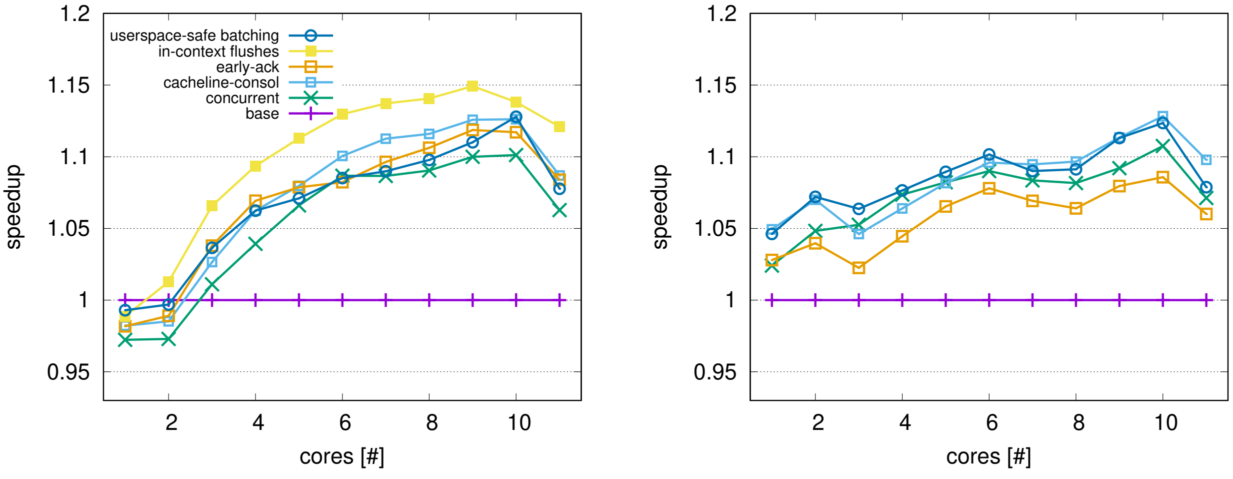
The Apache webserver is known to cause a significant number of TLB flushes when it is configured to use the `mpm_event` module. This module uses threads to serve web requests, and although this scheme saves memory, it triggers a significant number of TLB shootdowns, as Apache creates and tears down memory mappings of served files upon each request.

As a workload generator we choose `wrk`, which is multithreaded, unlike `ApacheBench`, and stresses the server more than other generators such as `Siege`. Configuring `wrk` must be done carefully, as the workload generator attempts to attain a given request rate per second. Setting the rate or the concurrency parameters to values that are too high results in connection errors when the number of threads that Apache uses is small, thereby distorting the results. We carefully choose a rate of 150k requests per second, 10 threads and concurrency level of 10. We change the number of cores that the server uses through `taskset`, and report the speedup on each point. As the benchmark performance plateaus after it uses 11 cores (at roughly 110k requests/second), we report the results up to this point.

When Apache webserver was executed with the default configuration, its results were very noisy. We therefore disabled address space layer randomization (ASLR), kernel ASLR (KASLR) and some kernel daemons (e.g, KSM). We also disabled the server access log. However, even with these modifications, we saw that the benchmark results varied whenever we restarted the Apache server, which was required whenever we booted a different kernel. This noise accounts for the performance dip when ≤ 2 threads are used in safe mode (Figure 11(a)). Nevertheless, the overall results have reasonable standard deviation (<3%) and the relative trend lines in Figure 11 are consistent, suggesting the results are merely skewed slightly down.

The results of this benchmark indicate that for this workload significant performance gains come primarily from two of the optimizations: concurrent flushes, which provide a speedup of up to 1.1x and in-context flushing, which provides a speed of up to 1.05x. Unlike the sysbench benchmark, this workload does not cause TLB flush storms that cause full TLB flushes to be performed instead of selective ones. As a result, the speedup from in-context flushing is higher.

The benefit of other optimizations is relatively small, which is reasonable given the workload. Recall from Section 5.1 that the cacheline consolidation optimization provides the greatest speedup when multiple sockets are used, and the early-ack optimization is most useful when inter-core communication and TLB flush time are both high. Since the workload is run on a single socket and the served webpages are smaller than 12KB (3 memory pages), these optimizations provide limited value.



(a) safe

(b) unsafe

Figure 11. Apache webserver using multithreaded modules and serving concurrent requests. We present the speedup in respect to the number of handled requests. We sequentially add each optimization, starting with concurrent flushes.

As for "userspace-safe batching", although we modified the `munmap` syscall to defer TLB flushes and indicate that other cores not send IPIs initiating TLB flushes during the system call, this optimization does not seem beneficial for this workload. Again, this is reasonable considering the fact that the Linux kernel already batches the flushes during `munmap`, and the workload spends time in the kernel executing other system calls (`mmap`, `send`) and handling page-faults. Hence further work is still needed to make this "userspace-safe batching" approach beneficial for general workloads while preventing it from introducing performance regressions.

It should be noted that the reduction in speedup that takes place when 11 cores are used is due to the fact that the benchmark performance saturates at this point, as we use 10 threads for our workload generator. This reduction is not related to the previously mentioned limitations of our solutions.

6 Discussion

Our evaluation shows that the significant performance gains from our optimizations reveal that current software does not extract maximum performance when managing TLBs. We are able to show performance improvements on unmodified workloads without adding additional interfaces or relaxing the semantic guarantees provided by the kernel.

Given that TLB shootdowns are a relatively mature part of modern operating systems, we believe that inefficiencies in how TLB shootdowns are dispatched and handled arise from a lack of principled software and hardware co-design. From the perspective of the operating system developer, the tools for manipulating TLB state are often a black box, where details about how the processor operates are hidden away under the guise of 'microarchitectural details'. For example, processor specifications often state that more TLB entries can be flushed than requested, even if this is never observed in practice. This leads to OS developers designing mechanisms which are much more conservative than they need to be. Meltdown [1] and Spectre [24] have both shown that microarchitectural details can and should matter to developers, as they are key to understanding and maximizing hardware performance. If hardware vendors provided less conservative guidance about the behavior of existing primitives, software developers could make better decisions on which primitive to use and how.

At the same time, during the development of our optimizations, we often found ourselves wishing that the hardware could provide basic primitives that would greatly enhance the performance of the TLB shootdown protocol. For instance, if it were possible to attach a message with a TLB shootdown, which sends a multicast message to other processors in hardware, we would have been able to avoid sending additional data through shared memory, which results in unnecessary coherence traffic. To truly maximize the performance of shootdowns, which generate significant overhead, hardware and software must be co-designed so that the OS has the right tools needed to manipulate the TLB efficiently. As many hardware features have migrated to microcode [6], many changes to the hardware can actually be implemented as software itself.

7 Future Work

We have submitted our optimizations as patches to Linux and have received positive feedback from the Linux community.

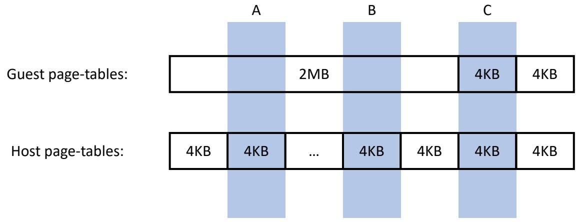


Figure 12. Page fracturing: the TLB can cache any of the guest-to-host mappings for a guest hugepage. For example, A and B can be separately cached in the TLB. This behavior causes both "page splintering" and an unnecessary number of TLB flushes on Intel CPUs. For example, even if C is flushed, the processor will initiate a full TLB flush to deal with possible page fracturing elsewhere.

We hope to upstream our changes and to integrate them into mainline Linux soon. However, much work is left in optimizing how Linux manages TLBs and uncovering behaviors that mismanage TLB state.

For example, one surprising behavior we have observed on Intel CPUs is the result of "page fracturing". Under virtualization, the TLB caches translations from guest virtual addresses (GVAs) directly to host physical addresses (HPAs), which merges the translations from GVAs to guest physical addresses (GPAs) in the guest page tables, and the translations from GPAs to HPAs from the host page tables. Consequently, a guest hugepage (2MB) can be associated with many host 4KB page mappings, thereby "fracturing" the guest page, as shown in Figure 12. The TLB is free to cache any of the 4KB page mappings of the address translation, a behavior known as "page splintering" [27], which causes TLB pressure and might increase the number of TLB misses.

Page fracturing, however, causes an additional, previously undiscussed challenge when a selective TLB flush is needed. The guest is allowed to selectively flush a 2MB page (e.g., using the INVLPG instruction), but flushing such a page requires the CPU to flush *all* the 4KB translations for the other guest addresses in the same 2MB page. Intel confirmed that this is the expected behavior of the CPU. We speculate that Intel CPUs maintain a flag that marks whether *any* cached TLB translation is a result of a page walk through a 2MB guest mapping and a 4KB host mapping. If this flag is set, *any* selective TLB flush might cause a flush of the entire TLB to prevent the TLB from holding stale 4KB translations.

Our experiments confirm this behavior. Table 4 summarizes the number of dTLB misses as reported by performance counters after either a full or selective (single page) flush. Note that these page sizes are not related to the size of the flushed pages: the flushed page was not mapped in the page-tables so it could not have been cached in the TLB.

	Host pg size	Guest pg size	Full Flush	Selective Flush
VM	4KB	4KB	103M	93K
	4KB	2MB	102M	102M
	2MB	4KB	103M	2.9K
	2MB	2MB	4M	2.5K
Bare-Metal	4KB	-	5M	789
	2MB	-	1M	537

Table 4. Number of dTLB misses after a full or selective page flush. Notice that if the guest has hugepages (2MB) that are mapped to 4KB pages on the host, a selective flush will cause an unusually large number of TLB misses, due to the processor actually executing a full TLB flush.

2MB pages are commonly used by operating systems, for example to hold kernel code, so issuing multiple selective TLB flushes in a VM instead of a single full TLB flush is not beneficial, since the TLB would be fully flushed anyway due to the special bit.

To mitigate the full TLB flush that deals with page fracturing, the CPU instruction set can be extended to allow the VM OS to convey the size of the invalidated page and avoid unnecessary full TLB flushes when a 4KB page is flushed (for example, C in Figure 12). As an intermediate software solution, the host may also inform the VM OS, using a paravirtual protocol, whether page fracturing may happen. This would allow the VM OS to avoid issuing multiple selective TLB flushes, which are slower than a full TLB flush and does not provide any benefit, since they do not preserve other TLB entries.

Page fracturing is only one of many undocumented behaviors which demonstrate that the nuances of the TLB are still poorly understood despite their maturity. We hope to address page fracturing and many other undocumented behaviors which lead to inefficient TLB usage in Linux in future work.

8 Conclusion

In this paper, we revisit TLB shootdowns and identify ample room for improvement. We present four techniques that improve the performance of a TLB shootdown: concurrent TLB flushing, cacheline consolidation, early remote acknowledgement, and in-context PTI flushing, as well as identify special cases during CoW faults and msync handling where TLB flushes can be avoided or TLB shootdowns can be batched. We can reduce the latency of a TLB shootdown by up to 58% for initiators and 22% for responders. We show that combined, our optimizations result in up to a 1.25 × performance improvement on workloads such as Sysbench and Apache. In summary, we observe that contrary to popular dogma, kernels should embrace TLB shootdowns in order to manage TLB state and correctness.

9 Acknowledgements

We thank our shepherd, Yubin Xia, for his helpful comments on our paper and the anonymous reviewers for their insightful feedback. We also thank the Linux kernel community for technical discussions and continued support and transparency.

References

[1] CVE-2017-5754. Available from NVD, CVE-ID CVE-2017-5754, https://nvd.nist.gov/vuln/detail/CVE-2017-5754, January 1 2018. [Online; accessed 21-May-2019].

[2] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *USENIX Annual Technical Conference (ATC)*, pages 27–39, 2017.

[3] ARM Ltd. ARMv8-A architecture reference manual, 2013.

[4] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287. IEEE, 2017.

[5] Manu Awasthi, David W Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 319–330, 2010.

[6] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[8] David L Black, Richard F Rashid, David B Golub, Charles R Hill, and Robert V Baron. Translation lookaside buffer consistency: a software approach. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 113–122, 1989.

[9] Alexandre Chartre. mm/x86: Introduce kernel address space isolation. Linux Kernel Mailing List, https://lkml.org/lkml/2019/7/11/364.

[10] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 211–224, 2013.

[11] Jonathan Corbet. Realtime and interrupt latency. LWN.net, https://lwn.net/Articles/139784/, 2005.

[12] Jonathan Corbet. Memory compaction, 2010.

[13] Jonathan Corbet. Memory management locking. LWN.net, https://lwn.net/Articles/591978/, 2014.

[14] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[15] Mel Gorman. TLB flush multiple pages per IPI v4. Linux Kernel Mailing List, https://lkml.org/lkml/2015/4/25/125, 2015.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[17] Dave Hansen. x86 tlb flushing: Invpcid vs. deferred cr3 write. Linux Kernel Mailing List, https://lkml.org/lkml/2017/12/5/1082.

[18] Intel Corporation. Intel 64 Architecture x2APIC Specification, 2008.

[19] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Reference number: 325462-057US, 2015. https://software.intel.com/en-us/articles/intel-sdm.

[20] Alexey Kopytov. SysBench: a system performance benchmark. sysbench.sourceforge.net.

[21] Mohan Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselỳ, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. LATR: Lazy translation coherence. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2018.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[23] Andy Lutomirski. x86/mm: PCID and INVPCID. https://lwn.net/Articles/671299/, 2016.

[24] MITRE. CVE-2017-5715: branch target injection, spectre-v2. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715, 2018. [Online; accessed 21-May-2019].

[25] Mark Oskin and Gabriel H Loh. A software-managed approach to die-stacked dram. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 188–200. IEEE, 2015.

[26] Binh Pham, Derek Hower, Abhishek Bhattacharjee, and Trey Cain. Tlb shootdown mitigation for low-power many-core servers with l1 virtual caches. *IEEE Computer Architecture Letters*, 17(1):17–20, 2017.

[27] Binh Pham, Ján Veselỳ, Gabriel H Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *IEEE/ACM International Symposium on Microarchitecture*, 2015.

[28] Rick van Riel. x86/mm/tlb: make lazy tlb mode even lazier. Linux Kernel Mailing List, https://lkml.org/lkml/2017/12/5/1082.

[29] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[30] The Linux Kernel. Page table isolation (pti). https://www.kernel.org/doc/html/latest/x86/pti.html.

[31] Volkmar Uhlig. *Scalability of microkernel-based systems*. PhD thesis, TH Karlsruhe, 2005. https://os.itec.kit.edu/downloads/publ_2005_uhlig_scalability_phd-thesis.pdf.

[32] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349. IEEE, 2011.

[33] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[34] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. Hardware translation coherence for virtualized systems. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 430–443. ACM, 2017.