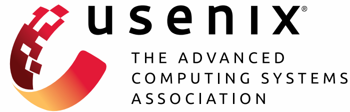


Optimizing the TLB Shootdown Algorithm with Page Access Tracking

Nadav Amit, *VMware Research*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).
July 12–14, 2017 • Santa Clara, CA, USA
ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

Optimizing the TLB Shootdown Algorithm with Page Access Tracking

Nadav Amit
VMware Research

Abstract

The operating system is tasked with maintaining the coherency of per-core TLBs, necessitating costly synchronization operations, notably to invalidate stale mappings. As core-counts increase, the overhead of TLB synchronization likewise increases and hinders scalability, whereas existing software optimizations that attempt to alleviate the problem (like batching) are lacking.

We address this problem by revising the TLB synchronization subsystem. We introduce several techniques that detect cases whereby soon-to-be invalidated mappings are cached by only one TLB or not cached at all, allowing us to entirely avoid the cost of synchronization. In contrast to existing optimizations, our approach leverages hardware page access tracking. We implement our techniques in Linux and find that they reduce the number of TLB invalidations by up to 98% on average and thus improve performance by up to 78%. Evaluations show that while our techniques may introduce overheads of up to 9% when memory mappings are never removed, these overheads can be avoided by simple hardware enhancements.

1. Introduction

Translation lookaside buffers (TLBs) are perhaps the most frequently accessed caches whose coherency is not maintained by modern CPUs. The TLB is tasked with caching virtual-to-physical translations ("mappings") of memory addresses, and so it is accessed upon every memory read or write operation. Maintaining TLB coherency in hardware hampers performance [33], so CPU vendors require OSes to maintain coherency in software. But it is difficult for OSes to *efficiently* achieve this goal [27, 38, 39, 41, 48].

To maintain TLB coherency, OSes employ the TLB shootdown protocol [8]. If a mapping m that possibly resides in the TLB becomes stale (due to memory mapping changes) the OS flushes m from the local TLB to restore coherency. Concurrently, the OS directs remote cores that might house m in their TLB to do the same, by sending them an inter-processor interrupt (IPI). The remote cores flush their TLBs according to the information supplied by the initiator core, and they report back when they are done. TLB shootdown can take microseconds, causing a notable slowdown [48]. Performing TLB shootdown in hardware, as certain CPUs do, is faster but still incurs considerable overheads [22].

In addition to reducing performance, shootdown overheads can negatively affect the way applications are constructed. Notably, to avoid shootdown latency, programmers are advised against using memory mappings, against unmapping them, and even against building multithreaded applications [28, 42]. But memory mappings *are* the efficient way to use persistent memory [18, 47], and avoiding unmappings might cause corruption of persistent data [12].

OSes try to cope with shootdown overheads by batching them [21, 43], avoiding them on idle cores, or, when possible, performing them faster [5]. But the potential of these existing solutions is inherently limited to certain specific scenarios. To have a generally applicable, efficient solution, OSes need do know which mappings are cached by which cores. Such information can in principle be obtained by replicating the translation data structures for each core [11], but this approach might result in significantly degraded performance and wasted memory.

We propose to avoid unwarranted TLB shootdowns in a different manner: by monitoring access bits. While TLB coherency is not maintained by the CPU, CPU architectures can maintain the consistency of access bits, which are set when a mapping is cached. We contend that these bits can therefore be used to reveal which mappings are cached by which cores. To our knowledge, we are the first to use access bits in this way.

In the x86 architecture, which we study in this paper, access bit consistency is maintained by the memory subsystem. Exploiting it, we propose techniques to identify two types of common mappings whose shootdown can be avoided: (1) short-lived private mappings, which are only cached by a single core; and (2) long-lived idle mappings, which are reclaimed after the corresponding pages have not been used for a while and are not cached at all. Using

these techniques, we implement a fully functional prototype in Linux 4.5. Our evaluation shows that our proposal can eliminate more than 90% of TLB shootdowns and improve the performance of memory migration by 78%, of copy-on-write events by 18–25%, and of multithreaded applications (Apache and parallel bzip2) by up to 12%.

Our system introduces a worst case slowdown of up to 9% when mappings are only set and never removed or changed, which means no shootdown activity is conducted. This slowdown is caused, according to our measurements, due to the overhead of our TLB manipulation software techniques. To eliminate it, we propose a CPU extension that would allow OSes to write entries directly into the TLB, and resembles the functionality provided by CPUs that employ software-TLB.

2. Background and Motivation

2.1 Memory Management Hardware

Virtual memory is supported by most modern CPUs and used by all the major OSes [9, 32]. Using virtual memory allows the OS to utilize the physical memory more efficiently and to isolate the address space of each process. The CPU translates the virtual addresses to physical addresses before memory accesses are performed. The OS sets the virtual address translations (also called "mappings") according to its policies and considerations.

The memory mappings of each address space are kept in a memory-resident data structure, which is defined by the CPU architecture. The most common data structure, used by the x86 architecture, is a radix-tree, which is also known as a page-table hierarchy. The leaves of the tree, called the page-table entries (PTEs), hold the translations of fixed-sized virtual memory pages to physical frames. To translate a virtual address into a physical address, the CPU incorporates a memory management unit (MMU), which performs a "page table walk" on the page table hierarchy, checking access permissions at every level. During a page-walk, the MMU updates the status bits in each PTE, indicating whether the page was read from and/or written to (dirtied).

To avoid frequent page-table walks and their associated latency, the MMU caches translations of recently used pages in a translation lookaside buffer (TLB). In the x86 architecture, these caches are maintained by the hardware, bringing translations into the cache after page walks and evicting them according to an implementation-specific cache replacement policy. Each x86 core holds a logically private TLB.

Unlike memory caches, TLBs of different CPUs are not maintained coherent by hardware. Specifically, x86 CPUs do not maintain coherence between the TLB and the page-tables, nor among the TLBs of different cores. As a result, page-table changes may leave stale entries in the TLBs until coherence is restored by the OS. The instruction set enables the OS to do so by flushing ("invalidating") individual PTEs or the entire TLB. Global and individual TLB flushes can only be performed locally, on the TLB of the core that executes the flush instruction.

Although the TLB is essential to attain reasonable translation latency, some workloads experience frequent TLB cache-misses [4]. Recently, new features were introduced into the x86 architecture to reduce the number and latency of TLB misses. A new instruction set extension allows each page-table hierarchy to be associated with an address-space ID (ASID) and avoid TLB flushes during address-space switching, thus reducing the number of TLB misses. Micro-architectural enhancements introduced page-walk caches that enable the hardware to cache internal nodes in the page-table hierarchy, thereby reducing TLB-miss latencies [3].

2.2 TLB Software Challenges

The x86 architecture leaves maintaining TLB coherency to the OSes, which often requires frequent TLB invalidations after PTE changes. OS kernels can make such PTE changes independently of the running processes, upon memory migration across NUMA nodes [2], memory deduplications [49], memory reclamation, and memory compaction for accommodating huge pages [14]. Processes can also trigger PTE changes by using system calls, for example `mprotect`, which changes protection on a memory range, or by writing to copy-on-write pages (COW).

These PTE changes can require a TLB flush to avoid caching of stale PTEs in the TLB. We distinguish between two types of flushes: *local* and *remote*, in accordance with the core that initiated the PTE change. Remote TLB flushes are significantly more expensive, since most CPUs cannot flush remote TLBs directly. OSes therefore perform a *TLB shootdown*: The initiating core sends an inter-processor interrupt (IPI) to the remote cores and waits for their interrupt handlers to invalidate their TLBs and acknowledge that they are done.

TLB shootdowns introduce a variety of overheads. IPI delivery can take several hundreds of cycles [5]. Then, the IPI may be kept pending if the remote core has interrupts disabled, for instance while running a device driver [13]. The x86 architecture does not allow OSes to flush multiple PTEs efficiently, requiring the OS to either incur the overhead of multiple flushes or flush the entire TLB and increase the TLB miss rate. In addition, TLB flushes may

indirectly cause lock contention since they are often performed while the OS holds a lock [11, 15]. It is noteworthy that while some CPU architectures (e.g., ARM) enable to perform remote TLB shootdowns without IPIs, remote shootdowns still incur higher performance overhead than local ones [22].

2.3 OS Solutions and Shortcomings

To reduce TLB related overheads, OSes employ several techniques to avoid unnecessary shootdowns, reduce their time, and avoid TLB misses.

A TLB shootdown can be avoided if the OS can ensure that the modified PTE is either not cached in remote TLBs or can be flushed at a later time, but before it can be used for an address translation. In practice, OSes can only avoid remote shootdowns in certain cases. In Linux, for example, each userspace PTE is only set in a single address space page-table hierarchy, allowing the OS to track which address space is active on each core and flush only the TLBs of cores that currently use this address space. The TLB can be flushed during context switch, before any stale entry would be used.

A common method to reduce shootdown time is to batch TLB invalidations if they can be deferred [21, 47]. Batching, however, cannot be used in many cases, for example when a multithreaded application changes the access permissions of a single page. Another way to reduce shootdown overhead is to acknowledge its IPI immediately, even before invalidation is performed [5, 43].

Flush time can be reduced by lowering the number of TLB flushes. Flushing multiple individual PTEs is expensive, and therefore OSes can prefer to flush the entire TLB if the number of PTEs exceeds a certain threshold. This is a delicate trade-off, as such a flush increases the number of TLB misses [23].

Linux tries to balance between the overheads of TLB flushes and TLB misses when a core becomes idle, using a lazy TLB invalidation scheme. Since the process that ran before the core became idle may be scheduled to run again, the OS does not switch its address space, in order to avoid potential future TLB misses. However, when the first TLB shootdown is delivered to the idle core, the OS performs a full TLB invalidation and indicates to the other cores not to send it further shootdown IPIs while it is idle.

Despite all of these techniques, shootdowns can induce high overheads in real systems. Arguably, this overhead is one of the reasons people refrain from using multithreading, in which mapping changes need to propagate to all threads. Moreover, application writers often prefer copying data over memory remapping, which requires TLB shootdown [42].

2.4 Per-Core Page Tables

Currently, the state-of-the-art software solution for TLB shootdowns is setting per-core page tables, and according to the experienced page-faults track which cores used each PTE [11,19]. When a PTE invalidation is needed, a shootdown is sent only to cores whose page tables hold the invalidated PTE.

Maintaining per-core page tables, however, can introduce substantial overheads when some PTEs are accessed by multiple cores. In such a case, OS memory management operations become more expensive, as mapping modifications require changes the of PTEs in multiple page-tables. The overhead of PTE changes is not negligible, as some require atomic operations. RadixVM [11] reduces this overhead by changing PTEs in parallel: sending IPIs to cores that hold the PTE and changing them locally. This scheme is efficient when shootdowns are needed, as one IPI triggers both the PTE change and its invalidation. Yet, if a shootdown is not needed, for example when the other cores run a different process, this solution may increase the overhead due to the additional IPIs.

Holding per-core page tables can also introduce high memory overheads if memory is accessed by multiple cores. For example, in recent 288 core CPUs [24], if half of the memory is accessed by all cores, the page tables will consume 18% of the memory or more if memory is overcommitted or mappings are sparse.

While studies showed substantial performance gains when per-core page tables are used, the limitations of this approach may have not been studied well enough. For example, in an experiment we conducted memory migration between NUMA nodes was 5 times slower when memory was mapped in 48 page-table hierarchies (of 48 Linux running processes in our experiment) instead of one. Previous studies may have not shown these overheads as they considered a teaching OS, which lacks basic memory management features [11]. In addition, previous studies experienced shootdown latencies of over 500k cycles, which is over 24x of the latency that we measured. Presumably, the high overhead of shootdowns could overshadow other overheads.

3. The Idea

The challenge in reducing TLB shootdown overhead is determining which cores, if at all, might be caching a given PTE. Although architectural paging structures do not generally provide this information, we contend that the OS can nevertheless deduce it by carefully tracking and manipulating PTE access-bits. The proclaimed goal of access bits is to indicate if memory pages have been accessed. This functional-

ity is declared by architectural manuals and is used by OSes to make informed swapping decisions. Our insight is that access bits can be additionally used for a different purpose: to indicate if PTEs are cached in TLBs, as explained next.

Let us assume: that (1) a PTE e might be cached by a set of cores S at time t_0; that (2) e's access bit is clear at t_0 (because it was never set, or because the OS explicitly cleared it); and that (3) this bit is still clear at some later time t_1. Since access bits are set by hardware whenever it caches the corresponding translations in the TLB [25], we can safely conclude that e is *not* cached by any core $c \notin S$ at t_1.

We note that our reasoning rests on the fact that last-level TLBs are private per core [6,27,29] and so translations are not transferred between them. Linux, for example, relies on this fact when shooting down a PTE of some address space α while avoiding the shootdown at remote cores whose current address spaces are different than α (§2.3). This optimization would have been erroneous if TLBs were shared, because Linux permits the said remote cores to freely load α while the shootdown takes place, which would have allowed them to cache stale mappings from a shared last-level TLB, thereby creating an inconsistency bug.

We identify two types of mappings that can help us optimize TLB shootdown by leveraging access-bit information. The first is *short-lived private mappings* of pages that are accessed exclusively by a single thread and then removed shortly after; this access pattern may be exhibited, for example, by multithreaded applications that use memory-mapped files to read data. The second type is *long-lived idle mappings* of pages that are reclaimed by the OS after they have not been accessed for a while; this pattern is typical for pages that cease to be part of the working set of a process, prompting the OS to unmap them, flush their PTEs, and reuse their frames elsewhere.

4. The System

Using the above reasoning (§3), we next describe the Linux enhancements we deploy on an x86 Intel machine to optimize TLB shootdown of short-lived private mappings (§4.1) and long-lived idle mappings (§4.2). We then describe "software-PTEs", the data structures we use when implementing our mechanisms (§4.3). To distinguish our enhancements from the baseline OS, we collectively denote them as ABIS—access-based invalidation system.

4.1 Private PTE Detection

To avoid TLB shootdown due to a private mapping, we must (1) identify the core that initially uses this mapping and (2) make sure that other cores have not used it too at a later time. As previously shown [27], the first item is achievable via demand paging, the standard memory management technique that OSes employ, which traps upon the first access to a memory page and only then sets a valid mapping [9]. The second item, however, is more challenging, as existing approaches to detect PTE sharing can introduce overheads that are much higher than those we set out to eliminate (§6).

Direct TLB Insertion Our goal is therefore to find a low-overhead way to detect PTE sharing. As a first step, we note that this goal would have been easily achievable if it was possible to conduct *direct TLB insertion*—inserting a mapping m directly into a TLB of a core c without setting the access bit of the corresponding PTE e. Given such a capability, as long as m resides in the TLB, subsequent uses of m by c would not set the access-bit of e, as no page table walks are needed. In contrast, if some other core $\bar{c}$ ends up using m as well, the hardware will walk the page table when inserting m to the TLB of $\bar{c}$, and it will therefore set e's access bit, thereby indicating that m is not private.

Direct TLB insertion would have thus allowed us to use turned-off access bits as identifiers of private mappings. We remark that this method is best-effort and might lead to false-positive indications of sharing in cases where m is evicted from the TLB and reinserted later. This issue does not affect correctness, however. It simply implies that some useless shootdown activity is possible. The approach is thus more suitable for short-lived PTEs.

Alas, current x86 processors do not support direct TLB insertion. One objective of this study is to motivate such support. When proposing a new hardware feature, architects typically resort to simulation since it is unrealistic to fabricate chips to test research features. We do not employ simulation for two reasons. First, because we suspect that it might yield questionable results, as the OS memory management subsystems that are involved are complex to realistically simulate. Second, since TLB insertion is possible on existing hardware even without hardware support, and can benefit workloads that are sensitive to shootdown overheads, shortening runtimes by 0.56x ($=\frac{1}{1.78}$; see Figure 5) at best. Although runtimes might be 1.09x longer in the worst case, our results indicate that real hardware support will eliminate this overhead (§5.1).

Note that although direct TLB insertion is not supported in the x86 architecture, it is supported in CPUs that employ software-managed TLBs. For example, Power CPUs support the `tlbwe` instruction

that can insert PTE directly into the TLB. We therefore consider this enhancement achievable with a reasonable effort.

Approximation Let us first rule out the naive approach to approximate direct TLB insertion by: (1) setting a PTE e; (2) accessing the page and thus prompting hardware to load the corresponding mapping m into the TLB and to set e's access bit; and then (3) having the OS clear e's access bit. This approach is buggy due to the time window between the second and third items, which allows other cores to cache m in their TLBs before the bit is cleared, resulting in a false sharing indications that the page is private. Shootdown will then be erroneously skipped.

We resolve this problem and avoid the above race by using Intel's address space IDs, which is known as process-context identifiers (PCIDs) [25]. PCIDs enable TLBs to hold mappings of multiple address spaces by associating every cached PTE with a PCID of its address space. The PCID of the current address space is stored in the same register as the pointer to the root of the page table hierarchy (CR3), and TLB entries are associated with this PCID when they are cached. The CPU uses for address translation only PTEs whose PCID matches the current one. This feature is intended to allow OSes to avoid global TLB invalidations during context switch and reduce the number of TLB misses.

PCID is not currently used by Linux due to the limited number of supported address spaces and questionable performance gains from TLB miss reduction. We indeed exploit this feature in a different manner. Nevertheless, our use does not prevent or limit future PCID support in the OS.

The technique ABIS employs to provide direct TLB insertion is depicted in Figure 1. Upon initialization, ABIS preallocates for each core a "secondary" page-table hierarchy, which consists of four pages, one for each level of the hierarchy. The uppermost level of the page-table (PGD) is then set to point to the kernel mappings (like all other address spaces). The other three pages are not connected at this stage to the hierarchy, but wired dynamically later according to the address of the PTE that is inserted to the TLB.

While executing, the currently running thread T occasionally experiences page faults, notably due to demand paging. When a page fault fires, the OS handler is invoked and locks the PT that holds the faulting PTE—no other core will simultaneously handle the same fault.

At this point, ABIS loads the secondary space to CR3 along with a PCID equal to that of T (Step 1 in Figure 1). After, ABIS wires the virtual-to-physical mapping of the target page in both primary and

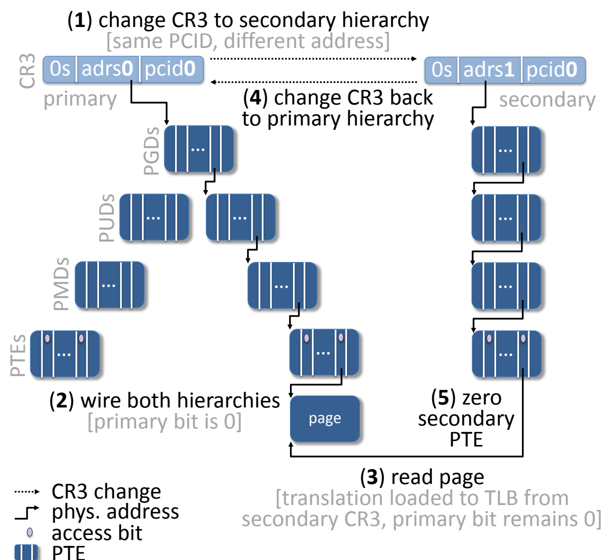


Figure 1: Direct TLB insertion using a secondary hierarchy.

secondary spaces, leaving the corresponding access bit in the primary hierarchy clear (Step 2).

Then, ABIS reads from the page. Because the associated mapping is currently missing from the TLB (a page fault fired), and because CR3 currently points to the secondary space, reading the page prompts the hardware to walk the secondary hierarchy and to insert the appropriate translation to the TLB, leaving the primary bit clear (Step 3). Importantly, the inserted translation is valid and usable within the primary space, because both spaces have the same PCID and point to the same physical page using the same virtual address. This approach eliminates he aforementioned race: no other core is able to access the secondary space, as it is private to the core.

After reading the page, ABIS loads the primary hierarchy back to CR3, to allow the thread to continue as usual (Step 4). It then clears the PTE from the secondary space, thereby preventing further use of translation data from the secondary hierarchy that may have been cached in the hardware page-walk cache (PWC). If the secondary tables are used by the CPU for translation, no valid PTE will be found and the CPU will restart a page-walk from the root entry.

Finally, using our "software-PTE" (SPTE) data structure (§4.3), ABIS associates the faulting PTE e with the current core c that has just resolved e. When the time comes to flush e, if ABIS determines that e is still private to c, it will invalidate e on c only, thus avoiding the shootdown overhead.

Coexisting with Linux Linux reads and clears architectural access bits (hwA-s) via a small API, allowing us to easily mask these bits while making sure

that both Linux and ABIS simultaneously operate correctly. Notably, when Linux attempts to clear an hwA, ABIS (1) checks whether the bit is turned on, in which case it (2) clears the bit and (3) records in the SPTE the fact that the associated PTE is not private (using the value `ALL_CPUS` discussed further below). Note, however, that Linux and ABIS can use the access bit in a conflicting manner. For example, after a page fault, Linux could expect to see the access bit turned on, whereas ABIS's direct TLB insertion makes sure that the opposite happens. To avoid any such conflicts, we maintain in the SPTE a new per-PTE "software access bit" (swA) for Linux, which reflects Linux's expectations. The swA bits are governed by the following rules: upon a page fault, we set the swA; when Linux clears the bit, we clear the swA; and when Linux queries the bit, we return an OR'd value of swA and hwA. These rules ensure that Linux *always* observes the values it would have observed in an ABIS-less system.

ABIS attempts to reduce false indications of PTE sharing when possible. We find that Linux performs excessive full flushes to reduce the number of IPIs sent to idle cores as part of the shootdown procedure (§2.3). In Linux, this behavior is beneficial as it reduces the number of TLB shootdowns at the cost of more TLB misses, whose impact is relatively small. In our system, however, this behavior can result in *more* shootdowns, as it increases the number of false indications. ABIS therefore relaxes this behavior, allowing idle cores to service a few individual PTE flushes before resorting to a full TLB flush.

Overhead Overall, the overhead of direct TLB insertions in our system is ≈550 cycles per PTE (responsible for the worst-case 9% slowdown mentioned earlier). This overhead is amortized when multiple PTEs are mapped together, for example, via one mmap system-call invocation, or when Linux serves a page-fault on a file-backed page and maps adjacent PTEs to avoid future page-faults [36].

4.2 TLB Version Tracking

Based on our observations from §3, we build a TLB version tracking mechanism to avoid flushes of long-lived idle mappings. Let us assume that a PTE e might be cached by a set of cores S at time t_0, and that each core $c \in S$ performed a full TLB flush during the time period (t_0, t_1). If at time t_1 the access bit of e remains clear (i.e., was not cleared by software), then we know for a fact e is not cached by any TLB. If the OS obtained the latter information by atomically reading and zeroing e, then all TLB flushes associated with e (local and remote) can be avoided. To detect such cases, we first need to maintain a "full-

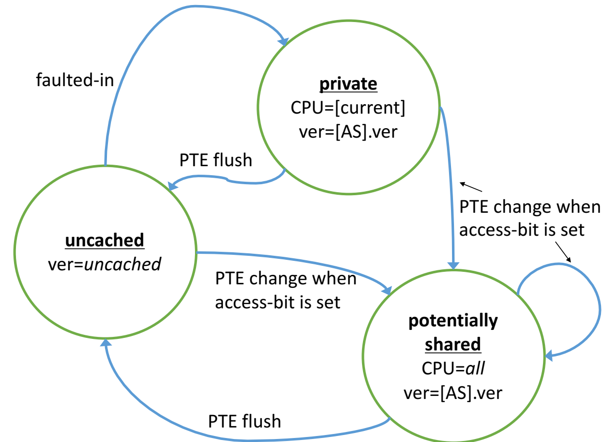


Figure 2: A finite state machine that describes the various states of a PTE. In each state, the assignment of the caching core and version are denoted. On each transition the access-bit is cleared.

flush version number" for S, such that the version is incremented whenever *all* cores $c \in S$ perform a full TLB flush. Recording this version for each e at the time e is updated would then allow us to employ the optimization.

TLB version tracking The most accurate way to track full flushes is by maintaining a version for each core, advancing it after each local full flush, and storing a vector of the versions for every PTE. Then, if a certain core's version differs from the corresponding vector coordinate (and the access-bit is clear), a flush on that core is not required. Despite its accuracy, this scheme is impractical, as it consumes excessive memory and requires multiple memory accesses to update version vectors. We therefore trade off accuracy in order to reduce the memory consumption of versions and the overheads of updating them.

ABIS therefore tracks versions for each address space (AS, corresponds to the above S) and not for each core. To this end, for every AS, we save a version number and a bitmask that marks which cores have not performed a full TLB flush in the current version. The last core to perform a full TLB flush in a certain version advances the version. At the same time, it marks in the bitmask which cores currently use this AS and can therefore cache PTEs in the next version. To mitigate cache line bouncing, the core that initiates a TLB shootdown updates the version on behalf the target cores.

Avoiding flushes After a PTE access-bit is cleared, ABIS stores the current AS version as the PTE version. Determining later whether a shootdown is needed requires some attention, as even if the PTE and the AS versions differ, a flush may be necessary. Consider

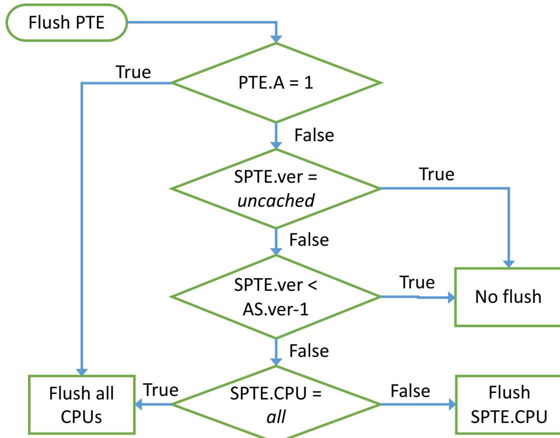


Figure 3: Flush type decision algorithm.

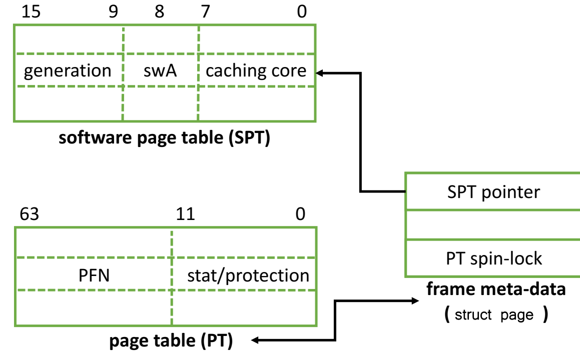


Figure 4: Software PTE (SPTE) and its association to the page table through the meta-data of the page-table frame.

a situation in which the access-bit is cleared, and the PTE version is updated to hold the AS version. At this time, some of the cores may have already flushed their TLB for the current AS version, and their respective bit in the bitmask is clear. The AS version may therefore advance before these cores flush their TLB again, and these cores can hold stale PTEs even when the versions differ. Thus, our system avoids shootdown only if there is a gap of at least one version between the AS and the PTE versions, which indicates a flush was performed on all cores.

Since flushes cannot be avoided when the access-bit is set, this bit should be cleared and the PTE version updated as frequently as possible, assuming it introduces negligible overheads. In practice, ABIS clears the bit and updates the version whenever the OS already accesses a PTE for other purposes, for example during an `mprotect` system-call or when the OS considers a page for reclamation.

Uncached PTEs The version tracking mechanism can also prevent unwarranted multiple flushes of the same PTE. Such flushes may occur, for example, when a user first calls an `msync` system call, which performs writeback of a memory mapped file, and then unmaps the file. Both operations require flushing the TLB since the first clears PTEs' dirty-bit and the second sets a non-present PTE. However, if the PTE was not accessed after the first flush, the second flush is unnecessary, regardless of whether a full TLB flush happened in between. To avoid this scenario, we set a special version value, `UNCACHED`, as the PTE version when it is flushed. This value indicates the PTE is not cached in any TLB if the access-bit is cleared, regardless of the current AS version.

Coexisting with Private PTE Detection Version tracking coexists with private PTE detection. The interaction between the two can be described in a state machine, as shown in Figure 2. In the "uncached" state a TLB flush is unnecessary; in the "private" state at most one CPU needs to perform a TLB flush; and in the "potentially shared" state all the CPUs perform TLB flush.[1] In the latter two states, a TLB flush may still be avoided if the access-bit is clear and the current address space version is at least two versions ahead of the PTE version. Figure 3 shows ABIS flush decision algorithm.

4.3 Software PTEs

As we noted before, for our system to perform informed TLB invalidation decisions, additional information must be saved for each PTE: the PTE version, the CPU which caches the PTE, and a software access-bit. Although we are capable of squeezing this information into two bytes, the architectural PTE only accommodates three bits for software use. We therefore allocate a separate "software page-table" (SPT) for each PT, which holds the corresponding "software-PTEs" (SPTEs). The SPTE is not used by the CPU during page-walks and therefore causes little cache pollution and overhead.

An SPTE is depicted in Figure 4. We use 7 bits for the version, 1 bit for the software access-bit, and another byte to track the core that caches the PTE if the access-bit is cleared. We want to define the SPTE in a manner that ensures a zeroed SPTE would behave in the legacy manner, allowing us to make fewer code changes. To do so, we reserve the zero value of the "caching core" field to indicate that the PTE may be cached by all CPUs (`ALL_CPUS`) and instead store the core number plus one.

When the OS wishes to access the SPTE of a certain PTE, it should be able to easily access it. Yet the PTE cannot accommodate a pointer to its SPTE. A possible solution is to allocate two page-frames for each page-

[1] A TLB flush is not required on CPUs that currently use a different page-table hierarchy as explained in §2

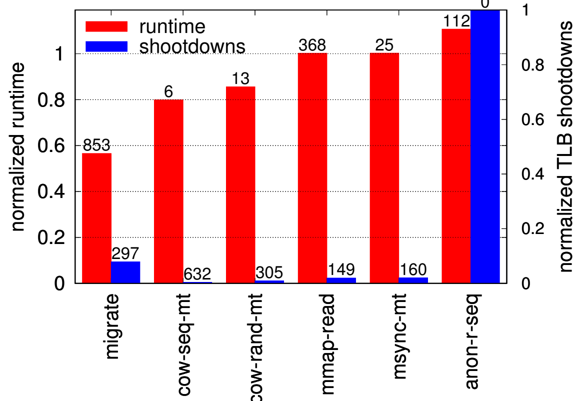


Figure 5: Normalized runtime and number of TLB shootdowns in ABIS when running vm-scalability benchmarks. The numbers above the bars indicate the baseline (left) runtime in seconds and (right) rate of TLB shootdowns in thousands/second.

table, one holding the CPU architectural PTEs and the second holding the corresponding SPTEs, each in a fixed offset from its PTE. While this scheme is simple, it wastes memory as it requires the SPTE to be the same size as a PTE (8B), when in fact SPTE only occupies two bytes.

We therefore allocate an SPT separately during the PT construction, and set a pointer to the SPT in the PT page-frame meta-data (`page struct`). Linux can quickly retrieve this meta-data, allowing us to access the SPTE of a certain PTE with small overhead. The SPTE pointer does not increase the page-frame meta-data, as it is set in an unused PT meta-data field (second quadword). The SPT therefore increases page table memory consumption by 25%. ABIS prevents races during SPT changes by protecting it with the same lock that is used to protect PT changes. It is noteworthy that although SPT management introduces a overhead, it is negligible relatively to other overheads in the workloads we evaluated.

5. Evaluation

We implemented a fully-functional prototype of the system, ABIS, which is based on Linux 4.5. As a baseline system for comparison we use the same version of Linux, which includes recent TLB shootdown optimizations. We run each test 5 times and report the average result. Our testbed consists of a two-socket Dell PowerEdge R630 with Intel 24-cores Haswell EP CPUs. We enable x2APIC cluster-mode, which speeds up IPI delivery.

In our system we disable transparent huge pages (THP), which may cause frequent full TLB flushes, increase the TLB miss-rate [4] and introduce additional overheads [26]. In practice, when THP is enabled, ABIS still shows benefit when small pages are used (e.g., in the Apache benchmark shown later) and no impact when huge pages are used (e.g., PBZIP2).

As a fast block device for our experiments we use ZRAM, a compressed RAM block device, which is used by Google Chrome OS and Ubuntu. This device latency is similar to that of emerging non-volatile memory modules. In our test, we disable memory deduplication and deep sleep states which may increase the variance of the results.

5.1 VM-Scalability

We use the vm-scalability test suite [34], which is used by Linux kernel developers to exercise the kernel memory management mechanisms, test their correctness and measure their performance.

We measure ABIS performance by running benchmarks that experience high number of TLB shootdowns.[2] To run the benchmarks in a reasonable time, we limit the amount of memory each test consumes to 32GB. Figure 5 presents the measured speedup, the runtime, the relative number of sent TLB shootdowns and their rate. We now discuss these results.

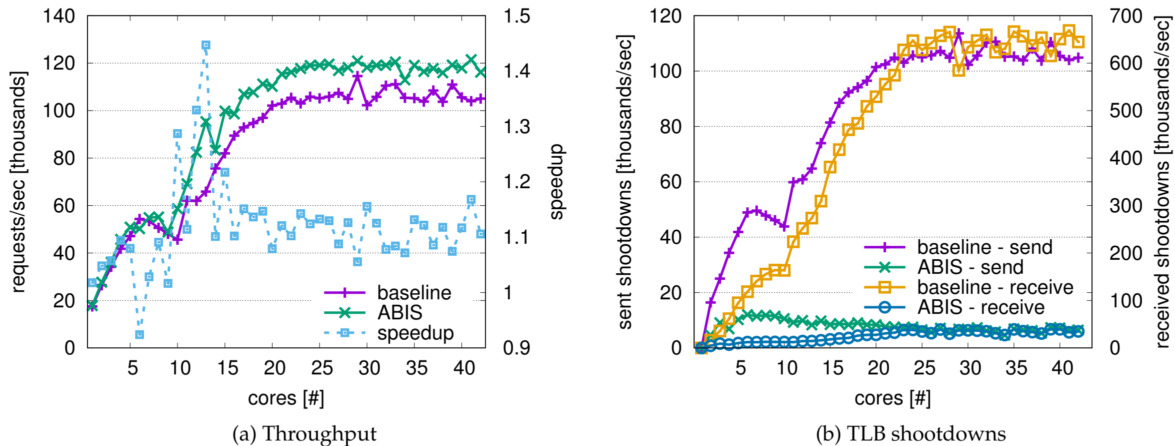
Migrate. This benchmark reads a memory mapped file and waits while the OS is instructed to migrate the process memory between NUMA nodes. During migration, we set the benchmark to perform a busy-wait loop to practice TLB flushes. We present the time that a 1TB memory migration would take. ABIS reduces runtime by 44% and shootdowns by 92%.

Multithreaded copy-on-write (cow-mt). Multiple threads read and write a private memory mapped file. Each write causes the kernel to copy the original page, update the PTE to point to the copy, and flush the TLB. ABIS prevents over 97% of the shootdowns, reducing runtime by 20% for sequential memory accesses and 15% for random by avoiding over 97%.

Memory mapped reads (mmap-read). Multiple processes read a big sparse memory mapped file. As a result, memory pressure builds up, and memory is reclaimed. While almost all the shootdowns are eliminated, the runtime is not affected, as apparently there are more significant overheads, specifically those of the page frame reclamation algorithm.

Multithreaded msync (msync-mt). Multiple threads access a memory mapped file and call the `msync` system-call to flush the memory changes to the file. `msync` can cause an overwhelming number of flushes, as the OS clears the dirty-bit. ABIS eliminates 98% of the shootdowns but does not reduce the runtime, as file system overhead appears to be the main performance bottleneck.

[2] We find that due to some benchmarks practice unrealistic scenarios. Our revised tests are released with ABIS code.



(a) Throughput

(b) TLB shootdowns

Figure 6: Execution of an Apache web server which serves the Wrk workload generator.

Anonymous memory read (anon-r-seq). To evaluate ABIS overheads we run a benchmark that performs sequential anonymous memory reads and does not cause TLB shootdowns. This benchmark's runtime is 9% longer using ABIS. Profiling the benchmark shows that the software TLB manipulations consume 9% of the runtime, suggesting that hardware enhancements to manipulate the TLB can eliminate most of the overheads.

5.2 Apache Web Server

Apache is the most widely used web server software. In our tests, we use Apache v2.4.18 and enable buffered server logging for more efficient disk accesses. We use the multithreaded Wrk workload generator to create web requests [50], and set it to repeatedly request the default Apache web page for 30 seconds, using 400 connections and 6 threads. We use the same server for both the generator and Apache, and isolate each one on a set of cores. We ensure that the generator is unaffected by ABIS.

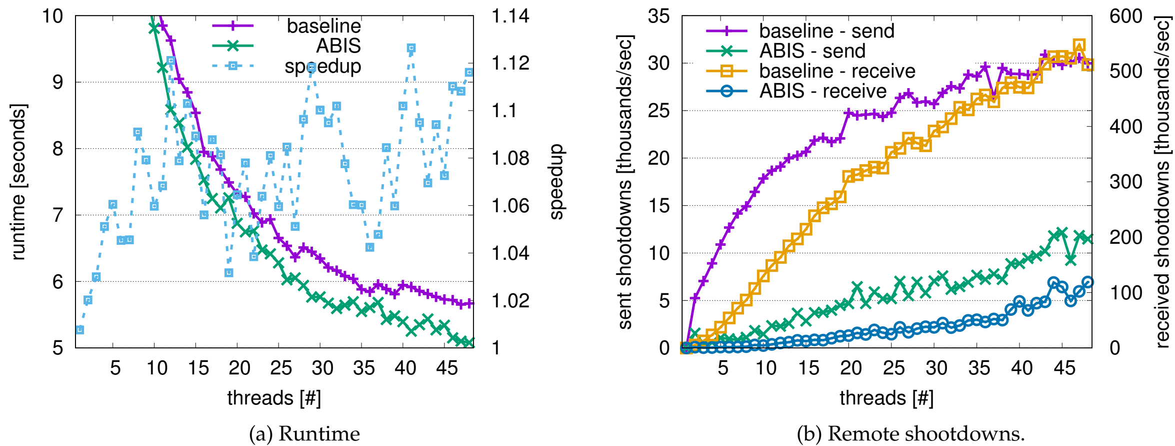
Apache provides several multi-processing modules. We use the default "mpm_event" module, which spawns multiple processes, each of which runs multiple threads. Apache serves each request by creating a memory mapping of the requested file, sending its content and unmapping it. This behavior effectively causes frequent invalidations of short-lived mappings. In the baseline system, the invalidation also requires expensive TLB shootdowns to the cores that run other threads of the Apache process. Effectively, when Apache serves concurrent requests using multiple threads, it triggers a TLB shootdown for each request that it serves.

Figure 6a depicts the number of requests per second that are served when the server runs on different number of cores. ABIS improves performance by 12% when all cores are used. Executing the benchmark reveals that the effect of ABIS on performance is inconsistent when the number of cores is low, as ABIS causes slowdown of up to 8% and speedups of to 42%. Figure 6b presents the number of TLB shootdown that are sent and received in the baseline system and ABIS. As shown, in the baseline system, as more cores are used, the amount of sent TLB shootdowns becomes almost identical to the number of requests that Apache serves. ABIS reduces the number of both sent and received shootdowns by up to 90% as it identifies that PTEs are private and that local invalidation would suffice.

5.3 PBZIP2

Parallel bzip2 (PBZIP2) is a multithreaded implementation of the bzip2 file compressor [20]. In this benchmark we evaluate the effect of reclamation due to memory pressure on PBZIP2, which in itself does not cause many TLB flushes. We use PBZIP2 to compress the Linux 4.4 tar file. We configured the benchmark to read the input file into RAM and split it between processors using 500k block size. We run PBZIP2 in a container and limit its memory to 300MB to induce swap activity. This activity causes the invalidation of long-lived idle mappings as inactive memory is reclaimed.

The time of compression is shown in Figure 7a. ABIS outperforms Linux by up to 12%, and the speedup grows with the number of cores. Figure 7b presents the number of TLB shootdowns per second when this benchmark runs. The baseline Linux system sends nearly 200k shootdowns regardless of the number of threads, and the different shootdown send rate is merely due to the shorter runtime when the number of cores is higher. The number of received shootdowns in the baseline system is



(a) Runtime

(b) Remote shootdowns.

Figure 7: Execution of PBZIP2 when compressing the Linux kernel. The process memory is limited to 300MB to practice page reclamation.

proportional to the number of cores, as the OS cannot determine which TLBs cache the entry, and broadcasts the shootdown messages to all the cores that run the process threads. In contrast, ABIS can usually determine that a single TLB needs to be flushed. When 48 threads are spawned, a shootdown is sent on average to 10 remote cores in ABIS, and to 18 cores using baseline Linux.

5.4 PARSEC Benchmark Suite

We run the PARSEC 3.0 benchmark suite [7], which is composed of multithreaded applications that are intended to represent emerging shared-memory programs. We set up the benchmark suite to use the native dataset and spawn 32 threads. The measured speedup, the runtime, the normalized number of TLB shootdowns and their rate in the baseline system are presented in Figure 8. As shown, ABIS can improve performance by over 3% but can also induce overheads of up to 2.5%. ABIS reduces the number of TLB shootdowns by 96% on average.

The benefit of ABIS appears to be limited by the overhead of the software technique it uses to insert PTEs into the TLB. As this overhead is incurred after each page fault, workloads which trigger considerably more page faults than TLB shootdowns experience slowdown. For example, "canneal" benchmark causes 1.5k TLB shootdowns per second in the baseline system, and ABIS prevents 91% of them. However, since the benchmark triggers over 55k page-faults per second, ABIS reduces performance by 2.5%. In contrast, "dedup" triggers 33k shootdowns and 370k page faults per second correspondingly. ABIS saves 39% of the shootdowns and improves performance by 3%. Hardware enhancements or selective enabling of ABIS could prevent the overheads.

5.5 Limitations

ABIS is not free of limitations. The additional operations and data introduce performance and memory overheads, specifically the insertions of PTEs into the TLB without setting the access-bit. However, relatively simple hardware enhancements could have eliminated most of the overhead (§7). In addition, the CPU incurs overhead of roughly 600 cycles when it sets the access-bit of shared PTEs [37].

To detect short-lived private mappings, our system requires that the TLB be able to accommodate them during their lifetime. New CPUs include rather large TLBs of up to 1536 entries, which may map 6MB of memory. However, non-contiguous or very large working sets may cause TLB pressure, induce evictions, and cause false indications that PTEs are shared. In addition, frequent full TLB flushes, for instance during address-space switching or when the OS sets the CPU to enter deep sleep-state have similar implications. Process migration between cores is also damaging as it causes PTEs to be shared between cores and requires shootdowns. These limitations are often irrelevant to a well-tuned system [30, 31].

Finally, our system relies on micro-architectural behavior of the TLBs. We assume the MMU does not perform involuntary flushes and that the same PTE is not marked as "accessed" multiple times when it is already cached. Experimentally, this is not always the case. We further discuss these limitations in §7.

6. Related Work

Hardware Solutions. The easiest solution from a software point of view is to maintain TLB coherency in hardware. DiDi uses a shared second-level TLB directory that tracks which PTEs are cached by which

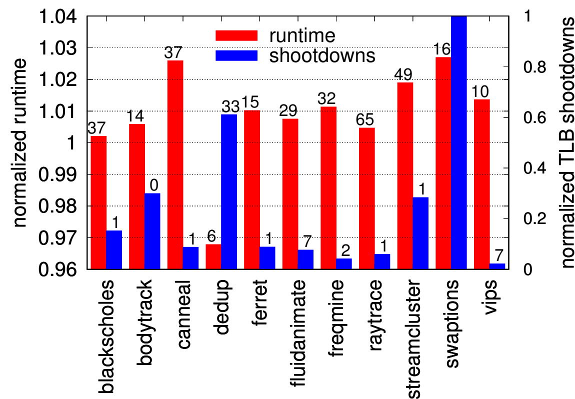


Figure 8: Normalized runtime and number of TLB shootdowns in ABIS when running PARSEC benchmarks. The numbers above the bars indicate the baseline (left) runtime in seconds and (right) rate of TLB shootdowns in thousands/second.

core and performs TLB flushes accordingly [48]. Teller et al. proposed that OSes save a version count for each PTE, to be used by hardware to perform TLB invalidations only when memory is addressed via a stale TLB entry [39]. Li et al. eliminate unwarranted shootdowns of PTEs that are only used by a single core by extending PTEs to accommodate the core that first accessed a page, enhancing the CPU to track whether a PTE is private and avoiding shootdowns accordingly [27].

These studies present compelling evaluation results; however, they require intrusive micro-architecture changes, which CPU vendors are apparently reluctant to introduce, presumably due to a history of TLB bugs [1, 16, 17, 35, 46].

Software Solutions. To avoid unnecessary recurring TLB flushes of invalidated PTEs, Uhlig tracks TLB versions and avoids shootdowns when the remote cores already performed full TLB flushes after the PTE changed [43, 44]. However, the potential of this approach is limited since even when TLB invalidations are batched, the TLB is flushed shortly after the last PTE is modified.

An alternative approach for reducing TLB flushes is to require applications to inform the OS how memory is used or to control TLB flushes explicitly. Corey OS avoids TLB shootdowns of private PTEs by requiring that user applications define which memory ranges are private and which are shared [10]. C4 uses an enhanced Linux version that allows applications to control TLB invalidations [40]. These systems, however, place an additional burden on application writers. Finally, we should note that reducing the number of memory mapping changes, for example by improving the memory reclamation policy, can reduce the number of TLB flushes. However, these solutions are often workload dependent [45].

7. Hardware Support

Although our system saves most of the TLB shootdowns, it does introduce some overheads. Hardware support that would allow privileged OSes to insert PTEs directly to the TLB without setting the access-bit would eliminate most of ABIS's overhead. Such an enhancement should be easy to implement as we achieve an equivalent behavior in software.

ABIS would able to save even more TLB flushes if CPUs avoid setting the PTE access-bit after the PTE is cached in the TLBs. We encountered, however, in situations where such events occur. It appears that when Intel CPUs set the PTE dirty-bit due to write access, they also set the access-bit, even if the PTE is already cached in the TLB. Similarly, before a CPU triggers a page-fault, it performs a page-walk to retrieve the updated PTE from memory and may set the access-bit even if the PTE disallows access. Since x86 CPUs invalidate the PTE immediately after, before invoking the page-fault exception handler, setting the access-bit is unnecessary.

CPUs should not invalidate the TLB unnecessarily, as such invalidations hurt performance regardless of ABIS. ABIS is further affected, as these invalidations cause the the access-bit to be set again when the CPU re-caches the PTE. We found that Intel CPUs (unlike AMD CPUs) may perform full TLB flushes when virtual machines invalidate huge pages that are backed by small host pages.

8. Conclusion

We have presented two new software techniques that prevent TLB shootdowns in common cases, without replicating the mapping structures and without incurring more page-faults. We have shown its benefits in a variety of workloads. While our system introduces overheads in certain cases, these can be reduced by minor CPU enhancements. Our study suggests that providing OSes better control over TLBs may be an efficient and simple way to reduce TLB coherency overheads.

Availability

The source code is publicly available at: `http://nadav.amit.to/publications/tlb`.

Acknowledgment

This work could not have been done without the continued support of Dan Tsafrir and Assaf Schuster. I also thank the paper reviewers and the shepherd Jean-Pierre Lozi.

References

[1] Lukasz Anaczkowski. Linux VM workaround for Knights Landing A/D leak. Linux Kernel Mailing List, lkml.org/lkml/2016/6/14/505, 2016.

[2] 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.

[3] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 48–59, 2010.

[4] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013.

[5] 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.

[6] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level TLBs for chip multiprocessors. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 62–63, 2011.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 72–81, 2008.

[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] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., 2005.

[10] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 43–57, 2008.

[11] 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.

[12] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 105–118, 2011.

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

[14] Jonathan Corbet. Memory compaction. LWN.net, https://lwn.net/Articles/368869/, 2010.

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

[16] Christopher Covington. arm64: Work around Falkor erratum 1003. Linux Kernel Mailing List, https://lkml.org/lkml/2016/12/29/267, 2016.

[17] Linux Kernel Driver DataBase. CONFIG_ARM_ERRATA_720789. http://cateee.net/lkddb/web-lkddb/ARM_ERRATA_720789.html, 2012.

[18] Jake Edge. Persistent memory. LWN.net, https://lwn.net/Articles/591779/, 2014.

[19] Balazs Gerofi, Akira Shimada, Atsushi Hori, and Yozo Ishikawa. Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 360–368, 2013.

[20] Jeff Gilchrist. Parallel data compression with bzip2. In *IASTED International Conference on Parallel and Distributed Computing and Systems (ICPDCS)*, volume 16, pages 559–564, 2004.

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

[22] Julien Grall. Force broadcast of TLB and instruction cache maintenance instructions. Xen development mailing list https://patchwork.kernel.org/patch/8955801/, 2016.

[23] Dave Hansen. Patch: x86: set TLB flush tunable to sane value. https://patchwork.kernel.org/patch/4460841/, 2014.

[24] Xeon phi processor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[25] 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.

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with Ingens. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 705–721, 2016.

[27] Yong Li, Rami Melhem, and Alex K Jones. PS-TLB: Leveraging page classification information for fast, scalable and efficient translation for future CMPs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):28, 2013.

[28] Likai Liu. Parallel computing and the cost of TLB shoot-down. http://lifecs.likai.org/2010/06/parallel-computing-and-cost-of-tlb.html, 2010.

[29] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1), 2013.

[30] Ophir Maor. What is CPU affinity? https://community.mellanox.com/docs/DOC-1924, 2014.

[31] Ophir Maor. Mellanox BIOS performance tuning example. https://community.mellanox.com/docs/DOC-2297, 2015.

[32] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[33] Jinzhan Peng, Guei-Yuan Lueh, Gansha Wu, Xiaogang Gou, and Ryan Rakvic. A comprehensive study of hardware/software approaches to improve TLB performance for Java applications on embedded systems. In *ACM Workshop on Memory System Performance and Correctness (MSPC)*, pages 102–111, 2006.

[34] Aristeu Rozanski. VM-scalability benchmark suite. https://github.com/aristeu/vm-scalability, 2010.

[35] Anand Lal Shimpi. AMD's B3 stepping Phenom previewed, TLB hardware fix tested. AnandTech http://www.anandtech.com/show/2477/2, 2008.

[36] Kirill A. Shutemov. mm: map few pages around fault address if they are in page cache. Linux Kernel Mailing List, https://lwn.net/Articles/588802, 2014.

[37] Kirill A. Shutemov. unixbench.score -6.3% regression. Linux Kernel Mailing List, http://lkml.kernel.org/r/20160613125248.GA30109@black.fi.intel.com, 2016.

[38] Patricia J Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, June 1990.

[39] Patricia J Teller, Richard Kenner, and Marc Snir. *TLB consistency on highly-parallel shared-memory multiprocessors*. Courant Inst. of Math. Sci, 1987.

[40] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *ACM International Symposium on Memory Management (ISMM)*, pages 79–88, 2011.

[41] Michael Y Thompson, JM Barton, TA Jermoluk, and JC Wagner. Translation lookaside buffer synchronization in a multiprocessor system. In *USENIX Winter*, pages 297–302, 1988.

[42] Linus Torvalds. Splice: fix race with page invalidation. http://yarchive.net/comp/linux/zero-copy.html, 2008.

[43] 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.

[44] Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *ACM SIGOPS Operating Systems Review (OSR)*, 41(4):49–58, 2007.

[45] Ahsen J Uppal and Mitesh R Meswani. Towards workload-aware page cache replacement policies for hybrid memories. In *International Symposium on Memory Systems (MEMSYS)*, pages 206–219, 2015.

[46] Theo Valich. Intel explains the Core 2 CPU errata. The Inquirer http://www.theinquirer.net/inquirer/news/1031406/intel-explains-core-cpu-errata, 2007.

[47] Brian Van Essen, Henry Hsieh, Sasha Ames, and Maya Gokhale. DI-MMAP: A high performance memory-map runtime for data-intensive applications. In *IEEE International Workshop on Data-Intensive Scalable Computing Systems (SCC)*, pages 731–735, 2012.

[48] 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 *ACM/IEEE International Conference on Parallel Architecture & Compilation Techniques (PACT)*, pages 340–349, 2011.

[49] 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.

[50] wrk. HTTP benchmarking tool. https://github.com/wg/wrk, 2015.