

Topology-aware TLB Shutdown Protocols: It's more complicated than you think

Ning Ding

University of British Columbia
Vancouver, Canada

Margo I. Seltzer

University of British Columbia
Vancouver, Canada

Reto Achermann

Technical University of Munich
Munich, Germany

Abstract

Translation Lookaside Buffers (TLBs) cache recent virtual-to-physical address translations. Each core has a private TLB. Because TLBs are not coherent, the OS must perform a TLB shutdown to invalidate stale translations when page-table entries change. TLB shutdowns are a major scalability bottleneck on large multicore systems. Prior work focused on avoiding expensive large-scale shutdowns by target-set reduction, deferred invalidation, batching, and early acknowledgments. However, the effect of hardware topology on TLB shutdown latency has been largely ignored.

We explore the effects of the hardware topology and the broadcast structure on the TLB shutdown latency in Linux's TLB shutdown mechanism. Our results show that topology-aware propagation can reduce TLB shutdown latency for large target sets by up to 1.9× over Linux's default mechanism. However, no single propagation structure is universally optimal. Performance depends on the hardware topology, workload behavior, and core count. Eager removal of cores from the target set further improves performance.

CCS Concepts: • **Software and its engineering** → **Virtual memory**; **Communications management**; **Multi-processing** / **multiprogramming** / **multitasking**.

Keywords: TLB Shutdown. Multi-processor Systems. Virtual Memory.

ACM Reference Format:

Ning Ding, Margo I. Seltzer, and Reto Achermann. 2026. Topology-aware TLB Shutdown Protocols: It's more complicated than you think. In *14th Workshop on Programming Languages and Operating Systems (PLOS '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3831586.3838155>

1 Introduction

Virtual memory enables flexible memory management and process isolation. On each memory access, the memory management unit (MMU) translates virtual to physical addresses

by walking the page table, a multi-level radix tree. Translation Lookaside Buffers (TLBs) are per-core caches that store recently used mappings to avoid repeated page-table walks.

TLBs are non-coherent: whenever the OS updates the page table entries, a TLB on any core might still cache the now-stale, previous mapping, allowing the process to access memory it should no longer be able to access. To ensure TLB coherence, the OS must invalidate all possible stale mappings in the TLBs of all processor cores in the system — a procedure called *TLB shutdown*. TLB shutdowns can incur significant overhead. Prior work optimized TLB shutdowns through target-set reduction using page-access tracking [3] or page-table replication [11], deferred or automatic TLB invalidation [8, 21], improved notification mechanisms [4], or shutdown avoidance in specific cases [4]. However, none of these works considered the effect of the hardware topology on TLB shutdown.

Mainstream operating systems, including Linux [18], Solaris [13], OpenBSD [23], and XNU [6], exhibit TLB shutdown initiator contention. They implement an inherently sequential TLB shutdown procedure that concentrates coordination on a single core. The initiator thread of the TLB shutdown must prepare and enqueue callback requests for all remote cores that might have cached stale translations, then send inter-processor interrupts (IPIs) to notify them, and finally wait for them to complete. Although IPI delivery is partially parallel on some architectures, the callback preparation, queue insertion, and completion synchronization remain serialized on the initiating core [19].

We address this initiator bottleneck by using a tree-based broadcast structure in the TLB shutdown protocol. The key idea is to break the initiator's serialized request preparation and enqueueing work into smaller propagation tasks and construct a hardware-aware broadcast tree that leverages hardware parallelism to speed up the TLB shutdown. Instead of coordinating the TLB shutdown sequentially and sending IPIs to target CPUs on a single core, the initiator first notifies a set of propagator cores that then forward the TLB shutdown concurrently to their assigned targets.

We implement and evaluate three topology-aware broadcast strategies for TLB shutdowns in Linux on two 192-logical-CPU platforms: a 4-socket Intel Xeon server and a 2-socket AMD EPYC server, using both microbenchmarks and application-level workloads.



This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2877-8/2026/09

<https://doi.org/10.1145/3831586.3838155>

Algorithm 1 Simplified Linux TLB Shutdown Dispatch

```

1: target_cores ← ∅
2: for each target core in mm_cpumask(mm) do
3:   if target core is not lazy then
4:     add target core to target_cores
5:     acquire CSD structure
6:     enqueue callback into target core's queue
7:   end if
8: end for
9: send IPIs to all target_cores
10: for each target core in target_cores do
11:   wait until target core acknowledges completion
12: end for

```

Our results show that broadcast topology can accelerate TLB shutdowns by up to 1.9× over Linux’s default protocol. Eagerly removing cores from the target set further improves latency; at the application level, this accounts for a large fraction of the performance improvement, exposing a second factor in TLB shutdown scalability. Crucially, no single strategy wins across all platforms and workloads: a strategy that helps one workload-machine combination can hurt another. Automatically selecting the optimal strategy based on workload and machine topology characteristics is nuanced, and we leave this for future work.

We make the following contributions:

- A characterization of the TLB shutdown cost on Linux showing initiator contention as a scalability bottleneck.
- We design and implement three tree-based broadcast strategies in the x86 Linux kernel.
- We empirically evaluate these strategies across two hardware platforms using micro- and application benchmarks.
- We identify eager reclamation of cores from the target set as a second factor in TLB shutdown scalability.

2 Background: TLB Shutdown on Linux

Translation Lookaside Buffers (TLBs) are non-coherent caches of recently used translations. This means that the operating system must invalidate stale TLB entries whenever it updates a page table entry. To do so, the Linux kernel initiates a TLB shutdown. On x86, this is a software-driven protocol that invalidates the corresponding TLB entries on all cores in the system that might have the translation cached.

During a TLB shutdown, the initiator sends IPIs to other cores that might hold stale mappings, interrupting their execution to trigger remote TLB invalidation. Linux tracks these targets with a bitmap mask (`mm_cpumask`), recording which cores recently used the address space and might therefore contain stale translations. The Linux kernel executes a TLB shutdown as follows (Algorithm 1). First, the initiator identifies the target set by reading `mm_cpumask`. Before dispatch, Linux filters out CPUs in lazy TLB mode, because they are not actively using the address space and can defer invalidation until a future context switch. However, these CPUs remain in `mm_cpumask`; the filter only excludes them from the current

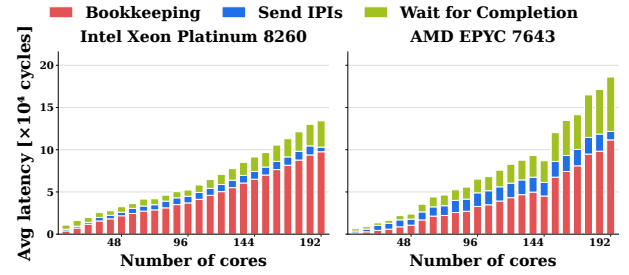


Figure 1. Latency breakdown of Linux TLB shutdown under the `madvise()` microbenchmark. The x-axis shows the shutdown target-set size, and the y-axis shows the average latency per shutdown.

shutdown. As a result, lazy CPUs can continue to appear in future target sets until they are explicitly removed from the bitmap on context switching. The initiator then acquires the corresponding Call Single Data (CSD) structures, marks them busy, and enqueues them into the target cores’ callback queues. Finally, it sends IPIs to the target cores and waits for completion. Each target signals completion by clearing its CSD busy bit after the TLB invalidation, while the initiator polls until all such bits are clear.

Although some architectures provide partial hardware assistance for TLB invalidation, such as AMD `INVLPGB` [2] and ARM broadcast TLBI instructions [7], software-based IPI broadcasts remain necessary on many platforms and are also used by CPU hotplugging, RCU, and scheduling.

3 The Cost of TLB Shutdown

We demonstrate the need for a topology-aware TLB shutdown by first showing initiator contention in the native Linux protocol (§3.1) and then evaluating the effects of the hardware topology on IPI propagation (§3.2).

3.1 Initiator Bottleneck

We now investigate the latency contributions of each part of Algorithm 1 by instrumenting the `smp_call_function_*()` function in the kernel and measuring three phases:

1) *Bookkeeping* covers waiting for each target’s CSD to become available, filling it with the flush metadata, and enqueueing it on the target CPU’s callback queue.

2) *Send IPIs* measures the time spent issuing the interrupts through the Interrupt Control Register (ICR).

3) *Wait for completion* begins after IPI dispatch and ends when all target CSD busy bits have been cleared. This phase includes IPI propagation, interrupt-handler entry, callback execution, TLB invalidation, and completion signaling.

Figure 1 shows the latency breakdown as the number of target cores increases. We observe that bookkeeping latency (acquiring and enqueueing CSDs) scales linearly with the

number of cores involved and is the dominant factor in TLB shutdowns, accounting for up to 74% of the total duration.

As Linux acquires the CSDs and atomically enqueues them on the target CPUs, one target core at a time, the cost of centralized SMP-call-dispatch bookkeeping grows with the size of the target set and becomes prohibitive on large machines. Although Linux avoids sending a new IPI when the target queue is non-empty, allowing batching of multiple callbacks behind a single interrupt, each initiator must still perform the atomic queue insertion [20].

3.2 Non-Uniform IPI Send and Propagation Delays

Modern processors have hierarchical topologies that can produce different communication latencies across sockets and chiplets within a socket.

While both of our evaluation platforms have 192 logical cores, their architectures differ significantly, as shown in Figure 2. The Intel Xeon Platinum 8260 system has four sockets, each with 48 logical cores that share an L3 cache [14]. The AMD EPYC 7643 platform has two sockets, each partitioned into eight Core Complex Dies (CCDs), each with its own L3 cache shared by twelve logical cores [1]. Thus, AMD has an additional CCD/L3 level within each socket. Both platforms expose one OS-visible NUMA node per socket, so NUMA-node and socket boundaries refer to the same partition of CPUs throughout this paper.

We evaluate the IPI send time and propagation latency across both platforms using a pairwise experiment similar to that of Smelt [15]. Instead of characterizing cache-to-cache communication latency, we focus on pairwise IPI latency between cores. Figure 3 shows the average send and propagation latencies across 1000 trials.

Propagation Time. Propagation time measures the interval between IPI issuing and the target CPU entering the interrupt handler. On both platforms, inter-NUMA IPIs are substantially more expensive than intra-NUMA IPIs, with latency increasing by up to 3.5×. The Intel Xeon platform shows relatively uniform intra-NUMA IPI latency with little variance. In contrast, the AMD EPYC platform, due to its chiplet-based design, exhibits different IPI latencies depending on whether communication crosses a chiplet boundary. We observe that cross-chiplet latency in the same NUMA node is up to 2× higher than communication within the same chiplet. On both platforms, sending an IPI to the sibling SMT thread of the same core has the lowest latency.

Send Time. This measures the time the initiator core spends sending the IPI, i.e., the time until `smp_call_function_*` returns without waiting for the handler to execute on the target core. We observe that the send time follows a pattern similar to the propagation delay: the IPI's sending time depends on the target core. We attribute this to CSD enqueueing, which is sensitive to memory locality. For each target, the initiator updates a target-indexed CSD and enqueues it on the target CPU's per-CPU callback queue. Because each

target-indexed CSD resides in the target CPU's per-CPU memory area, notifying a CPU in another NUMA node or L3 domain requires expensive remote memory accesses and cache-line ownership transfers, increasing latency.

4 Design and Implementation

We now present the design and implementation of topology-aware hierarchical propagation in the Linux TLB shutdown path. We begin by discussing the design goals (§4.1), followed by the three propagation strategies (§4.2) and the implementation details (§4.4).

4.1 Design Principles

Recall that in a large TLB shutdown, the initiator becomes a bottleneck, because it must serially coordinate all of the sending work on the same core. Instead of having a single CPU send all IPI messages, we distribute sending work across a few propagator cores, which forward requests in parallel. This increases the parallelism of sending and reduces the work the initiator must perform. However, hierarchical broadcast topologies also introduce additional forwarding and communication overhead. As a result, their effectiveness depends heavily on the underlying hardware topology and the inter-core communication cost. Thus, we design broadcast structures that follow these design principles:

P1: Distribute Work to Leverage Parallelism. Distribute the work among a set of propagator cores.

P2: Minimize the Longest Propagation Path. Balance the work distribution and maximum number of hops.

P3: Adapt to the Hardware Topology. Tailor the broadcast structure to the hardware topology.

P4: Reduce Target Set. Eagerly remove cores from the target set during TLB shutdown.

4.2 Tree-Based Broadcast Topologies

We design three tree-based propagation structures that adhere to the design principles. The key idea is to distribute request-sending work across multiple cores rather than having a single initiator core send requests directly to all targets. Our broadcast topologies organize request propagation based on the hardware topology to reduce forwarding costs. Figure 2 illustrates the topologies.

4.2.1 Static Tree Topologies. We first describe two tree topologies (Figure 2 (a) and (b)) that statically follow the hardware topology, i.e., taking the processor and chiplet boundaries into account.

NUMA-clustered. For each processor, we select a small set of propagator cores and evenly assign the remaining target cores among the initiator and propagators. During a TLB shutdown, the initiator sends requests only to the propagators, which then forward the shutdown to their assigned cores. This creates a two-level hierarchy that distributes the dispatch work across multiple cores while keeping most

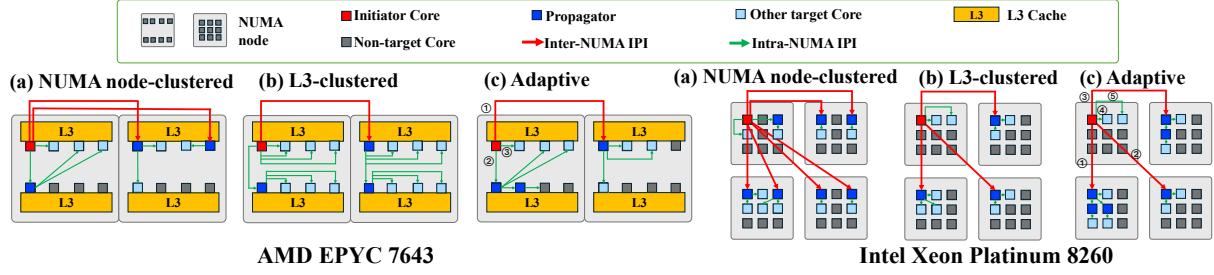


Figure 2. Illustration of the *simplified* hardware topologies of the Intel and AMD platforms with the topology-aware TLB shutdown broadcast topology overlaid. For the Adaptive topology, the numbered arrows indicate the order of request propagation, illustrating that the adaptive tree preserves a specific sending sequence.

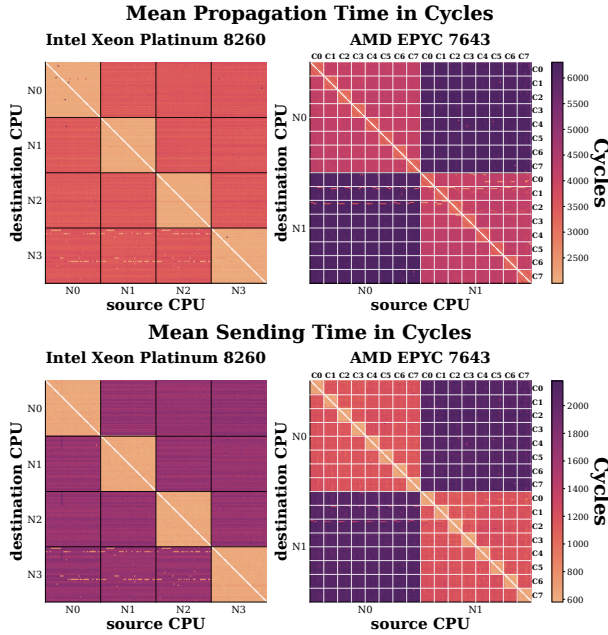


Figure 3. Pairwise IPI send and propagation latency between cores grouped by NUMA nodes and L3 caches.

communication within the same NUMA node. In our implementation, we use up to four propagators per NUMA node; Figure 2 shows only two for clarity.

L3-clustered. We extend NUMA-clustering by introducing an extra layer based on L3 cache domains. In a chiplet architecture, cores within the same CCD typically share an L3 cache. For each L3 cluster, we select one propagator core and assign the remaining cores to it. During a TLB shutdown, the initiator first sends requests to the NUMA-level propagators. Each NUMA propagator then forwards the request to the L3 propagators within its NUMA node, which finally notify the remaining cores in their respective L3 clusters.

On our Intel platform, all CPUs in a socket share one L3 cache. Thus, L3-clustered is equivalent to NUMA-clustered

configured with one propagator per node on that platform, but with additional implementation overheads.

4.2.2 Adaptive Tree Topology. For each possible initiator core, we precompute a separate adaptive TLB shutdown tree using the Smelt generator [15] and our measured pairwise IPI sending and propagation latencies, as illustrated in Figure 2(c).

The generator simulates propagation from the initiator, allowing each core to forward the request after receiving it. It reaches each remote NUMA node through the lowest-cost core in that node, prioritizing high-cost nodes, and then propagates within the node. It starts costly transfers early, uses only one inter-NUMA communication per remote node, and distributes the request across cores in the same NUMA node.

This approach increases intra-processor parallelism while reducing the number of expensive inter-socket communication.

4.3 Partial Tree Construction, Propagator Selection, and Load Balancing

A TLB shutdown typically involves only a subset of cores. We therefore build the trees at dispatch time according to the cpumask, ignoring non-participating cores while preserving the propagation paths to the target cores.

Concurrent TLB shutdowns can result in propagator contention. Selecting the same propagators for every TLB shutdown concentrates propagation work on a small number of cores. This overwhelms the propagator cores, leading to increased latency. To mitigate this effect, the NUMA-node clustered tree selects four participating CPUs per NUMA node as propagators using a round-robin policy and distributes the remaining targets evenly among them. This reduces each propagator’s workload and prevents successive shutdowns from repeatedly concentrating forwarding work on the same set of CPUs. The L3-clustered tree instead selects one propagator per CCD, with one also serving as the first-layer propagator for the NUMA node. Our AMD platform has 8 CCDs

per NUMA node and 12 CPUs per CCD. The first-layer propagator forwards to at most seven other CCD propagators and 11 CPUs in its own 12-CPU CCD, while each second-layer propagator handles at most 11 other CPUs. Since these fan-outs are already small, and our measurements in [Section 3.2](#) show little latency variation within an L3 group, we select propagators statically by choosing the lowest-indexed participating CPU, rather than using round-robin selection.

4.4 Implementation in Linux

We integrate the broadcast topologies into the Linux 5.15 TLB shutdown path by adding a hook to `flush_tlb_mm_range()`. This allows selecting the propagation strategies at runtime, providing an extensible framework implementing other topologies. When the hook is disabled, the system falls back to Linux's native TLB shutdown path. The initiator notifies the first-level propagators through the `smp_call_function_*`. Since invoking the interface from an IPI handler can lead to deadlocks, propagators use `irq_work` to notify the next level of targets. We use Linux's native TLB invalidation handlers to perform the actual flushing operations. Similar to Linux's native implementation, all topology metadata is pre-allocated in per-CPU data structures, eliminating memory allocation and synchronization on the TLB shutdown critical path.

5 Evaluation

We evaluate the effect of topology-aware propagation on TLB shutdown performance in [Section 5.2](#), while [Sections 5.3](#) and [5.4](#) show how shutdown latency affects application performance using Sysbench and Apache webserver.

Experimental Results Summary. The microbenchmark shows that the broadcast topology matters for large target sets, while Sysbench shows that waiting for remote completion can significantly affect performance. Apache reveals an orthogonal factor: shutdown cost is highly sensitive to target-set size. Retaining lazy CPUs in the target set indiscriminately can add substantial overhead on subsequent shutdowns ([Section 2](#)). Overall, a scalable shutdown design depends on the hardware topology, target-set size, broadcast topology, and workload.

5.1 Setup

We conduct all experiments on the two server-class platforms described in [Section 3.2](#). For all experiments, the CPU frequency governor is set to PERFORMANCE and Turbo Boost is disabled. We repeat each experiment three times. Intel results show little variation, while AMD results vary more but still exhibit stable trends.

We use Linux version 5.15 and compare against its native TLB shutdown protocol as the baseline.

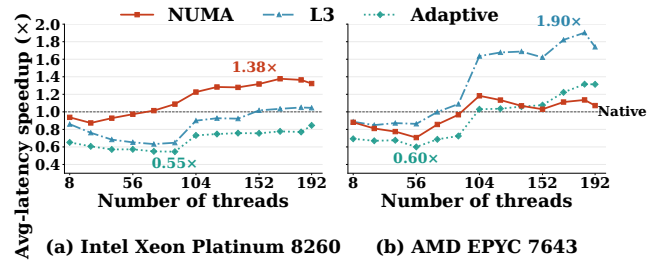


Figure 4. Normalized user-level `madvise()` latency speedup relative to native Linux TLB shutdown.

5.2 Microbenchmark

Methodology. The microbenchmark uses N threads, each pinned to a distinct logical core. One thread unmaps pages with `madvise(MADV_DONTNEED)` to trigger TLB shutdowns. To keep the target-set size stable, the remaining $N - 1$ threads use the address space on their respective cores.

We scale the number of threads and measure latency as the time from entering the system call to returning. We pin threads to cores and place them across NUMA nodes in round-robin order. We place one thread on each physical core before assigning additional threads to SMT siblings.

Results. [Figure 4](#) shows the latency speedup over Linux's default TLB shutdown. At small core counts, native Linux is best, because dispatch is still cheap, whereas tree-based strategies incur fixed costs for propagator selection and additional propagation delay. At larger core counts, initiator-side bookkeeping (e.g., cross-NUMA queue operations) becomes expensive, and tree-based broadcasts become beneficial.

However, the best topology depends on the platform. While NUMA-clustered performs best on Intel (up to 1.38 $\times$ speedup), L3-clustered performs best on AMD, reaching 1.90 $\times$ speedup. Adaptive performs poorly on both platforms, because the initiator-side savings do not amortize the additional pruning and forwarding overhead required to construct and traverse the adaptive tree.

We further observe interesting behavior on the AMD machine: First, the L3-clustered strategy has a noticeable performance drop at 152 threads. The cause remains unclear and needs further investigation. Second, all tree-based strategies except NUMA-clustered show a distinct increase in relative performance beyond 96 threads.

Since both platforms have 96 physical cores, additional threads begin using SMT threads. We attribute this increase to greater contention for the initiator's core-local resources, which affects the native strategy more because it concentrates dispatch on one CPU, while tree-based strategies distribute it across multiple cores.

Conclusion. The microbenchmark shows that the broadcast topology matters for large TLB shutdowns. Further, the optimal broadcast structure depends on the underlying hardware topology.

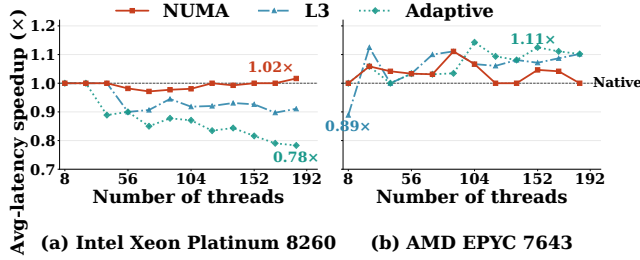


Figure 5. Normalized average per-operation Sysbench latency speedup relative to native Linux TLB shutdown.

5.3 Sysbench

Methodology. We configure Sysbench in random-write mode with memory-mapped I/O. We scale the number of worker threads that concurrently issue random writes through mmap’ed regions. The benchmark invokes `fsync()` to ensure data persistence, during which the kernel marks the pages as write-protected and finally triggers TLB shutdowns.

Results. Figure 5 shows the latency speedup over Linux’s default TLB shutdown. On AMD, tree-based propagation improves latency by up to 1.11×. Adaptive and L3-clustered show similar performance trends. On Intel, however, tree-based broadcast topologies show significantly lower performance than Linux’s default TLB shutdown for large core counts. The results suggest a possible contributing factor: on Intel, waiting time appears to dominate shutdown latency, while tree-based policies primarily reduce sender-side contention but may increase remote completion time. This trade-off appears less pronounced on AMD.

Conclusion. The results are consistent with a phase trade-off: hierarchical propagation helps when a reduction in sender-side contention outweighs forwarding overhead, but loses its advantage when waiting for remote completion or other kernel work dominates. Therefore, whether a tree-based strategy improves application performance depends not just on the hardware topology, but also on which phase of the shutdown dominates for a given workload and platform.

5.4 Apache Web Server

We evaluate the effect of the TLB shutdown broadcast topology on the throughput of the Apache Web Server.

Methodology. Following prior work [4], we run Apache with the event MPM and generate load using `wrk`. In this mode, listener threads handle connection management, while worker threads mainly process active requests [5]. Request processing frequently modifies memory mappings, triggering page-table updates and TLB shutdowns. We scale the number of Apache worker threads and report the bandwidth measured by `wrk`.

Results. Figure 6 (top) shows Apache’s throughput increase. The L3-clustered approaches result in the best performance, reaching 1.14× and 1.06× on Intel and AMD, respectively.

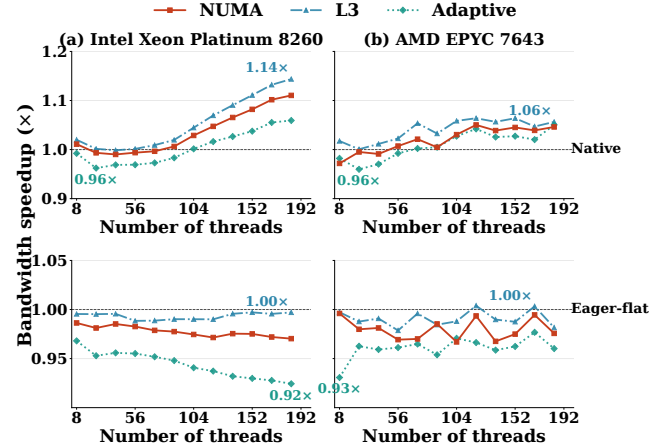


Figure 6. Apache throughput speedup. The top row normalizes the three tree strategies against native Linux; the bottom row normalizes them against an *eager-flat* baseline, i.e., the native Linux approach, but with the lazy-TLB filter removed to perform eager target-set cleanup.

At first, the topology appears to account for most of this increase. However, we observed an interesting side effect of the eager core removal from the cpumask: the TLB shutdown target set size decreases significantly with tree-based topologies, which accounts for a large fraction of the performance gain. In contrast, with the lazy-TLB filter, Linux skips cores to reduce IPI overhead, but this also delays the callback that would remove the core from the cpumask, thereby gradually increasing the target set size, making future TLB shutdowns more expensive.

Eagerly Cleaning Up the target set. We repeat the same experiment, but select a new baseline: *eager-flat*, which is Linux’s native flat topology with the lazy filter removed. Apache’s event MPM uses many threads but runs only a few at a time, leaving many CPU cores idle and making eager mask cleanup especially important.

Figure 6 (bottom) shows the resulting performance relative to this new baseline. Because cores are eagerly removed from the cpumask, the target set size decreases significantly, thereby reducing the send-side bottleneck. Consequently, tree-based structures show no benefit on either the Intel or the AMD platforms. The results are consistent with the microbenchmark data from Figure 4.

Conclusion. For workloads that leave many CPUs in lazy TLB mode, such as the Apache web server, eagerly removing them from the cpumask significantly impacts performance. Application performance improvements can come from target-set maintenance, not only from broadcast topology. For small target sets, a tree-based broadcast does not pay off.

6 Discussion and Future Work

Our results show that no single propagation strategy is universally best: broadcast topology is a real but conditional factor on TLB shutdown latency. Tree-based strategies can improve TLB shutdown latency with large target sets and initiator contention but degrade performance when remote completion becomes the bottleneck or with small target sets.

Automatic Selection of Topologies. Our results show that the best propagation structure depends on the platform, workload, and shutdown target-set size. An ideal system would dynamically select the topology using a cost model that balances request sending parallelism against the additional forwarding and completion overhead introduced by hierarchical propagation. Developing such a lightweight and accurate selection policy is left to future work.

Adapting Smelt [15] to TLB shutdowns. Smelt produces efficient broadcast topologies. However, TLB shutdowns require a more lightweight runtime model. Smelt assumes a static core set and communication costs when constructing broadcast trees. However, for TLB shutdowns, the target set and communication latencies vary with the workload. We precompute a full Smelt tree at boot and prune it for each shutdown to avoid generating the tree on the hot path. Unfortunately, this naive pruning does not preserve all of the tree's benefits. As a result, the Smelt tree is not the best fit for minimizing the TLB shutdown latency, and a dynamic, lightweight tree still needs further investigation. A possible approach is to periodically measure TLB shutdown phase costs. When sending costs change, a worker uses a cheaper algorithm to rebuild and cache the tree off the shutdown path. Subsequent shutdowns use the latest cached tree.

Inter-socket and intra-socket message ordering. In our current design, reaching a remote NUMA node target begins with an inter-socket IPI from the initiator, while subsequent propagation remains intra-socket. Future work could reverse this order, allowing the initiator to issue only intra-socket IPIs and the propagators to issue inter-socket ones. This could parallelize inter-socket dispatch, but whether that would further improve latency warrants investigation.

Broader workloads and platforms. During our evaluation, we also tested several other workloads, but many TLB shutdowns involved relatively few CPUs. For example, the RocksDB target set contained fewer than 100 CPUs, even with 192 threads. Tree-based propagation is most useful when many threads in the same process are simultaneously active and frequently trigger TLB shutdowns. Moreover, on non-fully connected platforms, the broadcast topology may require additional constraints that influence tree construction. We leave this scenario for future work.

The Cost of Lazy TLB. Linux's lazy-TLB optimization avoids unnecessary IPIs, but stale entries can remain in the target set and continue to contribute to shutdown overhead. Our Apache results suggest that target-set maintenance might be

as important as the broadcast structure itself. Understanding this tradeoff is an interesting direction for future work.

Hardware Accelerated TLB Shutdown. Some architectures, e.g., Arm and recently AMD, provide hardware-based TLB shutdown support. We believe that similar techniques presented in this work could also be applicable to improve hardware-assisted TLB shutdowns.

Applicability to other Mechanisms. Our work focused on TLB shutdowns, as they are an important factor for performance in many workloads. However, there might be other uses that can leverage a fast, IPI-based notification mechanism, e.g., CPU hotplugging/offlining or read-copy-update grace periods for lockless synchronization. Hence, leveraging tree-based broadcast structures can further speed up those mechanisms in the kernel.

7 Related Work

TLB shutdown overhead has long been recognized as an important scalability problem on multicore systems [3, 21, 24, 26]. Prior work explored a wide range of approaches to mitigate this overhead, including shrinking the shutdown target set through more precise tracking of TLB residency [3, 11, 26], avoiding shutdowns under certain conditions [4, 8, 21, 25], batching multiple invalidations into fewer cross-core notifications [9, 12], and optimizing the execution of issued shutdowns [4]. Most of these approaches reduce the frequency or scale of expensive large shutdowns. However, none of these works investigated initiator contention or the effect of machine topology on TLB shutdown.

Optimizing tree-based communication has been explored in the context of NUMA-machines [10, 15, 16] and using MPI collective communication primitives [17, 22]. Specifically, we leverage Smelt's ideas [15] in constructing our adaptive tree for TLB shutdowns. While those approaches include optimizations for broadcasts, none of them include IPIs or have been applied to TLB shutdowns.

8 Conclusion

We demonstrated a scalability bottleneck in Linux's TLB shutdown protocol. Our results point to two largely independent factors in scalable shutdown: the topology-aware propagation structure, which parallelizes request sending and thus reduces initiator contention, and the size of the target set. This reduces TLB shutdown latency by up to 1.90× and increases application throughput by up to 14% on large multicore machines. Tree-based topologies are better for large target sets, while flat topologies are better for smaller ones. Ultimately, the best propagation structure depends on hardware topology, workload behavior, and the target set.

References

- [1] Advanced Micro Devices. [n. d.]. AMD EPYC 7643 Processor Specifications. <https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7643.html> Accessed May 2026.

- [2] Advanced Micro Devices. 2026. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Advanced Micro Devices. https://docs.amd.com/v/u/en-US/24593_3.44_APM_Vol2 Revision 3.44. Accessed May 2026.
- [3] Nadav Amit. 2017. Optimizing the TLB Shutdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, 27–39. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit>
- [4] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't Shoot Down TLB Shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. ACM, 35:1–35:14. doi:10.1145/3342195.3387518
- [5] Apache Software Foundation. [n. d.]. Apache MPM event – Apache HTTP Server Version 2.4. <https://httpd.apache.org/docs/2.4/mod/event.html> Accessed May 2026.
- [6] Apple Inc. 2026. XNU pmap TLB shutdown interface (pmap_flush_tlbs). https://github.com/apple-oss-distributions/xnu/blob/xnu-12377.101.15/osfmk/i386/pmap_internal.h#L2391-L2405 XNU tag xnu-12377.101.15. Accessed May 2026.
- [7] Arm Ltd. [n. d.]. *TLBI (A64)*. Arm Ltd. <https://developer.arm.com/documentation/dui0801/I/A64-General-Instructions/TLBI--A64-> Accessed May 2026.
- [8] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*. IEEE, 273–287. doi:10.1109/PACT.2017.38
- [9] Vinay Banakar. 2025. Re: [PATCH] mm: Optimize TLB Flushes During Page Reclaim. Linux Kernel Mailing List. <https://lkml.iu.edu/2501.2/05507.html>
- [10] Matthieu Dorier, Misbah Mubarak, Robert B. Ross, Jianping Kelvin Li, Christopher D. Carothers, and Kwan-Liu Ma. 2016. Evaluation of Topology-Aware Broadcast Algorithms for Dragonfly Networks. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER '16)*, 40–49. doi:10.1109/CLUSTER.2016.26
- [11] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. 2024. Scalable and Effective Page-table and TLB management on NUMA Systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 445–461. <https://www.usenix.org/conference/atc24/presentation/gao-bin-scalable>
- [12] Mel Gorman. 2015. TLB Flush Multiple Pages per IPI v4. Linux Kernel Mailing List. <https://lkml.org/lkml/2015/4/25/125>
- [13] illumos Developers. 2026. illumos hat_tlb_inval() TLB shutdown path. https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/i86pc/vm/hat_i86.c Accessed May 2026.
- [14] Intel Corporation. [n. d.]. Intel Xeon Platinum 8260 Processor Specifications. <https://www.intel.com/content/www/us/en/products/sku/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz/specifications.html> Accessed May 2026.
- [15] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. 2016. Machine-Aware Atomic Broadcast Trees for Multicores. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 33–48. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kaestle>
- [16] Christer Karlsson and Zizhong Chen. 2015. Optimising MPI Tree-Based Communication for NUMA Architectures. *International Journal of Autonomous and Adaptive Communications Systems* 8, 4 (Nov. 2015), 407–423. doi:10.1504/IJAACS.2015.073190
- [17] Shigang Li, Torsten Hoefler, and Marc Snir. 2013. NUMA-Aware Shared-Memory Collective Communication for MPI. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/2462902.2462903
- [18] Linux Kernel Developers. [n. d.]. Linux Kernel Source Code: arch/x86/mm/tlb.c. <https://elixir.bootlin.com/linux/v5.15.193/source/arch/x86/mm/tlb.c#L1004> Linux kernel version 5.15.193. Accessed May 2026.
- [19] Linux Kernel Developers. [n. d.]. Linux smp_call_function_many() implementation. <https://elixir.bootlin.com/linux/v5.15.193/source/kernel/smp.c#L974> Linux kernel version 5.15.193. Accessed May 2026.
- [20] Linux Kernel Developers. 2011. Linux Kernel Linked List API (llist.h). <https://elixir.bootlin.com/linux/v5.15.193/source/include/linux/llist.h> Linux kernel version 5.15.193. Accessed May 2026.
- [21] Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. 2020. ecoTLB: Eventually Consistent TLBs. *ACM Transactions on Architecture and Code Optimization* 17, 4, Article 27 (Sept. 2020), 24 pages. doi:10.1145/3409454
- [22] Emin Nuriyev, Juan Rico-Gallego, and Alexey Lastovetsky. 2022. Model-Based Selection of Optimal MPI Broadcast Algorithms for Multi-Core Clusters. *J. Parallel and Distrib. Comput.* 165 (July 2022), 1–16. doi:10.1016/j.jpdc.2022.03.012
- [23] OpenBSD Developers. 2026. OpenBSD pmap_tlb_shutdown() source. <https://github.com/openbsd/src/blob/master/sys/arch/amd64/amd64/pmap.c> OpenBSD source tree. Accessed May 2026.
- [24] Stijn Schildermans, Kris Aerts, Jianchen Shan, and Xiaoning Ding. 2020. Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency. In *2020 IEEE International Conference on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking*. 76–83. doi:10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00036
- [25] Frederic Schimmelpfennig, André Brinkmann, Hossein Asadi, and Reza Salkhordeh. 2024. Skip TLB Flushes for Reused Pages within mmap's. arXiv:2409.10946 [cs.OS]
- [26] Carlos Villavieja, Vasileios Karakostas, Lluís Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE, 340–349. doi:10.1109/PACT.2011.64