LATR: Lazy Translation Coherence

Mohan Kumar[*†] Steffen Maass[*†] Sanidhya Kashyap[†] Ján Veselý[‡] Zi Yan[‡]
Taesoo Kim[†] Abhishek Bhattacharjee[‡] Tushar Krishna[†]

[†]*Georgia Institute of Technology* [‡]*Rutgers University*

Abstract

We propose LATR—lazy TLB coherence—a software-based TLB shootdown mechanism that can alleviate the overhead of the synchronous TLB shootdown mechanism in existing operating systems. By handling the TLB coherence in a lazy fashion, LATR can avoid expensive IPIs which are required for delivering a shootdown signal to remote cores, and the performance overhead of associated interrupt handlers. Therefore, virtual memory operations, such as *free* and *page migration* operations, can benefit significantly from LATR's mechanism. For example, LATR improves the latency of `munmap()` by 70.8% on a 2-socket machine, a widely used configuration in modern data centers. Real-world, performance-critical applications such as web servers can also benefit from LATR: without any application-level changes, LATR improves Apache by 59.9% compared to Linux, and by 37.9% compared to ABIS, a highly optimized, state-of-the-art TLB coherence technique.

CCS Concepts • **Software and its engineering → Operating systems**; **Memory management**; **Virtual memory**;

Keywords TLB; Translation Coherence; Asynchrony.

1 Introduction

Translation lookaside buffers (TLBs) are frequently accessed per-core caches that store recently used virtual-to-physical address mappings. TLBs enable fast virtual address translation that is critical for application performance. Since TLBs are per-core caches, TLB entries should be kept coherent with their corresponding page table entries. The lack of hardware support for TLB coherence implies that software should provide the necessary coherence. In most existing systems, system software such as an operating system (OS) maintains the TLB coherence with the page table.

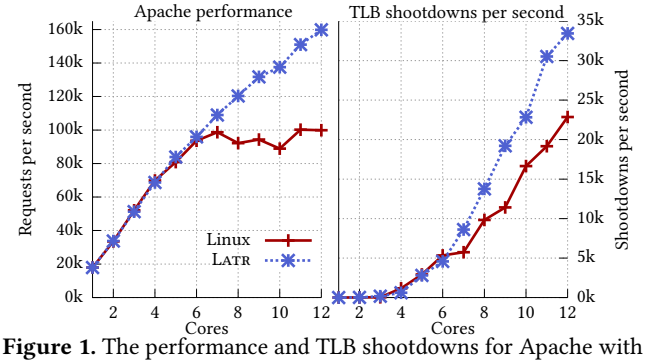


Figure 1. The performance and TLB shootdowns for Apache with Linux and LATR. LATR improves Apache's performance of serving 10 KB static web pages by 59.9%; it removes the cost of TLB shootdowns from the critical path and handles 46.3% more TLB shootdowns.

To provide TLB coherence, an OS performs a *TLB shootdown*, which is a mechanism to invalidate stale TLB entries on remote cores. TLB shootdowns are triggered by various virtual memory operations that modify page table entries, such as freeing pages (`unmap()`), page migration [20, 44], page permission changes (`mprotect()`), deduplication [49, 63], compaction [19], and Copy-on-Write operations (CoW).

Unfortunately, the existing TLB shootdown mechanism is very *expensive*—a shootdown takes up to 80 μs for 120 cores with 8 sockets and 6 μs for 16 cores with 2 sockets that is a widely used configuration in modern data centers [43]. This is mainly because most existing systems use expensive inter-processor interrupts (IPIs) [1] to deliver a TLB shootdown: e.g., an IPI takes up to 6.6 μs for 120 cores with 8 sockets and 2.7 μs for 16 cores with 2 sockets. Even worse, current TLB shootdown mechanisms handle invalidation in a *synchronous* manner. That is, a core initiating a TLB shootdown first sends IPIs to all remote cores and then waits for their acknowledgments, while the corresponding IPI interrupt handlers on the remote cores complete the invalidation of a TLB entry (see §2.1 for details).

Such an expensive TLB shootdown severely affects the overall performance of applications that frequently trigger memory management operations that change the page table [2, 62], such as web servers like Apache [4] and data analytic engines like MapReduce [25, 52]. For example, a typical Apache workload serving small static web pages or files

[*]Joint first authors

ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173198

[1]IPIs are the mechanism used in x86 to communicate with different cores.

does not scale beyond six cores with the current TLB shootdown mechanism in Linux (see Figure 1). Once alleviated by our new mechanism, Apache can handle 46.3% more TLB shootdowns and thus improve its throughput by 59.9%. More importantly, it is all possible without any application-level modifications.

To solve this problem, there have been two broad categories of research, namely, hardware- and software-based approaches. Hardware-based approaches strive to provide TLB cache coherence in hardware (see §2.2), but require expensive hardware modifications and introduce additional verification challenges to the microarchitecture, which is known to be bug-prone [3, 22, 24, 40, 53, 55, 61]. Software-based approaches, on the other hand (see §2.3), focus on reducing the number of necessary IPIs to be sent, either by batching TLB invalidations (e.g., identifying the sharing cores [2]), or using alternative mechanisms instead of IPIs (e.g., message passing [10]). However, current software-based approaches still handle TLB shootdowns *synchronously* and do not eradicate the overheads associated with the TLB shootdown. It means that, even with a message-passing alternative [10], a core initiating a TLB shootdown should wait for acknowledgments from participating remote cores. A synchronous TLB-shootdown mechanism increases the latency by several micro-seconds for certain virtual address operations, which is known to a culprit that contributes to the tail latency of some critical services in data centers [9].

To solve this inherent synchronous behavior of TLB shootdowns, we propose a software-based, *lazy shootdown* mechanism, called LATR, that can asynchronously maintain TLB coherence. The key idea of LATR is to use *lazy memory reclamation* and *lazy page table unmap* to perform an asynchronous TLB shootdown. By handling TLB shootdowns in a lazy fashion, LATR can eliminate the performance overheads associated with IPI mechanisms as well as the waiting time for acknowledgments from remote cores. In addition, as a software mechanism, LATR is readily implementable in commodity OSes. In fact, as a proof-of-concept, we implement LATR in Linux 4.10.

We enumerate in Table 1 the operations in which a lazy TLB shootdown is possible. For *free* operations, such as `munmap()` and `madvise()`[2], lazy memory reclamation enables a lazy TLB shootdown. Similarly, a lazy TLB shootdown is applicable to *migration* operations, such as AutoNUMA page migration, where page table entries can be lazily unmapped to enable a lazy TLB shootdown. However, LATR's lazy approach is not applicable to operations such as permission changes, ownership changes, and remap (`mremap()`), where the page table changes should be synchronously applied to the entire system. LATR supports common operations, such as free and migration, and improves real-world applications such as Apache, Graph500, PBZIP2, and Metis. In addition,

[2]For example for the case of `MADV_DONTNEED` and `MADV_FREE`.

Classification	Operations	Lazy operation possible
Free	`munmap()`: unmap address range	✓
	`madvise()`: free memory range	✓
Migration	AutoNUMA [20]: NUMA page migration	✓
	Page swap: swap page to disk	✓
	Deduplication [49, 63]: share similar pages	✓
	Compaction [19]: physical pages defrag.	✓
Permission	`mprotect()`: change page permission	-
Ownership	CoW: Copy on Write	-
Remap	`mremap()`: change physical address	-

Table 1. Overview of virtual address operations and whether a lazy TLB shootdown is possible. A lazy TLB shootdown is not possible when PTE changes should be immediately applied to the entire system for their correct behavior.

the proposed lazy migration approach can play a critical part in emerging systems using heterogeneous memory where pages are migrated to faster on-chip memory [14, 42, 44] and in emerging disaggregated memory systems in data centers where pages are swapped to remote memory using RDMA [27, 36].

However, there are a few challenges in handling TLB coherence in a lazy manner. First and foremost, LATR should guarantee the correctness of the new, lazy approach (i.e., how does LATR ensure that stale entries do not have negative, or adversarial impacts on the kernel and to applications?). We laid out the correctness sketch in §4.2 and §4.3. Second, a lazy TLB mechanism should make non-trivial design decisions: how the shootdown information is communicated to the remote cores without relying on IPIs, and when the remote cores should invalidate their TLB entries (§4.1).

We developed LATR as a proof-of-concept in Linux 4.10, and compared it with both Linux 4.10 and ABIS [2], a recent, state-of-the-art approach to reduce the number of TLB shootdowns that can be complementary to LATR. LATR makes the following contributions:

- LATR provides a lazy TLB shootdown for *free* operations using a lazy memory reclamation mechanism, and for *migration* operations using a lazy page table unmap mechanism.
- We also reason about why such lazy operations are still correct for both free and migration operations in commodity operating systems.
- We demonstrate LATR's approach is effective on both small (2 sockets, 16 cores) and large NUMA machines (8 sockets, 120 cores) when running real-world applications (Apache, PARSEC, Graph500, PBZIP2, and Metis). With a large NUMA machine, LATR reduces the cost of `munmap()` by up to 66%. In addition, LATR improves Apache's performance by up to 37.9% compared to ABIS and 59.9% compared to Linux.

2 Background and Motivation

2.1 Existing OS Designs

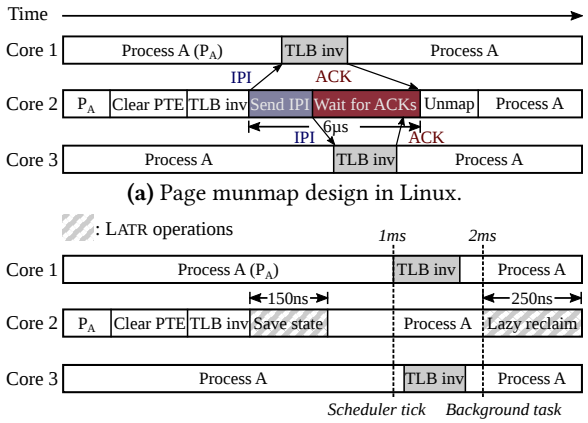
Most architectures, including x86, do not support TLB cache coherence. The current x86 architecture allows two operations on TLBs: invalidating a TLB entry using the `INVLPG` instruction and flushing all local TLB entries by writing to the `CR3` register. However, both instructions provide control only over the local, per-core TLB. To invalidate entries in remote TLBs on other cores, a process known as *TLB shootdown*, commodity OSes use an expensive, IPI-based mechanism. IPIs are individually delivered to each remote cores via the Advanced Programmable Interrupt Controller (APIC) [32] as it does not support flexible multicast delivery [44].

Free operations. We analyze the existing handling of a free operation (`munmap()`) in Linux, on a system with three cores (as shown in Figure 2a). The OS receives an `munmap()` system call from the application to remove a set of virtual addresses on core `C2` with the current process running on all existing cores (`C1`, `C2`, and `C3`). The `munmap()` handler removes the page table mappings for the set of virtual addresses and frees the virtual addresses and physical pages associated with the virtual addresses. In addition, `C1` performs a local TLB invalidation for the set of virtual addresses before initiating an IPI (to `C1` and `C3`) to perform the TLB shootdown. On receipt of the interrupt, `C1` and `C3` perform a local TLB invalidation in their IPI handlers and send an `ACK` to `C2` by the means of cache coherence. After receiving both `ACK`s from `C1` and `C3`, the `munmap()` handler on `C2` finishes processing the `munmap()` system call and returns control back to the application. The same TLB shootdown mechanism is used for all virtual address operations, though the page table changes are different.

The TLB shootdown mechanism outlined above shows three types of performance overheads: sending IPIs to remote cores, which has an increased overhead on large NUMA machines; handling interrupts on remote cores, which might be delayed due to temporarily disabled interrupts; and the wait time for `ACK`s on the initiating core.

AutoNUMA page migration. AutoNUMA page migration is a feature provided by Linux to migrate pages to the NUMA node where they are being frequently accessed from, to avoid costly cross-NUMA-domain accesses.

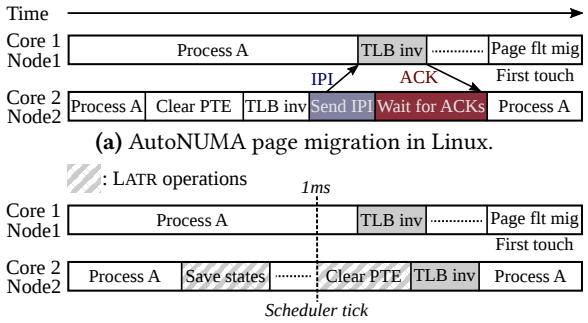
To identify pages that are predominantly used on a remote NUMA node, the AutoNUMA background task periodically scans a process' address space and changes page table entries which triggers frequent TLB shootdowns. After the TLB shootdown for a specific virtual address, any subsequent memory access to this virtual address triggers a page fault. If, during this page fault, a page is accessed twice from a NUMA node different from the current node the page resides on, the page will be migrated to the node accessing the page, pending other factors such as enough free memory on the target node. Figure 3a gives a high-level overview of this process in Linux



(a) Page munmap design in Linux.

(b) Page munmap design in LATR.

Figure 2. An overview of the operations involved in unmapping a page in both Linux and LATR. LATR removes the instantaneous TLB shootdown from the critical path by executing it asynchronously.



(a) AutoNUMA page migration in Linux.

(b) AutoNUMA page migration in LATR.

Figure 3. AutoNUMA page migration in both Linux and LATR. LATR removes the need for an immediate TLB shootdown when sampling pages for NUMA migration.

(shown with a single page fault for simplicity). As explained above, if the page is accessed from the same NUMA node, the page migration is not performed even though the expensive TLB shootdown (5.8%, with one 4 KB page, to 21.1%, with 512 4 KB pages, of the overall migration cost) was performed.

The goal of LATR is to provide a lazy mechanism for TLB shootdowns during free and migration operations (summarized in Table 1). With its lazy mechanism, LATR eliminates three types of performance overheads associated with the current TLB shootdown, namely, sending IPIs and waiting for the `ACK`s in the initiating core, and handling interrupts in the remote cores.

2.2 Hardware-based Approaches

Hardware-based research approaches provide cache coherence to the TLB. UNITD [54], a scalable hardware-based TLB coherence protocol, uses a bit on each TLB entry to store sharing information, thereby eliminating the use of IPIs. However, UNITD still resorts to broadcasts for invalidating shared mappings. Furthermore, UNITD adds a costly content-addressable memory (CAM) to each TLB to perform reverse address translations when checking whether a

Properties	DiDi [62]	Oskin et al. [44]	ARM TLBI [5, 6, 47]	UNITD [54]	HATRIC [64]	ABIS [2]	Barrelfish [10]	Linux [60]	LATR
Asynchronous approach	-	-	-	-	-	-	-	-	✓
Non-IPI-based approach	✓	-	✓	✓	✓	-	✓	-	✓
No remote core involvement	✓	✓	✓	✓	✓	-	-	-	✓
No hardware changes required	-	-	-	-	-	✓	✓	✓	✓

Table 2. Comparison between LATR and other approaches to TLB shootdowns.

page translation is present in a specific TLB, thereby greatly increasing the TLB's power consumption. HATRIC [64] is a hardware mechanism similar to UNITD and piggybacks translation coherence information using the existing cache coherence protocols.

DiDi [62] employs a shared second-level TLB directory to track which core caches which PTE. This allows efficient TLB shootdowns, while DiDi also includes a dedicated per-core mechanism that provides support for invalidating TLB entries on remote cores without interrupting the instruction stream they execute, thereby eliminating costly IPIs. Similarly, other approaches provide microcode optimizations to handle IPIs without remote core interventions [44]. Though these approaches remove interrupts on remote cores, the wait time on the core initiating the TLB shootdown is not removed. Finally, these approaches require intrusive changes to the micro-architecture, which adds additional verification cost to ensure correctness [3, 22, 24, 40, 53, 55, 61].

2.3 Software-based Approaches

Commodity OSes, such as Linux and FreeBSD, implemented a set of non-trivial software-base optimizations. For example, Linux made two important optimizations for TLB shootdowns: 1) batched remote TLB invalidation [30], where multiple invalidation entries are batched together in one IPI, and 2) lazy TLB invalidation that balances between the overheads of TLB flushes and TLB misses when a core becomes idle. It is worth noting that Linux's lazy invalidation mechanism refers to lazily invalidating entries on the local TLB in *idle cores*, which is different from LATR's lazy mechanism that lazily invalidates entries on *remote cores*. More specifically, it works as follows: when a core becomes idle, the OS speculates that the same process may get scheduled on the same core, so it defers the invalidation of the local TLB to avoid potential future TLB misses. However, if the idle core subsequently receives a TLB shootdown, the OS performs a full TLB invalidation and indicates to the other cores not to send further shootdown IPIs to this core while it remains idle. Unfortunately, all these optimizations do not eliminate the IPIs needed for TLB invalidation.

Barrelfish [10], a research multi-kernel OS, uses message passing instead of IPIs to shoot down remote TLB entries. Thus, it eliminates the interrupt handling on remote cores. However, it still has to wait for the `ACK` from all remote cores participating in the shootdown. We note that Barrelfish thereby still takes a synchronous approach for TLB shootdowns. ABIS [2], a recent state-of-the-art research prototype based on Linux, uses page table access bits to reduce the number of IPIs sent to remote cores by tracking the set of CPUs sharing a page, which can be complementary to LATR. However, the operations in ABIS to track page sharing introduce additional overheads.

We conclude that these hardware- and software-based approaches for TLB shootdowns do not eliminate all TLB shootdown overheads and are not easy to apply in current systems due to their required hardware changes. Table 2 provides a comprehensive comparison of existing approaches with LATR.

3 Overview

LATR proposes a lazy TLB shootdown approach for virtual memory operations such as *free* (e.g., `munmap()` and `madvise()`) and *page migration* (e.g., AutoNUMA page migration), as shown in Table 1. The key idea that drives LATR is the *delayed* reuse of virtual and physical memory for free operations. Currently, the immediate reuse of the virtual and physical pages involved in a `munmap()` operation necessitates an immediate TLB shootdown, e.g., via a mechanism like inter-processor interrupts (IPIs). However, considering the large virtual address space (2^{48} bytes) and the amount of RAM (64 GB and more) available in current servers, not reusing the virtual and physical pages immediately enables an asynchronous TLB shootdown.

Support for free operations. LATR relies on the following invariant for the correctness of free operations: virtual and physical pages can be reused only after the associated TLB entries have been cleared on all cores. To ensure that the invariant holds for a free operation, LATR stores the virtual and physical pages to be freed in a separate lazy-reclamation list instead of adding them to the free pool immediately, which avoids their immediate reuse across all cores. LATR issues a local TLB invalidation for the TLB entries on the current executing core. In addition, instead of sending IPIs to the other participating cores, the state needed for the TLB shootdown is recorded in per-core invalidation states (referred to as LATR states §4.1). During a context switch or scheduler tick, the participating cores perform a local TLB invalidation by sweeping the other cores' LATR states via regular memory reads. The context switch and scheduler tick provide a periodic transition to the OS, which provides

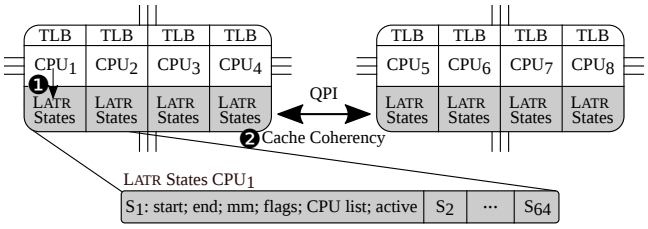


Figure 4. Overview of LATR's interaction with the system and its data structures. LATR uses per-CPU *states* (❶) to identify the cores included in a TLB shootdown. The states are made accessible to remote cores via the cache coherence protocol (❷) that remote cores use to clear entries in their local TLB.

the opportunity to perform the state sweep and a TLB invalidation. Since the TLB invalidation is performed during a context switch or scheduler tick, the scheduler tick interval (1 ms in Linux x86) establishes an upper bound time limit for a TLB shootdown. Based on this upper bound, LATR releases the virtual and physical pages using a background thread after a scheduler tick on the cores. As these scheduler ticks are not synchronized across all the cores, LATR delays the reclamation by twice the scheduler tick interval (2 ms).

Support for page migration operations. In addition to free operations, LATR's lazy TLB shootdown mechanism is applicable to page migration (e.g., AutoNUMA in Linux). For AutoNUMA, lazily changing the page table enables a lazy TLB shootdown. In LATR, the background task records the LATR state without changing the page table immediately. During the scheduler tick, the first core that reads the LATR state changes the page table, followed by local TLB invalidation. The other cores that read the LATR state during the scheduler tick, perform only the local TLB invalidation. The existing AutoNUMA page-fault handling and page-migration algorithm handle page migration, which is not modified by LATR. A similar algorithm can be used for other migration operations such as page swapping, deduplication, and compaction. For example, with a least recently used (LRU) based page swapping algorithm, the page table unmap and swap operation can be performed lazily after the last core has invalidated the TLB entry. LATR's proposed lazy AutoNUMA and page swap algorithms are important for emerging systems with heterogeneous memory [44] and disaggregated memory [27, 36].

4 Design

Using the idea introduced in §3, we describe the design of LATR for x86-based Linux machines in detail. We first introduce the states needed by LATR (referred to henceforth as LATR states), and explain the TLB shootdown operation using the LATR states. Using the LATR states component, we describe the free operations (e.g., `munmap()` and `madvise()` in §4.2) and migration operations (e.g., AutoNUMA in §4.3). An overview of the LATR states is given in Figure 4.

4.1 LATR States

LATR saves the shootdown information in the LATR states, which are used for asynchronous TLB invalidation. The LATR states are a per-core cyclic lock-less queue (as shown in Figure 4), which is allocated from a contiguous memory region. Each entry in the LATR states holds the following information: the addresses *start* and *end* of the virtual address for the TLB shootdown, a pointer to the `mm_struct` to identify the current running process, a *bitmask* to identify the remote CPUs involved, *flags* to identify the reason for the shootdown (e.g., to distinguish migration and free operations), and an *active* flag. The CPU bitmask identifies the remote cores on which the TLB invalidation should be performed for a particular entry's address. The active flag is used to identify currently valid entries. To ensure ordering between memory instructions, an entry is activated after setting all the fields using an atomic instruction coupled with a memory barrier.

Storage overhead. In LATR, each core stores 64 LATR states. The size of each state is 68 B, while all LATR states on a system with 32 cores amount to 136 KB, occupying less than 1% of the last-level caches (LLC) of recent processors (e.g., at least 16 MB for an 8-core Intel CPU [33]). Even on an 8-socket, 192-core machine, the total size of all LATR states grows to only 816 KB, which corresponds to less than 1.3% of the LLC [35].

State update. The core initiating the TLB shootdown sets all fields of a LATR state, including the CPU bitmask. Currently, Linux calculates the CPU bitmask for sending IPIs based on the cores where the process is currently scheduled. LATR uses the same logic to update the CPU bitmask in the state. Since the states are in memory, the state updates are available to all other cores using the cache-coherence protocol. We show an example in Figure 5, where CPU_1 initiates the TLB shootdown. It includes CPU_2 and CPU_5 in the bitmask, as the process is currently scheduled on both these CPUs (❶). The state update in LATR eliminates the overhead of sending IPIs waiting for the `ACK`s that present in Linux and existing OSes.

Asynchronous remote shootdown. During a periodic interval (scheduler tick or context switch), each core sweeps the LATR states of all available cores. The *state sweep* operation checks the LATR states from all cores, taking advantage of hardware prefetching since the states are allocated as contiguous memory blocks. Using the active flag and the CPU bitmask available in a LATR state, a core identifies states relevant to itself and invalidates the TLB entries on the core. In addition, after the TLB invalidation, the core removes itself from the CPU bitmask of the respective state. During the state sweep operation, each core updates the CPU bitmask and the active flag using an atomic operation that eliminates the need for locks. For example, in Figure 5, CPU_2 and CPU_5 invalidate their local TLB entries during the state sweep operation before resetting the CPU bitmask in the state (❷

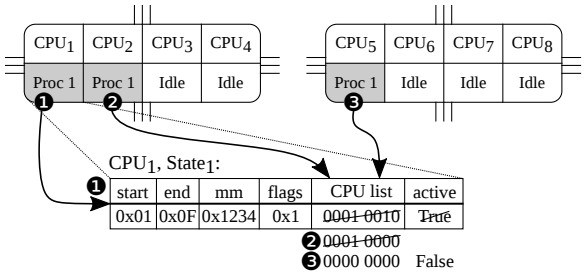


Figure 5. Example of the usage of a LATR state. CPU_1 unmaps a page (❶) which is also present in CPU_2 and CPU_5. At the scheduler tick, all other CPUs will use the LATR state to determine if a local TLB invalidation is needed and CPU_2 (❷) and CPU_5 (❸) will invalidate their local TLB entry before resetting the LATR state to be reclaimed.

and ❸). In addition, CPU_5, the last core performing the local TLB invalidation, resets the active flag in the LATR state (❸). By means of this lazy asynchronous shootdown, LATR inherently provides batched TLB invalidation without using IPIs. For example, similar to Linux where the entire TLB is flushed if there are more than 33 TLB invalidations (i.e., half the size of the L1 D-TLB), LATR flushes the entire TLB during state sweep.

The LATR shootdown is performed during the scheduler tick or a context switch, whichever event happens first. The scheduler tick or context switch event provides an existing transition mechanism in the current OS, which we leverage for the state sweep and TLB invalidation. The LATR TLB invalidation during a scheduler tick or a context switch eliminates the interrupt handling overheads associated with IPIs. In addition, it reduces the cache pollution overhead resulting from IPI interrupts (see Table 4).

4.2 Handling Free Operations

In this section, we analyze the handling of free operations using the LATR states.

Lazy memory reclamation. A key part in the design of LATR, as outlined in §3, is the lazy reclamation of both virtual and physical pages. LATR establishes the following *invariant*: during free operations, virtual or physical pages are released only after associated TLB entries have been cleared. In LATR, to honor this invariant, we explore the following relaxation: By allowing a delay (e.g., 2 ms, twice the scheduler tick interval as introduced in §3) before reclaiming virtual and physical pages, we can remove both the reclamation of memory and the need for a TLB shootdown from the critical path of free operations such as `munmap()` and `madvise()`.

LATR deletes the mappings from the page table entry (PTE) during free operations; however, instead of freeing the virtual and physical pages, LATR maintains the list of virtual and physical pages to be freed lazily in the `mm_struct`. In addition, LATR maintains a global list of `mm_structs`, synchronized using a global spin lock, to identify the tasks participating in a lazy reclamation. To ensure that the virtual address is not reused, the lazy virtual address list is traversed during any memory allocation, and the addresses in the lazy list are not reused. Similarly, since the physical page reference count is non-zero, LATR ensures that the physical pages are not reused.

Handling `munmap()`. Using the *lazy memory reclamation* and the LATR states, LATR removes the instantaneous shootdown from the critical path of free operations. Instead, on execution of these operations, LATR simply records the states to shootdown a set of virtual addresses (the *state* information) but does not send an IPI immediately, as outlined in §4.1. In the case where there are more shootdowns per interval than there are per-core LATR states (i.e., 64 LATR states per core), LATR issues IPIs as a fallback mechanism. Furthermore, LATR's lazy free operation introduces new race conditions that LATR solves, these conditions are discussed in §4.4.

A detailed example of LATR handling an `munmap()` operation is shown in Figure 2b as a timeline. Core 2 executes the `munmap()` system call resulting in a TLB invalidation followed by core 2 saving the LATR state which includes the cores 1 and 3 in the CPU bitmask. The `munmap()` execution adds the page to the lazy free list. Due to the CPU bitmask, core 1 and core 3 invalidate their local TLB entry during their respective scheduler ticks (after 1 ms) and reset their respective CPU bitmask in the LATR state. Core 2 runs the LATR background thread (after 2 ms), and frees the virtual and physical pages in the lazy list.

Lazy TLB shootdown correctness. The correctness of LATR's handling of TLB shootdowns for free operations relies on the invariant introduced in §3: Virtual and physical pages can only be reused *after* associated TLB entries have been invalidated. To fulfill this invariant, LATR waits two full cycles of TLB invalidations (i.e., two scheduler ticks and 2 ms) to ensure that all associated entries have definitely been invalidated by at least one scheduler tick.

4.3 Handling NUMA Migration

In addition to supporting free operations, LATR's design also provides a lazy mechanism for migration operations, such as AutoNUMA page migration. We discuss the LATR mechanism for AutoNUMA page migration in this section.

The current AutoNUMA design in Linux includes a remote TLB shootdown (see §2.1), which accounts for up to 5.8–21.1% (page count ranging from 1 to 512) of the overall time in the case of a page migration. However, this TLB shootdown cost is paid even if the page fault handling decides to not migrate the page. LATR's mechanism for AutoNUMA provides a lazy TLB shootdown approach, eliminating the expensive TLB shootdown operation.

Lazy page table change. For AutoNUMA, a key part of LATR design is the lazy page table change after an interval (1 ms). LATR maintains an invariant that the pages are migrated, after the interval, only after all cores performed a

TLB shootdown. LATR uses the state abstraction to maintain this invariant and to perform the lazy page table change.

AutoNUMA mechanism. We illustrate the approach taken with LATR in Figure 3b, which exemplifies LATR's key design change (shown with two cores on two different sockets): When the AutoNUMA daemon decides to unmap a page from the page table to test for a potential migration, LATR records only this state into a *LATR state* instead of unmapping the page immediately, delaying the TLB shootdown. This state simply informs all cores to invalidate their local TLB for the specified page at the next scheduler tick. Any memory access before the scheduler tick can proceed without interruption. In addition to invalidating the local TLB entry, the first core performs the page table unmap operation (shown as "Clear PTE") before invalidating its TLB entry. The page unmap operation results in a page fault when the page is next touched, resulting in a potential page migration, similar to the existing design in Linux.

LATR removes the need for an instantaneous and costly IPI while retaining the design of AutoNUMA. As LATR delays the page table unmap operation until the next scheduler tick (1 ms), there is no additional overhead imposed on the application. This design trades off the expensive IPI-based TLB shootdown for additional waiting time until the next scheduler tick (up to 1 ms). Furthermore, LATR's lazy migration introduces new race conditions; their handling in LATR is discussed in section §4.4.

Figure 3b shows an example of the LATR mechanism used in conjunction with the AutoNUMA page migration. The AutoNUMA background daemon on core 2 adds a state to the LATR states instead of unmapping the page from the page table. This state includes the CPU bitmask of all the cores, including core 1 and core 2. The scheduler tick on core 2 initiates the unmap operation (shown as "Clear PTE") and then invalidates its local TLB. The scheduler tick on core 1 only invalidates its local TLB. The next page access on core 1 triggers the page fault handler and subsequently the page migration due to a page access from a different NUMA node.

Correctness for AutoNUMA migration. We show the correctness of LATR's AutoNUMA migration. Memory accesses during the interval (1 ms) proceed normally. The first core performing the TLB shootdown, after the interval, will unmap the page from the page table. Memory accesses after the interval thus result in a page fault. If the page fault handler migrates pages, LATR holding a lock until all cores perform their local TLB invalidation ensures that parallel writes are not allowed during the migration.

4.4 LATR Race Conditions

In this section, we discuss the possible race conditions introduced by a lazy TLB shootdown and their handling with LATR.

Reads before a TLB shootdown. For free operations, an application error can result in reading already freed memory before the scheduler tick (1 ms). On cores where the respective TLB entry is not invalidated yet, LATR serves the read from the old, not yet freed page. However, after the LATR TLB shootdown during the scheduler tick, any further reads will result in a page fault, which eventually results in a segmentation fault.

Writes before a TLB shootdown. For free operations, an application error can result when writing values to the unmapped memory before the scheduler tick (1 ms). On cores where the TLB entry is not invalidated yet, LATR allows writes to the old page that is not yet freed. However, after the scheduler tick interval, any further writes will result in a page fault, which eventually results in a segmentation fault.

For both reads and writes, LATR does not prohibit applications to read or write to an unmapped page for a specific interval (until the scheduler tick, up to 1 ms) although this application behavior is the result of an application error. However, LATR prevents the consequences (e.g., page corruption) of these reads or writes to impact other processes or the kernel by not releasing the physical pages before the LATR TLB shootdown is complete.

AutoNUMA balancing. With LATR's delayed page table unmap operation for the case of AutoNUMA, there is a possibility that a page fault could occur on any core before the page table unmap operation is complete, for example if a page fault occurs simultaneously with the first core unmapping the page from the page table. However, both the page fault and the AutoNUMA page table unmap operation are guarded by the `mmap_sem` semaphore, which ensures that the unmap operation is completed before the page fault handler can proceed. Similarly, the page fault is handled only after all cores have performed the LATR TLB invalidation for the AutoNUMA migration; otherwise, cores that have not invalidated their TLB entries yet could proceed in writing to the page under migration. To avoid such a race condition, the first core that performs the page table unmap releases the `mmap_sem` only after all CPU bitmasks in the LATR state are cleared, indicating that all cores have invalidated their TLB entries. Thus, the next page fault can then trigger the NUMA page migration.

4.5 Approach With and Without PCID

Process-context identifiers (PCIDs) are available in x86 to allow a processor to cache TLB entries for multiple processes and to preserve it across context switches. LATR's lazy invalidation approach is applicable regardless of the OS' use of PCIDs. When PCIDs are not used (as Linux 4.10 elects to do), invalidating TLB entries pertaining to the states during the scheduler tick is important to remove stale entries within a bounded time period (e.g., 1 ms). During a context switch, however, the TLB is flushed, eliminating the need for a LATR

Machine Type	Commodity data center [43]	Large NUMA
Model	E5-2630 v3	E7-8870 v2
Frequency	2.40 GHz	2.30 GHz
# Cores	16	120
Layout (cores × sockets)	8 × 2	15 × 8
RAM	128 GB	768 GB
LLC (size × sockets)	20 MB× 2	30 MB× 8
L1 D-TLB entries (per core)	64	64
L2 TLB entries (per socket)	1024	512
Hyperthreading	Disabled	Disabled

Table 3. The two machine configurations used to evaluate LATR.

invalidation. In the case where PCIDs are being used, only TLB entries matching the currently active PCID can be invalidated. Since the invalidation should be performed before the PCID is changed, LATR's TLB invalidation during a context switch is mandatory. When using PCIDs, TLB invalidations can be triggered only for the currently active PCID. When a different PCID is active, LATR aborts migrations similar to a case where AutoNUMA aborts a page migration if the page fault does not indicate a remote socket accessing the page.

4.6 Large NUMA Machines

LATR eliminates the use of expensive IPIs to disseminate the TLB shootdown information. Instead, LATR requires writing the LATR states to memory, which implicitly propagate to the LLCs of all sockets via the hardware cache coherence protocol, making LATR highly scalable. The lazy approach employed by LATR eliminates the synchronous TLB shootdown overhead, which amounts to up to 80 μs on an 8-socket machine (as shown in Figure 7).

5 Implementation

We implemented the LATR prototype in 1,012 lines of code by extending Linux 4.10. We modified the kernel's TLB shootdown handler to save the LATR states instead of sending IPIs. We extend the kernel's `munmap()` and `madvise()` handlers to perform the lazy memory reclamation, and the AutoNUMA page table unmap handler to save the LATR states.

Lazy TLB shootdown. LATR's lazy TLB shootdown is handled in `native_flush_tlb_others`, in which sending of IPIs is replaced with saving the corresponding LATR states. The state sweep to invalidate the LATR states is handled in `scheduler_tick` and `__schedule`.

Lazy memory reclamation. The `VMA` and `page` pointers are added to a lazy list in the `mm_struct`, and the `mm_struct` is added to a global list. The background kernel thread frees the `VMA` and `page` using `remove_vma` and `free_pages`, respectively.

NUMA page migration. The function `task_numa_work` is modified to not trigger the page table unmap via `change_prot_numa`. Instead, a handler saving the LATR state is invoked while, `change_prot_numa` is invoked later during the `scheduler_tick`, along with the local TLB invalidation.

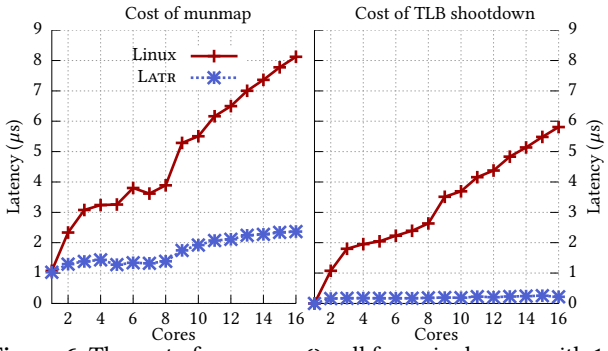


Figure 6. The cost of an `munmap()` call for a single page with 1 to 16 cores in our microbenchmark. TLB shootdowns account for up to 71.6% of the total time. LATR is able to improve the time taken for `munmap()` by up to 70.8% with its asynchronous mechanism.

6 Evaluation

We implemented a proof-of-concept of LATR based on Linux 4.10. The baseline for the evaluation is Linux 4.10, while we also compare a subset of cases against ABIS [2], which is based on Linux 4.5. ABIS is a recent research prototype that aims at reducing the number of IPIs sent by tracking the set of cores sharing a page via the page table access bits [2].

Using this setup, we evaluate LATR by answering the following questions:

- Does LATR show benefits with microbenchmarks on machines with a larger number of NUMA sockets?
- What are the benefits of LATR for applications with heavy usage of free operations (introduced in Table 1)?
- What are the benefits of LATR in the context of AutoNUMA page migration?
- What is the impact of LATR for applications which show few TLB shootdowns and what is the overhead of LATR for memory usage and cache misses?

6.1 Experiment Setup

We evaluate LATR on two different machine setups, as shown in Table 3. The primary evaluation target is the *2-socket, 16-core* machine, while we also show the impact of LATR on a large NUMA machine with *8 sockets and 120 cores* . We run each benchmark five times and report the average results.

The machines are configured without support for transparent huge pages, as this mechanism is known to increase overheads and introduce additional variance to the benchmark results [38]. Furthermore, to reduce variance in the results, all benchmarks are run on the physical cores only, without hyperthreads. We furthermore deactivate Linux's automatic balancing of memory pages between NUMA nodes, AutoNUMA, unless specifically noted, as it might introduces TLB shootdowns during the migration of a page (see §2.1).

6.2 Impact on Free Operations

First, we discuss the impact of LATR on operations centered around freeing virtual and physical addresses, such as `munmap()` and `madvise()` (as introduced in Table 1).

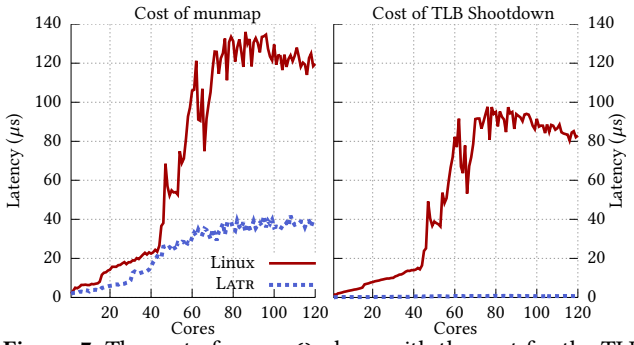


Figure 7. The cost of `munmap()` along with the cost for the TLB shootdown for a single page in Linux compared to LATR on an 8-socket, 120-core machine. TLB shootdowns account for up to 69.3% of the overall cost, while LATR is able to improve the cost of `munmap()` by up to 66.7%.

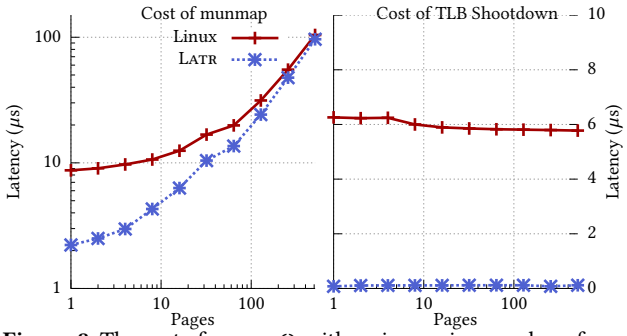


Figure 8. The cost of `munmap()` with an increasing number of pages along with the cost of the TLB shootdowns for Linux and LATR. For a small number of pages, LATR shows improvements of up to 70.8%, while the impact of the TLB shootdown diminishes with a larger number of pages. At 512 pages, LATR still retains a 7.5% benefit over Linux.

6.2.1 Microbenchmarks

To understand the scalability behavior of LATR in isolation, we compare LATR to Linux while we exclude ABIS [2], as its behavior matches Linux in a microbenchmark where all cores actually access a shared page. We devise a microbenchmark that shares a set of pages between a specified number of cores. A subsequent call to `munmap()` on this set of pages will then force a TLB shootdown on the participating cores. Each data point is run 250,000 times. The microbenchmark records the time taken for the call to `munmap()`, as well as the time taken for the TLB shootdown, excluding other overheads, e.g., the page table modifications and syscall overheads.

The results of this microbenchmark using one page on our 2-socket, 16-core machine are shown in Figure 6 and exemplify the overheads introduced by the TLB shootdown: In the baseline Linux system, the TLB shootdowns contribute up to 71.6% to the overall execution time of `munmap()`, while a single `munmap()` call for 16 cores takes up to 8 μs. LATR on the other hand is able to reduce almost all of this overhead and improves the latency of `munmap()` by 70.8% by recording only the LATR states on the critical path of `munmap()`. LATR thus reduces the latency for `munmap()` to 2.4 μs for 16 cores.

Large NUMA machine. To investigate the behavior of LATR and the baseline Linux on a large NUMA machine, we run this microbenchmark on the 8-socket, 120-core machine and show the results in Figure 7. These results show a drastic increase in latency for `munmap()` on Linux when using more than 45 cores (more than 3 sockets), as the IPI delivered through the APIC needs two hops to reach the destination CPU. At 120 cores, the latency for a single `munmap()` rises to more than 120 μs, with the TLB shootdown accounting for up to 82 μs or 69.3%. LATR on the other hand is able to efficiently use the cache coherence protocol to complete the `munmap()` operation in less than 40 μs on 120 cores, reducing the latency by 66.7% compared to Linux, as LATR's `munmap()` does not rely on expensive IPIs and eliminates the `ACK` wait time on the initiating core.

Increasing number of pages. We investigate the behavior of LATR compared to Linux when using more than one page; the results for up to 512 pages on 16 cores are shown in Figure 8. The impact of the TLB shootdown diminishes with a larger number of pages, as the overhead of clearing TLB entries is amortized by more costly operations, such as changing the page table. Furthermore, Linux elects to fully flush the TLB when more than 32 pages are being invalidated at once, which furthermore limits the maximum possible overhead. Even though the impact of the TLB shootdown reduces, LATR still improves the performance at 512 pages by 7.5% while showing larger benefits with fewer pages. Furthermore, applications can use *huge pages* (either 2 MB or 1 GB pages on x86), either directly or via transparent huge pages support in the OS [38], to mitigate the effects of unmapping many pages at once.

6.2.2 Impact on Applications

We evaluate LATR by quantifying its impact on free operations with real-world applications, using both Apache and the PARSEC [13] benchmark suite.

Apache webserver. We compare the requests per second of Apache with Linux, ABIS [2], and LATR on the 2-socket, 16-core machine. We use the Wrk [29] HTTP request generator, using four threads with 400 connections each for 30 seconds, to send requests to Apache, which hosts a static, 10 KB webpage. Wrk and Apache run on the same machine (to avoid the network stack becoming the bottleneck) but on a distinct set of cores to isolate the two applications. This configuration leaves up to 12 cores available for Apache. We disable logging in Apache and use the default `mpm_event` module to process requests. This module spawns a (small) set of processes that in turn spawn many threads to handle the requests. To serve an individual request, Apache `mmap()`s the requested file to serve a request and `munmap()`s the file after the request has been served. This behavior generates many TLB shootdowns due to the frequent unmapping of (potentially) shared pages. The results are shown in Figure 9

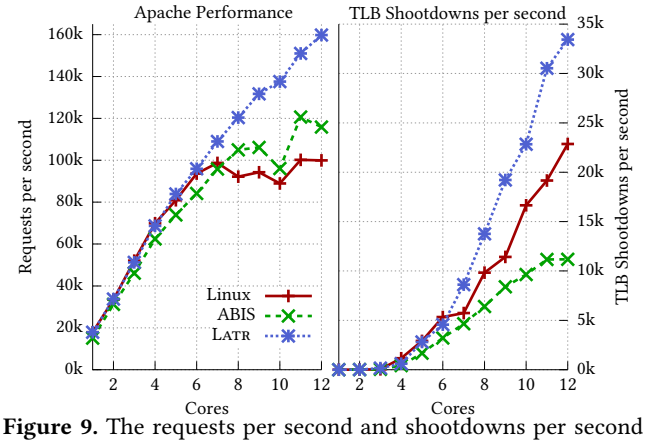


Figure 9. The requests per second and shootdowns per second of Apache using Latr, Linux and ABIS [2]. Latr shows a similar performance as Linux for lower core counts while outperforming it by 59.9% on 12 cores. ABIS initially shows overhead from frequent unmapping, while Latr performs up to 37.9% better at 12 cores than it.

and show both the requests per second served by Apache, as well as the TLB shootdowns per second. Latr outperforms Linux by up to 59.9% and ABIS by up to 37.9%. ABIS shows a reduced performance on lower core counts (< 8 cores) because of the overhead of frequent changes to access bits while outperforming Linux for larger core counts because of the significantly reduced number of TLB shootdowns. Latr outperforms both Linux and ABIS because of its efficient asynchronous handling of TLB shootdowns, even though the rate of shootdowns is up to 46% higher due to the increased performance of Latr. Furthermore, Latr does not need to use any fallback IPIs (see §4.2) during the execution of this test.

Parsec application benchmark. We show the performance of Latr compared with Linux across a wider range of PARSEC application benchmarks. The normalized runtime (with respect to Linux) along with the TLB shootdowns per second is shown in Figure 10. Latr shows improvements of up to 9.6% on cases with a larger number of TLB shootdowns (e.g., `dedup`) due to frequent calls to `madvise()` and shows small improvements for most of the other benchmarks. The reason for the small improvement for most benchmarks is that Latr optimizes background operations of the system such as reading files via `mmap()`/`munmap()`. For one benchmark, `canneal`, Latr shows a slight degradation of 1.7% due to frequent context switches for this benchmark, which triggers frequent state sweeps. Overall, Latr shows an improvement of 1.5% on average over Linux across all PARSEC benchmarks.

6.3 Impact on NUMA Migration

In contrast to previous benchmarks, we enable AutoNUMA for the following experiments. We evaluate the impact of Latr on AutoNUMA with a subset of applications (`fluidanimate` from PARSEC and `ocean_cp` from

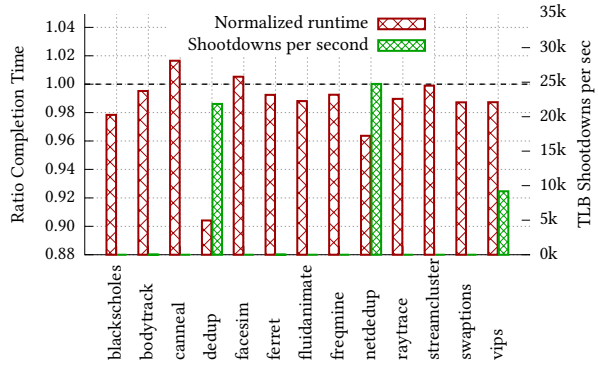


Figure 10. The normalized runtime and the rate of shootdowns for the PARSEC benchmark suite, comparing Latr and the Linux baseline using all 16 cores. Latr imposes at most a 1.7% percent overhead while improving the runtime on average by 1.5% and by up to 9.6% for `dedup`.

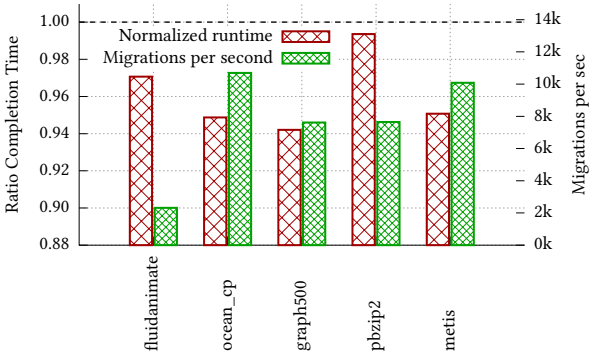


Figure 11. Impact of NUMA balancing on the overall runtime as well as the overall number of page migrations of Latr compared to Linux on 16 cores. Latr performs up to 5.7% better, showing a larger improvement with more page migrations.

the SPLASH-2x benchmark suite) that benefit from enabling NUMA memory balancing. Additionally, we evaluate Latr with three real-world applications: Graph500 [31], PBZIP2 [28], and Metis [17, 18, 41]. Graph500 is a graph analytics workload running a breadth-first search on a problem of size 20. PBZIP2 allows compressing files in parallel and in memory. We compress the Linux 4.10 tarball while splitting the input file among all processors. Finally, Metis is a Map-Reduce framework optimized for a single-machine, multi-core setup.

The results are given in Figure 11 and show the normalized runtime of Latr compared to Linux, as well as the number of page migrations per second. The results show an improvement of up to 5.7% for the Graph500 benchmarks while showing similar benefits on other benchmarks that show a large number of migrations. On PBZIP2, Latr improves only marginally compared with Linux, as for this application other application-level overheads dominate the runtime. As AutoNUMA migrates one page at a time, this result aligns with the cost breakdown of a migration operation showing a 5.8% (one 4 KB page) to 21.1% (512 4 KB pages) overhead for the TLB shootdown.

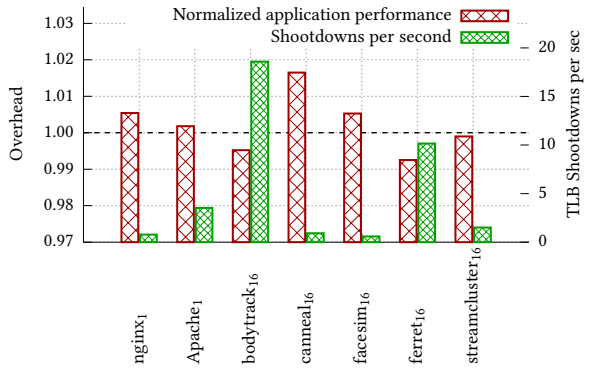


Figure 12. The overhead imposed by LATR on applications with few TLB shootdowns; subscripts indicate the number of cores. LATR shows small overheads of up to 1.7% for one benchmark.

6.4 Overheads of LATR

We investigate the overheads imposed by LATR in three aspects: what is the memory overhead of LATR, how does LATR impact applications with few TLB shootdowns, and how does LATR impact the cache locality of applications?

Memory utilization. We perform an analysis based on the microbenchmarks presented to show LATR's overhead in terms of memory utilization. In each time period (e.g., 1 ms), LATR shows an overhead of up to 21 MB of physical and virtual pages (for the case of 16 cores and 512 pages per `munmap()` call). If fewer cores and pages are being used, the overhead ranges from 3 MB (for 2 cores sharing a single page) to 1.5 MB (for 16 cores sharing a single page). Using more pages, the memory overhead stays bounded by 21 MB, as the overhead of page table modifications and related operations dominates the cost of the TLB shootdown (as seen in Figure 8). Considering the large virtual address space (2^{48} bytes) and the amount of RAM (64 GB and more) available in current servers, the memory overhead is not high (smaller than 0.03%) and is released back within a short time interval (2 ms).

Overhead on applications. We show the overhead imposed by LATR on real-world applications with few TLB shootdowns in Figure 12. This focuses on applications being run only on a single core (two webservers: Nginx and Apache) and a subset of PARSEC benchmarks which show very few TLB shootdowns. LATR shows a maximum overhead of 1.7% due to a larger number of context switches while imposing a smaller overhead on other applications. LATR is even able to improve the performance of some of the benchmarks due to optimizing various background activity in the system, as well as the general benefit of faster unmapping and freeing of shared memory.

An interesting analysis is the percentage of LLC cache misses for various applications and core counts. These results are shown in Table 4 and show that LATR *improves* cache misses for a number of applications while only imposing a maximum overhead of 0.8% on other cases. LATR's improvements in cache misses are due to the removed handling of IPI interrupts on remote cores. These benefits outweigh the increased cache utilization of the LATR states, which, however, occupy only a small portion of the LLC of modern processors (less than 1%, see §4.1).

Application	Cache Misses		Relative Change
	Linux	LATR	
$Apache_1$	6.08%	6.13%	+0.84%
$Apache_6$	1.60%	1.55%	-3.27%
$Apache_{12}$	1.23%	1.22%	-1.32%
$canneal_{16}$	80.51%	79.94%	-0.71%
$dedup_{16}$	18.33%	18.14%	-1.09%
$ferret_{16}$	48.02%	48.21%	+0.38%
$streamcluster_{16}$	95.42%	95.25%	-0.18%
$swaptions_{16}$	47.48%	47.23%	-0.54%

Table 4. The ratio of L3 cache misses between Linux and LATR; subscript indicates the number of cores the benchmark ran on. LATR shows cache misses to be very close (or better) to the Linux baseline due to the minimal cache footprint of LATR's states.

Operation	Time spent
Saving a LATR state	132.3 ns
Performing single state sweep with LATR	158.0 ns
Single TLB shootdown in Linux	1594.2 ns

Table 5. A breakdown of operations in LATR compared to Linux when running the Apache benchmark. LATR reduces the time taken for a single shootdown by up to 81.8% due to its asynchronous mechanism.

Breakdown of operations. We show a breakdown of operations in LATR compared to Linux when running Apache on 12 cores in Table 5. This breakdown shows the two separate phases of LATR: saving the LATR states on a per-core basis as well as invalidating local TLB entries based on the other's LATR states. This breakdown only includes the time taken for the TLB shootdown, excluding other effects such as modifying the page table or the syscall overhead, allowing for a fair comparison between Linux and LATR. Overall, LATR reduces the time taken for the TLB shootdown by up to 81.8%.

7 Discussion

Hardware support. We identify a number of hardware elements that could help to improve the performance of LATR. First, the LATR states can be allocated using Intel's cache allocation technology (CAT) [34], which allows fine-grained allocations in the LLC. This ensures that the LATR states do not interfere with the applications' cache usage. Second, LATR could benefit from the availability of a globally coherent scratchpad memory [1]. This would remove the need for duplicating the states in the LLCs of each processor, and simplify the lookup of states on remote cores. Furthermore, such a hardware element would reduce the time taken to access a LATR state and to perform the state sweep operation.

***Free operation* semantics.** LATR changes the semantics of free operations (`munmap()` and `madvise()`) by not freeing the physical pages immediately. This changed behavior impacts applications that unmap a page to force a page fault (e.g. to detect use-after-frees). To avoid such impacts, LATR's asynchronous mechanism could be selectively enabled by including a new flag in the API of free operations (e.g., `munmap()`) for existing OSes, such as Linux and FreeBSD.

Huge page support. LATR currently does not support transparent huge pages. However, the LATR states could be extended with an additional flag to support a lazy TLB shootdown for transparent huge pages as well. Additionally, memory compaction, an important component for the transparent huge page mechanism, performs similar mechanism as AutoNUMA's page migration. This operation has high TLB shootdown overheads, which can be optimized using LATR's AutoNUMA mechanism.

LATR's impact on debuggability. With stock Linux on 120 cores, remote shootdowns could take up to 80 μs. During this interval, reads and writes to pages will proceed on remote cores that did not receive the IPI yet. This renders debugging corruptions due to reads and writes during remote shootdowns difficult even in stock Linux. Similarly, LATR retains the existing challenges to debug these kind of scenarios.

Tickless Kernel. LATR supports the lazy TLB shootdown mechanism in tickless kernels [21]. In the Linux kernel, the tickless (`CONFIG_NO_HZ`) configuration disables the scheduler ticks on idle cores. In such tickless kernels, the idle core's id is not set in a LATR state's CPU bitmask because none of the processes are scheduled on the idle core. This eliminates the need for a state sweep during a scheduler tick on the idle core. In addition, when a core transitions from an idle state to a running state, the existing, stale TLB entries are removed by flushing the entire TLB.

8 Limitations

We identify the set of operations that can potentially benefit from an asynchronous approach in Table 1. LATR's lazy approach is not applicable to operations such as permission changes (`mprotect()`), ownership changes (CoW), and remap (`mremap()`), where the page table changes should be synchronously applied to the entire system. LATR maintains a limited number (64) of per-core LATR states and falls back to the existing IPI mechanism when the LATR states are fully occupied. Thus, LATR creates a trade-off between the number of per-core LATR states and the cost of state sweeps. And finally, LATR currently does not support transparent huge pages.

9 Related Work

Hardware-based TLB shootdown. There have been a number of approaches to handle the problem of TLB cache coherence at the hardware layer [7, 44, 50, 51, 54, 62, 64]. These approaches, however, are only being adopted slowly by hardware vendors, likely due to increased verification cost as well as the potential bugs they introduce in TLB cache coherence [3, 22, 24, 40, 53, 55, 61].

Software approaches. Similarly, there are a number of approaches in the OS [2, 8, 10, 15, 18, 45, 56, 57, 59] to optimize TLB shootdowns. However, none of them eliminate the synchronous TLB shootdown overhead. An alternative approach for reducing TLB shootdown is that applications can inform the OS on how memory is used or handle TLB flushes explicitly. Corey OS [16] avoids TLB shootdowns of private PTEs by requiring the user applications to explicitly define shared and private pages. Apart from this, FreeBSD uses versions with process context identifiers (PCIDs) [26] to eliminate the IPI operation. However, this approach invalidates all the TLB entries by using a version-based mechanism, which induces TLB misses.

Other TLB-related optimizations. SLL TLBs introduced a shared last-level TLB [12, 39] and evaluated the benefit of using a shared last-level TLB compared to a private second-level TLB. However, their design still relies on IPI-based coherency transactions. In addition, research approaches showed that TLB misses are predictable and that inter-core TLB cooperation and prefetching mechanisms can be applied to improve TLB performance [46, 58]. However, this implies that a TLB shootdown must also invalidate mappings in the TLB prefetch buffers. In addition, other approaches improve TLB misses, which is an orthogonal problem [11, 23, 37, 48].

10 Conclusion

We present LATR, a lazy software-based TLB shootdown mechanism that is readily implementable in modern OSes, for significant operations such as *free* and *page migration*, without requiring hardware changes. In addition, the proposed lazy migration approach can play a critical role in emerging heterogeneous memory systems and in disaggregated data centers. LATR reduces the cost of `munmap()` by up to 70.8% on multi-socket machines while improving the performance of applications like Apache by up to 59.9% compared to Linux and 37.9% compared to ABIS.

11 Acknowledgment

We thank the anonymous reviewers for their helpful feedback. We also thank Nadav Amit for his help with setting up ABIS. This research was supported, in part, by the NSF awards DGE-1500084, CNS-1563848, CNS-1704701, CRI-1629-851, SHF-1319755, XPS-1337147, CAREER-1253700, ONR under grant N000141512162, DARPA TC (No. DARPA FA8650-15-C-7556), ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] Lluc Alvarez, Lluís Vilanova, Miquel Moreto, Marc Casas, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Eduard Ayguadé, and Mateo Valero. Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 720–732, Portland, OR, June 2015.

[2] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 27–39, Santa Clara, CA, July 2017.

[3] Lukasz Anaczkowski. Linux VM workaround for Knights Landing A/D leak, 2016. https://lkml.org/lkml/2016/6/14/505.

[4] Apache. Apache HTTP Server Project, 2017. https://httpd.apache.org/.

[5] Ravi Arimilli, Guy Guthrie, and Kirk Livingston. Multiprocessor system supporting multiple outstanding TLBI operations per partition, October 2004. US Patent App. 10/425,425.

[6] ARM. ARM Compiler Reference Guide: TLBI, 2014. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/TLBI_SYS.html.

[7] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding TLB Shootdowns through Self-invalidating TLB Entries. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287, Portland, OR, September 2017.

[8] Ramesh Balan and Kurt Gollhard. A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machine. In *Proceedings of the Summer 1992 USENIX Annual Technical Conference (ATC)*, pages 107–115, San Antonio, TX, June 1992.

[9] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, March 2017.

[10] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, Big Sky, MT, October 2009.

[11] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–76, Xi'an, China, April 2017.

[12] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. In *Proceedings of the 17th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 62–73, San Antonio, TX, February 2011.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Toronto, Canada, October 2008.

[14] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–479, Orlando, FL, December 2006.

[15] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the 3rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–122, Boston, MA, April 1989.

[16] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 43–57, San Diego, CA, December 2008.

[17] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, October 2010.

[18] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, pages 211–224, Prague, Czech Republic, April 2013.

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

[20] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling, 2012. https://lwn.net/Articles/488709/.

[21] Jonathan Corbet. (Nearly) full tickless operation in 3.10, 2013. https://lwn.net/Articles/549580/.

[22] Christopher Covington. arm64: Work around Falkor erratum 1003, 2016. https://lkml.org/lkml/2016/12/29/267.

[23] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 435–448, Xi'an, China, April 2017.

[24] Linux Kernel Driver Database. CONFIG_ARM_ERRATA_720789, 2017. http://cateee.net/lkddb/web-lkddb/ARM_ERRATA_720789.html.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, San Francisco, CA, December 2004.

[26] FreeBSD. FreeBSD - PCID implementation, 2015. https://reviews.freebsd.org/rS282684.

[27] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–264, Savannah, GA, November 2016.

[28] Jeff Gilchrist. Parallel Compression with BZIP2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 559–564, Cambridge, MA, November 2004.

[29] Will Glozer. wrk - a HTTP benchmarking tool, 2015. https://github.com/wg/wrk.

[30] Mel Gorman. TLB flush multiple pages per IPI , 2015. https://lkml.org/lkml/2015/4/25/125.

[31] Graph500 Reference Implementations, 2017. http://graph500.org/?page_id=47.

[32] Intel. Multiprocessor Specification, 1997.

[33] Intel Xeon Processor E5-4610 v2, 2014. http://ark.intel.com/products/75285/Intel-Xeon-Processor-E5-4610-v2-16M-Cache-2_30-GHz.

[34] Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[35] Intel Xeon Processor E7-8894 v4, 2017. http://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz.

[36] Gu Juncheng, Lee Youngmoon, Zhang Yiwen, Chowdhury Mosharaf, and Shin Kang. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, April 2017.

[37] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Ünsal. Energy-Efficient Address Translation. In *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, Barcelona, Spain, March 2016.

[38] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 705–721, Savannah, GA, November 2016.

[39] 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):2:1–2:38, April 2013.

[40] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 233–247, Atlanta, GA, April 2016.

[41] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing MapReduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, May 2010.

[42] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *Proceedings of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, San Francisco, CA, February 2015.

[43] Timothy Prickett Morgan. AMD Disrupts The Two-Socket Server Status Quo, 2017. https://www.nextplatform.com/2017/05/17/amd-disrupts-two-socket-server-status-quo/.

[44] Mark Oskin and Gabriel H. Loh. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 188–200, San Francisco, CA, September 2015.

[45] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from Multithreading System V Release 4. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, SEDMS III, pages 77–91, 1992.

[46] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 20th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, Orlando, FL, USA, February 2014.

[47] Binh Pham, Derek Hower, Abhishek Bhattacharjee, and Trey Cain. TLB Shootdown Mitigation for Low-Power, Many-Core Servers with L1 Virtual Caches. *IEEE Computer Architecture Letters*, PP(99), June 2017.

[48] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–269, Vancouver, Canada, December 2012.

[49] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Waikiki, Hawaii, December 2015.

[50] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 743–758, Salt Lake City, UT, March 2014.

[51] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *Proceedings of the 20th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578, Orlando, FL, USA, February 2014.

[52] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Phoenix, AZ, February 2007.

[53] Bogdan F. Romanescu, Alvin R. Lebeck, and Daniel J. Sorin. Specifying and Dynamically Verifying Address Translation-aware Memory Consistency. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 323–334, Pittsburgh, PA, March 2010.

[54] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *Proceedings of the 16th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Bangalore, India, January 2010.

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

[56] Patricia Teller. Translation-Lookaside Buffer Consistency. *Computer*, 23(6):26–36, June 1990.

[57] Patricia J. Teller, Richard Kenner, and Marc Snir. TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences. Volume I: Architecture Track*, volume 1, pages 184–193, 1988.

[58] Scott Rixner Thomas Barr, Alan Cox. SpecTLB: a Mechanism for Speculative Address Translation. In *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 307–318, San Jose, California, USA, June 2011.

[59] Michael Y Thompson, JM Barton, TA Jermoluk, and JC Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of the Winter 1988 USENIX Annual Technical Conference (ATC)*, Dallas, TX, 1988.

[60] Linus Torvalds. Linux Kernel, 2017. https://github.com/torvalds/linux.

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

[62] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Ünsal. 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)*, pages 340–349, Galveston Island, TX, October 2011.

[63] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Boston, MA, December 2002.

[64] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 430–443, Toronto, Canada, June 2017.