# ECOTLB: Eventually Consistent TLBs

STEFFEN MAASS*, Georgia Institute of Technology
MOHAN KUMAR KUMAR*, Georgia Institute of Technology
TAESOO KIM, Georgia Institute of Technology
TUSHAR KRISHNA, Georgia Institute of Technology
ABHISHEK BHATTACHARJEE, Yale University

We propose ECOTLB—software-based eventual translation lookaside buffer (TLB) coherence—that eliminates the overhead of the synchronous TLB shootdown mechanism in operating systems that use address space identifiers (ASIDs). With an eventual TLB coherence, ECOTLB improves the performance of *free* and *page swap* operations, by removing the inter-processor interrupt (IPI) overheads incurred to invalidate TLB entries. We show that the TLB shootdown has implications for page swapping in particular in emerging, disaggregated data centers and demonstrate that ECOTLB can improve both the performance and the specific swapping policy decisions using ECOTLB's asynchronous mechanism. We demonstrate that ECOTLB improves the performance of real-world applications, such as Memcached and Make, that perform page swapping using INFINISWAP, a solution for next generation data centers that use disaggregated memory, by up to 17.2%. Moreover, ECOTLB improves the 99th percentile tail latency of Memcached by up to 70.8% due to its asynchronous scheme and improved policy decisions. Furthermore, we show that recent features to improve security in the linux kernel, like kernel page table isolation (KPTI), can result in significant performance overheads on architectures without support for specific instructions to clear single entries in tagged TLBs, falling back to full TLB flushes. In this scenario, ECOTLB is able to recover the performance lost for supporting KPTI due to its asynchronous shootdown scheme and its support for tagged TLBs. Finally, we demonstrate that ECOTLB improves the performance of free operations by up to 59.1% on a 120-core machine and improves the performance of Apache on a 16-core machine by up to 13.7% compared to baseline Linux, and by up to 48.2% compared to ABIS, a recent state-of-the-art research prototype that reduces the number of IPIs.

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

Additional Key Words and Phrases: TLB; Translation Coherence; Asynchrony.

Extension of Conference Paper: "LATR: Lazy Translation Coherence"[31] was published in ASPLOS 2018. ECOTLB provides an asynchronous TLB shootdown mechanism for page swap operations in next-generation data centers that use disaggregated memory. In addition, ECOTLB supports asynchronous TLB shootdown mechanism in operating systems that use address space identifiers (ASIDs).
*Both authors contributed equally to this research.

Authors' addresses: Steffen Maass Georgia Institute of Technology, steffen.maass@gatech.edu; Mohan Kumar Kumar Georgia Institute of Technology, mohankumar@gatech.edu; Taesoo Kim Georgia Institute of Technology, taesoo@gatech.edu; Tushar Krishna Georgia Institute of Technology, tushar@ece.gatech.edu; Abhishek Bhattacharjee Yale University, abhishek.bhattacharjee@yale.edu.

1544-3566/2020/7-ART1
https://doi.org/10.1145/3409454

**ACM Reference Format:**
Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. 2020. ecoTLB: Eventually Consistent TLBs. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (July 2020), 24 pages. https://doi.org/10.1145/3409454

## 1 INTRODUCTION

Memory-intensive applications, such as web servers, key value stores, and graph analytics engines, are widely used today for low-latency services in data centers. Such memory-intensive applications are built on existing architectures in current data centers, and research systems propose novel designs for applications, such as Infiniswap [28], in next-generation data centers using disaggregated memory [23, 28, 30]. For memory-intensive applications, in both existing and next-generation systems, maintaining translation lookaside buffer (TLB) coherence is critical in terms of operational correctness and performance, which needs to be performed by an operating systems (OS).

Existing OSs' TLB design can be classified into two categories: one that does not use application space identifiers (ASIDs), and another that uses ASIDs. With a design that does not use ASIDs (e.g., versions before 4.14 in Linux), TLB entries are not tagged with process identifiers which require a TLB flush[1] during every context switch. The design that uses ASIDs (FreeBSD and versions after 4.14 in Linux) tags TLB entries by using process identifiers that eliminate the need for a TLB flush and preserves TLB entries across context switches, thereby improving TLB hit rates. In particular, using ASIDs, called process context identifiers (PCID) in the x86 architecture, is a performance-critical optimization for the Linux kernel that enables Kernel Page Table Isolation (KPTI) [18], which attempts to mitigate the Meltdown [35] security vulnerability.

Even with a design using ASIDs, an OS maintains the hardware TLB coherence by performing a *TLB shootdown*, the process of invalidating specific, stale TLB entries on remote cores, using a synchronous inter-processor interrupt (IPI) mechanism, which is very expensive in modern architectures (e.g., an IPI takes up to 6.6 $\mu$s for 120 cores with 8 sockets [31]). With a synchronous IPI, the core initiating a TLB shootdown first sends IPIs to all remote cores and then waits for their acknowledgments while the corresponding IPI handlers on the remote cores complete the local invalidation of a TLB entry. For example, with Linux 4.14 that uses the PCID support available in the x86 architecture, a TLB shootdown takes around 108 $\mu$s for 120 cores with eight sockets and 2.5 $\mu$s for 16 cores with two sockets.

Such an expensive TLB shootdown in turn affects the performance of applications that trigger frequent memory management operations, such as `munmap()` and page swapping, that need to change their page table entries [1, 60]. For example, we observe that an Apache web server suffers from high TLB shootdown overheads while serving static files, which requires frequent `munmap()` operations (e.g., around 80K per second). Worse yet, the commonly enabled kernel page-table isolation (KPTI) feature increases the shootdown overhead by 8.4% on various x86 architectures, such as Ivy Bridge and below, that is provided by popular cloud providers [25].

In addition to existing architectures, TLB shootdown overhead plays an important role in next-generation disaggregated data centers. For example, we observe that Memcached, a key value store that can trigger page swap operations using Infiniswap [28], suffers from an increase of up to 2× in tail latency due to shootdown overheads. The above observations show that applications and features, developed on current and next-generation architectures, sometimes ignore the interactions of these innovations with existing system abstractions.

To address the synchronous TLB shootdown overhead, hardware-based approaches [7, 42, 48, 49, 51, 60, 62] strive to provide TLB coherence in an efficient manner. Nevertheless, such hardware

[1] The process of invalidating *all* TLB entries using one IPI per remote core.

| Operations | DiDi<br>[60] | Oskin et al.<br>[42] | ARM TLBI<br>[4, 5, 47] | UNITD<br>[51] | HATRIC<br>[62] | ABIS<br>[1] | Linux<br>[58] | Latr<br>[31] | ecoTLB |
|---|---|---|---|---|---|---|---|---|---|
| Non-IPI-based approach | ✓ | × | ✓ | ✓ | ✓ | × | × | ✓ | ✓ |
| Software-based approach | × | × | × | × | × | ✓ | ✓ | ✓ | ✓ |
| Lazy TLB shootdown with PCIDs | × | × | × | × | × | × | × | × | ✓ |
| Disaggregated memory (Infiniswap) | × | × | × | × | × | × | × | × | ✓ |

Table 1. Comparison between ecoTLB and other approaches to TLB shootdowns. ecoTLB provides a lazy shootdown mechanism leveraging PCIDs, which is not available in Latr. Importantly, ecoTLB provides a lazy shootdown mechanism for swapping pages to disaggregated memory using RDMA.

mechanisms need expensive hardware modification and are not available as part of the existing x86 architecture. Most existing software approaches [1, 8, 11, 15, 17, 44, 54, 55, 57] to synchronous TLB shootdowns strive to reduce the number of necessary IPIs to be sent, either by batching TLB shootdowns, or by using alternative mechanisms instead of IPIs (e.g., message passing [11]). However, the above hardware- and software-based approaches continue to use a synchronous TLB shootdown mechanism, and do not eradicate its overhead. Latr [31], a software-based mechanism, proposes an orthogonal, asynchronous lazy TLB shootdown mechanism that eliminates the shootdown overhead only for *free* and *AutoNUMA* page migration operations. However, Latr does not address the TLB shootdown overhead with next-generation architectures, both for addressing the emerging usage of PCIDs as well as the impact of *swapping* in disaggregated datacenter environments. In addition, none of the above research work bring out the intrinsic interplay between process-context identifiers (PCIDs) and the TLB shootdown on various x86 architectures. Table 1 compares ecoTLB to existing research approaches.

ecoTLB provides an asynchronous TLB shootdown mechanism for page swap operations in next-generation data centers that use disaggregated memory, and shows the impact of the asynchronous mechanism for free operations using PCIDs with security features such as KPTI on various x86 architectures. For free operations, ecoTLB stores the versions needed by a PCID design, and lazily updates these versions during a context switch. In addition, ecoTLB provides a lazy page swap mechanism that addresses the TLB shootdown overhead in next-generation data centers. For *free* and *page swap* operations, ecoTLB removes the performance overheads associated with a synchronous TLB shootdown.

We developed ecoTLB as a proof-of-concept in Linux, comparing it with stock Linux and ABIS, a recent approach to reduce the IPI overheads. ecoTLB makes the following contributions:

- ecoTLB provides an eventual TLB coherence mechanism for *free* operations that use PCIDs, and shows the benefits of ecoTLB on various x86 architectures.
- ecoTLB provides a lazy mechanism for *page swapping* with Infiniswap, a solution for next-generation data centers that use disaggregated memory.
- We show the benefits of ecoTLB with real-world applications, such as Memcached, Make, and Mosaic, running with Infiniswap. In addition, ecoTLB improves the performance of *free* operations by 59.1% on a 120-core machine and improves the performance of Apache by up to 13.7% compared to baseline Linux, and by up to 48.2% compared to ABIS on a 16-core machine.

## 2 BACKGROUND AND MOTIVATION

We first provide a primer on a current OS' `munmap()` (using PCIDs) and page swapping design. We then provide the needed background on Latr, which provides the components needed by ecoTLB. The explanation about Linux is based on version 4.14, unless specified.

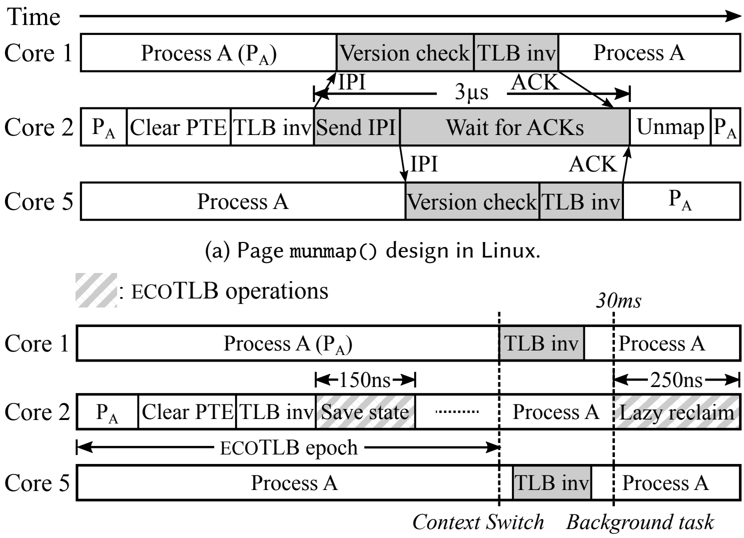
## 2.1 Existing OS Design using PCIDs

Most architectures, including x86, do not support TLB coherence. In x86, the `PCIDE` bit in the `CR4` register enables the PCID feature. When PCIDs are enabled, x86 supports two operations on TLBs: invalidating a TLB entry based on a PCID using the `INVPCID` (only on supported architectures) or `INVLPG` instruction and flushing all local TLB entries associated with the current PCID by writing to the `CR3` register with the 63rd bit set to zero. `INVPCID`, which invalidates TLB entries of a given PCID, is not supported by all architectures that support PCIDs (e.g., Intel Ivy bridge and before). In such architectures, in addition to flushing the entire TLB, `INVLPG` can be used to invalidate TLB entries associated with the *current* PCID. Even with PCIDs, the x86 mechanisms provide control only over the local, per-core TLB. Operating systems that use PCIDs, such as Linux and FreeBSD, use IPIs to invalidate entries in remote TLBs, a process known as *TLB shootdown*. IPIs are delivered via the Advanced Programmable Interrupt Controller (APIC) [26] that limits flexible multicast delivery which induces software overheads [42].

***Free* operations with PCIDs.** Since PCIDs are limited in number (e.g., 4096 in x86 architectures), Linux uses a small pool of PCIDs per core (e.g., six PCIDs per core) instead of assigning a unique PCID per process, thereby allowing multiple running processes on a given CPU to take advantage of the PCID mechanism. In addition, the kernel maintains two TLB versions: a per-process version that tracks the page table modifications and a per-CPU-PCID version that tracks the TLB state. These versions track the deviation of the per-core TLB state from the per-process TLB state, which is used to perform a shootdown or a flush during a free operation. We analyze the existing handling of a free operation (`munmap()` in Linux), which uses the PCID and version mechanism on a system with three cores (as shown in Figure 1a). The OS receives an `munmap()` system call from the application to remove a set of virtual addresses on core 2 with the current process running on all existing cores (1, 2, and 5). The `munmap()` handler removes the page table mappings for the set of virtual addresses, and frees the virtual addresses and its associated physical pages. In addition, the `munmap()` handler increments the per-process version by one. Core 2 performs a local TLB invalidation for the set of virtual addresses, if the per-process version is greater (newer) than the per-CPU-PCID version by one, before initiating an IPI (to cores 1 and 5) to perform the TLB shootdown. The TLB is flushed if the per-process version is greater (newer) than the per-CPU-PCID version by more than one, which is used by idle cores that deviate from the per-process version to flush their TLB entries. On receipt of the interrupt, cores 1 and 5 perform a local TLB invalidation or flush, depending on the per-process and per-core-PCID version, in their IPI handlers and send an `ACK` to core 2 by the means of cache coherence. After the TLB shootdown, each core sets the per-core-PCID version to the per-process version. After receiving both `ACK`s from cores 1 and 5, the `munmap()` handler on core 2 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 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.

Though the above overheads are addressed by LATR using a lazy TLB shootdown mechanism, ECOTLB addresses two challenges in a lazy mechanism that uses PCIDs: one challenge is handling the version update, and another challenge is to provide a generalized lazy mechanism for x86 architectures that do not support `INVPCID`.

**Kernel page table isolation.** KPTI is a feature available in the Linux kernel to mitigate the Meltdown [35] security vulnerability, which splits the page tables into kernel and user-space page tables. With this patch, any kernel to user-space transition, and vice versa, results in a context switch



(a) Page munmap() design in Linux.

(b) Page munmap() design in ecoTLB.

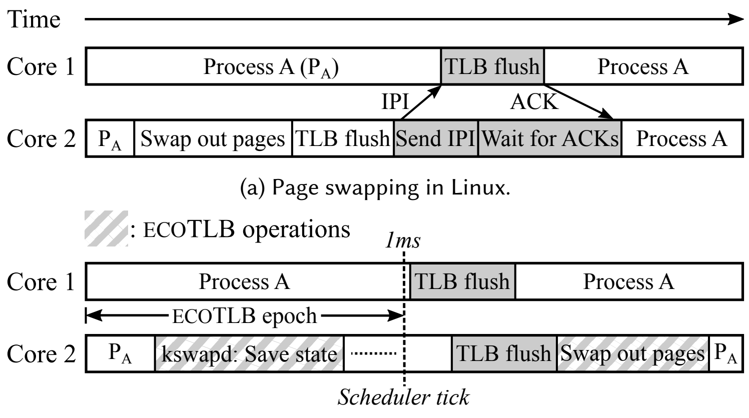
Fig. 1. An overview of the operations involved in unmapping a page in both Linux and ecoTLB. ecoTLB removes the instantaneous TLB shootdown from the critical path by executing it asynchronously.

that updates the CR3 register. The PCID feature in the Linux kernel avoids the expensive TLB flush needed during a CR3 register update. However, an munmap() system call results in an expensive TLB flush with x86 architectures that do not support INVPCID, as a targeted TLB shootdown is not possible with the PCIDs for the kernel and user space being different. ecoTLB's approach optimizes the PCID mechanisms needed to support KPTI with various x86 architectures.

## 2.2 Page swapping

Page swapping is a feature in commodity OSes, such as Linux and FreeBSD, to swap least recently-used (LRU) pages to disk during high memory pressure. In Linux, a kernel background task (kswapd) maintains an active and inactive list of pages. To begin with, the kernel adds the allocated pages to the inactive list. By tracking the page table entry (PTE) access bits, kswapd takes an informed decision to move pages from the inactive to the active list, and vice versa. During high memory pressure, pages are swapped out to disk from the inactive list, which triggers a synchronous TLB flush as shown in Figure 2a.

With the advent of disaggregated data centers, the paradigm for page swapping shifts from disks to remote memory using fast network interconnects. Instead of swapping pages to disk, recent research systems, such as Infiniswap [28], advocate for the usage of remote memory using Infiniband RDMA, which reduces the tail latency of page swapping by up to 61×. Due to the reduced remote paging latency, the TLB shootdowns needed for swapping become an important contributor to the cost of page swapping (contributing up to 18% for a Memcached workload using Infiniswap). ecoTLB changes the swap mechanism to swap pages after its lazy TLB shootdown, eliminating expensive IPIs and interrupt handlers. Importantly and due to its lazy technique, ecoTLB operates on the pages that are being swapped out in the background (e.g., via a swapping daemon) rather than directly in the critical path when running into memory pressure during allocation of new memory.



(a) Page swapping in Linux.

(b) Page swapping in ecoTLB.

Fig. 2. Page swapping in Linux and ecoTLB. ecoTLB removes the need for an immediate TLB flush after pages are swapped out.

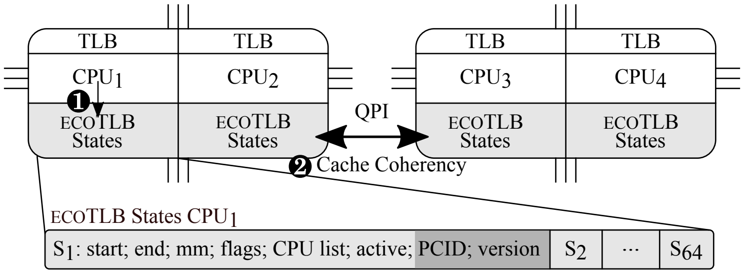


Fig. 3. Overview of ecoTLB, its interaction with the system, and the ecoTLB *states* (❶). The states are shared with cores on other sockets via the cache coherency (❷) protocol. ecoTLB extends the Latr states and adds a versioning record in accordance with the PCID versioning approach (as shown in dark gray).

## 2.3 Lazy TLB shootdown

Latr [31] proposes an asynchronous approach to eliminate the remote TLB shootdown overhead when using IPIs. During a *free* operation, it saves the TLB shootdown information in Latr states without issuing a synchronous TLB shootdown. During the next *scheduler tick*, all cores invalidate their TLB entries lazily using the Latr states. Latr frees the virtual and physical address only after this lazy TLB flush, which retrains the correctness for the lazy approach.

However, Latr was built and evaluated on Linux 4.10, without support for PCIDs and did not address the issue of emerging, disaggregated data centers. ecoTLB addresses these shortcomings by extending Latr to an Infiniswap-enabled environment and uses PCIDs to support the latest changes of the Linux kernel including the KPTI feature. ecoTLB extends Latr's states, by including the PCID and a version, as shown in Figure 3.

## 2.4 Challenges

An asynchronous approach to TLB shootdowns introduces various challenges, in the form of corner cases, for the correctness of the virtual-memory operations. With an asynchronous approach, the operation that triggers a TLB shootdown returns immediately, and the cores perform a lazy TLB

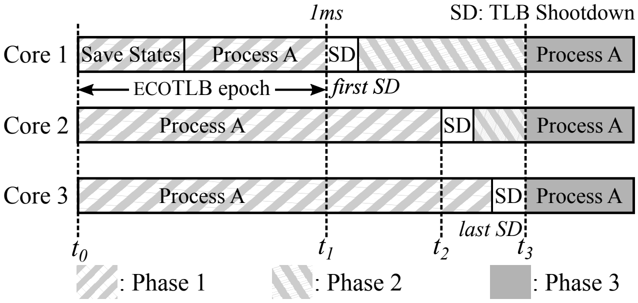


Fig. 4. An overview of exemplary cases that an eventual approach to TLB coherence has to cover for correctness. We identify cases 1 through 3 which are covered in ecoTLB's design to ensure correctness.

shootdown, which introduces different phases of TLB coherence. We define the different phases, as shown in Figure 4, of the lazy TLB shootdown below:

- Phase 1 ranges from initiating a TLB shootdown to the time a remote core performs a TLB shootdown, which is specific to each core. For example, on core 2, this phase ranges from $t_0$ to $t_2$.
- Phase 2 ranges from the current core's TLB shootdown to the time of the last core's shootdown, which is specific to each core. For example, on core 2 this phase ranges from $t_2$ to $t_3$. The last core performing the TLB shootdown (e.g., core 3) does not enter this phase. The cores that did not cache the TLB entries will be in this phase up to $t_3$.
- Phase 3 is any time after $t_3$. In this phase, all TLB entries are coherent with the corresponding page table entries.

During an *ecoTLB epoch*, each core in the system will either be in phase one or phase two. The TLBs become eventually consistent at the end of phase two after all cores performed their TLB shootdown. ecoTLB should provide correctness in all the three phases for the supported virtual-memory operations.

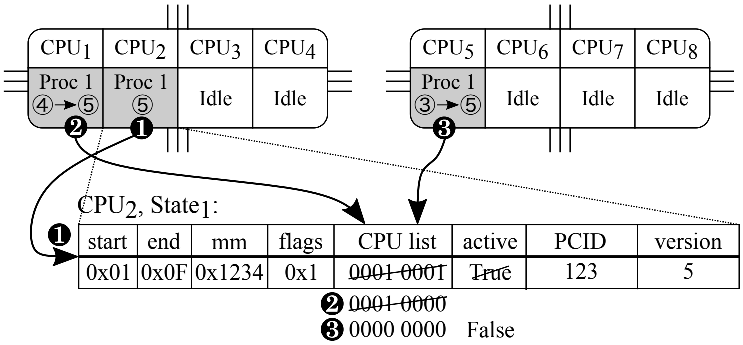
## 3 OVERVIEW

ecoTLB proposes an eventual TLB shootdown approach, that uses PCIDs, for virtual memory operations such as *free* (`munmap()` and `madvise()`) and *page swapping* that uses Infiniswap.

**ecoTLB epoch.** ecoTLB defines *TLB shootdown epochs*, an interval in which the TLB shootdowns to the remote cores are recorded in ecoTLB states. These epochs serve as a batching interval, in which remote shootdowns are not performed synchronously, but are only recorded to ecoTLB states. During this interval, the TLB entries are inconsistent. At the end of an epoch, each core invalidates the particular TLB entries from its recorded ecoTLB state which makes the TLB eventually consistent.

**Support for free operations with PCID.** The key idea for *free* operations with ecoTLB is the batched per-core version update which enables the delayed reuse of virtual and physical pages and a lazy TLB shootdown. For free operations, a context switch is used as an epoch interval (e.g., up to 30 ms). During every context switch, the states saved during the free operations are used to perform the lazy TLB shootdown.

**Support for page swapping operations.** The key idea for *page swapping* with ecoTLB is the delayed swap out operation of inactive pages, after an ecoTLB epoch. The current epoch interval used for page swapping is the scheduler tick (e.g., 1 ms). During an epoch, the TLB entries of inactive pages are invalidated on all cores. Read or write accesses to inactive pages, during an



| start | end | mm | flags | CPU list | active | PCID | version |
|---|---|---|---|---|---|---|---|
| 0x01 | 0x0F | 0x1234 | 0x1 | ~~0001 0001~~ | ~~True~~ | 123 | 5 |

Fig. 5. Executing a TLB shootdown using the ecoTLB states, ❹ indicates a local TLB version of 4. Core 2 unmaps a page and sets up the ecoTLB state (❶) to allow cores 1 (❷) and 5 to clear/flush their local TLBs. Core 5 is the last core and resets the *active* state (❸). Cores 2 and 5 transition their local TLB version to ❺.

ecoTLB epoch, are tracked using the PTEs' access bits, allowing ecoTLB to prevent swapping out pages accessed during an epoch.

## 4 DESIGN

We describe the design of ecoTLB for x86-based Linux 4.14. We first introduce the additional fields needed in ecoTLB states, and explain their usage to maintain the PCID version. We further explain ecoTLB's design for *free* and *page swap* operation.

### 4.1 Handling Free Operations with PCIDs

**ecoTLB states.** To maintain the per-process and per-core-PCID version, ecoTLB extends the Latr states to store the per-process version corresponding to each TLB shootdown. The core initiating the TLB shootdown adds a state entry that contains the per-process version. During the ecoTLB state sweep, the version available in the state is used to update the per-core-PCID version. Figure 3 shows ecoTLB's extension to Latr states.

**State sweep after an epoch.** ecoTLB performs the state sweep only during the context switch, instead of performing a state sweep during both context switch and scheduler ticks in Latr. If the TLBs are lazily invalidated during a scheduler tick, the `INVPCID` instruction does not point to the correct PCID. Though this can be solved by adding the PCID to ecoTLB states, invalidating the TLB entries using the appropriate PCID, some x86 architectures (e.g., Ivy bridge) do not support `INVPCID`. By performing the state sweep only during the context switch, ecoTLB invalidates the TLB entries pertaining to a PCID before switching to another PCID.

The advantage of performing the state sweep during a context switch is two-fold: one is avoiding the full TLB flushes on architectures that do not support `INVPCID` which potentially causes more TLB misses; another is the applicability of this approach to tickless kernels (e.g., using `CONFIG_NO_HZ_FULL` option in Linux) which do not have scheduler ticks.

**Handling `munmap()` with PCIDs.** For `munmap()` operations, ecoTLB performs *lazy memory reclamation* that frees the virtual and physical pages only at the end of phase 2. Lazy memory reclamation ensures that the virtual and physical pages are not reused until the TLB entries are eventually consistent. Using the lazy memory reclamation and the ecoTLB states, ecoTLB removes the synchronous shootdown from the critical path of free operations that use PCIDs. Instead, on execution

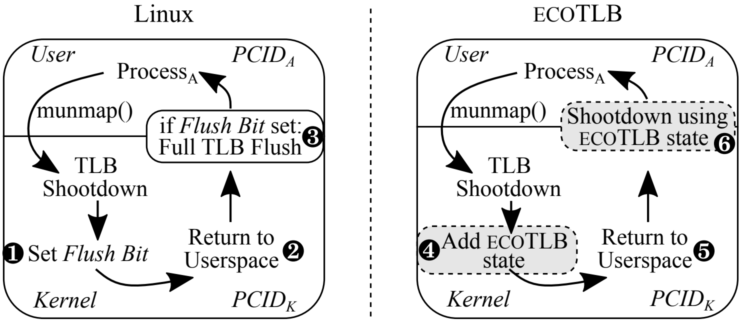


Fig. 6. The process of completing an `munmap()` call with KPTI enabled on a system without support for `INVPCID`. Linux can only handle this situation using a full TLB flush while ecoTLB can use its states to facilitate a targeted shootdown (changes to Linux in gray).

of these operations, ecoTLB clears the PTEs, and simply saves the states, including the per-process version, without sending an IPI immediately.

A detailed example of ecoTLB handling an `munmap()` operation is shown in Figure 1b and Figure 5. Core 2 executes the `munmap()` system call resulting in a TLB invalidation on core 2, followed by saving the ecoTLB state which includes the cores 1 and 5 in the CPU bitmask, the PCID, and the per-process version. The saved per-process version is one greater than the per-CPU-PCID version stored in each core. In addition, core 2 adds the virtual and physical pages to a lazy list, without freeing them immediately. Due to the CPU bitmask, core 1 and 5 invalidate their local TLB entry at the end of an epoch, during a context switch, and reset their respective CPU bitmask in the ecoTLB state. However, if any of the core's per-CPU-PCID versions deviate from the version in the state entry by more than one, then the respective core flushes the TLB instead of performing a TLB shootdown (e.g., core 5 has to increment from version core 3 to core 5). This mechanism, similar to stock Linux, takes care of cores that deviate from the per-process version. In addition, core 1 and 5 update their respective per-CPU-PCID version with the per-process version stored in the corresponding state entry.

**Kernel page table isolation.** With a separate address space between the kernel and the user space in KPTI, ecoTLB provides a general solution that supports architectures that do not support `INVPCID` (such as Ivy bridge). During an `munmap()` operation with KPTI, since the kernel is running in a different address space, the kernel needs to use the `INVPCID` instruction to invalidate the user space TLB entry. If the architecture does not support `INVPCID` the user space TLB entries are flushed after a context switch to user space (by updating the `CR3` register). With ecoTLB, even without the `INVPCID` instruction, the full flushes can be avoided by using ecoTLB states. With ecoTLB, the kernel only records ecoTLB states, and uses them to perform TLB shootdowns using `INVLPG` during a context switch to user space. Since the TLB shootdown is performed after a context switch, `INVPCID` is not mandatory for PCID support. We give an example of ecoTLB handling this situation in Figure 6: Linux, due to missing `INVPCID` support, has to set up a bit to indicate flushing of the TLB (❶) which on return to user space (❷) is executed for the user's *PCID*$_A$ (❸). ecoTLB avoids the full TLB flush by setting up an ecoTLB state (❹) instead of the *Flush bit.* While returning to user space (❺), ecoTLB uses the state to accomplish a targeted shootdown using `INVLPG` (❻).

To perform TLB shootdowns after updating the `CR3` register, ecoTLB currently maps the ecoTLB states to both the kernel and user address space. With this design, the ecoTLB states expose a range of virtual addresses that need to be flushed to the user space process. However, this can be

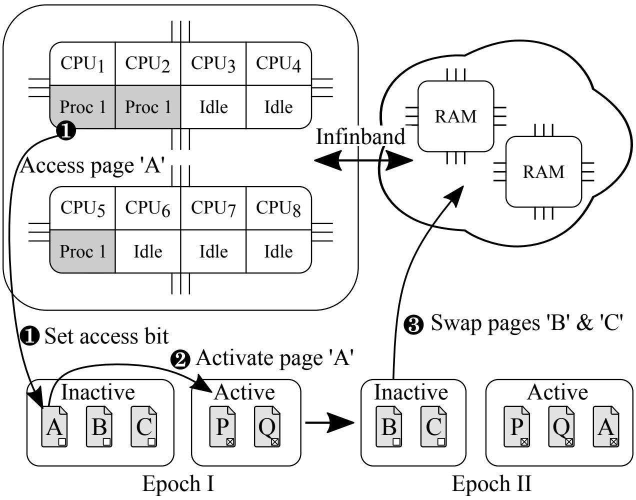


Fig. 7. Infiniswap with support for lazy swapping with ecoTLB. Access bits are used to move pages on access (❶) to the *active* list (❷). After the ecoTLB epoch is finished and all TLB invalidations have taken place, the pages are swapped out to remote memory (❸).

avoided by having a specific user space address that represents ecoTLB states, and copying the process specific entries into these user ecoTLB states.

**Correctness for free operations.** We show the correctness of ecoTLB free operations during the three phases ( described in §2.4). Accesses to unmapped memory on cores in phase one continue to proceed using the TLB entry available in their cores. As the pages are not freed yet, which ensures that the virtual address and physical page are not reused, reads and writes are allowed to proceed. Accesses to unmapped memory on cores in phase two and three results in a page fault leading to, e.g., a segmentation error, depending on the OS implementation. During phase three, before freeing the pages, file backed dirty pages are written back, which accommodates the writes during phase one.

### 4.2 Handling Page Swapping with Infiniswap

In addition to supporting free operations with PCIDs, ecoTLB's design provides a lazy mechanism for page swap operations. We discuss the ecoTLB mechanism for page swapping in this section. The current page swap design in Linux includes a remote TLB shootdown (see Figure 2a), which accounts for up to 18% of the overall time in the case of a page swap using Infiniswap. ecoTLB's mechanism for page swapping provides a lazy TLB shootdown approach, eliminating the expensive TLB shootdown operation.

**Lazy page swapping.** The key part of ecoTLB's design for background page swapping (e.g., kswapd in Linux), is to swap pages lazily after ecoTLB's epoch of 1 ms, after the inactive pages' TLB entries are invalidated. ecoTLB maintains an invariant that the pages are swapped out, after an ecoTLB epoch, only if their TLB entries are not present in any of the TLBs. The pages accessed during the epoch, whose TLB entries are present in any of the TLBs, are not swapped out and added to the active list.

ecoTLB extends Latr's states to record the TLB shootdown states needed by a page swap. The page swap background task instead of swapping out inactive pages immediately, adds the state to the ecoTLB states. Due to the large inactive list, the added state indicates a full TLB flush on all

CPU cores, similar to the existing page swap mechanism (handled using IPIs). After an ecoTLB epoch, when all cores have fully flushed their TLBs, the inactive pages are swapped out to remote memory.

To swap out pages to remote memory after an epoch, ecoTLB has to ensure that the TLB entries of inactive pages are not present in any cores. However, some cores could access the page in phase two, which would set up the TLB entry again. ecoTLB tracks such inactive page accesses using the access bit in the PTE. For tracking accesses to inactive pages, the background task, in addition to adding the states, resets the access bit in the PTE for all the pages in the inactive list. The current active page detection mechanism in Linux is best effort, which does not trigger a TLB shootdown when resetting an access bit. After an ecoTLB epoch inactive pages that have their access bit set, indicating that those pages were accessed during phase 2, are moved to the active list while other pages are (potentially) swapped out. After a page is swapped with ecoTLB mechanism, none of the TLBs contain an entry for the swapped page eliminating the need for a TLB shootdown.

**Page swap policy change.** ecoTLB's lazy mechanism delays page swapping by an ecoTLB epoch, providing an additional epoch to track page accesses. By tracking accesses during an epoch, ecoTLB changes the existing page swapping policy by not swapping out pages accessed during an ecoTLB epoch. Using its lazy swapping policy, ecoTLB improves, in addition to removing the TLB shootdown overhead, the temporal locality of pages accessed during an epoch by not swapping them out.

**Correctness for page swapping.** We show the correctness of ecoTLB's page swapping operation during the three phases. Page accesses for inactive pages during phase one and two proceed as before, with the hardware setting the access bit when needed. In phase two, any page access to an inactive page sets the access bit, as the TLB is flushed and the access bit is reset when setting up the ecoTLB state. Correctness for page swapping is thus maintained by not swapping out any pages with the access bit set, maintaining the invariant.

One potential race condition is in phase three: when pages are being swapped out, the access bit could be set in parallel with the page being accessed, e.g., one core is performing swapping while another core accesses the page resulting in the access bit set. ecoTLB avoids this race condition by checking the access bit immediately after resetting the present flag in the PTE. If the access bit is set, the page is moved back to the active list and is not swapped out.

## 5 IMPLEMENTATION

We implemented the ecoTLB prototype in about 1K lines of code by extending Linux 4.14 for PCIDs and Linux 4.10 for page swapping. We used Linux 4.10 for page swapping as Infiniswap was supported on Linux 4.10.

**Free operation.** The `context_switch` function is modified to perform the state sweep after the user space context switch is performed. For KPTI, the function `__native_flush_tlb_single` is modified to not update the flag `user_pcid_flush_mask` that indicates to flush the TLB during a context switch.

**Page Swapping.** The function `shrink_list` is modified to not immediately trigger `shrink_active_list`. Instead, a new handler that adds an entry to the ecoTLB state for each inactive page is invoked, while `shrink_active_list` is invoked later using the background thread. The new handler, in addition to adding entries to the ecoTLB states, resets the access bit in the PTE for all the pages in the inactive list. The logic to move a page from the inactive to the active list based on the access bit is part of `shrink_active_list`.

| Machine Type | Commodity data center [41] | No `INVPCID` | RDMA | Large NUMA |
|---|---|---|---|---|
| Model | E5-2630 v3 | E5-2620 v2 | E5-2658 v3 | E7-8870 v2 |
| Frequency | 2.40 GHz | 2.10 GHz | 2.20 GHz | 2.30 GHz |
| # Cores | 16 | 12 | 12 | 120 |
| Cores × Sockets | 8 × 2 | 6 × 2 | 6 × 2 | 15 × 8 |
| RAM | 128 GB | 64 GB | 128 GB | 768 GB |
| LLC | 20 MB× 2 | 15 MB× 2 | 30 MB× 2 | 30 MB× 8 |
| L1 D-TLB entries | 64 | 64 | 64 | 64 |
| L2 TLB entries | 1024 | 512 | 1024 | 512 |
| RDMA | × | × | ✓ | × |
| `INVPCID` | ✓ | × | ✓ | × |

Table 2. The four machine configurations used to evaluate ecoTLB. We run ecoTLB on a commidity data center machine, a previous-generation commodity data center machine without support for the `INVPCID` instruction, an RDMA-enabled machine for supporting a setting of disaggregated memory, and a large, 120-core NUMA machine.

## 6 EVALUATION

We implement our proof-of-concept prototype of ecoTLB on top of Linux 4.14 for the *free* operations and Linux 4.10 for the *page swapping* with Infiniswap. We use the respective versions of Linux as the baseline for our evaluation and include ABIS [1] for a subset of experiments. ABIS is a recent research prototype, based on Linux 4.5, which uses PCIDs and access bits to track the sharing of pages between cores to avoid potentially unnecessary IPIs.

We evaluate ecoTLB, using the mentioned setup, to answer the following questions:

- What is the benefit of ecoTLB for microbenchmarks on a small and a large NUMA machine?
- What is ecoTLB's impact on a real-world application, Apache, which generates a large number of TLB shootdowns?
- What is ecoTLB's benefit when running in an Infiniswap-enabled environment when swapping pages via RDMA?
- What is the overhead of ecoTLB in terms of memory utilization and for applications with a low number of TLB shootdowns?

### 6.1 Experiment Setup

We evaluate ecoTLB on four different machine setups, as shown in Table 2. The primary evaluation target is the *2-socket, 16-core* machine, a commodity data center configuration [41], while we also show the impact of ecoTLB on a large NUMA machine with *8 sockets and 120 cores*. Furthermore, we show the effects of swapping with Infiniswap in an RDMA-enabled setup with a *2-socket, 12-core* machine. Each benchmark is run five times and we report the average results.

The machines are configured without support for transparent huge pages, as this mechanism is known to increase overheads and introduces additional variance to the benchmark results [33]. Furthermore, to reduce variance in the results, we run the benchmarks on the physical cores only. We furthermore deactivate Linux's automatic balancing of memory pages between NUMA nodes, AutoNUMA, as it might introduces TLB shootdowns during the migration of a page [31]. We evaluate ecoTLB with PCID support using Linux 4.14, the first Linux version to add support for PCIDs. Unless otherwise noted, we disable the new KPTI feature (see §2.1). For the case of Infiniswap, we evaluate ecoTLB using Linux 4.10 due to Infiniswap's build requirements. For Infiniswap, we set up two machines with Infiniband (using the ConnectX3 adapter [39]), both machines have 12 physical cores on two sockets. The main machine, hosting the applications, runs ecoTLB while the remote machine, hosting the in-memory swapping device, runs an unmodified Linux 4.10 kernel. We compare ecoTLB to Latr in Table 3 for a simple microbenchmark and note that Latr is not included for other evaluations as it supports neither PCIDs nor swapping with Infiniswap.

**Machine types and linux kernel version.**

The *commodity data center* machine type is used with Linux kernel version 4.14 for all experiments that evaluate ecoTLB's PCID approach. The Linux kernel version 4.14 is used for the PCID experiments as PCIDs are supported in the Linux kernel only in the 4.14 version. The *no `INVPCID`* machine type is used to demonstrate the impact on previous-generation commodity data center machines which support PCIDs but not the newer `INVPCID` instruction, rendering context switches more expensive. The Linux kernel version 4.14 is used with this machine type. The *RDMA* machine type is used with Linux kernel version 4.10 for all experiments with Infiniswap. Infiniswap's kernel patch is ported to Linux kernel 4.10 to keep the kernel version consistent for all the experiments, excluding the PCID experiments. The *Large NUMA* machine type is used with Linux kernel version 4.10 for the memory `unmap()` experiments. This machine type is used to show the scalability bottleneck of a synchronous TLB shootdown mechanism in current machines with an increased number of sockets. The *Commodity data center* machine type is used with Linux kernel version 4.10 for all experiments unless explicitly stated. The *commodity data center* machine type is used to show the impact of a synchronous TLB shootdown mechanism in machines that are widely deployed in existing data centers. As detailed above, excluding the experiments that evaluate ecoTLB's PCID approach, all other evaluations are done using the Linux kernel version 4.10.

**Benchmarks.** We evaluate ecoTLB on a set of microbenchmarks, as well as full-application benchmarks. For evaluating the improvements realized with ecoTLB when using PCIDs, we use the Apache web-serving benchmark to show the impact of frequent TLB shootdowns on end-to-end throughput. We furthermore evaluate ecoTLB with Infiniswap using three applications to show the impact of TLB shootdowns during swapping: We choose Memcached, a key-value store to demonstrate the impact on tail-latency, Make as an example of a throughput-oriented application, and Mosaic, a graph-processing engine, as an example of in-memory computing with low locality and high throughput demands.

## 6.2 Microbenchmarks

We first evaluate ecoTLB's impact on Linux with PCIDs using the *commodity data center* and the *large NUMA* machine types. We devise a microbenchmark that simply `mmaps` a page, shares it with the specified number of cores and subsequently `munmaps` the page again, resulting in a TLB shootdown per `munmap()`. We run this microbenchmark on both the 2-socket, 16-core machine (the *commodity data center* machine type) and the 8-socket, 120-core machine (the *large NUMA* machine type) and show the impact of the synchronous TLB shootdown with the PCID-based Linux design.

**Small NUMA machine.** For the case of the 2-socket, 16-core machine, the results are shown in Figure 8 and demonstrate improvements for ecoTLB in the cost of an `munmap()` call by up to 49.6%. Furthermore, we show the cost of only the TLB shootdown to increase to more than 2 $\mu$s per `munmap()` call with 16 cores (for a proportion of 48.0%), while ecoTLB's mechanism of saving the states consistently takes less than 500 ns. In addition, Figure 8 shows the improvements of ecoTLB over Latr for `munmap()` and TLB shootdown. We also compare these results to Latr and its baseline, Linux 4.10, in Table 3 for the case of the 2-socket, 16-core machine and the 16-core data point. These results show that ecoTLB performs slightly better compared to Latr (by 6.9%), while Linux 4.14 (with its PCID-based design) outperforms Linux 4.10 by 37.7%. Both ecoTLB and Linux 4.14 profit from various optimizations in the path of the `munmap()` call between the two versions

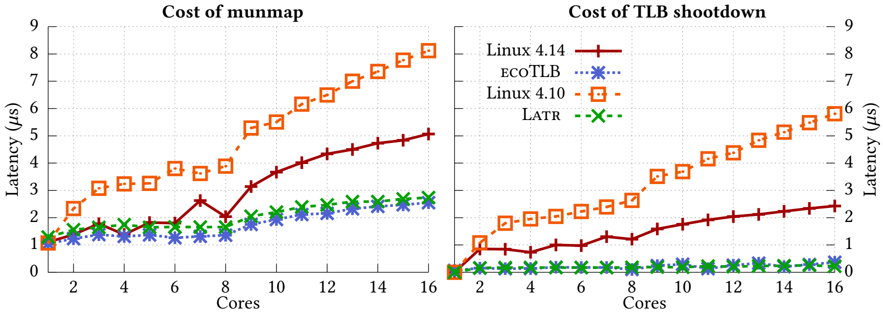


Fig. 8. The cost of an munmap() call for a single page with 1 to 16 cores in our microbenchmark, using the *commodity data center* machine type. TLB shootdowns account for up to 48.0% of the total time and ecoTLB is able to improve munmap() by up to 49.6%.

| System | munmap() Cost | Cost of TLB Shootdown |
|---|---|---|
| Linux 4.10 | 8.12 $\mu$s | 5.81 $\mu$s |
| Latr | 2.74 $\mu$s | 0.43 $\mu$s |
| Linux 4.14 | 5.06 $\mu$s | 2.43 $\mu$s |
| ecoTLB | 2.55 $\mu$s | 0.37 $\mu$s |

Table 3. Comparison of our microbenchmark at 16 cores for Linux 4.10, Latr, Linux 4.14, and ecoTLB with the *commodity data center* machine type. Linux 4.14 improves its cost of munmap() and the TLB shootdown as a result of improved idle-core tracking and a faster shootdown routine while ecoTLB still retains its advantage over Linux 4.14.

while Linux 4.14 also improves the cost of a single TLB shootdown drastically by 58.2%. This is attributed to both a shorter function that needs to be executed on the remote cores via the IPI mechanism as well as a much improved tracking and handling of idle cores and their specific TLB characteristics. With this, Linux 4.14 is able to remove more CPUs from the set of CPUs that need to be involved in a single TLB shootdown, thus cutting the cost of the whole shootdown procedure. In spite of these optimizations in Linux 4.14, the TLB shootdown still attributes to 48.3% of the munmap() cost. We conclude that ecoTLB improves the munmap() cost by 49.6% compared to Linux 4.14 by removing the need for a synchronous TLB shootdown.

**Large NUMA machine.** We also run this microbenchmark on our 8-socket, 120-core machine and show the results in Figure 9. On this machine, ecoTLB shows a significant benefit of up to 59.1% (i.e., more than 88 $\mu$s) per invocation of munmap(). For the lower 20 cores, ecoTLB shows a similar or at times slightly slower behavior as Linux, however ecoTLB outperforms Linux consistently starting at about 30 cores. We note that the PCID-based mechanism of Linux 4.14 does not scale to 120 cores as the overhead of IPI operations as well as global atomic operations when accessing the global version number for Linux result in a scalability bottleneck from about 30 cores. Eventually, the TLB shootdown, using IPIs, inside the munmap() call takes up to 69.2% of the overall time of munmap(). ecoTLB on the other hand shows an initial increase in the cost of munmap() due to increased cost in the cache coherency protocol but levels off at around 50 cores once the most-remote socket (which is 2 hops away from the local socket) has been reached.

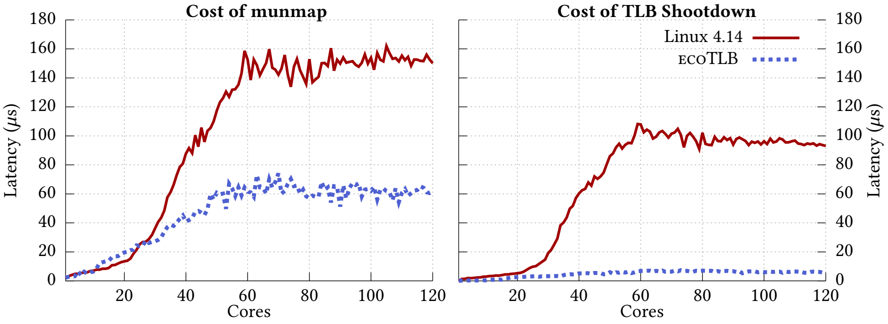


Fig. 9. The cost of munmap() for a single page when running on the *large NUMA* machine type (8 sockets, 120 cores) with Linux and ECOTLB. For Linux, the TLB shootdown accounts for up to 69.2% of the total time while ECOTLB improves the cost of munmap() by up to 59.1%.

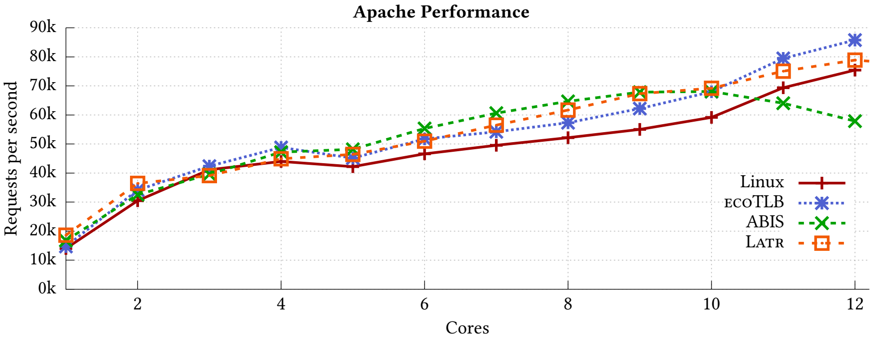


Fig. 10. The requests per second when running Apache on the *commodity data center* machine type (2 sockets, 16 cores) with Linux, ABIS, LATR, and ECOTLB. ECOTLB outperforms LATR by up to 8.7% and Linux by 13.7%. In addition, ECOTLB performs up to 48.2% better compared to ABIS, which is suffering from overhead to track sharing of the pages.

## 6.3 Apache

We evaluate ECOTLB by running Apache, a web server, serving static webpages using the *commodity data center* machine type. We compare the requests per second of Apache with Linux 4.14, ABIS [1], LATR [31], and ECOTLB on the 2-socket, 16-core machine.[2] We use the Wrk [24] HTTP request generator, using four threads with 200 connections each for 30 seconds, to send requests to Apache, which hosts a static 10 KB webpage. Apache and Wrk are run on the same machine to avoid the network from becoming the bottleneck. We ensure that Wrk and Apache are each running on a distinct set of cores, leaving up to 12 cores to Apache. We configure Apache without logging and use the (default) mpm_event module to process incoming requests. This module spawns a small number of processes which in turn spawn another set of threads (e.g., 30 threads) per process to handle the incoming requests. In the process of handling a request, Apache mmaps the requested, static file to process and sends a response before calling munmap to clear the mapping. This behavior of frequent mapping and unmapping of a few pages across generates a large number of remote TLB shootdowns due to the (potential) sharing of the mapped pages between multiple cores.

[2] Note that the machine configuration between this experiment and the one found in the LATR paper [31] differs

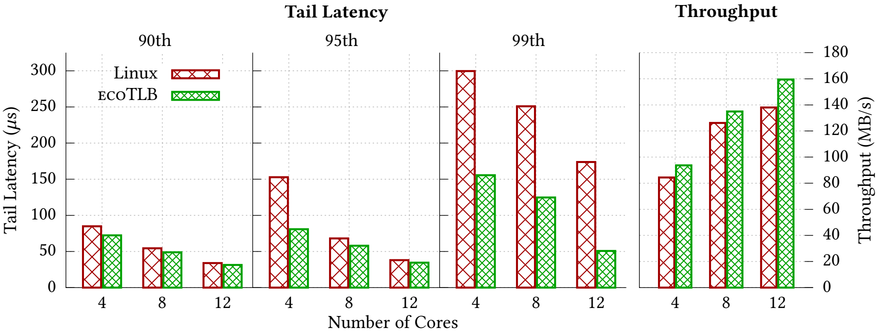


Fig. 11. The impact of swapping using Infiniswap with Linux and ecoTLB for Memcached in terms of both latency and throughput for a varying number of cores using the *RDMA* machine type. ecoTLB improves the 99th percentile tail latency by up to 70.8% and the throughput by up to 13.5% by reducing the impact of synchronous TLB shootdowns.

The results of the Apache benchmark are shown in Figure 10, highlighting the improvements possible with ecoTLB. When running on all 16 cores (12 cores for Apache and 4 for Wrk), ecoTLB shows a 13.7% improvement over the baseline Linux 4.14 PCID-based design. Overall, compared to Linux, ecoTLB consistently outperforms Linux by up to 15%. This is attributed to ecoTLB's efficient handling of TLB shootdowns compared to Linux and its synchronous, PCID-based design. Compared to ABIS, ecoTLB performs up to 48.2% better (on 12 cores) while temporarily showing a small overhead of up to 12.1%, as ABIS reduces the number of shootdowns needed as part of its design. However, we show that, with higher core count, ABIS's overhead to keep track of page sharing using access bits becomes a scalability bottleneck. In addition, compared to Latr, ecoTLB improves the performance of Apache by up to 8.7%.

### 6.4 Page Swapping

We evaluate ecoTLB's benefits with page swapping using Infiniswap [28] that uses remote memory as the swap device, using the *RDMA* machine type with 2 sockets and 12 cores. We constrain the applications inside an `lxc` container and set the soft-memory limit of the container to about half of the applications working set to induce swapping via `kswapd`. Overall, the TLB shootdown accounts for up to 20% of the swapping time with Infiniswap when running Memcached with 5M keys using a recently published workload (ETC) by Facebook [6]. This workload shows around 100,000 TLB shootdowns per second from background swapping, as the kernel has to ensure that dirty pages are unmapped and not being written to on all cores before swapping them out.

**Memcached.** We use the mutilate tool [34] to send requests to Memcached [40], constraining Memcached and mutilate to separate NUMA nodes to minimize interference effects. We show the results with a differing number of cores when running Memcached with Infiniswap, on Linux and ecoTLB, for both tail latency and throughput in Figure 11. We show the tail latency for the 90th, 95th and 99th percentile, using 4, 8, and 12 cores. ecoTLB shows benefits for all of these percentiles, with a larger benefit being visible for higher percentiles (up to a reduction of 14.8%, 47.2%, and 70.8% for the 90th, 95th, and 99th percentile, respectively). ecoTLB helps in reducing the tail latency for in-memory caches, which is a critical goal to many data-center applications [9]. ecoTLB also shows a larger benefit when Memcached is run on less cores (e.g., on four cores) as Memcached doesn't scale well when increasing the number of cores [52], thus allowing for more

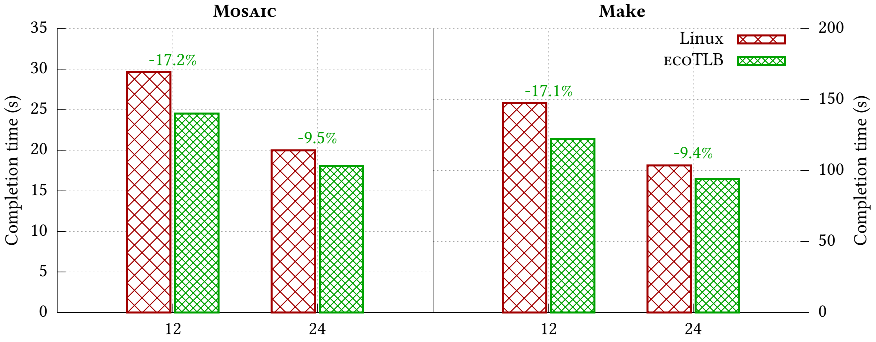


Fig. 12. The impact of swapping using Infiniswap with Linux and ecoTLB for Mosaic and Make using the *RDMA* machine type. ecoTLB is able to improve the applications' completion time by up to 17.2% as a result of the lazy swapping approach.

idle CPUs when using more cores which in turn results in a lower tail latency. ecoTLB is also able to improve the throughput of Memcached by up to 13.5% and 9.8% on average.

ecoTLB's deferred TLB shootdown algorithm allows pages to move from the *inactive* list to the *active* list (with the help of the active bits in the page table entry) during the epoch before the TLB shootdown is completed on all cores (e.g., 1 ms). For example, this *swap policy change* allows ecoTLB to move around 180 pages per second from the *inactive* to the *active* list when running Memcached on 12 cores, thus saving the overhead of having to swap out a page that actually would have been used soon after and would need to be swapped in again.

**Make and Mosaic.** We demonstrate ecoTLB's benefits when swapping with Infiniswap with two more applications, building a Linux kernel with Make and running a graph processing engine with Mosaic, focusing on the application's completion time. In more detail, Make compiles the Linux 4.10 kernel on a specified number of cores in a massively parallel fashion, loading the source files into memory in the process. On the other hand, Mosaic [38] is a graph processing engine, running in an in-memory mode, executing the pagerank algorithm algorithm on the twitter graph [32]. Mosaic initially loads the graph into memory before executing 10 iterations of the pagerank algorithm. We report the overall time taken for all 10 iterations.

For both applications, we run two configuration for both Linux and ecoTLB, using 12 and 24 cores. The results are given in Figure 12 and show that ecoTLB achieves a speedup of up to 17.2% for Make and 17.1% for Mosaic. ecoTLB shows a smaller benefit (of 9.5% and 9.4%, respectively) for the 24 core configuration as the system has to use all physical and virtual cores for that configuration which results in only marginal speedup, thereby reducing the impact of ecoTLB's improved handling of swapping via Infiniswap.

## 6.5 KPTI

We demonstrate the overhead imposed by the recent page table isolation (KPTI) mechanisms, introduced as a mitigation against the meltdown [35] attack, for both ecoTLB and Linux when running Apache. We run this benchmark on an Intel Sandy Bridge machine with 12 cores on 2 sockets which supports PCIDs but not the newer `INVPCID` instruction, using the *no `INVPCID`* machine type with Linux kernel 4.14. This results in a complete TLB flush on any *free operation*, as the kernel now operates using a different PCID as the application. Thus, clearing a single entry from the application's PCID space is not possible with the only fallback mechanism being a complete flush of the applications' PCID space.

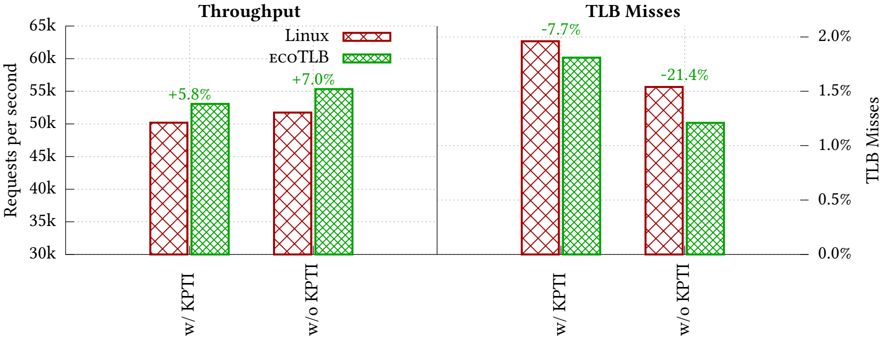


Fig. 13. The overhead imposed on both Linux and ECOTLB by enabling KPTI using the *no `INVPCID`* machine type (with support for PCIDs but without support for `INVPCID`) when running on all available cores. Even with KPTI enabled, ECOTLB still outperforms Linux *without* KPTI enabled by 3.5%, highlighting ECOTLB's ability to reclaim some of the performance lost with KPTI enabled in scenarios with a large number of *free operations*.

We show the results of this experiment in Figure 13 which demonstrate that the KPTI mechanism has a 4.5% overhead when running with KPTI enabled. ECOTLB also incurs an overhead of 4.2% when running with KPTI enabled due to the additional overheads from KPTI like cache and TLB pollution due to replicated page table hierarchies. However, ECOTLB is still able to improve the overall performance of Apache, compared to Linux *without* the KPTI mechanism, by 3.5%, demonstrating ECOTLB's real-world benefits with the new KPTI mechanism. ECOTLB accomplishes this by using its states to invalidate TLB entries during a context switch, after switching back to the application's PCID but before returning control back to the application. The benefits for ECOTLB with KPTI are due to two reasons: First, ECOTLB removes the overhead of sending IPIs and second, ECOTLB avoids flushing the complete TLB for the application's PCID. The benefits of the second reason are also seen when comparing the rate of TLB misses: ECOTLB is able to improve the rate of TLB misses by 7.7% when KPTI is enabled and 21.4% when KPTI is disabled. The larger improvement for the case of KPTI being disabled is attributed to the reduced cache and TLB pollution of the baseline design without a separate kernel address space. Finally, with this experiments, we acknowledge that ECOTLB improves the performance of KPTI, and does not completely solve the problem.

### 6.6 Overheads of ECOTLB

We evaluate the overhead of ECOTLB in terms of *memory utilization* and ECOTLB's impact on applications with few TLB shootdowns, using the *commodity data center* machine type.

**Memory utilization.** We perform a worst-case analysis in terms of memory utilization of ECOTLB's lazy memory reclamation based on the microbenchmarks presented. For one ECOTLB epoch (e.g., the default time slice on 16 cores of Linux's completely fair schedulers, CFS, is 30 ms), ECOTLB shows an overhead of up to 630 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 overheads ranges from 60 MB (for 2 cores sharing a single page) to 45 MB (for 16 cores sharing a single page). Using more pages, the memory overhead stays bounded by 630 MB, as the overhead of page table modifications and related operations then dominate the cost of the TLB shootdown. Considering the large virtual address space ($2^{48}$ bytes, with newer processors and 5-level page tables even $2^{57}$ bytes [27]) and the amount of RAM (64 GB and more) available in current servers, the memory overhead is not high (smaller than 0.97%) and is released back within a short interval (at most 30 ms).

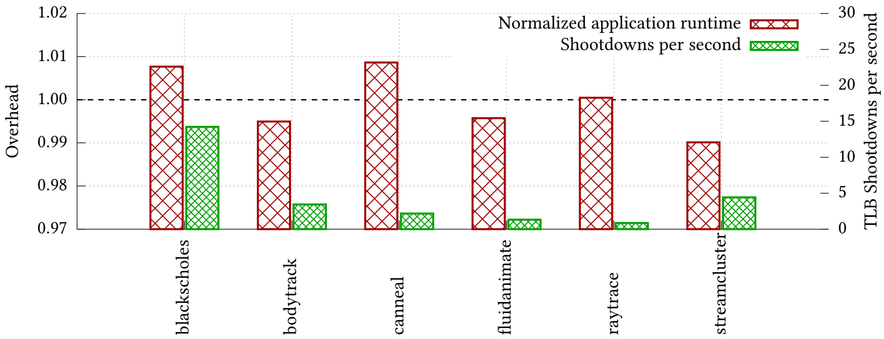


Fig. 14. The overhead of running ECOTLB is less than 1%, using the *commodity data center* machine type with 16 cores with a subset of PARSEC applications that show low number of TLB shootdowns due to added operations during a context switch.

**Application overhead.** We show the overhead for ECOTLB on a subset of PARSEC applications that perform very few TLB shootdowns in Figure 14. ECOTLB shows overheads of less than 1% as a result of a larger number of context switches and thus a larger number of potential ECOTLB state sweeps. However, ECOTLB also *improves* the performance of a subset of these applications as a result of optimizing background system activity and allowing applications to benefit from faster unmapping and freeing of memory.

## 7 DISCUSSION AND LIMITATIONS

**ECOTLB supported operations.** For *free* operations, such as `munmap()` and `madvise()`[3], lazy memory reclamation enables an eventual TLB shootdown. Similarly, an eventual TLB shootdown is applicable to *migration* operations, such as page swapping using INFINISWAP, where page table entries can be unmapped lazily, enabling the eventual TLB shootdown. However, ECOTLB's lazy approach is not applicable to operations such as permission changes, ownership changes, and remap (`mremap()`), where page table changes should be synchronously applied to the entire system. ECOTLB supports common operations, such as free and page swap, and improves real-world applications such as Apache, Memcached, and Make.

***Free* operation semantics.** ECOTLB 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). Nevertheless, these semantics are mainly used for debugging purposes [21]. To allow such applications to function correctly, ECOTLB could be selectively enabled by including a new flag in the API of free operations (e.g., `munmap()`) for existing OSes.

**Huge page support.** ECOTLB currently does not support transparent huge pages (THPs). However, ECOTLB's states could be extended with an additional flag to support an eventual TLB shootdown for THPs as well. In addition, INFINISWAP already supports THPs as page swapping splits huge pages into 4 KB pages, thus, as a result, ECOTLB supports swapping THPs as well.

## 8 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, 10, 12, 42, 43, 48, 49, 51, 60, 62]. Several

[3] for the case of `MADV_DONTNEED` and `MADV_FREE`.

of these hardware-based approaches attempt to squeeze performance using non-traditional TLB designs, such as multi-level TLB hierarchies. The alternate hardware designs may be possible using multi-level TLB hierarchies, similar to the data caches, where it may be beneficial to back latency-critical higher level L1/L2 TLBs with slower but considerably larger in-DRAM TLBs.

Hardware-based research approaches provide cache coherence to the TLB. UNITD [51], 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 page translation is present in a specific TLB, thereby greatly increasing the TLB's power consumption. HATRIC [62] is a hardware mechanism similar to UNITD and piggybacks translation coherence information using the existing cache coherence protocols.

DiDi [60] 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 [42]. 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.

Hardware-based approaches, however, are 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, 19, 22, 37, 50, 53, 59].

**Software approaches.** Similarly, there are a number of approaches in the operating system [1, 8, 11, 15, 17, 44, 54, 55, 57, 63] to optimize TLB shootdowns. Barrelfish [11], 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 [1], 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. In addition, research approaches propose further optimizations for the synchronous TLB shootdown mechanism in Linux kernel [2]. However, none of them eliminate the synchronous TLB shootdown overhead. The Corey OS [16] avoids TLB shootdowns of private PTEs by requiring the user applications to explicitly define shared and private pages. Finally, Latr [31] proposes a lazy mechanism to eliminate the overheads due to a synchronous TLB shootdown. However, Latr's design is not easily applicable to the current setting of tagged TLBs with PCIDs. ecoTLB bridges this shortcoming while also providing a solution for the emerging setting of next-generation data centers with swapping in disaggregated memory using RDMA.

**Other TLB-related optimizations.** SLL TLBs introduced a shared last-level TLB [14, 36], and showed the benefits of using such a TLB. However, their design still relies on IPI-based coherency transactions which is an expensive operation (§2). 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, 56, 61]. However, this implies that a TLB shootdown must also invalidate mappings in the TLB prefetch buffers which incurs additional overhead to the expensive TLB shootdown operation. In addition, other approaches are aimed at mainly improving TLB misses which is orthogonal to the TLB shootdown problem [13, 20, 29, 45].

## 9 CONCLUSION

We present ecoTLB, a software-based eventual TLB coherence scheme for improving the performance of emerging, disaggregated data centers while being readily implementable in modern OSes that use ASIDs, for *free* operations. We demonstrate that ecoTLB's eventual TLB coherence scheme plays an important role in emerging, disaggregated data centers: ecoTLB demonstrates the overhead of the existing TLB coherence scheme with fast I/O devices using RDMA, and provides an eventual coherence scheme for *page swapping* which can play a critical role in next-generation data centers that use disaggregated memory. Additionally, ecoTLB's TLB coherence scheme plays a significant role to improve the performance of security features, such as kernel page table isolation, in the Linux kernel. Finally, we show that ecoTLB reduces the cost of `munmap()` by up to 59.1% on multi-socket machines while improving the throughput and 99th percentile tail latency of Memcached with Infiniswap by up to 13.5% and 70.8%, respectively, when they are run in a prototype for future, disaggregated data centers.

## 10 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF awards 1253700, 1916817, 1337147, CNS-1749711, and ETRI IITP/KEIT[2014-3-00035].

## REFERENCES

[1] N. 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.

[2] N. Amit, A. Tai, and M. Wei. Don't shoot down TLB shootdowns! In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, pages 1–14, Heraklion, Greece, Apr. 2020.

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

[4] R. Arimilli, G. Guthrie, and K. Livingston. Multiprocessor system supporting multiple outstanding TLBI operations per partition, Oct. 2004. URL https://www.google.com/patents/US20040215898. US Patent App. 10/425,425.

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

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 53–64, London, UK, June 2012.

[7] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. 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, Sept. 2017.

[8] R. Balan and K. 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] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, Mar. 2017.

[10] T. Baruah, Y. Sun, A. T. Dinçer, S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *Proceedings of the 26th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 596–609, San Diego, CA, Feb. 2020.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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, Oct. 2009.

[12] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee. Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 271–284, Fukuoka, Japan, Oct. 2018.

[13] A. 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, Apr.

2017.

[14] A. Bhattacharjee, D. Lustig, and M. 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, Feb. 2011.

[15] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. 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, Apr. 1989.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. 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, Dec. 2008.

[17] A. T. Clements, M. F. Kaashoek, and N. 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, Apr. 2013.

[18] J. Corbet. The current state of kernel page-table isolation, 2017. https://lwn.net/Articles/741878/.

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

[20] G. Cox and A. 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, Apr. 2017.

[21] T. H. Dang, P. Maniatis, and D. Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 815–832, Vancouver, BC, Aug. 2017.

[22] L. K. D. Database. CONFIG_ARM_ERRATA_720789, 2017. http://cateee.net/lkddb/web-lkddb/ARM_ERRATA_720789.html.

[23] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. 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, Nov. 2016.

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

[25] Google. CPU Platforms, 2018. https://cloud.google.com/compute/docs/cpu-platforms.

[26] Intel. Multiprocessor Specification, 1997.

[27] Intel. 5-Level Paging and 5-Level EPT, 2017. https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[28] G. Juncheng, L. Youngmoon, Z. Yiwen, C. Mosharaf, and S. Kang. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2017.

[29] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Energy-Efficient Address Translation. In *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, Barcelona, Spain, Mar. 2016.

[30] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash Storage Disaggregation. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 29:1–29:15, London, UK, Apr. 2016.

[31] M. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna. LATR: Lazy Translation Coherence. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 651–664, Williamsburg, VA, Mar. 2018.

[32] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International World Wide Web Conference (WWW)*, pages 591–600, Raleigh, NC, Apr. 2010.

[33] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. 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, Nov. 2016.

[34] J. Leverich. Mutilate: High-performance memcached load generator, 2017. https://github.com/leverich/mutilate.

[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[36] D. Lustig, A. Bhattacharjee, and M. 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, Apr. 2013.

[37] D. Lustig, G. Sethi, M. Martonosi, and A. 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, Apr. 2016.

[38] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 527–543, Belgrade, SR, Apr. 2017.

[39] Mellanox. ConnectX-3 Single/Dual-Port Adapter with VPI, 2017. http://www.mellanox.com/page/products_dyn?product_family=119&mtag=connectx_3_vpi.

[40] Memcached. A high-performance, distributed memory object caching system, 2017. http://memcached.org/.

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

[42] M. Oskin and G. 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, Sept. 2015.

[43] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 913–925, Valencia, Spain, May 2020.

[44] J. K. Peacock, S. Saxena, D. Thomas, F. Yang, and W. 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.

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

[46] B. Pham, A. Bhattacharjee, Y. Eckert, and G. 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, Feb. 2014.

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

[48] B. Pichai, L. Hsu, and A. 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, Mar. 2014.

[49] J. Power, M. D. Hill, and D. 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, Feb. 2014.

[50] B. F. Romanescu, A. R. Lebeck, and D. 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, Mar. 2010.

[51] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. 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, Jan. 2010.

[52] ScyllaDB. Memcached Benchmark, 2015. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark.

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

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

[55] P. J. Teller, R. Kenner, and M. 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.

[56] S. R. 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.

[57] M. Y. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of the Winter 1988 USENIX Annual Technical Conference (ATC)*, Dallas, TX, 1988.

[58] L. Torvalds. Linux Kernel, 2017. https://github.com/torvalds/linux.

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

[60] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. 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, Oct. 2011.

[61] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, and S. Jiang. Supporting Superpages and Lightweight Page Migration in Hybrid Memory Systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2): 11:1–11:26, Apr. 2019.

[62] Z. Yan, J. Veselý, G. Cox, and A. 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.

[63] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 331–345, Providence, RI, Apr. 2019.