Copyright
by
Yashwant Marathe
2018

The Thesis Committee for Yashwant Marathe
Certifies that this is the approved version of the following thesis:

Improving Virtual Memory Performance in Virtualized Environments

APPROVED BY

SUPERVISING COMMITTEE:

Lizy K. John, Supervisor

Nagendra Gulur Dwarakanath

Improving Virtual Memory Performance in Virtualized Environments

by

Yashwant Marathe

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2018

Abstract

Improving Virtual Memory Performance in Virtualized Environments

Yashwant Marathe, M.S.E.
The University of Texas at Austin, 2018

Supervisor: Lizy K. John

Virtual Memory is a major system performance bottleneck in virtualized environments. In addition to expensive address translations, frequent virtual machine context switches are common in virtualized environments, resulting in increased TLB miss rates, subsequent expensive page walks and data cache contention due to incoming page table entries evicting useful data. Orthogonally, translation coherence, which is currently an expensive operation implemented in software, can consume up to 50% of the runtime of an application executing on the guest. To improve the performance of virtual memory in virtualized environments, two solutions have been proposed in this thesis - namely, (1) Context Switch Aware Large TLB (CSALT), an architecture which addresses the problem of increased TLB miss rates and their adverse impact on data caches. CSALT copes with the increased demand of context switches by storing a large number TLB entries. It mitigates data cache contention by

employing a novel *TLB-aware cache partitioning scheme*. On 8-core systems that switch between two virtual machine contexts executing multi-threaded workloads, CSALT achieves an average performance improvement of 85% over a baseline with conventional L1-L2 TLBs and 25% over a baseline which has a large L3 TLB (2) Translation Coherence using Addressable TLBs (TCAT), a hardware translation coherence scheme which eliminates almost all of the overheads associated with address translation coherence. TCAT overlays translation coherence atop cache coherence to accurately identify slave cores. It then leverages the *addressable Part-Of-Memory TLB (POM-TLB)* to eliminate expensive Inter Processor Interrupts (IPI) and achieve precise invalidations on the slave core. On 8-core systems with one virtual machine context executing multi-threaded workloads, TCAT achieves an average performance improvement of 13% over the `kvmtlb` baseline.

Table of Contents

Abstract **iv**

List of Figures **ix**

Chapter 1. Introduction **1**

1.1 Address Translation and Context Switching 1

1.2 Translation Coherence 5

Chapter 2. Virtual Memory in Virtualized Systems **10**

2.1 Hardware Virtualization 10

2.2 Address Translation 11

2.2.1 Address Translation in Native Systems 12

2.2.2 Address Translation in Virtualized Systems 12

2.3 Translation Coherence 14

2.3.1 Translation Coherence in Native Systems 15

2.3.2 Translation Coherence in Virtualized Systems 16

Chapter 3. Motivation: Virtual Memory Overheads in Virtualized Environments **21**

3.1 Address Translation Overheads 21

3.2 Virtual Memory Associated Overheads induced by Context Switching 22

3.3 Translation Coherence Overheads 26

Chapter 4. Related Work **33**

4.1 Address Translation in Virtualized Environments 33

4.1.1 Translation Caching Structures 33

4.1.2 Speculation Schemes 34

4.1.3 Hiding Page Walk Latency and Increasing TLB reach . 34

4.1.4 Translation-aware Cache Replacement and Cache Partitioning . . . 35
4.2 Translation Coherence . . . 35
4.2.1 Software Approaches . . . 36
4.2.2 UNITD: UNified Instruction/Translation/Data Coherence 38
4.2.3 DiDi: A Shared TLB Directory . . . 39
4.2.4 HATRIC: Hardware Translation Invalidation and Coherence . . . 39

Chapter 5. Architecture Description & Evaluation 41
5.1 Context Switch Aware Large TLB . . . 41
5.1.1 CSALT with Dynamic Partitioning (CSALT-D) . . . 43
5.1.2 CSALT with Criticality Weighted Partitioning (CSALT-CD) . . . 49
5.1.3 Hardware Overhead . . . 55
5.1.4 Effect of Replacement Policy . . . 56
5.2 Translation Coherence using Addressable TLBs (TCAT) . . . 57
5.2.1 Overview of our scheme . . . 57
5.2.2 Overlaying translation coherence atop cache coherence . 58
5.2.3 Coherence lookups in the TLB . . . 59
5.2.4 Achieving Translation Coherence . . . 61
5.2.4.1 Initiator core operations . . . 61
5.2.4.2 Slave core operations . . . 63
5.2.5 Discussion . . . 66
5.2.5.1 Why is Cache Line Flush required? . . . 66
5.2.5.2 Assumptions . . . 68
5.2.5.3 Precision of Invalidation . . . 69
5.2.6 Putting it all together . . . 69

Chapter 6. Results 73
6.1 Experimental Set-Up . . . 73
6.2 Context-Switch Aware Large TLB . . . 75
6.2.1 Workloads . . . 75
6.2.2 Simulation . . . 75

6.2.3 CSALT Performance 77
6.2.3.1 CSALT Performance in Native Systems 82
6.2.4 Comparison to Prior Works 82
6.2.5 Sensitivity Studies 85
6.3 Translation Coherence using Addressable TLBs (TCAT) 88
6.3.1 Workloads 88
6.3.2 Simulation 89
6.3.3 TCAT performance 90
6.3.4 TCAT performance in Native Systems 92
Chapter 7. Conclusion **95**
Bibliography **97**

List of Figures

2.1 Page table walk in Native Systems (1D-page walk) 13
2.2 Page Table Walks in Virtualized Systems 14
2.3 TLB shootdowns in Native Systems 17
2.4 TLB shootdowns in Virtualized Systems 20

3.1 Increase in TLB Misses due to Context Switches. Ratio of L2 TLB MPKIs in Context Switch Case to Non-Context Switch Case 23
3.2 Fraction of Cache Capacity Occupied by TLB Entries 25
3.3 Percentage Execution overhead of TLB shootdown (vCPU overcommit) 31
3.4 Percentage Execution overhead of TLB shootdown (VM overcommit) 31
3.5 Average TLB Shootdown Latency 32

5.1 CSALT System Architecture 42
5.2 LRU Stack Example 49
5.3 CSALT Overall Flowchart 50
5.4 Coherence Scheme in action 72

6.1 Performance Improvement of CSALT (normalized to POM-TLB) 77
6.2 POM-TLB: Fraction of Page Walks Eliminated 78
6.3 Fraction of TLB Allocation in Data Caches 79
6.4 Relative L2 Data Cache MPKI over POM-TLB 80
6.5 Relative L3 Data Cache MPKI over POM-TLB 80
6.6 Performance Improvement of CSALT-CD in the native context 83
6.7 Performance Comparison of CSALT with Other Comparable Schemes 85
6.8 Performance of CSALT with Different Number of Contexts . . 86
6.9 Performance of CSALT with Different Epoch Lengths 87

6.10 Performance of CSALT with Different Context Switch Intervals 87
6.11 Performance improvement of TCAT 92
6.12 Percentage stall cycles eliminated 93
6.13 Performance improvement in Native systems 94

Chapter 1

Introduction

1.1 Address Translation and Context Switching

Computing in virtualized cloud environments [1–5] has become a common practice for many businesses in order to reduce capital expenditures. Many hosting companies have found that the utilization of their servers is low (see [6] for example). In order to keep the machine utilization high, the hosting companies that maintain the host hardware typically attempt to keep just *enough* machines to serve the computing load, and allowing multiple virtual machines to coexist on same physical hardware [7–9]. High CPU utilization has been observed in many virtualized workloads [10–12].

The aforementioned trend means that the host machines are constantly occupied by applications from different businesses, and frequently, different contexts are executed on the same machine. Although it is ideal for achieving high utilization, the performance of guest applications suffer from frequent context switching. The memory subsystem has to maintain consistency across the different contexts, and hence traditionally, processors used to flush caches and TLBs. However, modern processors adopt a more efficient approach where each entry contains Address Space Identifier (ASID)[13]. Tagging the entry

with ASID eliminates the needs to flush the TLB upon a context switch, and when the swapped-out context returns, some of its previously cached entries will be present. Although these optimizations worked well with traditional benchmarks where the working set, or memory footprint, was manageable between context switches, this trend no longer holds for emerging workloads. The memory footprint of emerging workloads is orders of magnitude larger than traditional workloads, and hence the capacity requirement of TLBs as well as data caches is much larger. This means the cache and TLB contents of previous context will frequently be evicted from the capacity constrained caches and TLBs since the applications need a larger amount of memory. Although there is some prior work that optimizes context switches [14–16], there is very little literature that is designed to handle the context switch scenarios caused by huge footprints of emerging workloads that flood data caches and TLBs.

Orthogonally, the performance overhead of address translation in virtualized systems is considerable as many TLB misses incur a full 2-dimensional page walk. The page walk in virtualized system begins with guest virtual address (gVA) when an application makes a memory request. However, since the guest and host system keep their own page tables, the gVA has to be translated to host physical address (hPA). First, gVA has to be translated to guest physical address (gPA), which is the host virtual address (hVA). This hVA is finally translated to gPA. This involves walking down a 2-dimensional page table. Current x86-64 employs a 4-level page table[17], so the 2-dimensional

page walk may require up to 24 accesses. Making the situation worse, emerging architectures [18] introduce a 5-level page table resulting in the page walk operation to only get longer. Also, even though the L1-L2 TLBs are constantly getting bigger, they are not large enough to handle the huge footprint of emerging applications, and expensive page walks are becoming frequent.

Context switches in virtualized workloads are expensive. Since both the guest and host processes share the hardware TLBs, context switches across virtual machines can impact performance severely by evicting a large fraction of the TLB entries held by processes executing on any one virtual machine.

Conventional page walkers as well as addressable large-capacity translation caches (such as Oracle SPARC TSB [19]) generate accesses that get cached in the data caches. In fact, these translation schemes rely on successful caching of translation (or intermediate page walk) entries in order to reduce the cost of page walks. There has also been some recent work that attempts to improve the address translation problem by implementing a very large L3 TLB that is a part of the addressable memory[20]. The advantage of this scheme titled POM-TLB is that since the TLB is very large (several orders of magnitude larger than conventional on-chip TLBs), it has room to hold most required translations, and hence most page walks are eliminated. However, since the TLB request is serviced from the DRAM, the latency suffers. The POM-TLB entries are cached in fast data caches to reduce the latency problem, however, all of the aforementioned caching schemes suffer from the problem of cache contention due to the additional load on data caches caused

by the cached translation entries.

As L2 TLB miss rates go up, proportionately, the number of translation-related accesses also go up, resulting in congestion in the data caches. Since a large number of TLB entries are stored in data caches, now the data traffic hit rate is affected. When the cache congestion effects are added on top of cache thrashing due to context switching, which is common in modern virtualized systems, the amount of performance degradation is not negligible.

In this thesis, we present CSALT (read as "sea salt") which employs a novel dynamic cache partitioning scheme to reduce the contention in caches between data and TLB entries. CSALT employs a partitioning scheme based on monitoring of data and TLB stack distances and marginal utility principles. We architect CSALT over a large L3 TLB which can practically hold all the required TLB entries. However, CSALT can be easily architected atop any other translation scheme. CSALT addresses increased cache congestion when L3 TLB entries (or entries pertaining to translation in other translation schemes) are allowed to be cached into L2 and L3 data caches by means of a novel cache partitioning scheme that separates the TLB and data traffic. This mechanism helps to withstand the increased memory pressure from emerging large footprint workloads especially in the virtualized context switching scenarios.

1.2 Translation Coherence

Beyond the problem of expensive address translations and TLB misses due to context switching, translation coherence has become a major performance sink in virtualized environments. Often, OSes perform modifications to the virtual-to-physical address translations to increase performance. For example, the OS may want to swap a page out to the disk or relocate a frequently used page to a faster memory. In either case, the OS must invalidate a virtual-to-physical address translation of the physical page being swapped out or relocated. To maintain a consistent view of virtual memory across cores, the OS must inform all the processors in a Chip Multi Processor (CMP) about this invalidation. Modern systems utilize Inter Processor Interrupts (IPIs) for this communication. The core which initiates the address translation modification, called the *initiator core*, relays IPIs to cores which might potentially hold the modified address translation in their private TLBs, called the *slave cores*. The initiator core then enters a busy wait loop waiting for acknowledgement from all the cores to ensure remote invalidation before proceeding further. Upon receiving the IPIs, the slave cores jump to an interrupt handler and perform invalidations in their own private TLBs. This process is called the *TLB shootdown*. There are several overheads involved in this process. First, processing the IPIs involve expensive pipeline flushes and protection level switches. Second, for the initiator core, remote invalidations pose a foreground overhead because of the busy-wait loop. In native systems, we observe that the initiator core spends thousands of CPU cycles waiting for acknowledgement from slave

cores. In dedup benchmark from the PARSEC benchmark suite, we observe upto *8us* of delay on a state-of-the-art Intel Skylake system. Third, the list of slave cores is approximated. The OS conservatively relays IPIs to cores which may potentially contain the modified translation. The slave may not contain the modified mapping due to reasons like TLB evictions[21][22].

In virtualized systems, TLB shootdowns occur between virtual processors (vCPUs) instead of the physical cores. The overall process of a TLB shootdown in virtualized environments looks very similar to native TLB shootdowns, albeit a few key differences. First, vCPUs generate virtual IPIs instead of physical IPIs. These virtual IPIs are emulated by the underlying Virtual Machine Manager (VMM), which performs the task of communicating to the slave vCPU that a translation has been modified, either by relaying a physical IPI to the physical core running the target vCPU, or by setting a flag to enforce coherence when target vCPU resume execution. TLB shootdowns in virtualized environments suffer from the same set of problems as their native counterparts: expensive pipeline flushes and protection level switches on the physical core upon delivery of the physical IPI, approximation of the list of slave vCPUs, and busy wait loop on the initiator vCPU. Interestingly, busy wait on the initiator vCPU is exacerbated because the slave vCPU might potentially get pre-empted *while* processing the virtual interrupt, delaying the acknowledgement [23]. In addition, due to imprecision of invalidation, the slave core must perform a TLB flush instead of an invalidation of a single translation entry. Therefore, we observe an order of magnitude increase in the

busy wait to *60us* on the initiator in dedup on Intel Skylake machine with 8 cores running a single VM with 8 vCPUs. Virtualized systems also suffer from the overhead of emulation of virtual interrupts.

TLB shootdowns in virtualized environments are woefully imprecise, imposing additionally overheads on an already worse baseline. Current VMMs do not track the guest virtual address. Since modern processors only permit invalidations of individual TLB entries when the gVA is known, when the VMM updates the nested page table, translation structures are completely flushed [24]. Additionally, VMMs track address translations at VM-granularity - VMMs track the subset of cores that a vCPU runs on, but do not track the subset of cores that a process on a vCPU runs on. To make matters worse, even the initiator vCPU can only approximate the list of slave vCPUs which have potentially accessed the translation being modified. As a result, in addition to the list of slave vCPUs approximated by the guest OS, the list of slave cores is approximated by the VMM. Combined with the flushing of translation structures upon a nested page table update, these approximations result in translations pertaining to other processes that run on the target vCPU being needlessly invalidated[24].

There have been various techniques proposed in both hardware and software to eliminate these overheads. Software schemes like Lazy Translation Coherence [25], ABIS[26] and hardware schemes like DIDI[21], UNITD[22] and HATRIC [24] aim to reduce the number of IPIs or eliminate them altogether. Hardware schemes allow precise identification of the slave cores and eliminate

busy waits on the initiator. However, some of these schemes are applicable only to native environments. While HATRIC[24] does account for virtualized environments, it suffers from two major drawbacks. First, HATRIC tags the TLB entries with the physical address of the last-level PTE of the host page table. As a result, it can only track changes to the host page table, and cannot track changes to the guest page table. Also, since the TLB entries are tagged with the physical address of the last-level PTE, it cannot track changes to the intermediate host PTEs. Second, HATRIC does not solve the problem of imprecision in virtualized environments. Even with HATRIC, when the VMM updates the nested page table, translation structures are completely flushed.

In this thesis, we present TCAT (Translation Coherence using Addressable TLBs), a hardware translation coherence scheme to reduce the cost of address translation coherence. TCAT overlays translation coherence atop cache coherence. It then leverages the addressable Part of Memory TLB (POM-TLB) [27] to enable precise invalidations in virtualized environments. TCAT tracks changes to guest page table by tying each guest page table translation to a single address in POM-TLB. On a guest-page table updates, TCAT generates coherence messages for that POM-TLB address. When the other cores see these coherence messages, they know which entry in their TLB needs to be invalidated, because each guest page table translation is tied to a single address in the POM-TLB. This allows precise invalidations on the slave vCPUs/cores. This mechanism helps withstand the overheads of page-table modifications by the guest OS.

This thesis is structured as follows. Chapter 2 provides a brief background on address translation and translation coherence in both native and virtualized environments. In Chapter 3, we motivate the need for our solutions by demonstrating the overheads that exist in virtual memory in current virtualized systems. In Chapter 4, we explore the existing solutions and their drawbacks. In Chapters 5 and 6, we present our design and evaluate the results. Chapter 7 concludes the thesis.

Chapter 2

Virtual Memory in Virtualized Systems

2.1 Hardware Virtualization

Hardware virtualization is an abstraction of computing resources from the software that uses those resources. In a traditional computing environment, the operating system or an application has direct access to the underlying computing resources such as the processor, memory and storage. In order to facilitate server consolidation, hardware virtualization installs a hypervisor or virtual machine manager (VMM), which creates an abstraction layer between the software and the underlying computing resources. In the presence of a hypervisor, software uses the virtual representations of underlying computing resources such as virtual processors (vCPUs) or guest memory instead of using those resources directly. Virtualized computing resources are provisioned into isolated independent instances called virtual machines (VMs) where operating systems and applications can be installed. The operating system running on the VM along with its associated software is termed the guest, and the underlying hardware which provides the computing resources for the guest to run is termed the host. Virtualized systems can host multiple VMs simultaneously, but every VM is logically isolated from every other VM.

2.2 Address Translation

All modern computer systems provide an abstraction of main memory known as Virtual Memory. This abstraction ensures efficient use of main memory, provides an uniform address space to all processes, and achieves memory isolation between processes. In a system with virtual memory support, while each byte on the main memory is addressable by a unique address called the Physical Address (PA), a processor which intends to perform a memory access generates a Virtual Address (VA). Before accessing memory, VA is converted to PA in a process known as Address Translation. This process is performed at the granularity of a fixed-size block, rather than the granularity of a single virtual address. Virtual memory is partitioned into fixed-size blocks of $P = 2^p$ bytes called Virtual Pages. Main memory, addressed by PA, is partitioned into fixed-size blocks of the same size ($P = 2^p$ bytes) called Page Frames or Physical Pages. Address translation maps a virtual page to a physical page, and is achieved by means of a data structure stored in physical memory called the Page Table that stores mappings of virtual pages to physical pages. A page table is organized as an array of page table entries (PTEs). A PTE comprises of the VA-PA mapping, a valid bit indicating if the mapping is valid or not, and metadata associated with the mapping. Metadata fields vary widely across architectures. Each page in the virtual address space has a PTE at a fixed offset in the page table. Therefore, given a VA, the address translation hardware can index into the page table with an appropriate offset and obtain the PA from the PTE. To reduce the memory requirements of a page table,

multi-level page tables are used in modern systems. In case of today's x86-64, a four-level page table is adopted. To obtain the VA-PA mapping, the address translation hardware has to access all page table levels in a process called the page table walk. Since address translation needs to be performed on every memory access, it is critical to processor performance. Therefore, all processors employ a Translation Lookaside Buffer (TLB), which caches VA-PA mappings in an on-chip, content-addressable memory and eliminates the need for a full page table walk in the common case.

2.2.1 Address Translation in Native Systems

The procedure to perform the full translation in a native setting as is shown in Figure 2.1. The first few most significant bits (MSBs) of the VA are used along with the CR3 register to index into the first level of the page table, which is denoted as L4 in the figure. The entry at that index in L4 points to the base of L3. The next few MSBs of the VA are used to index into L3, and the entry at that index points to the base of L2. This process is continued until the last level L1 yields the PA corresponding to the VA. Therefore, in order to obtain a full translation, upto 4 memory accesses have to performed with a four-level page table.

2.2.2 Address Translation in Virtualized Systems

With hardware virtualization, to fully virtualize memory, two levels of address translation are used: (1) guest virtual address (gVA) to guest physical

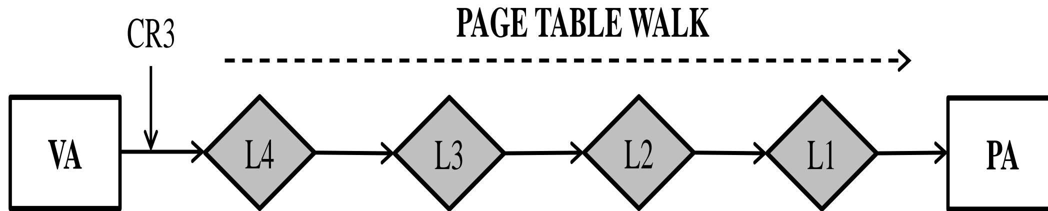


Figure 2.1: Page table walk in Native Systems (1D-page walk)

address translation (gPA) via a per-process guest OS page table (gPT) (2) guest physical address (gPA) to host physical address (hPA) via a per-VM host page table (hPT). Address translation in modern systems with hardware virtualization support is usually performed by a hardware technique called Nested Paging. The processor has two page table pointers to perform the two levels of address translation: one points to the guest page table (gCR3) and the other points to the host page table (hCR3). The guest page table holds gVA to gPA translation and the host page table holds gPA to hPA translations. In the best case, the gVA to hPA translation is available in the TLB. In the worst case, a TLB miss necessitates a 2D page walk. In a 2D page walk, a guest page table walk proceeds in the same way a native page table walk does. However, since guest memory is virtual, each guest page table access necessitates a host page table walk (which again proceeds in the same way as a native page table walk) to obtain the hPA. Once the hPA is obtained, the guest page table is accessed on the host physical memory, and the pointer to the next level of the guest page table is obtained. Page table memory references grow from a native 4 to a virtualized 24 references: 4 accesses to translate gCR3 (since each gPA requires access to host page table) and each of the 4 levels of the guest page

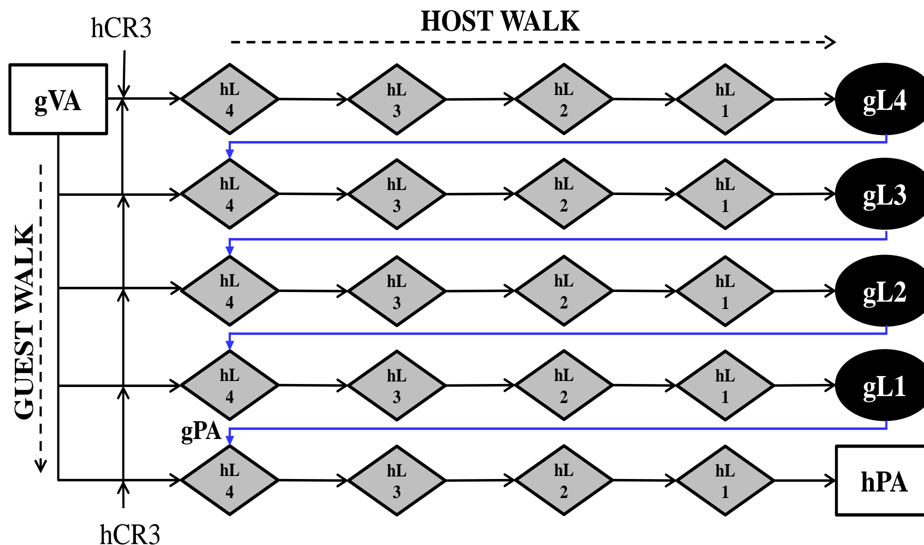


Figure 2.2: Page Table Walks in Virtualized Systems

table (guest page table holds gPA) plus 4 accesses for the guest page table itself; to obtain the final hPA: $4 \times 5 + 4 = 24$ references.

2.3 Translation Coherence

As TLBs are critical to processor performance, each core in a chip multiprocessor (CMP) has its own private TLB. Additionally, depending on the architecture, each core might contain private translation structures such as MMU caches and Nested TLBs (nTLB) to mitigate the overheads of full page walk in the common case. This replication mandates that all translation structures must be kept consistent with the OS page tables, so that the information cached in each one of these structures preserves a globally consistent view of virtual memory. There are a number of scenarios in which

the information associated with a translation may change: free operations like *munmap*() and *madvise*(); migration operations like page swap, deduplication and compaction; permission change operations like *mprotect*(); ownership change operations like copy-on-write optimization; and remap operations like *mremap*(). The aforementioned operations either update the mapping itself, or update the metadata bits associated with the mapping. During such events, consistency has to be maintained across private TLBs for correctness - no core should access a stale translation. Translation coherence is a way to enforce this consistency. In modern systems, TLBs are kept coherent at the software-level by the operating system (OS). Whenever the OS modifies a translation, it must initiate a coherency transaction among TLBs, a process known as a *TLB shootdown*. Current CMPs rely on the OS to approximate the set of TLBs caching a mapping and synchronize TLBs using costly Inter-Processor Interrupts (IPIs) and software handlers.

2.3.1 Translation Coherence in Native Systems

A TLB shootdown is an elaborate transaction in which the core initiating a modification the page-table ensures that all cores invalidate the affected mapping from their TLBs. The time line of a TLB shootdown in native environments is as illustrated below. Figure 2.3 depicts the sequence of events in a native system when a TLB shootdown is triggered to invalidate a mapping T_g[21][22].

1. One of the physical cores executing an operation that modifies the page-

table, referred to as the initiator core, prompts the OS to lock the corresponding page table entry and disables kernel pre-emption.

2. The OS forms a list of physical cores, called the slave cores or target cores, that requested a translation of the modified page-table entry in the past. The list of slave cores is conservatively approximated, since all cores in the list might not contain the modified translation due to TLB evictions.

3. The initiator sends an IPI to all the slave cores, requesting them to invalidate the TLB entries referring to the modified mapping. Meanwhile, the initiator core invalidates the mapping in its private TLB and waits for acknowledgments from all the slave cores.

4. All slave cores receive the IPI and execute the IPI handler for TLB invalidations. The interrupt handler code invalidates any affected TLB entries and sends an acknowledgement to the initiator core.

5. Once all acknowledgements are received by the initiator core, the OS unlocks the modified page-table entry, enables kernel pre-emption and continues its execution.

2.3.2 Translation Coherence in Virtualized Systems

With hardware virtualization, there is an abstraction layer sitting atop the underlying hardware - as a result, the guest processes execute on virtual

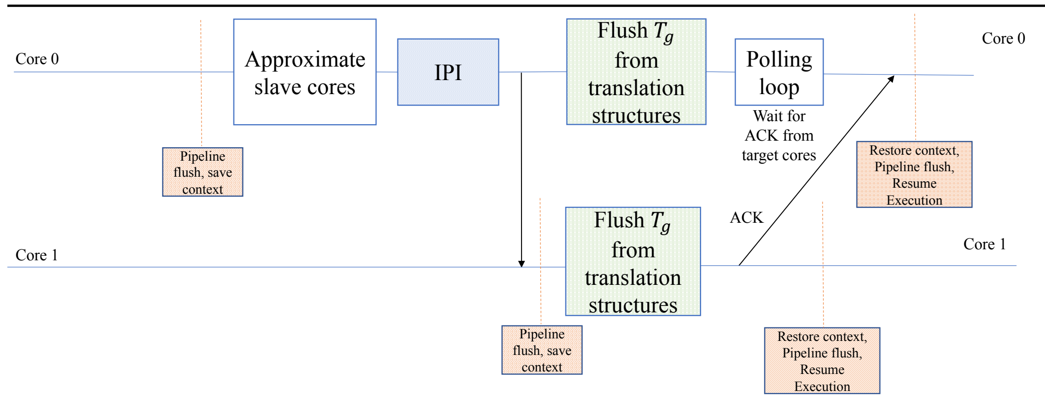


Figure 2.3: TLB shootdowns in Native Systems

processors (vCPUs) and not directly on the physical cores. TLB shootdown in virtualized environments must therefore ensure that all vCPUs invalidate the affected mapping from their virtual TLBs. The time line of a TLB shootdown in virtualized environments is as illustrated below. Figure 2.4 depicts the sequence of events in a virtualized system when a TLB shootdown is triggered to invalidate a mapping T_g[24][23].

1. One of the vCPUs executing a page-table modification operation, referred to as the initiator vCPU, prompts the guest OS to lock the corresponding page table entry.

2. The guest OS approximates a list of vCPUS, called the target vCPUs, that requested a translation of the modified page-table entry in the past.

3. The initiator vCPU sends an IPI with a specific vector number to a set of target vCPUs. The initiator then enters a polling based loop until all

target vCPUs have processed and acknowledged the requests by setting a flag located in shared memory.

4. The IPI transmission by the vCPU causes the hardware to trap into the VMM. The VM emulates the IPI and generates a new IPI that is transmitted to the VMM on the physical cores hosting the target vCPUs.

5. When the IPI is delivered to the physical core hosting the target vCPU, it causes a VM trap. The IPI is handed off the VMM running on the physical core. The VMM then injects a virtual interrupt to the target vCPU.

6. As soon as the target vCPU resumes execution, it executes the IPI handler for TLB invalidations. The interrupt handler code invalidates any affected TLB entries and sends an acknowledgement to the initiator vCPU by setting a flag located in shared memory.

7. Once all acknowledgements are received by the initiator vCPU, the guest OS unlocks the modified page-table entry and resumes normal execution.

In native environments, TLB shootdowns are comparatively faster. This is because the initiator physical core has to wait for target physical cores to acknowledge before the IPI is handled -and the target physical core is always available. However, in virtualized environments, since vCPUs are merely processes running on the host, a target vCPU could potentially be preempted by the host scheduler. In other words, target vCPUS are not always available to

send the acknowledgement to the initiator vCPU. Target vCPUs can perform the invalidation and send the acknowledgement only when they are scheduled again. This can result in the latencies of TLB flush operations increasing by orders of magnitude depending on the scheduling state of the target vCPUs. This problem is referred to as the *TLB shootdown preemption* problem[23]. In order to circumvent this problem, KVM utilizes a scheme called KVM paravirtual remote flush TLB scheme (kvmtlb) developed by the Linux community. This scheme records the preemption state of all vCPUs inside the VMM and shares this information with the VM. When there is a nested page table update, if the target vCPU is running, TLB shootdown proceeds like in the native case. However, if the target vCPU is preempted, a *should_flush* flag is set on that target vCPU and the IPI is not sent. When rescheduling a vCPU, the VMM checks the *should_flush* flag. If set, the VMM invalidates all TLB entries of that vCPU. This eliminates the problem of increased latencies on the initiator vCPUs waiting for acknowledgements from the remote vCPU[23].

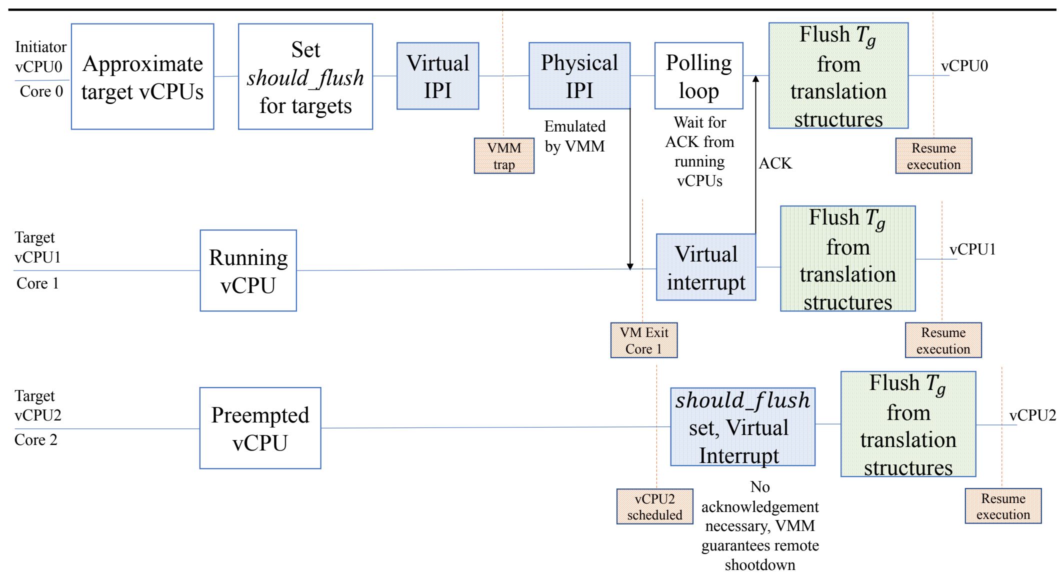


Figure 2.4: TLB shootdowns in Virtualized Systems

Chapter 3

Motivation: Virtual Memory Overheads in Virtualized Environments

3.1 Address Translation Overheads

In native systems, address translation overhead is considerable as a result of multiple memory references (proportional to the depth of the page table) required to obtain a full translation. Although techniques like caching of page table entries in data caches, MMU caches and caching intermediate translations can remove some of the memory references for a TLB miss, address translation in native systems still incurs a non-negligible performance overhead. In virtualized systems, the address translation overhead increases as a result of the increased number of memory references required to obtain a full translation due to a 2D page table walk. Table 3.1 lists the measured page walk cost per L2 TLB miss in both native and virtualized systems for some PARSEC and graph workloads. Measurements were made on a state-of-the-art Intel Skylake system with extended page tables. While some workloads (e.g., streamcluster) have very similar page walk costs in both native and virtualized, others (e.g., connectedcomponent, gups) show significant increase under virtualization as expected. With the increase in the number of page table levels, these overheads will only be exacerbated.

Benchmark	Native	Virtualized
canneal	53	61
connectedcomponent	44	1158
graph500	79	80
gups	43	70
pagerank	51	61
streamcluster	74	76

Table 3.1: Average Page Walk Cycles Per L2 TLB miss

3.2 Virtual Memory Associated Overheads induced by Context Switching

Today, it is common to have multiple VM instances to share a common host system as cloud vendors try to maximize hardware utilization. The aforementioned trend means that the host machines are constantly occupied by applications from different businesses, and frequently, different contexts are executed on the same machine. Although it is ideal for achieving high utilization, the performance of guest applications suffer from frequent context switching.

Context switches in virtualized workloads are expensive. Since both the guest and host processes share the hardware TLBs, context switches across virtual machines can impact performance severely by evicting a large fraction of the TLB entries held by processes executing on any one virtual machine. To quantify this, we measured the increase in the L2 TLB MPKI of a context-switched system (2 virtual machine contexts, switched every $10ms$) over a non-context-switched baseline. Figure 3.1 illustrates the increase in L2 TLB MPKIs for several multi-threaded workloads, when additional virtual machine context

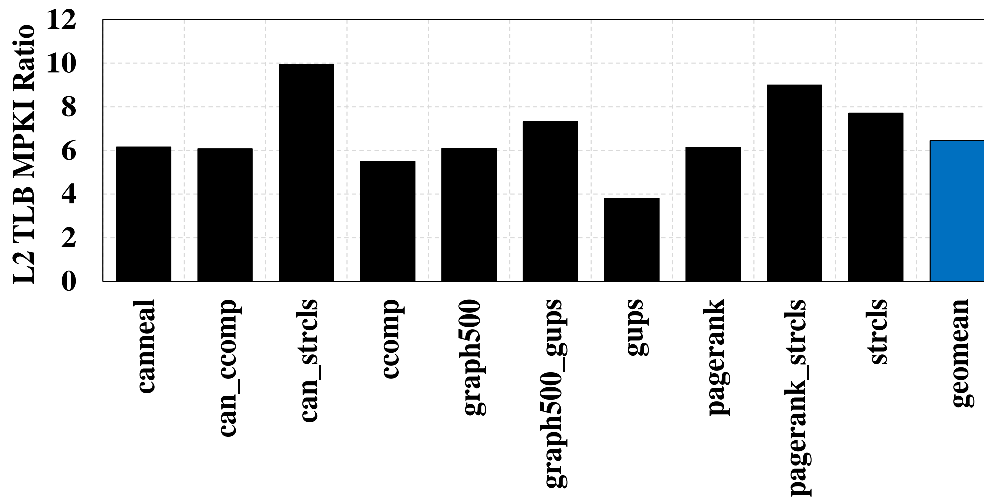


Figure 3.1: Increase in TLB Misses due to Context Switches. Ratio of L2 TLB MPKIs in Context Switch Case to Non-Context Switch Case

switches are considered. Despite only two VM contexts, the impact on the the L2 TLB is severe: an average increase in TLB MPKI of over $6X$. This leads to an overall degradation in performance of the context-switched workloads. For instance, when 1 VM instance of *pagerank* was context-switched with another VM instance of the same workload, the total program execution cycles for each instance went up by a factor of $2.2X$.

The higher miss rate of the L2 TLB leads to increased translation traffic to the data caches. In the conventional radix tree based page table organization, the additional page walks result in the caching of intermediate page tables. In the POM-TLB organization, the caches store *translation entries* instead of page table entries. By *translation entry* we refer to a TLB entry that stores the translation of a virtual address to its physical address. While

caching of TLB entries inherently causes less congestion (one entry per translation as opposed to multiple intermediate page table entries), it still results in polluting the data caches when the L2 TLB miss rates are high. This scenario creates an undesirable situation where neither data nor TLB traffic achieves the optimal hit rate in data caches. A conventional system is not designed to handle such scenarios as the conventional cache replacement policy does not distinguish different types of cache contents. This is no longer true as some contents are data contents while others are TLB contents. When a replacement decision is made, it does not distinguish TLB contents versus data contents. But the data and TLB contents impact system performance differently. For example, data requests are overlapped with other data requests with the help of MSHR. On the other hand, an address translation request is a blocking access, so it stalls the pipeline. Although newer processor architectures such as Skylake have simultaneous page table walkers to allow up to two page table walks, the page table walk being a blocking access does not change. In the end, the conventional content-oblivious cache replacement policy makes both the TLB and data access performance suffer by making them compete for entries in capacity constrained data caches. This problem is exacerbated when frequent context switches occur between virtual machines.

To quantify the cache congestion problem, we measure the *occupancy* of TLB entries in L2 and L3 data caches. We define *occupancy* as the average fraction of cache blocks that hold TLB entries. To collect this data, we modified our simulator to maintain a type field (TLB or data) with each cache

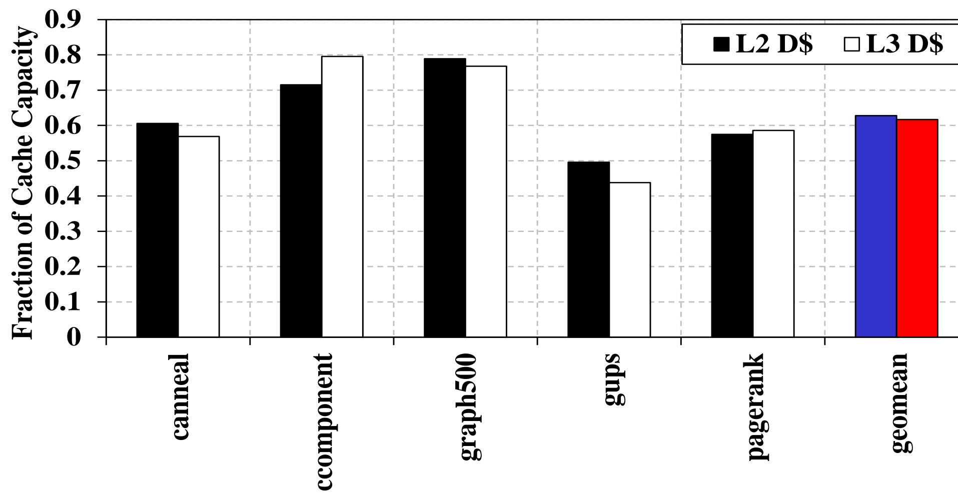


Figure 3.2: Fraction of Cache Capacity Occupied by TLB Entries

block; periodically the simulator scanned the caches to record the fraction of TLB entries held in them. Figure 3.2 plots this data for several workloads. We observe that an average of 60% of the cache capacity holds translation entries. In one workload (*connectedcomponent*), the TLB entry occupancy is as high as 80%. This is because the L2 TLB miss rate is approximately 10 times the L1 data cache miss rate, as a result of which translation entries end up dominating the cache capacity.

In summary, context switching increases L2 TLB MPKI, which in turn causes increased translation traffic and congestion in the data caches. While caching of the translation entries is useful to avoid DRAM accesses, the above data suggests that unregulated caching of translation entries has a flip side of causing cache pollution or creating capacity conflict with data entries. If the congestion favors TLB contents while data contents are of more importance,

performance will suffer. On the other hand, if the congestion favors data contents while TLB contents are of more importance, performance will still suffer. However, in the latter case, in virtualized environments, an expensive 2D page walk has to be performed to obtain a gVA to hPA translation, which further worsens the performance impact of such a scenario.

3.3 Translation Coherence Overheads

TLB shootdowns are employed in modern CMPs to maintain a globally consistent view of virtual memory. As discussed in Section 2.3.2, TLB shootdowns utilize costly IPIs and protection level switches to maintain coherence. As a result, they impose a considerable execution overhead. In native systems, these overheads can be broken up into three distinct components [21] in the following manner: (1) **IPI delivery overhead**: Overhead incurred by the initiator core in approximating the set of slave cores and delivering IPIs to each one of them (2) **TLB invalidation overhead**: Overhead of executing the handler which performs TLB invalidation on the initiator core (local shootdown), and TLB invalidation on the slave core (remote shootdown). This overhead is incurred by both the initiator core and the slave core (3) **Busy wait overhead**: Overhead of busy wait for acknowledgements from all the slave cores indicating that they have flushed the modified mapping from their TLBs. Only the initiator core sees this overhead.

TLB shootdowns in virtualized systems incur a few additional overheads. Since vCPUs are processes running on the host, IPIs delivered by the

initiator vCPUs must be transmitted to the physical cores running the target vCPUs. Additionally, the physical core running the target vCPU must guarantee a remote TLB invalidation. This is enabled by the emulation of virtual interrupts by the VMM, which involves the penalty of conversion of virtual IPI into physical IPI and back into virtual IPI, and the penalty of a VM-exit on the slave physical core. In addition to these undesirable overheads, there are several inefficiencies associated with TLB shootdown in virtualized systems, as summarized below.

1. **Imprecise invalidations**: In a virtualized system, when the VMM updates the nested page table, it only tracks the gPA and the hPA. It does not track the gVA. Modern processors do not permit precise invalidations if the gVA is not known. This results in a complete flush of all translation structures on the cores containing the modified mapping. Repopulating these flushed structures incur the penalty of a 2D-page table walk[24].

2. **Expensive remote TLB invalidations**: TLB invalidations on the slave core/vCPU involve expensive pipeline flushes and protection level switch to execute privileged instructions. This interferes with the workloads executing on the slave cores[21].

3. **Slave cores/vCPUs are approximated**: In a native system, the list of slave cores is approximated. Therefore, some cores which may not possess the modified mapping needlessly flush their pipeline and execute

the interrupt handler upon IPI delivery. In a virtualized system, VMMs track the subset of cores that a vCPU runs on, and do not track the subset of cores that a process on a vCPU runs on. In other words, mappings are tracked at the VM-granularity rather than a process-within-a-VM granularity. Therefore, when the VMM remaps a page pertaining to a process running on the initiator vCPU, it conservatively issues IPIs to all physical cores that may potentially have executed any target vCPU. As in the native case, if the target physical core (which has potentially run the target vCPU in the past) does not have the modified mapping, it incurs the overhead of pipeline flush and interrupt handler execution. Additionally, translation entries pertaining to other processes that run on the target vCPU get needlessly invalidated due to flushing of the translation structures[24].

To summarize these overheads in a succinct manner, in our measurements, we have broken up TLB shootdown overheads into two distinct components: (1) Initiator core (native) or Initiator vCPU (virtualized) overheads (2) Slave core (native) or Target vCPU (virtualized) overheads. Initiator overheads are comprised of the IPI delivery overhead, TLB invalidation overhead, and the busy wait overhead. Slave/Target overheads are solely constituted by remote TLB invalidation overheads. We measure these components independently on a real system. We compute the total execution overhead of TLB shootdowns as the summation of all the initiator overheads and all the slave overheads.

We perform the measurements a 8-core state-of-the-art Intel Skylake

system with and without hardware virtualization. To collect this data, we use the Linux *perf* utility in conjunction with the Linux *ftrace* utility. *perf* utility provides precise information about the local and remote shootdown events, while *ftrace* utility is used to measure latencies of these individual events.

While performing measurements, we demonstrate the behavior with two different modes of oversubscription: (1) A single VM with a vCPU: CPU ratio of 1:1, 2:1 3:1 and 4:1. In this mode, oversubscription is due to running multiple vCPUs. (2) Multiple VMs, each with a vCPU: CPU ratio of 1:1. In this mode, oversubscription is due to the presence of multiple VMs. In this mode, a single VM runs the benchmark of interest, while all the other VMs run a CPU-intensive benchmark from *sysbench*[28][23]. Both these modes are commonly used in datacenters: the first mode is used typically used to boost CPU utilization, while the second mode is used to run multiple services at once. Typically, the ratio of the number of virtual CPUs (vCPUs) in the virtualized system to the number of physical cores present on the host is called the overcommit ratio. Overcommit ratios ranging from 2:1 to 6:1 are commonplace for the maximizing CPU utilization[4].

Figure 3.3 plots the percentage of execution time spent in TLB shootdowns with the first mode of oversubscription for four workloads: *apache*, *dedup*, *ferret* and *vips*. For comparison, we plot the percentages for native environment, and virtualized environments with overcommit ratios of 1:1, 2:1, 3:1 and 4:1. We observe that the time spent doing TLB shootdowns typically increases with overcommit and can consume a considerable portion of

the application runtime across benchmarks. The proportion of time spent in the initiator and slave cores varies widely across benchmarks, although the overheads on the initiator usually larger. In *apache*, time spent in the initiator doing TLB shootdowns constitutes around 25% the execution for higher overcommit ratios. This can be attributed to wasteful busy waits on the initiator. In *vips*, the proportion of time spent on the slave increases with overcommit. This can be attributed to the imprecision of invalidation in virtualized environments.

Figure 3.4 plots the percentage of execution time spent in TLB shootdowns with the second mode of oversubscription. In *apache*, the percentage overhead increases from 25% in the previous mode to 50%. Although *vips* and *dedup* spend more time doing TLB shootdowns, the percentage overhead decreases due to the large increase in the execution time. However, in *ferret*, percentage overhead increases and reaches upto 15% in the 4:1 overcommit case.

Figure 3.5 plots the average TLB shootdown latency with different overcommit ratios with the first mode of oversubscription. We consistenty observe higher average TLB shootdown latencies in virtualized environments. In most of the benchmarks, there is a noticeable jump in the shootdown latency from the 1:1 overcommit case to the 2:1 overcommit case. In *apache*, the latency increases 20X from the 1:1 to 2:1 overcommit case. In *vips*, we observe a 10-fold increase in latency from the 1:1 to 2:1 overcommit case. From 10us in the native case, average TLB shootdown latencies exhibit an order of magni-

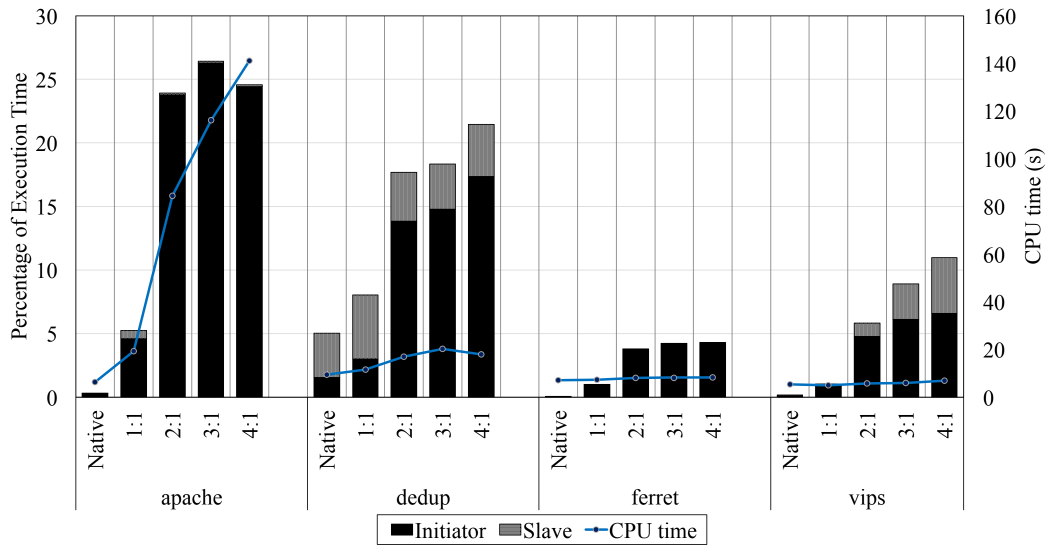


Figure 3.3: Percentage Execution overhead of TLB shootdown (vCPU overcommit)

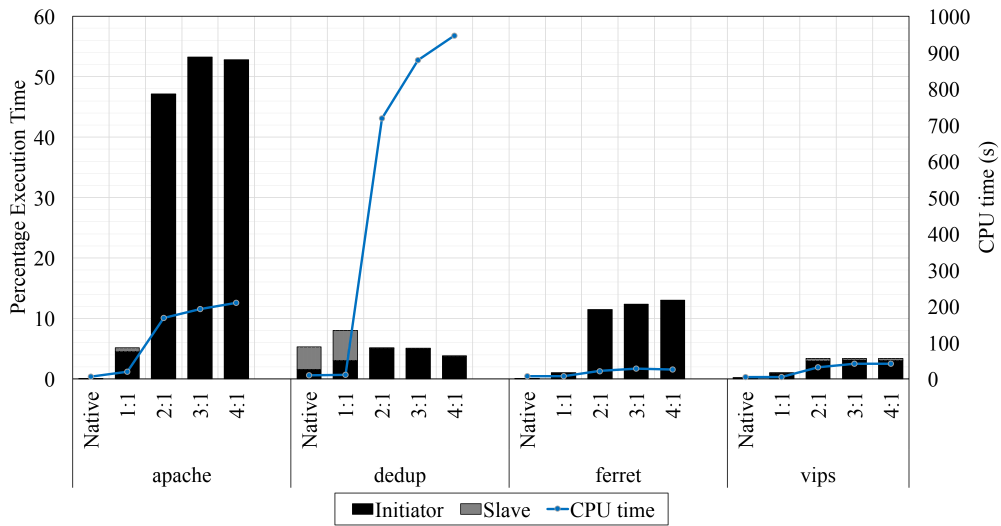


Figure 3.4: Percentage Execution overhead of TLB shootdown (VM overcommit)

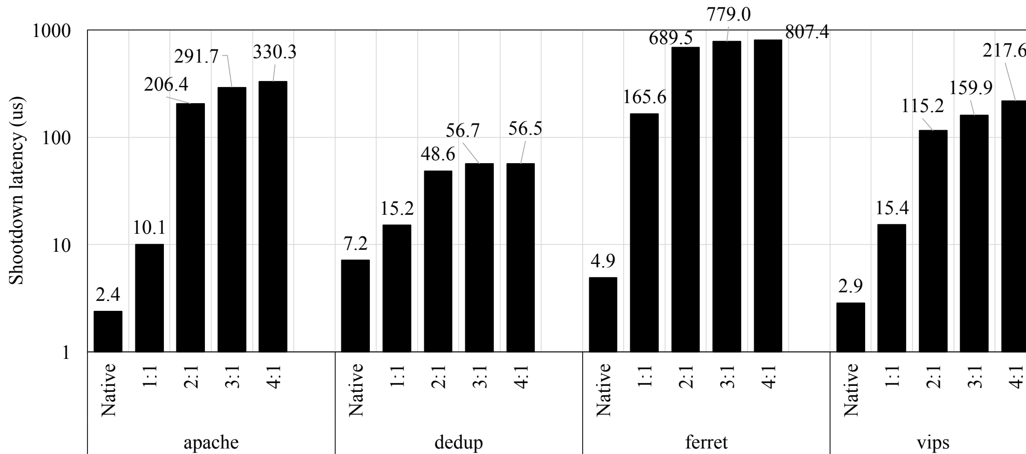


Figure 3.5: Average TLB Shootdown Latency

tude increase to 100-1000us in virtualized environments with oversubscription. These behaviors can be attributed to the additional overheads associated with virtualization. We see that reducing the TLB shootdown costs would result in a considerable performance boost. This motivates us to design a translation coherence scheme which amortizes the overheads associated with TLB shootdowns in virtualized environments - like the busy wait on the initiator and imprecise invalidations on the slave.

Chapter 4

Related Work

4.1 Address Translation in Virtualized Environments

4.1.1 Translation Caching Structures

Oracle UltraSPARC mitigates expensive software page walks by using TSB [19]. Upon TLB misses, the trap handling code quickly loads the TLB from TSB where the entry can reside anywhere from the L2 cache to off-chip DRAM. However, TSB requires multiple memory accesses to load the TLB entry in virtualized environments as opposed to a single access in our scheme (refer to Figure 15 in [29] for an overview of the TSB address translation steps in virtualized environments). Further, our TLB-aware cache partitioning scheme is applicable to the TSB as well, and as demonstrated in Section 6.2.2, TSB architecture also sees performance improvement.

Modern processors implement MMU caches such as Intel's PSC [17] and AMD's PWC [30] that store partial translation to eliminate page walks. However, the capacity is still much smaller than application footprints that a large number of page walks are still inevitable. Other proposals like cooperative caching [31], shared last level TLBs [32], and cooperative TLBs [33] exploit predictable memory access patterns across cores. These techniques are orthogonal to our approach and can be applied on top of our scheme since

we use a shared TLB implemented in DRAM. Although software-managed TLBs have been proposed for virtualized contexts [34], we limit our work on hardware managed TLBs.

4.1.2 Speculation Schemes

Speculation schemes [35,36] continue the processor execution with speculated page table entries and invalidate speculated instructions upon detecting the mispeculation. These schemes can effectively hide the overheads of page table walks. On the other hand, our scheme addresses the problem of TLB capacity. We aim to reduce the number of page walks significantly by having much larger capacity.

4.1.3 Hiding Page Walk Latency and Increasing TLB reach

Huge pages (e.g., 2MB or 1GB in x86-64) can reduce TLB misses by having a much larger TLB reach [37–39]. Our approach is orthogonal to huge pages since our TLB supports caching TLB entries for multiple page sizes. Various prefetching mechanisms [33, 40] have been explored to fetch multiple TLB or PTE entries to hide page walk miss latency. However, the fundamental problem that the TLB capacity is insufficient is not addressed in prior work. Hybrid TLB coalescing[41] aims to increase TLB coverage by encoding memory contiguity information and does not deal with managing cache capacity. Page Table Walk Aware Cache Management [42] uses a cache replacement policy to preferentially store page table entries on caches and does not use cache

partitioning.

4.1.4 Translation-aware Cache Replacement and Cache Partitioning

Recent cache replacement policy work such as DIP [43], DRRIP [44], SHiP [45] focuses on homogeneous data types, which means they are not designed to achieve the optimal performance when different data types of data (e.g., POM-TLB and data entries) coexist. Hawkeye cache replacement policy [46], also targets homogeneous data types, has a considerable hardware budget for LLC, and cannot be implemented for L2 data caches. EVA cache replacement policy[47] cannot be used in this case due to a similar problem.

Cache partitioning is an extensively researched area. Several previous works ([48–62]) have proposed mechanisms and algorithms for partitioning shared caches with diverse goals of latency improvement, bandwidth reduction, energy saving, ensuring fairness and so on. However, none of these works take into account the adverse impact of higher TLB miss rates due to virtualization and context switches. As a result, they fail to take advantage of this knowledge to effectively address the TLB induced cache congestion.

4.2 Translation Coherence

We have shown that translation coherence, which is currently an expensive operation implemented in software on most systems, can consume 10 to 30 percent of the application runtime. This observation is corraborated by

various studies which show similar results[63]. These overheads mainly arise due to various inefficiencies of TLB shootdowns. In native environments, inefficiencies arise due to long latencies between with IPI delivery and reception of acknowledgement from the target cores, and inaccurate approximations of the slave cores causing unnecessary pipeline flushes. In virtualized environments, the latency problem is worsened due to TLB shootdown pre-emption. Additionally, precise invalidation of TLB entries is not possible in virtualized environments because hypervisors only track the guest physical and host physical page numbers. In order to eliminate these inefficiencies, various techniques have been proposed in both software and hardware.

4.2.1 Software Approaches

Oskin et al [64] propose a scheme for hardware-assisted TLB shootdown. Hardware-assisted TLB shootdown approach introduces a special form of IPI, called the *REMOTE_INVLPG*, and an associated microcode change that receives this special IPI and issues a TLB shoot-down process entirely in microcode without necessitating any OS interaction. This approach eliminates OS interaction on the slave cores. However, this approach still necessitates the initiator core to deliver IPIs and wait for acknowledgements from all the slave cores, therefore, it incurs the entire initiator overhead component elaborated in the previous section.

Amit et al [26] introduce Access Based Invalidation System (ABIS). They assert that that access bits of the PTE can be used to determine if the

PTE is cached in any of the several private TLBs on the CMP. They use this information to avoid sending IPIs upon updates to mappings that are private, thereby preventing some of the penalties associated with both the initiator and the slave. However, ABIS requires extensive hardware support for direct TLB insertion and complex software infrastructure like a secondary page-hierarchy.

Mohan et al [25] propose a software-based TLB shootdown mechanism called Lazy-Translation Coherence that can alleviate the overhead of the TLB shootdown mechanism by handling TLB coherence in a lazy manner for page-table update operations that do not enforce synchronous updates. Lazy Translation Coherence avoids expensive IPIs which are required for delivering a shootdown signal to remote cores and the performance overhead of associated interrupt handlers. However, Lazy-translation coherence is only applicable for operations that can update the TLB asynchronously and cannot be applied for operations that enforce synchronous translation updates like *mprotect* or *mremap*.

Ouyang et al [23] propose a paravirtual TLB shootdown scheme named Shoot4U, which eliminates TLB shootdown preemptions in virtualized environments. It does so by intercepting remote vCPU TLB flush operations and performing the invalidations directly in the VMM instead of handling them in the guest environment. This optimization allows Shoot4U to avoid any delays caused by a vCPU which has been pre-empted and to ensure that the delays are consistent. Although Shoot4U decouples TLB shootdown completion from the host scheduler behaviors, it does not eliminate the busy wait loop on the

physical core on which the initiator vCPU is running. Additionally, it does not eliminate expensive IPIs between the physical cores.

4.2.2 UNITD: UNified Instruction/Translation/Data Coherence

Romanescu et al [22] propose a scalable hardware-based TLB coherence protocol called UNITD. The idea behind UNITD is to identify whether or not a page translation is present in a specific TLB by augmenting each TLB entry with the physical address of the last-level PTE of the translation that it holds. This is done by using structure called the Page Table Entry CAM (PCAM). Whenever the OS changes a PTE, the change propagates to the last-level PTE, and the cache-coherence protocols detect and relay this information to all the translation structures. Lookups are performed in the PCAM using the physical address of this last-level PTE, and TLB entries which are tagged with the same physical address are invalidated. There are two major drawbacks to this scheme: (1) It cannot track changes to intermediate PTEs. Although the authors argue that all the changes to the intermediate PTE need to be propagated to the last-level PTE, it might not be true in the case where a mapping changes (2) It increases coherence traffic to the TLBs to a large extent. Since the data caches do not distinguish data from page-table entries, cache coherence messages are relayed to the TLBs upon each update to the data cache contents.

4.2.3 DiDi: A Shared TLB Directory

Villavieja et al [21] propose a scalable architectural mechanism that couples shared TLB directory with load/store queue support for lightweight TLB invalidation and eliminates the need for costly IPIs. The scheme has two components - a second-level TLB that acts as a Dictionary Directory (DiDi), and a Pending TLB Invalidation (PTLBI) buffer. DiDi eliminates unnecessary IPIs by tracking the location of every address translation stored on the first-level TLBs of the whole system using the Directory Bitmap. On the other hand, the PTLBI buffer eliminates the overheads of interrupt processing performing the following operations: storing all broadcasted invalidations, injecting a memory barrier into the load/store queue, actually performing the invalidation, and sending acknowledgement back to the DiDi. DiDi lacks support for hardware virtualization. Although it eliminates expensive IPI delivery, it does not eliminate the busy wait on the initiator core waiting for acknowledgements from the slave cores.

4.2.4 HATRIC: Hardware Translation Invalidation and Coherence

Yan Zi et al [24] propose a hardware mechanism called HATRIC to piggyback translation coherence atop existing cache coherence protocols. In a manner similar to UNITD, by adding co-tags (system physical address of the nested page table entry) to translation structures, HATRIC obviates the need for full translation structure flushes by more precisely identifying invalidation targets. HATRIC then exposes these co-tags to the cache coherence protocol to

precisely identify coherence targets and to eliminate VM exits. HATRIC works well in virtualized environments with heterogeneous memories with potentially a large number of host-page remappings. In modern virtualized systems with no heterogeneous memories, we have observed that the host-page remappings are quite infrequent. Instead, we observe a lot of guest-page table remappings and associated overheads as detailed in the background section. However, HATRIC does not support hardware coherence for guest-page table remappings. Moreover, since it tags the TLB entries with the physical address of last-level PTE, it suffers from the same drawback as UNITD - it cannot track changes to intermediate PTEs.

Chapter 5

Architecture Description & Evaluation

5.1 Context Switch Aware Large TLB

The address translation overhead in virtualized systems comes from one apparent reason, the lack of TLB capacity. If the TLB capacity were large enough, most of page table walks would have been eliminated. The need for a larger TLB capacity is also seen as a recent generation of Intel processors [65] doubled the L2 TLB capacity from the previous generation. Traditionally, TLBs are designed to be small and fast, so that the address translation can be serviced quickly. Yet, emerging applications require much more memory than traditional server workloads. Some of these applications have terabytes of memory footprint, so that TLBs, which were not initially designed for such huge memory footprint, suffer significantly.

Recent work [20] by Ryoo et al. uses a part of main memory to be used as a large capacity TLB. They use 16MB of the main memory, which is negligible considering high-end servers have terabytes of main memory these days. However, 16MB is orders of magnitude higher than today's on-chip TLBs, and thus, it can eliminate virtually all page table walks. This design achieves the goal of eliminating page table walks, but now this TLB suffers

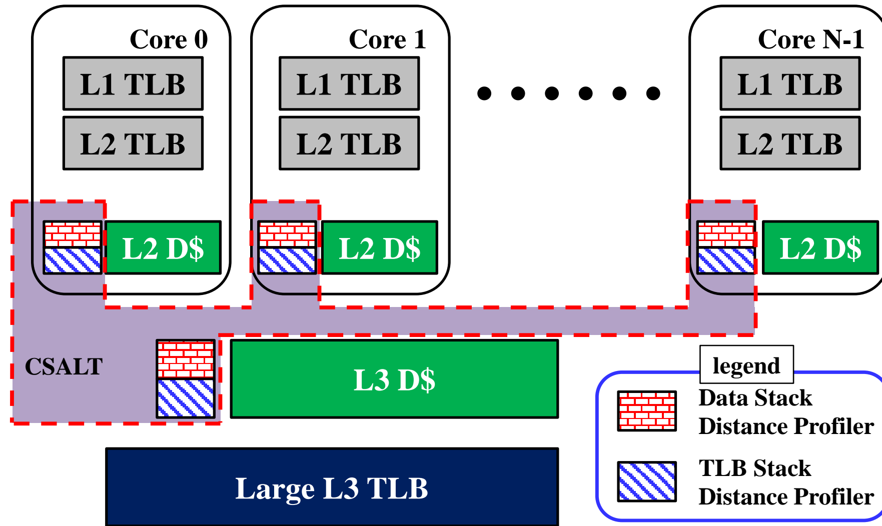


Figure 5.1: CSALT System Architecture

from slow access latency since off-chip DRAM is much slower than on-chip SRAMs. Consequently, they make this high-capacity TLB as addressable, so TLB entries can be stored in data caches. They call this TLB as POM-TLB (Part of Memory TLB) as the TLB is given an explicit address space. CSALT uses the POM-TLB organization as its substrate. It may be noted that CSALT is a cache management scheme, and can be architected over other translation schemes such as conventional page tables.

Figure 5.1 depicts the system architecture incorporating CSALT architected over the POM-TLB. CSALT encompasses L2 and L3 data cache management schemes. The role of the stack distance profilers shown in the figure is described in Section 5.1.1. In the following subsections, we describe the architecture of our Context-Switch Aware Large TLB (CSALT) scheme.

First, we explain the dynamic partitioning algorithm that helps to find a balanced partitioning of the cache between TLB and data entries to reduce the cache contention. In Section 5.1.2, we introduce a notion of "criticality" to improve the dynamic partitioning algorithm by taking into account the relative costs of data cache misses. We also describe the hardware overheads of these partitioning algorithms.

5.1.1 CSALT with Dynamic Partitioning (CSALT-D)

Since prior state-of-the-art work [20] does not distinguish data and TLB entries when making cache replacement decisions, it achieves a suboptimal performance improvement. The goal of CSALT is to profile the demand for data and TLB entries at runtime and adjust the cache capacity needed for each type of cache entry.

CSALT dynamic partitioning algorithm (*CSALT-D*) attempts to maximize the overall hit rate of data caches by allocating an optimal amount of cache capacity to data and TLB entries. In order to do so, CSALT-D attempts to minimize interference between the two entry types. Assuming that a cache is statically partitioned by half for data and TLB entries, if data entries have higher miss rates with the current allocation of cache capacity, CSALT-D would allocate more capacity for data entries. On the other hand, if TLB entries have higher miss rates with the current partitioning scheme, CSALT-D would allocate more cache for TLB entries. The capacity partitioning is adjusted at a fixed interval, and we refer to this interval as an epoch. In

order to obtain an estimate of cache hit/miss rate for each type of entry when provisioned with a certain capacity, we implement a cache hit/miss prediction model for each type of entry based on Mattson's Stack Distance (MSA) algorithm [66]. The MSA uses the LRU information of set-associative caches. For a K-way associative cache, LRU stack is an array of $(K+1)$ counters, namely $Counter_1$ to $Counter_{K+1}$. $Counter_1$ counts the number of hits to the Most Recently Used (MRU) position, and $Counter_K$ counts the number of hits to the LRU position. $Counter_{K+1}$ counts the number of misses incurred by the set. Each time there is a cache access, the counter corresponding to the LRU stack distance where the access took place is incremented.

LRU stack can be used to predict the hit rate of the cache when the associativity is increased/reduced. For instance, consider a 16-way associative cache where we record LRU stack distance for each of the accesses in a LRU stack. If we decrease the associativity to 4, all the accesses which hit in positions $LRU4 - LRU15$ in the LRU stack previously would result in a miss in the new cache with decreased associativity ($LRU0$ is the MRU position). Therefore, an estimate of the hit rate in the new cache with decreased associativity can be obtained by summing up the hit rates in the LRU stack in positions $LRU0 - LRU3$.

For a K-way associative cache, our dynamic partitioning scheme works by allocating certain ways $(0 : N-1)$ for data entries and the remaining ways for TLB entries $(N : K-1)$ in each set in order to maximize the overall cache hit rate. For each cache which needs to be dynamically partitioned,

we introduce two additional structures: a data LRU stack, and a TLB LRU stack corresponding to data and TLB entries respectively. The data LRU stack serves as a cache hit rate prediction model for data entries whereas the TLB LRU stack serves as as a cache hit rate prediction model for TLB entries. Estimates of the overall cache hit rates can be obtained by summing over appropriate entries in the data and TLB LRU stack. For instance, in a 16-way associative cache with 10 ways allocated for data entries and remaining ways allocated for TLB entries, an estimate of the overall cache hit rate can be obtained by summing over $LRU0 - LRU9$ in Data LRU stack and $LRU0 - LRU5$ in the TLB LRU stack.

This estimate of the overall cache hit rate obtained from the LRU stack is referred to as the *Marginal Utility* of the partitioning scheme[67]. Consider a K-way associative cache. Let the data LRU stack be represented as `D_LRU` and the TLB LRU stack be represented as `TLB_LRU`. Consider a partitioning scheme P that allocates N ways for data entries and $K - N$ ways for TLB entries. Then the *Marginal Utility* of P, denoted by MU_N^P is given by the following equation,

$$MU_N^P = \sum_{i=0}^{N-1} \texttt{D_LRU}(i) + \sum_{j=0}^{K-N-1} \texttt{TLB_LRU}(j). \tag{5.1}$$

CSALT-D attempts to maximize the marginal utility of the cache at each epoch by comparing the marginal utility of different partitioning schemes. Consider the example shown in Figure 5.2 for an 8-way associative cache. Suppose the current partitioning scheme assigns $N = 4$ and $M = 4$. At the

Algorithm 1 Dynamic Partitioning Algorithm

1: `N` = Number of ways to be allocated for data
2: `M` = Number of ways to be allocated for TLB
3:
4: **for** n in $N_{min} : K - 1$ **do**
5: $MU_n = \texttt{compute_MU}(n)$
6: **end for**
7:
8: `N` = $\underset{N}{\boldsymbol{arg\ max}}(MU_{N_{min}}, MU_{N_{min}+1}, ..., MU_{K-1})$
9: `M` = K - `N`

end of an epoch, the `D_LRU` and `TLB_LRU` contents are shown in Figure 5.2. In this case, the dynamic partitioning algorithm finds the marginal utility for the following partitioning schemes (not every partitioning is listed):

$$\begin{aligned} MU_4^{P1} &= \sum_{i=0}^{3} \texttt{D_LRU}(i) + \sum_{j=0}^{3} \texttt{TLB_LRU}(j) = 34 \\ MU_5^{P2} &= \sum_{i=0}^{4} \texttt{D_LRU}(i) + \sum_{j=0}^{2} \texttt{TLB_LRU}(j) = 30 \\ MU_6^{P3} &= \sum_{i=0}^{5} \texttt{D_LRU}(i) + \sum_{j=0}^{1} \texttt{TLB_LRU}(j) = 40 \\ MU_7^{P4} &= \sum_{i=0}^{6} \texttt{D_LRU}(i) + \sum_{j=0}^{0} \texttt{TLB_LRU}(j) = 50 \end{aligned} \tag{5.2}$$

Among the computed marginal utilities, our dynamic scheme chooses the partitioning that yields the best marginal utility. In the above example, CSALT-D chooses partitioning scheme $P4$. This is as elaborated in Algorithm 1 and Algorithm 2.

Once the partitioning scheme P_{new} is determined by the CSALT-D

algorithm, it is enforced globally on all cache sets. Suppose the old partitioning scheme P_{old} allocated N_{old} ways for data entries, and the updated partitioning scheme P_{new} allocates N_{new} ways for data entries. We consider two cases: (a) $N_{old} < N_{new}$ and (b) $N_{old} > N_{new}$ and discuss how the partitioning scheme P_{new} affects the cache lookup and cache replacement. While CSALT-D has no affect on cache lookup, CSALT-D does affect replacement decisions. Here, we describe the lookup and replacement policies in detail.

Cache Lookup: All K-ways of a set are scanned irrespective of whether a line corresponds to a data entry or a TLB entry during cache lookup. In case (a), even after enforcing P_{new}, there might be TLB entries resident in the ways allocated for data (those numbered N_{old} to $N_{new} - 1$). On the other hand, in case (b), there might be data entries resident in the ways allocated for TLB entries (ways numbered N_{new} to $N_{old} - 1$). This is why all ways in the cache is looked up as done in today's system.

Cache Replacement: In the event of a cache miss, consider the case where an incoming request corresponds to a data entry. In both case (a) and (b), CSALT-D evicts the LRU cacheline in the range $(0, N_{new} - 1)$ and places the incoming data line in its position. On the other hand, if the incoming line corresponds to a TLB entry, in both case (a) and (b), CSALT-D evicts the LRU-line in the range $(N_{new}, K - 1)$ and places the incoming TLB line in its position.

Classifying Addresses as Data or TLB: Incoming addresses can be classified as data or TLB by examining the relevant address bits. Since the POM-

Algorithm 2 Computing Marginal Utility

```
1: N = Input
2: D_LRU = Data LRU Stack
3: TLB_LRU = TLB LRU Stack
4: MU = 0
5:
6: for i in 0 : N − 1 do
7:     MU += D_LRU (i)
8: end for
9: for j in 0 : K − N − 1 do
10:    MU += TLB_LRU (j)
11: end for
12: return MU
```

TLB is a memory mapped structure, the cache controller can identify if the incoming address is to the POM-TLB or not. For stored data in the cache, there are two ways by which this classification can be done: i) by adding 1 bit of metadata per cache block to denote data (0) or TLB (1), or ii) by reading the tag bits and determining if the stored address falls in the L3 TLB address range or not. We leave this as an implementation choice. In our work, we assume the latter option as it does not affect metadata storage.

Finally, the overall flow is summarized in Figure 5.3. Each private L2 cache maintains its own stack distance profilers and updates them upon accesses to it. When an epoch completes, it computes marginal utilities and sets up a (potentially different) configuration of the partition between data ways and TLB ways. Misses (and writebacks) from the L2 caches go to the L3 cache which performs a similar update of its profilers and configuration outcome. A TLB miss from the L3 data cache is sent to the L3 TLB. Finally,

	DATA LRU Stack		TLB LRU Stack
LRU0	8	LRU0	5
LRU1	9	LRU1	1
LRU2	2	LRU2	0
LRU3	1	LRU3	8
LRU4	4	LRU4	15
LRU5	10	LRU5	1
LRU6	11	LRU6	10
LRU7	3	LRU7	7
LRU8	12	LRU8	12

Figure 5.2: LRU Stack Example

a miss in the L3 TLB triggers a page walk.

5.1.2 CSALT with Criticality Weighted Partitioning (CSALT-CD)

CSALT-D assumes that the impact of data cache misses is equal for both data and TLB entries, and as a result, both the data and TLB LRU stacks had the same weight when computing the marginal utility. However, this is not necessarily true since a TLB miss can cause a long latency page walk. Note that even if the translation request misses in an L3 data cache, the entry may still hit in the L3 TLB thereby avoiding a page walk. In order to maximize the performance, the partitioning algorithm needs to take the relative performance gains obtained by TLB entry hit and the data entry hit in the data caches into account.

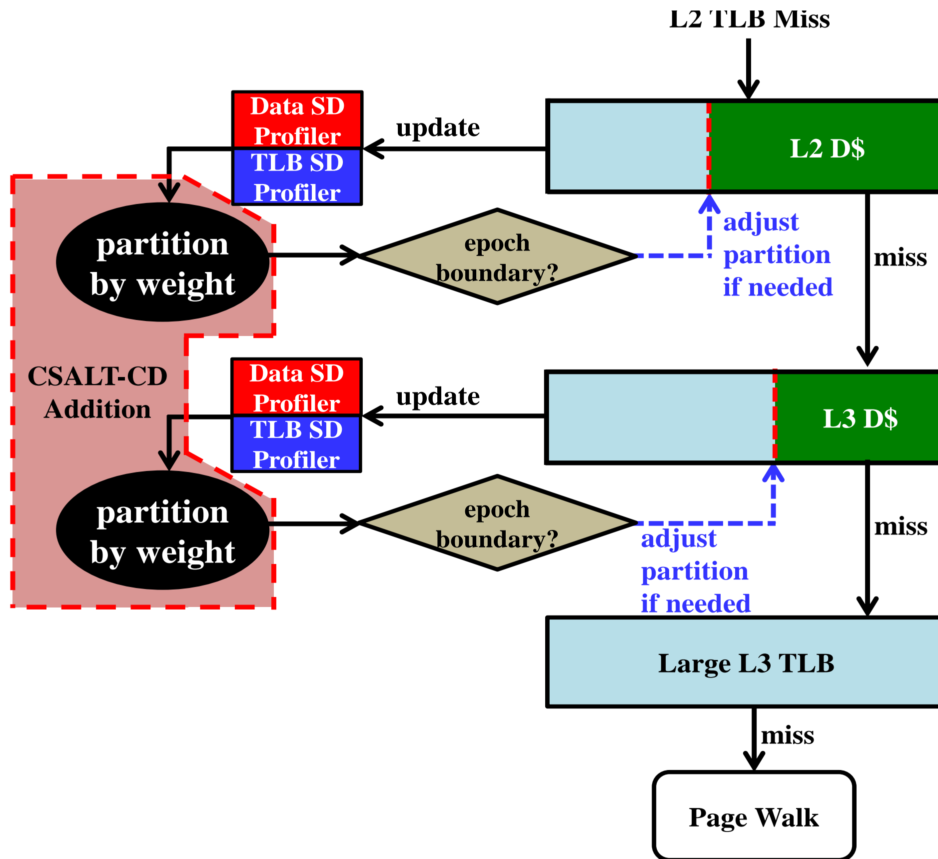


Figure 5.3: CSALT Overall Flowchart

Therefore, we propose a dynamic partitioning scheme that considers criticality of data entries, called Criticality Weighted Dynamic Partitioning (*CSALT-CD*). We use the insight that data and TLB misses incur different penalties on a miss in the data cache. Hence, the outcome of stack distance profiler is scaled by its importance or weight, which is the performance gain obtained by a hit in the data cache. Figure 5.3 shows an overall flowchart with additional hardware to enable such scaling (the red shaded region shows the additional hardware).

In *CSALT-CD*, a performance gain estimator is added to estimate the impact of a TLB entry hit and a data entry hit on performance. In an attempt to minimize hardware overheads, *CSALT-CD* uses existing performance counters. For estimating the hit rate of the L3 data cache, *CSALT-CD* uses performance counters that measures the number of L3 hits and the total number of L3 accesses that are readily available on modern processors. For estimating the L3 TLB hit rate, a similar approach is used. Utilizing this information, the total number of cycles incurred by a miss for each kind of entry is computed dynamically. The ratio of the number of cycles incurred by a miss to the number of cycles incurred by a hit for each kind of entry is used to estimate the performance gain on a hit to each kind of entry. For instance, if a data entry hits in the L3 cache, the performance gain obtained is the ratio of the average DRAM latency to the total L3 access latency. If a TLB entry hits in the L3 cache, the performance gain obtained is the ratio of the sum of the TLB latency and the average DRAM latency to the total L3 access latency.

These estimates of performance gains are directly plugged in as *Criticality Weights* which are used to scale the *Marginal Utility* from the stack distance profiler. To estimate these performance gains, the following approach is used.

Estimating performance gains: We use the notation $L1D_{Acc}, L2_{Acc}$, and $L3D_{Acc}$ to represent the access times for L1, L2, and L3 data caches respectively. $L1T_{Acc}, L2T_{Acc}$, and $L3T_{Acc}$ represent the access times for L1 TLB, L2 TLB and L3 TLB respectively. $L2D_{hit}, L2D_{miss}, L3D_{hit}$, and $L3D_{miss}$ represent the hit rate and miss rates of L2 and L3 data caches. $L3T_{hit}, L3T_{miss}$ represents the hit and miss rate for L3 TLB respectively. PW_{Acc} represents the page walk cycles, and $DRAM_{Acc}$ represents average DRAM access latency.

Consider a virtual address *VA* whose translation resident neither in L1 TLB nor L2 TLB. If its translation is found in the L2 data cache, the process of translation is sped up by a certain measure as a result of not incurring a page walk. The speedup S_{Tr} obtained is given by

$$S_{Tr} = \frac{\text{Translation time on a L2 data cache miss}(N_{Tr})}{\text{Translation time on a L2 data cache hit}(D_{Tr})} \tag{5.3}$$

The denominator is a summation of the access times of L1 TLB, L2-TLB and L2 data cache.

$$D_{Tr} = L1T_{Acc} + L2T_{Acc} + L2D_{Acc} \tag{5.4}$$

The numerator, however, is dependent on other factors. In the event on a L2 data cache miss, it is possible that the translation entry is resident in the L3 data cache or the L3 TLB. In the worst case, the virtual address *VA* might

incur a page walk if the the translation for VA is resident neither in L3 data cache nor L3 TLB. The numerator is approximated by utilizing hit rate of the L3 data cache and the hit rate of the L3 TLB as follows.

$$\begin{aligned} N_{Tr} = D_{Tr} + L3D_{hit} \times (D_{Tr} + L3D_{Acc}) + \\ (L3D_{miss}) \times (L3T_{hit}) \times (D_{Tr} + L3D_{Acc} + L3T_{Acc}) + \\ (L3D_{miss}) \times (L3T_{miss}) \times (D_{Tr} + L3D_{Acc} + \\ L3T_{Acc} + PW_{Acc}) \end{aligned} \quad (5.5)$$

On the other hand, consider a data cache line DL. A hit in the L2/L3 data cache for DL eliminates the off-chip memory access latency. The relative speedup S_{Dat} obtained in this case is

$$S_{Dat} = \frac{\text{Access time on a L2 data cache miss}(N_{Dat})}{\text{Access time on a L2 data cache hit}(D_{Dat})} \quad (5.6)$$

where the denominator is a summation of access times of L1 and L2,

$$D_{Dat} = L1D_{Acc} + L2D_{Acc} \quad (5.7)$$

and the numerator is estimated by utilizing information about the hit rate of L3 data cache.

$$\begin{aligned} N_{Dat} = D_{Dat} + (L3D_{hit}) \times (D_{Dat} + L3D_{Acc}) + \\ (L3D_{miss}) \times (D_{Dat} + L3D_{Acc} + DRAM_{Acc}) \end{aligned} \quad (5.8)$$

Therefore, a TLB entry hit in the L2 data cache results in a speedup of S_{Tr}, and a data entry hit in the L2 data cache results in speedup of S_{Dat}. Speedups for the L3 data cache can be obtained in a similar manner.

We define a new quantity called the *Criticality Weighted Marginal Utility*. For a partitioning scheme P which allocates N data ways out of K ways, Criticality Weighted Marginal Utility (CWMU), denoted as $CWMU_N^P$, is given by the following equation. We could normalize the values in the LRU stack with respect to the number of data and TLB entry accesses, but we do not do so for the sake of simplicity.

$$CWMU_N^P = S_{Dat} \times \sum_{i=0}^{N-1} \texttt{D_LRU}(i) + S_{Tr} \times \sum_{j=0}^{K-N-1} \texttt{TLB_LRU}(j). \tag{5.9}$$

The partitioning scheme with the highest CWMU is used for the next epoch. Figure 5.3 shows the overall flow chart of CSALT-CD with the additional step required (the red shaded is the addition for CSALT-CD). We have used separate performance estimators for L2 and L3 data caches as the performance impact of L2 and L3 data caches is different. Algorithm 3 shows the pseudocode of *CSALT-CD*. For a data entry, this performance gain is denoted by S_{Dat}, and for a TLB entry, by S_{Tr}. A method to estimate these performance gains have been elaborated previously. These criticality weights are dynamically estimated using the approach elaborated earlier. The rest of the flow (cache accesses, hit/miss evaluation, replacement decisions) is the same as in CSALT-D.

Criticality weighted partitioning attempts to maximize the CWMU at each epoch by comparing CMWU of different partitioning schemes and choosing the one with highest CWMU. Computation of CWMU is as elaborated Algorithm 3. Maximization of CWMU is achieved as shown in Algorithm 1.

Algorithm 3 Computing CWMU

```
1: N = Input
2: D_LRU = Data LRU Stack
3: TLB_LRU = TLB LRU Stack
4: CWMU = 0
5:
6: for i in 0 : N − 1 do
7:     CWMU += S_Dat × D_LRU (i)
8: end for
9: for j in 0 : K − N − 1 do
10:    CWMU += S_Tr × TLB_LRU (j)
11: end for
12: return CWMU
```

The values of S_{Dat} and S_{Tr} are calculated using equations in 5.6 and 5.3 at each epoch.

5.1.3 Hardware Overhead

Both CSALT-D and CSALT-CD algorithms use stack distance profilers for both data and TLB. The area overhead for each stack distance profiler is negligible. This structure requires the MSA LRU stack distance structure, which is equal to the number of ways, so in case of L3 data cache, it is 16 entries. Computing the marginal utility only requires a few adders that will accumulate the sum of a few entries in the stack distance profiler. Both CSALT-D and CSALT-CD also require an internal register per partitioned cache which contains information about the current partitioning scheme, specifically, N, the number of ways allocated for data in each set. The overhead of such a register is minimal, and depends on the associativity of the cache. Furthermore,

the CSALT-CD algorithm uses a few additional hardware structures, which include the hit rates of L3 data cache and L3 TLB. However, these counters are already available on modern processors as performance monitoring counters. Thus, estimating the performance impact of data caches and TLBs will only require a few multipliers that will be used to scale the marginal utility by weight. Therefore, we observe that the additional hardware overhead required to implement CSALT with criticality weighted partitioning is minimal.

5.1.4 Effect of Replacement Policy

Until this point, we assumed a True-LRU replacement policy for the purpose of cache partitioning. However, True-LRU is quite expensive to implement, and is rarely used in modern processors. Instead, replacement policies like Not Recently Used (NRU) or Binary Tree (BT) pseudo-LRU are used [68]. Fortunately, the cache partitioning algorithms utilized by CSALT are not dependent on the existence of True-LRU policy. There has been prior research to adapt cache partitioning schemes to Pseudo-LRU replacement policies[68], and we leverage it to extend CSALT.

For NRU replacement policy, we can easily estimate the LRU stack positions depending on the value of the NRU bit on the accessed cache line. For Binary Tree-pseudoLRU policy, we utilize the notion of an Identifier (ID) to estimate the LRU stack position. Identifier bits for a cache line represent the value that the the binary tree bits would assume if a given line held the LRU position. In either case, estimates of LRU stack positions can be used to update

the LRU stack. It has been shown that using these estimates instead of the actual LRU stack position results in only a minor performance degradation[68].

5.2 Translation Coherence using Addressable TLBs (TCAT)

In this section, we describe TCAT, our hardware translation coherence scheme which precisely captures nested-page table updates initiated by the guest OS. Our design alleviates virtually all foreground overheads associated with translation coherence as listed in Section 3.3. In a manner similar to UNITD[22] and HATRIC[24], our design integrates TLBs into the existing cache coherence protocol. It exploits the addressing scheme of the Part-Of-Memory TLB (POM-TLB)[27] architecture to enable sending coherence messages to achieve precise invalidation on the slave cores upon a guest page table update.

5.2.1 Overview of our scheme

TCAT is a hardware translation coherence scheme designed for virtualized environments. While HATRIC[24] can track host page table changes, TCAT can track guest page table changes by overlaying TLB coherence atop cache coherence. Additionally, unlike current VMMs[69] which do not track guest virtual pages, and hence cannot perform precise invalidation on the slave cores, TCAT enables invalidation of TLB entries for a specified page. It does so by leveraging the POM-TLB. Each translation is tied to a specific location in the POM-TLB - therefore, when translation structures receive a coherence

message for a POM-TLB address, they know which translation entries to invalidate.

5.2.2 Overlaying translation coherence atop cache coherence

Our design integrates TLBs into the existing cache coherence protocol. From the cache coherence protocol perspective, a TLB is just another cache which houses translations instead of data. TLBs use the same set of coherence states that a data/instruction cache uses. If a MOESI protocol is used, the possible coherence states in the TLB are: Exclusive (E), Shared (S) and Invalid (I), since translations cannot be modified in the TLBs themselves. When a translation T is brought into the TLB, it is marked as Exclusive (E). When another core tries to read T and incurs a L2 TLB miss, it goes through the POM-TLB address translation procedure, either installing the translation in the L2 data cache of that core or reading the translation from the L2 data cache of that core where it is already resident. In either case, the L2 data cache sends a coherence message to all the TLBs, and the translation is marked as Shared (S). The cached translation can be accessed by the local core as long as it is in the Shared (S) state. The translation remains in this state until the TLB receives a coherence message invalidating the translation. The translation is then marked Invalid (I). Coherence messages invalidating a translation can be received due to page table updates on any other core or due to page table updates on the the local core. Page table updates on the local core are captured by self-snooping. Page table updates on a remote core are relayed by the

directory. Subsequent memory accesses which need the translation will miss in the TLB and go through the POM-TLB address translation procedure.

Note that in native environments, TLBs store $VA - PA$ mappings. Therefore, coherence states need to be updated when a $VA - PA$ mapping changes. In virtualized environments, TLBs store $gVA - hPA$ mappings. In this case, coherence states need to be updated in two cases: (1) $gVA - gPA$ mapping changes (2) $gPA - hPA$ mapping changes. The first case corresponds to a guest page table update, and the second case corresponds to a host-page table update. Our design, currently, can capture only case (1). We leave case (2) as future work.

When a coherence message is received by the TLB, it has to identify translations whose coherence states must change. This is achieved by exploiting certain features of the POM-TLB design and is elaborated in the subsequent sections.

5.2.3 Coherence lookups in the TLB

While cache coherence is based on physical addresses of blocks, a translation cached in a TLB is not directly addressable by the physical addresses on which it resides. For the TLBs to participate in the coherence protocol, the hardware coherence scheme must be able to perform coherence lookups in the TLB. In native environments, UNITD accomplishes this by augmenting the TLBs with the physical address of the last-level PTE. Similarly, in virtualized environments, HATRIC augments the TLBs with the physical address of

the last-level host PTE to capture updates to the nested page table initiated by the host. In our design, we exploit features of the Part-Of-Memory TLB (POM-TLB) addressing scheme to enable coherence lookups.

As referenced in the previous section, POM-TLB is a large shared L3 TLB after the private L2 TLBs, and is a part of the main memory. Each entry in the POM-TLB has a valid bit, process ID, Virtual Address (VA), and Physical Address (PA) as in on-chip TLBs. To facilitate the translation in virtualized platforms, it also has Virtual Machine (VM) ID to distinguish addresses coming from different virtual machines. The attributes include information such as replacement and protection bits. POM-TLB is designed as a 4-way set associative TLB - each entry is 16B and four entries make 64B. As each set holds four 16-byte TLB entries, POM-TLB comprising N sets is assigned an address range of $64 \times N$ bytes. The virtual address (VA) of the L2 TLB miss is converted to a POM-TLB set index by extracting $log_2(N)$ bits of the VA. The memory address of the set that the VA maps to is given by:

$$ADDR_{POM-TLB}(VA) = (((VA \oplus VM_ID) >> 6) \,\&\, ((1 << log_2(N)) - 1)) * 64 + BASE_ADDR_{POM-TLB} \quad (5.10)$$

Due to the POM-TLB addressing scheme and layout, any guest virtual address (gVA) to host physical address (hPA) translation from a single VM maps to a fixed location in the POM-TLB. Due to the entirety of the virtual address space mapping to a fixed-size POM-TLB, many gVA-hPA mappings map to the same location in the POM-TLB. As a result, the POM-TLB design

enforces a many-to-one mapping between guest virtual address (gVA) and POM-TLB address ($ADDR_{POM-TLB}$). We utilize this feature to devise a method to acheive coherence lookups in the TLB.

5.2.4 Achieving Translation Coherence

Consider a guest page table mapping T_g which maps guest virtual address gVA_1 to guest physical address gPA_1, and the host page table mapping T_h which maps guest physical address gPA_1 to host physical address hPA_1. In virtualized environments, TLBs store the mapping $gVA_1 - hPA_1$, which we denote by T_{tlb}. When the guest OS initiates an guest page table update U on Core C, it updates the $gVA_1 - gPA_1$ mapping to $gVA_1 - gPA_2$. The update U has resulted in the mapping $gVA_1 - hPA_1$ becoming stale. Suppose another core $C' \neq C$ has the translation $gVA_1 - hPA_1$ resident on its private TLBs. Translation coherence necessitates that this stale translation be invalidated from the private TLBs of C'. This is achieved by our translation coherence scheme in the following manner.

5.2.4.1 Initiator core operations

After the guest OS performs writes to gPAs corresponding the guest PTEs of the translation T_g, it performs a hypercall to the underlying VMM with information about the gVA of translation that it modified. In the hypercall, the VMM performs the following sequence of steps on the initiator core C i.e. the physical core on which an update to mapping T_g has been initiated.

- **Calculate the POM-TLB address corresponding to** gVA_1: From equation 5.10, the POM-TLB address A corresponding to gVA_1 is calculated. Any mapping from gVA_1 to any of one of the numerous host physical addresses must reside at this fixed location A owing to the many-to-one mapping property of the POM-TLB addressing scheme.

- **Perform a Read-Modify-Write (RMW) to POM-TLB address** A: The address A may or may not contain the translation T_{tlb}. This is because T_{tlb} may not be a recently-used translation in any of the cores, and hence, might not be resident on the POM-TLB. The RMW inspects the contents of address A to determine whether T_{tlb} is resident or not. If it is resident, the RMW changes the valid bit of T_{tlb} to 0. If it is not resident, the RMW writes back the same contents to address A. This RMW operation results in two things: (1) Coherence messages are generated from Core C for physical address A. These coherence messages are relayed to the private translation structures of all cores by the directory (TLBs, MMU caches, nTLBs) in addition to their data caches (2) Contents of address A are resident on L1 data cache of Core C.

- **Perform a Cache Line Flush of address** A: Cache Line Flush of address A performs the following set of operations: (1) invalidates the cache line that contains A from all levels of the processor cache hierarchy (2) the invalidation is broadcast throughout the cache coherence

domain (3) If, at any level of the cache hierarchy, the line is dirty, it is written to memory before invalidation. Almost all architectures enable cache line flush either through a single instruction or a sequence of instructions. For instance, in x86-64, CLFLUSH instruction performs the exact set of operations as described above[70]. In ARM, this operation can be achieved through a combination of writes to $CP15$ register and the MCR instruction[71]. Therefore, we assume support for this operation across microarchitectures. Cache Line Flush for A is always valid because of the guarantee provided by the RMW: A is resident on the L1 data cache. After Cache Line Flush, the following properties are guaranteed (1) The translation T_{tlb} is not resident on any of the data caches or translation structures on any core. Non-residence on the data caches of the local core is guaranteed by Cache Line Flush. Non-residence on translation structures of local core is guaranteed by RMW operation and self-snooping (2) If the translation T_{tlb} was resident on the POM-TLB at address A, the valid bit of T_{tlb} has been set to 0 on account on writing back to memory.

5.2.4.2 Slave core operations

On the other hand, the following sequence of operations occur on the slave core C' i.e. the core which needs to perform invalidation of T_{tlb} on its private translation structures.

- **Translation structures receive coherence messages**: All private

translation structures on Core C' receive coherence messages relayed by the directory on account of a translation update on on the initiator core C. These coherence messages are generated by the RMW step on the initiator core for Address A.

- **Precisely identify invalidation targets**: While this operation is performed on each one of the private translation structures, we demonstrate the operation for private TLBs. On other translation structures, the operation can be performed in a similar manner. We propose two approaches to precisely identify targets for invalidation: *TCAT-cotag* and *TCAT-cotagless*. In TCAT-cotag, we tag each TLB entry with the POM-TLB address, like HATRIC's cotags. This allows invalidation at a translation entry granularity. In TCAT-cotagless, we do not utilize cotags. Instead, we compute the index of the set on the private TLB which contains the invalidation target by using the POM-TLB address. This approach leverages certain features of the POM-TLB addressing scheme. Computing the set index on the private TLB e comprises of two steps (1) Calculate the POM-TLB set-index $S_{POM-TLB}(A)$ for address A (2) Determine the set index on the private TLB, $S_{cur-tlb}(A)$, from $S_{POM-TLB}(A)$. To calculate the POM-TLB set index, a modified form of Equation 5.10 is used.

$$\begin{aligned} S_{POM-TLB}(A) &= \frac{ADDR_{POM-TLB} - BASE_ADDR_{POM-TLB}}{64} \\ &= (ADDR_{POM-TLB} - BASE_ADDR_{POM-TLB}) >> 6 \end{aligned} \tag{5.11}$$

Due to the many-to-one mapping nature of the POM-TLB addressing scheme, all translations which are potentially affected by the received coherence messages can reside only in this POM-TLB set. POM-TLB, by nature, is always the the largest TLB in the system. Therefore, the index bits of POM-TLB subsume the index bits of any private TLB. In other words, if a private TLB contains M sets, then $N > M$, where N is the number of POM-TLB sets. Therefore, the $log_2(M)$ least significant bits of the POM-TLB set index $S_{POM-TLB}(A)$ can be used to determine the set index of the private TLB $S_{cur-tlb}(A)$ in the following fashion.

$$S_{cur-tlb}(A) = S_{POM-TLB}(A) \;\&\; ((1 << log_2(M)) - 1) \tag{5.12}$$

Again, due to the many-to-one mapping nature of the POM-TLB addressing scheme, all translations which are potentially affected by the received coherence messages can reside only in this private-TLB set.

- **Invalidate entire set**: On each translation structure, all translations resident in the set $S_{cur-tlb}$ computed in the previous step are invalidated.

Non-residence on data caches of remote cores is guaranteed due to coherence messages generated by RMW operation. Non-residence on translation structures of remote cores in guaranteed by RMW operation and slave-core operations. These guarantees, coupled with the guarantees provided by Cache Line Flush on the initiator core, ensure that when any of the cores in the system try to obtain the host physical address corresponding to gVA_1, they always incur

a 2D-page walk and read the updated translation. This proves the correctness of our translation coherence scheme.

5.2.5 Discussion

5.2.5.1 Why is Cache Line Flush required?

Cache Line Flush is necessary to ensure that the modified translation has been invalidated in the all caching structures in the system. In other words, Cache Line Flush in necessary for the correctness of our coherence scheme. Suppose core C updates the mapping T_g. This necessitates invalidation of the mapping T_{tlb} as explained previously. Assume that we do not perform Cache Line Flush after the RMW operation, and that the mapping T_{tlb} was resident in the POM-TLB. After RMW, contents of POM-TLB address A containing T_{tlb} are resident in the L1 and L2 data caches of Core C (assuming non-inclusive caches). They are also resident in the LLC. However, the valid bit on T_{tlb} is guaranteed to be 0 only in L1 data cache of C on account of the RMW. In the L2 data cache of C and the LLC, the mapping T_{tlb} can still be valid. Coherence messages generated by the RMW to address A invalidate the mapping T_{tlb} in the private data caches and private translation structures all remote cores.

Consider the scenario in which a core $C' \neq C$ tries to get the host physical address corresponding to the guest virtual address gVA_1. Translation coherence enforces the requirement that core C' must always fetch the latest mapping. However, the following sequences of operations ensue: Due to non-residence in the private translation structures of Core C', POM-TLB address

translation procedure kicks in, and tries to read contents of the address A from the L2 data cache of core C'. As data caches on C' cannot contain contents of A due to RMW, it incurs a L2 data cache miss. The L2 data cache miss may result in two different sequences of events: (1) A cache-to-cache transfer request is initiated from the private caches of Core C' to the private caches of Core C. This is because the L2 data cache of Core C still contains the contents of address A (2) A LLC access is initiated. In case of the cache-to-cache transfer, once the transfer completes, the POM-TLB address translation procedure inspects the contents of A and finds out that the translation T_{tlb} has its valid bit set to 0 because of the previous RMW. As a result, the POM-TLB address translation procedure initiates an LLC access. Therefore, both sequences culminate in a LLC access. When the POM-TLB address translation procedure accesses the LLC for contents of address A, barring a LLC eviction, it will find the mapping T_{tlb} in the LLC in valid state. As a result, it is possible for core C' to pick up the stale translation and install it in its own private TLBs. There is no guarantee that the mapping T_{tlb} is invalidated from LLC, which in turn provides no guarantee that Core C' does not pick up the stale translation. This precludes the correctness of TCAT.

A Cache Line Flush performed after the RMW circumvents this correctness issue by flushing contents of A from all levels of the cache hierarchy. It also writes back to memory i.e. POM-TLB with the valid bit of T_{tlb} cleared. As a result, the POM-TLB address translation procedure on Core C' fails to find the mapping for guest virtual address gVA_1 in any of its translation/-

caching structures, and a 2D page table walk is initiated. Since the guest PTEs corresponding to the mapping have already been updated by the guest OS, Core C' is ensured to find the latest mapping for the guest virtual address gVA_1 upon a 2D page table walk.

5.2.5.2 Assumptions

We assume that the system is architected atop a POM-TLB. We also assume a POM-TLB addressing scheme which results in a many-to-one mapping between the guest virtual addresses and the POM-TLB addresses. For TCAT-cotagless, additionally, we assume that the POM-TLB set index bits subsume the index bits of the private TLBs. This is a reasonable assumption because of two reasons: (1) POM-TLB is a very large shared TLB by design, and supposed to contain more sets than any of the private TLBs (2) We have a $POM-TLB_{small}$ dedicated to small pages (4kB) and a $POM-TLB_{large}$ dedicated to large pages (2MB/4MB). In case of a large page, the set-index bits of $POM-TLB_{large}$ subsume the set-index bits of the private TLBs dedicated to large pages. In case of a small page, the set-index bits of $POM-TLB_{small}$ subsume the set-index bits of the private TLBs dedicated to small pages. For a skewed-associative TLB, it is not straightforward to compute the index of the set to be invalidated on receiving a coherence message. However, since the POM-TLB address is broadcast to all translation structures, a skewed-associative TLB can still precisely identify the target - although it might have to go through an additional set of computations.

5.2.5.3 Precision of Invalidation

In *TCAT-cotag*, upon reception of coherence messages for POM-TLB address A, the slave core performs invalidations of entries which are tagged with A. This results in precise invalidations at the granularity of a translation entry. In *TCAT-cotagless*, given a coherence message for POM-TLB address A, the slave core identifies the set in the TLB which can potentially contain the modified mapping and invalidates the entire set. The precision of invalidation achieved by *TCAT-cotagless* scheme is at the granularity of a set. The overhead of such imprecision is minimal as evident from our results.

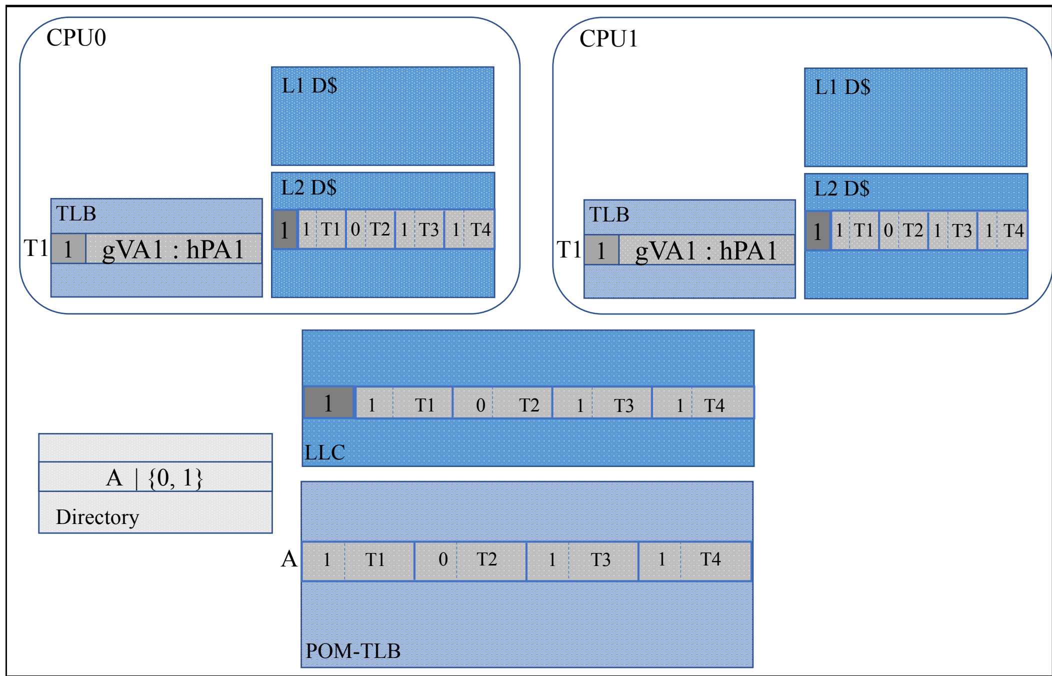
5.2.6 Putting it all together

Figure 5.4 details our coherence scheme's overall operation. Figure 5.4(a) shows the initial state of the system with two cores. A mapping $T1$ from guest virtual address gVA_1 to host physical address hPA_1 is resident on TLBs of CPU0 and CPU1. A single 64B POM-TLB entry, which holds 4 16-byte TLB entries containing translations $T1$, $T2$, $T3$, and $T4$ respectively, is resident on the on-chip data caches and on the POM-TLB. The dark-gray box on the far-left of the POM-TLB entry on the on-chip data caches represents the valid bit.

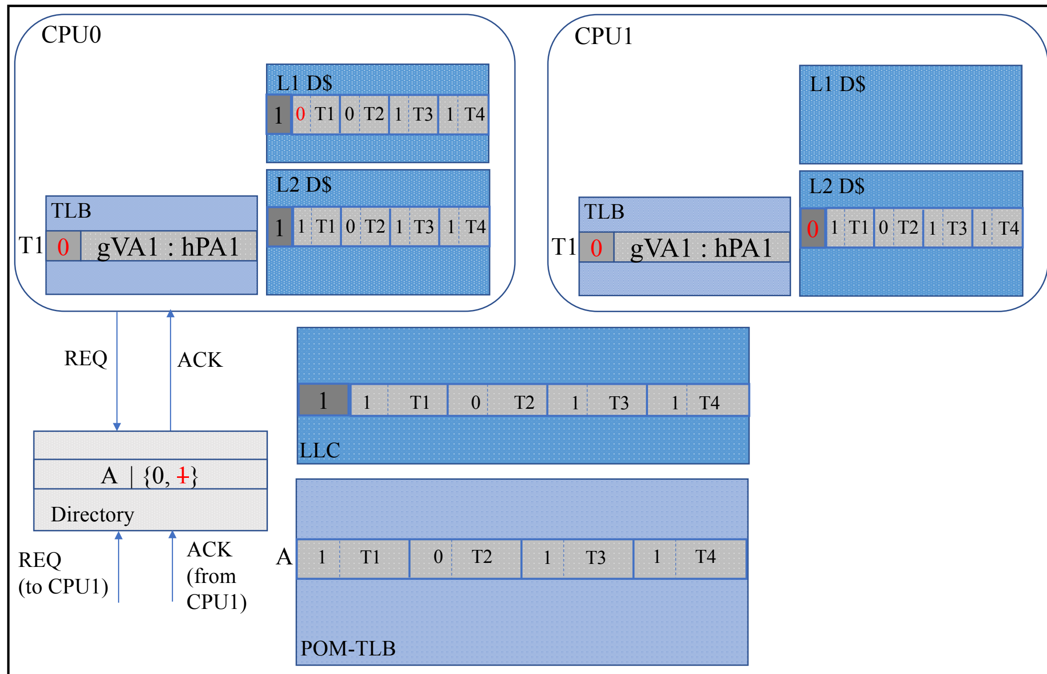
A vCPU running on CPU0 changes the mapping $T1$. To ensure translation coherence, the VMM performs the initiator core operations. The VM first computes the POM-TLB address corresponding to gVA_1, which is A. It then reads the contents of address A. This results in the 64-byte POM-TLB entry

containing translation $T1$ to be brought into the L1 data cache of CPU0. The VMM inspects the contents of the POM-TLB entry, and finds that the POM-TLB entry does contain the translation $T1$. Therefore, it performs a write to A changing the valid bit of translation $T1$ to 0. This generates coherence requests to the directory for address A. The directory forwards the request to the list of sharers, in this case CPU1. Cache coherence clears the valid bit of the POM-TLB entry resident on the L2 data cache of CPU1. Slave operations detailed in the previous section ensure that the valid bit of translation $T1$ on the TLB of CPU1 is cleared. CPU1 sends an acknowledgement to the directory conveying that it has performed the necessary invalidations. The directory removes CPU1 from the sharer list and forwards the acknowledgement to CPU0. Self-snooping on CPU0 ensures that the valid bit of translation $T1$ on the TLB of CPU0 is cleared. The sequence of activities occuring on hardware upon a RMW, and the state of the system after the RMW is depicted in Figure 5.4(b).

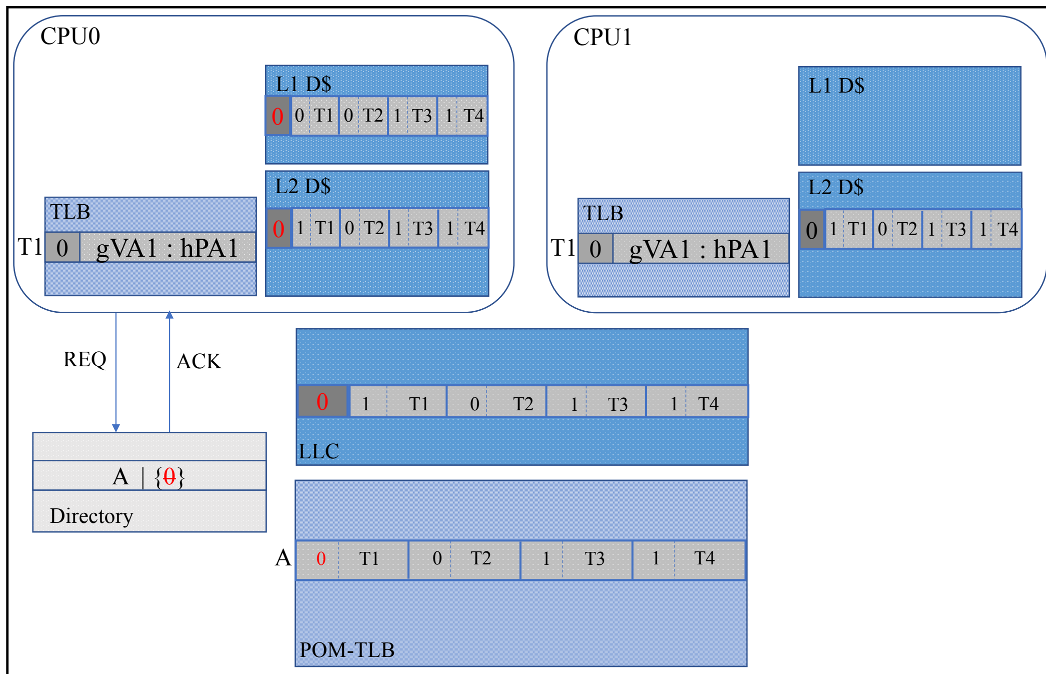
The VMM then initiates a Cache Line Flush of address A. Cache Line Flush clears the valid bit of the cache lines which hold the POM-TLB entry containing translation $T1$. It also performs a write-back of the dirty POM-TLB entry to the POM-TLB. The state of the system after the Cache Line Flush is depicted in Figure 5.4(c). In this state, note that any core in the system is mandated to perform a 2D-page walk to obtain the address translation of gVA_1, as the translation $T1$ is not valid on any of the translation-caching structures.



(a) Initial State: Initiating coherence for $T1$ resident in POM-TLB address A



(b) Read-Modify-Write to A



(c) Cache Line Flush of A

Figure 5.4: Coherence Scheme in action

Chapter 6

Results

6.1 Experimental Set-Up

We evaluate the performance of CSALT and POM-TLB translation coherence using a combination of real system measurements, Pin tool [72], and heavily modified Ramulator [73] simulation. The virtualization platform is QEMU [74] 2.0 with KVM [75] support. Our host system is Ubuntu 14.04 running on Intel Skylake [17] with Transparent Huge Pages (THP) [76] turned on. The system also has Intel VT-x with support for Extended Page Tables [77]. The host system parameters are shown in Table 6.1 under Processor, MMU, and PSC categories. The guest system is Ubuntu 14.04 also with THP turned on. Although the host system has a separate L1 TLBs for 1GB pages, we do not make use of it. The L2 TLB is a unified TLB for both 4KB and 2MB pages. In order to measure page walk overheads, we use specific performance counters (e.g., 0x0108, 0x1008, 0x0149, 0x1049), which take MMU caches into account. The page walk cycles used in this section are the average cycles spent after a translation request misses in L2 TLB.

Processor	Values
Frequency	4 GHz
Number of Cores	8
L1 D-Cache	32KB, 8 way, 4 cycles
L2 Unified Cache	256KB, 4 way, 12 cycles
L3 Unified Cache	8MB, 16 way, 42 cycles
MMU	**Values**
L1 TLB (4KB)	64 entry, 9 cycles
L1 TLB (2MB)	32 entry, 9 cycles
	L1 TLBs 4 way associative
L2 Unified TLB	1536 entry, 17 cycles
	L2 TLBs 12 way associative
PSC	**Values**
PML4	2 entries, 2 cycle
PDP	4 entries, 2 cycle
PDE	32 entries, 2 cycle
Die-Stacked DRAM	**Values**
Bus Frequency	1 GHz (DDR 2 GHz)
Bus Width	128 bits
Row Buffer Size	2KB
tCAS-tRCD-tRP	11-11-11
DDR	**Values**
Type	DDR4-2133
Bus Frequency	1066 MHz
	(DDR 2133 MHz)
Bus Width	64 bits
Row Buffer Size	2KB
tCAS-tRCD-tRP	14-14-14

Table 6.1: Experimental Parameters

VM1	VM2
canneal_x8	connected component_x8
canneal_x8	streamcluster_x8
graph500_x8	gups_x8
pagerank_x8	streamcluster_x8

Table 6.2: Heterogeneous Workloads Composition

6.2 Context-Switch Aware Large TLB

6.2.1 Workloads

The main focus of this work is on memory subsystems, and thus, applications, which do not spend a considerable amount of time in memory, are not meaningful. Consequently, we chose a subset of PARSEC [78] applications that are known to be memory intensive. In addition, we also ran graph benchmarks such as the *graph500* [79] and big data benchmarks such as *connected component* [80] and *pagerank* [81]. We paired two multi-threaded benchmarks (two copies of the same program, or two different programs) to study the problems introduced by context switching. The heterogeneous workload composition is listed in Table 6.2. The x8 denotes the fact that all our workloads are run with 8 threads.

6.2.2 Simulation

Our simulation methodology is different from prior work [36, 82] that relied on a linear additive performance model. The drawback of the linear model is that it does not take into account the overlap of instructions and address translation traffic, but merely assumes that an address translation re-

quest is blocking that the processor immediately stalls upon a TLB miss. This is not true in modern hardware as the remaining instructions in the ROB can continue to retire as well as some modern processors [17] allow simultaneous page walkers. Therefore, we use a cycle accurate simulator that uses a heavily modified Ramulator. We ran each workload 10 billion instructions. The front-end of our simulator uses the timed traces collected from real system execution using the Pin tool. During playback, we simulate two contexts by switching between two input traces every 10ms. We choose 10ms as the context switch granularity based on measured data from prior works [83, 84].

In our simulation, we model the TLB datapath where the TLB miss still lets the processor to flush the pipeline, so the overlap aspect is well modeled. We simulate the entire memory system accurately, including the effects of translation accesses on L2 and L3 data caches as well as the misses from data caches that are serviced by POM-TLB or off-chip memory. The timing details of our simulator are summarized in Table 6.1.

The performance improvement is calculated by using the ratio of improved IPC (geometric mean across all cores) over the baseline IPC (geometric mean across all cores), and thus, higher normalized performance improvement indicates a higher performing scheme.

This section presents simulation results from a conventional system with only L1-L2 TLBs, a POM-TLB system, and various CSALT configurations. POM-TLB is the die-stacked TLB organization using the LRU replacement scheme in L2 and L3 caches [20]. CSALT-D refers to proposed scheme

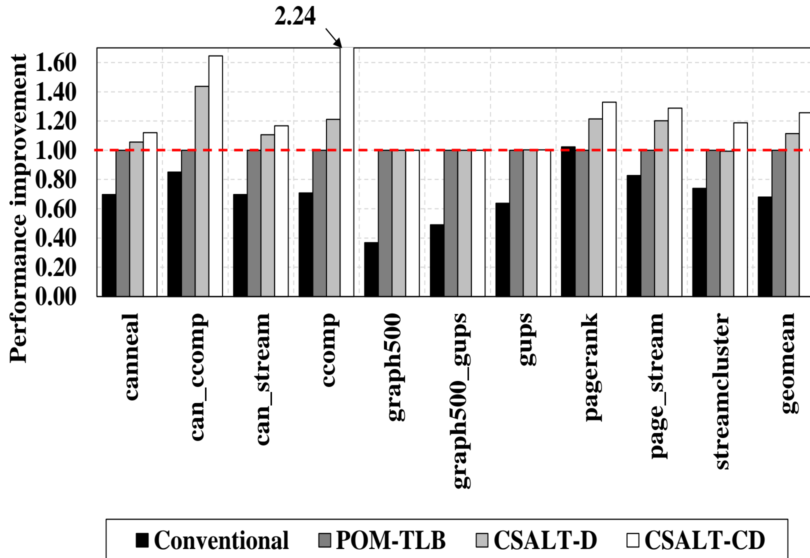


Figure 6.1: Performance Improvement of CSALT (normalized to POM-TLB)

with dynamic partitioning in L2, L3 data caches. CSALT-CD refers to proposed scheme with *Criticality-Weighted* dynamic partitioning in L2, L3 data caches.

6.2.3 CSALT Performance

We compare the performance (normalized IPC) of the baseline, POM-TLB, CSALT-D and CSALT-CD in this section. Figure 6.1 plots the performance of these schemes. Note that we have normalized the performance of all schemes using the POM-TLB. POM-TLB, CSALT-D and CSALT-CD all gain over the conventional system in every workload. The large shared TLB organization helps reduce expensive page walks and improves performance in the presence of context switches and high L2 TLB miss rates. This is confirmed

by Figure 6.2 which plots the reduction in page walks after the POM-TLB is added to the system. In the presence of context switches (that cause L2 TLB miss rates to go up by $6X$), the POM-TLB eliminates the vast majority of page walks, with average reduction of 97%. It may be emphasized that no prior work has explored the use of large L3 TLBs to mitigate the page walk overhead due to context switches.

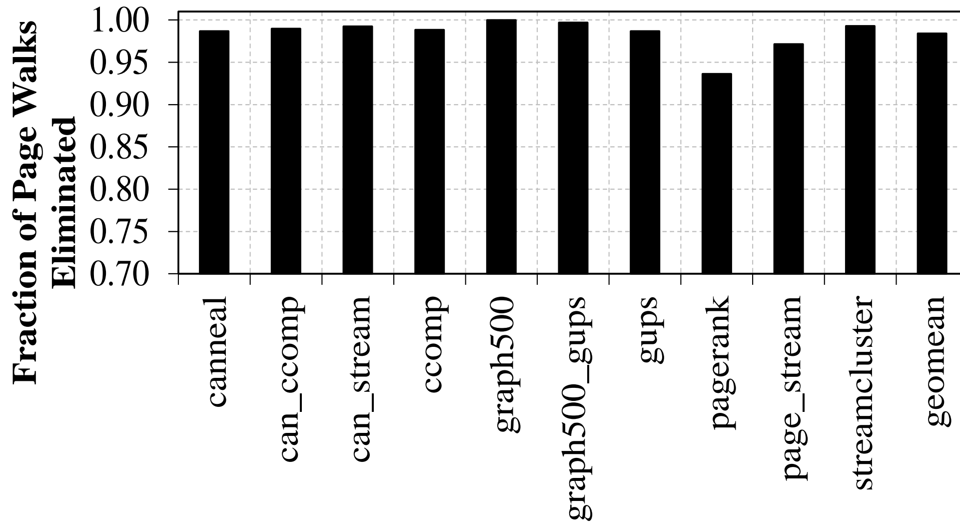


Figure 6.2: POM-TLB: Fraction of Page Walks Eliminated

Both CSALT-D and CSALT-CD outperform POM-TLB, with average performance improvements of 11% and 25% respectively. Both the dynamic schemes show steady improvements over POM-TLB highlighting the need for cache de-congestion on top of reducing page walks. In the *connected-component* workload[1], CSALT-CD improves performance by a factor of $2.2X$ over POM-

[1]When we refer to a single benchmark, we refer to two instances of the benchmark

TLB demonstrating the benefit of carefully balancing the shared cache space to TLB and data storage. In gups and graph500, just having a large L3 TLB improves performance significantly but then there is no additional improvement obtained by partitioning the caches.

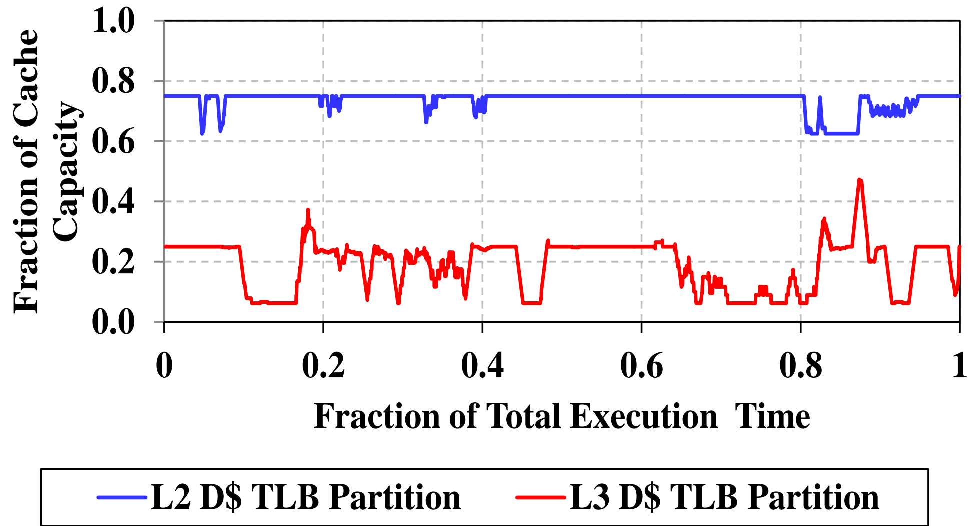


Figure 6.3: Fraction of TLB Allocation in Data Caches

In order to analyze how well our CSALT scheme works, we deep dive into one workload, *connected_component*. Figure 6.3 plots the fraction of L2 and L3 cache capacity allocated to TLB entries during the course of execution for *connected_component*. The TLB capacity allocation follows closely with the application behaviors. For example, the workload processes a list of active vertices (a segment of graph) in each iteration. Then, a new list of active vertices is generated based on the edge connections of vertices in the current

co-scheduled.

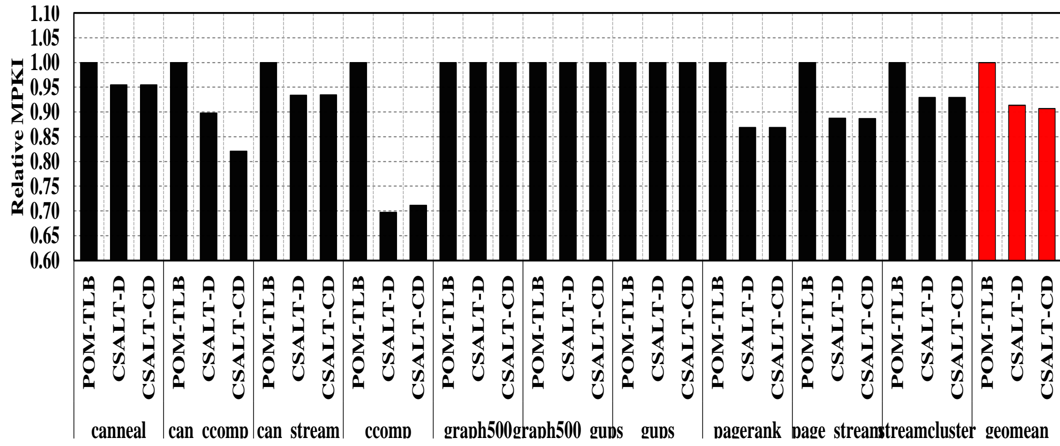


Figure 6.4: Relative L2 Data Cache MPKI over POM-TLB

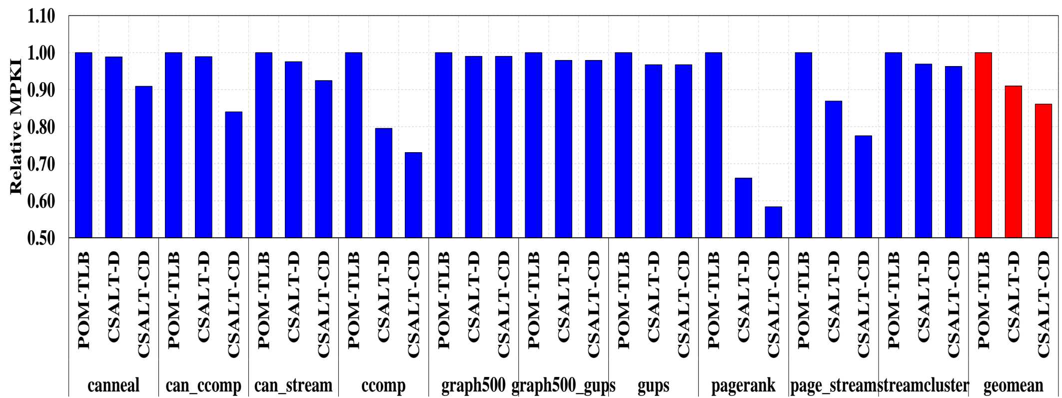


Figure 6.5: Relative L3 Data Cache MPKI over POM-TLB

list. Since vertices in the active list are placed in random number of pages, this workloads produces different levels of TLB pressure when a new list is generated. This is apparent that the L2 data cache, which is more performance critical, favors TLB entries in some execution phases. This phase is when the new list is generated. By dynamically assessing and weighing the data and TLB traffic, CSALT-CD is able to vary the proportion allocated to TLB, which satisfies the requirements of application. Interestingly, when more of L2 data cache capacity is allocated to TLB entries, we see a drop in L3 allocation for TLB entries. Since a larger L2 capacity for TLB entries reduces the number of TLB entry misses, the L3 data cache needs lesser capacity for TLB entries. Even though L2 and L3 data cache partitioning works independently, our stack distance profiler as well as performance estimators work cooperatively and optimize the overall system performance. The significant improvement in performance of CSALT over POM-TLB can be quantitatively explained by examining the reduction in the L2 and L3 MPKIs. Figures 6.4 and 6.5 plot the relative MPKIs of POM-TLB, CSALT-D and CSALT-CD in L2 and L3 data caches respectively (relative to POM-TLB MPKI). Both CSALT-D and CSALT-CD achieve MPKI reductions in both L2 and L3 data caches. In *connected-component*, both CSALT-D and CSALT-CD reduce MPKI of the L2 cache by as much as 30%. CSALT-CD achieves a reduction of 26% in the L3 MPKI as well. These reductions indicate that CSALT is successfully able to reduce cache misses by making use of the knowledge of the two streams of traffic.

These results also show the effectiveness of our *Criticality-Weighted* Dynamic partitioning. In systems subject to virtual machine context switches, since the L2 TLB miss rate goes up significantly, a careful management of cache capacity factoring in the TLB traffic becomes important. While TLB traffic is generally expected to be a small fraction in comparison to data traffic, our investigation shows that this is not always the case. In workloads with large working sets, frequent context switches can result in generating significant TLB traffic to the caches. CSALT-CD is able to handle this increased demand by judiciously allocating cache ways to TLB and data.

6.2.3.1 CSALT Performance in Native Systems

While CSALT is motivated by the problem of high translation overheads in context switched virtualized workloads, it is equally applicable to native workloads that suffer high translation overheads. Figure 6.6 shows that CSALT achieves an average performance improvement of 5% in native context-switched workloads with as much as 30% improvement in the *connectedcomponent* benchmark.

6.2.4 Comparison to Prior Works

Since CSALT uses a combination of an addressable TLB and a dynamic cache partitioning scheme, we compare its performance against two relevant existing schemes: i) Translation Storage Buffers (TSB, implemented in Sun Ultrasparc III, see [19]), and ii) DIP [85], a dynamic cache insertion policy

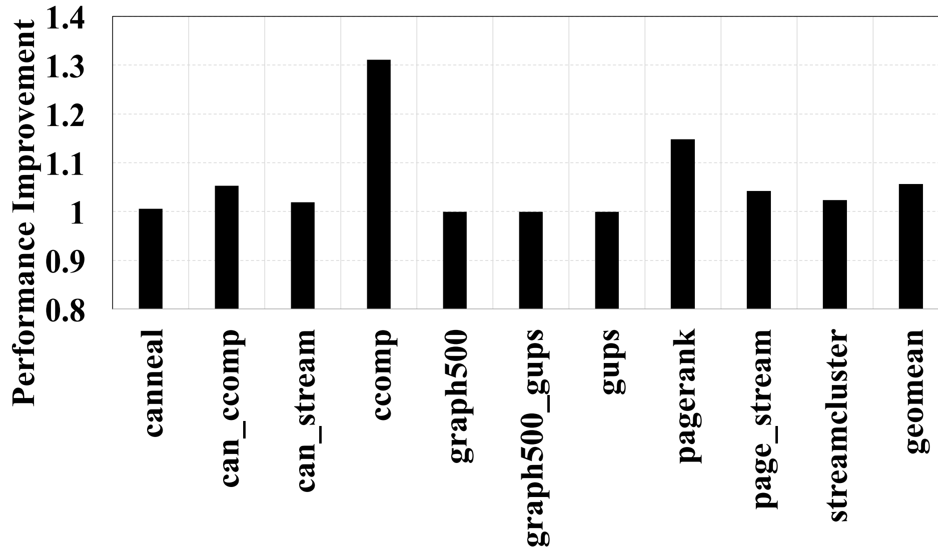


Figure 6.6: Performance Improvement of CSALT-CD in the native context

which we implemented on top of POM-TLB.

We chose TSB for comparison as it uses addressable software-managed buffers to hold translation entries. Like POM-TLB, TSB entries can be cached. However, unlike POM-TLB, the TSB organization requires multiple look-ups to perform guest-virtual to host-physical translation.

DIP is a cache insertion policy, which uses two competing cache insertion policies and selects the better one to reduce conflicts in order to improve cache performance. We chose DIP for comparison as we believed that the TLB entries may have different reuse characteristics that would be exploited by DIP (such as inserting such entries into cache sets at non-MRU positions in the recency stack). As DIP is not a page-walk reduction scheme, for a fair comparison, we implemented DIP on top of POM-TLB. By doing so, this

scheme leverages the benefits of POM-TLB (page walk reduction) while also incorporating a dynamic cache insertion policy that is implemented based on examining all of the incoming traffic (data + TLB) into the caches.

Figure 6.7 compares the performance of TSB, DIP and CSALT-CD on context-switched workloads. Clearly, CSALT-CD outperforms both TSB and DIP. Since TSB requires multiple cacheable accesses to perform guest-virtual to host-physical translation, it causes greater congestion in the shared caches. Since it has no cache-management scheme that is aware of the additional traffic caused by accesses to the software translation buffers, the TSB suffers from increased load on the cache, often evicting useful data to make room for translation buffer entries. This results in the TSB under-performing all other schemes (except in *connected-component*, where it performs superior to DIP, but inferior to CSALT-CD). It may also be noted that the TSB system organization can leverage CSALT cache partitioning schemes.

As such, DIP does not distinguish between data and TLB entries in the incoming traffic and is unable to exploit this distinction for cache management. As a result, DIP achieves nearly the same performance as that of POM-TLB. This is not surprising considering that we implemented DIP on top of POM-TLB. CSALT-CD, by virtue of its TLB-conscious cache allocation, leverage cache capacity much more effectively and as a result, performs 30% better than DIP, on average.

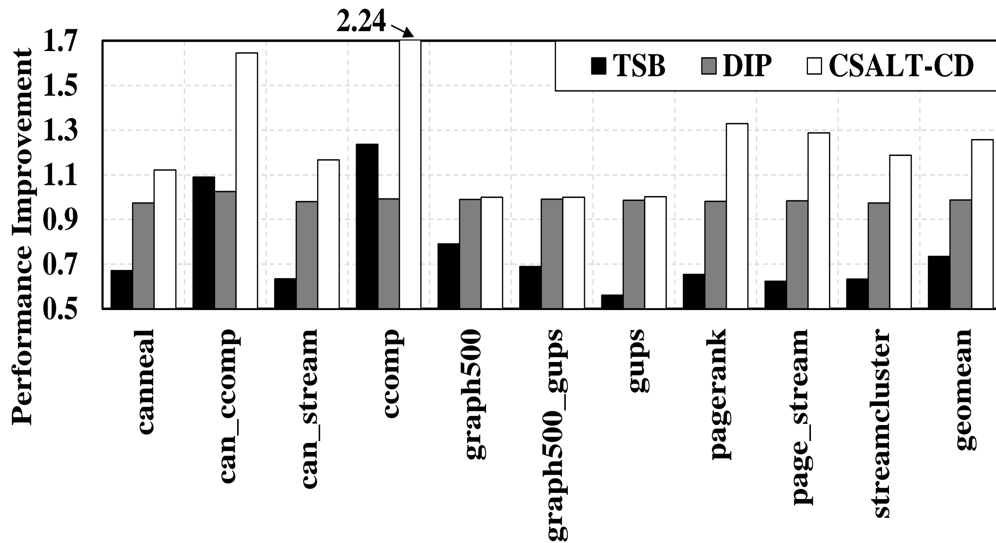


Figure 6.7: Performance Comparison of CSALT with Other Comparable Schemes

6.2.5 Sensitivity Studies

In this section, we vary some of our design parameters to see their performance effects.

Number of contexts sensitivity: The number of contexts that can run on a host system vary across different cloud services. Some host machines can choose to have more contexts running than others depending on the resource allocations. In order to simulate such effects, we vary the number of contexts that run on each core. We have used a default value of 2 contexts per core, but in this sensitivity analysis, we vary it to 1 context and 4 contexts per core. We present the results on how well CSALT is able to handle the increased resource pressure. Figure 6.8 shows the performance improvement results for varying number of contexts. The results are normalized to POM-TLB. As

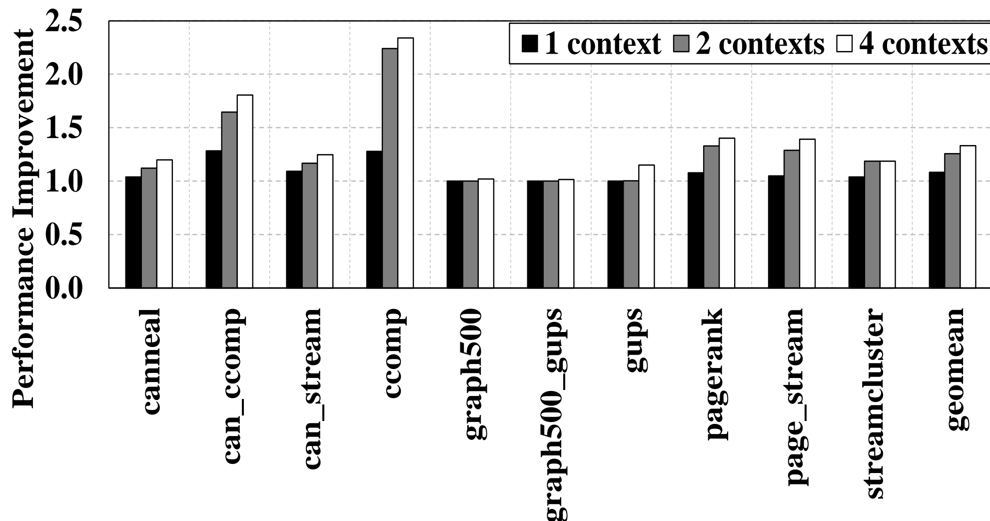


Figure 6.8: Performance of CSALT with Different Number of Contexts

expected, 1 context achieves the lowest performance improvement as there is no resource contention between multiple threads. Likewise, when we further increased the pressure by executing 4 contexts (doubled the default 2 context case), the performance increase is only 33%. This study shows that CSALT is very effective at withstanding increased system pressure by reducing the degree of contention in shared resources such as data caches.

Epoch length sensitivity: The dynamic partitioning decision is made in CSALT at regular time intervals, referred to as epochs. In CSALT design, the default epoch length was 256,000 accesses for both L2 and L3 data cache. The epoch length at which the partitioning decision is made determines how quickly our scheme reacts to changes in the application phases. We change this epoch length after experimental evaluation. Figure 6.9 shows the perfor-

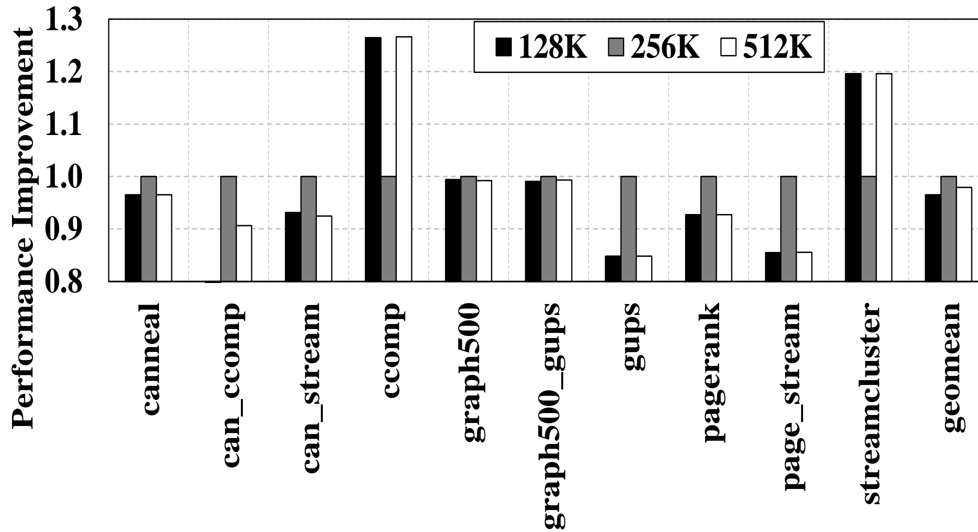


Figure 6.9: Performance of CSALT with Different Epoch Lengths

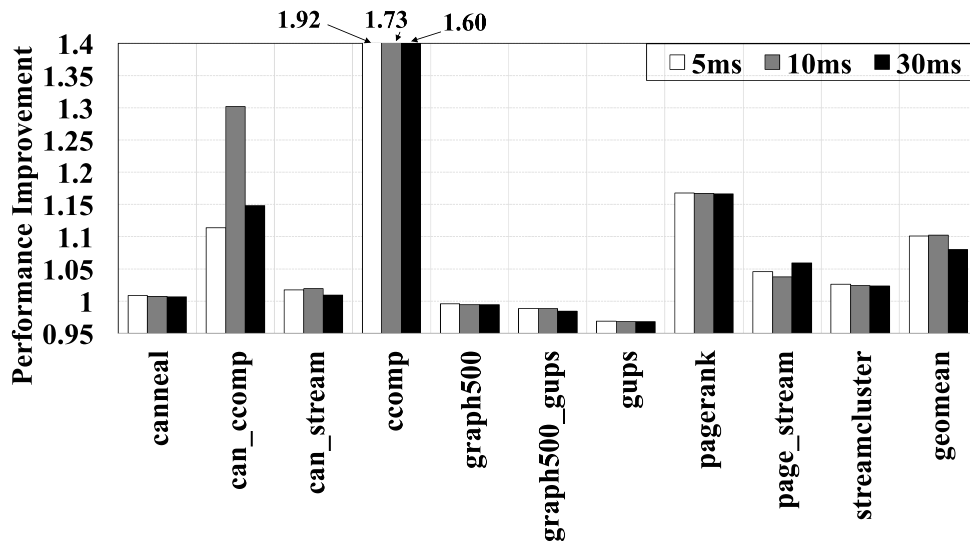


Figure 6.10: Performance of CSALT with Different Context Switch Intervals

mance improvement normalized to our default epoch length of 256K accesses when the epoch length, at which the dynamic partitioning decision is made,

is changed. In some cases such as *connected_component* and *streamcluster*, shorter and longer epoch length achieve higher performance improvement than our default case. This indicates that our default epoch length is not chosen well for these workloads as it results in making a partitioning decision based on non-representative regions of workloads. However, in all other workloads, our default is able to achieve the highest performance improvement. Therefore, for CSALT, we chose the default of 256K accesses as the epoch length.

Context Switch Interval Sensitivity: The rate of context switching affects the congestion/interference on data caches and results in eviction of useful data/TLB entries. Figure 6.10 plots the performance gain achieved by CSALT (relative to POM-TLB) at context-switch intervals of $5, 10$, and 30 ms. CSALT exhibits steady performance improvement at each of these intervals, with a slightly lower (8%) average improvement at 30 ms in comparison to 10 ms.

6.3 Translation Coherence using Addressable TLBs (TCAT)

6.3.1 Workloads

The main focus of this work is on address translation coherence, and thus, applications which do not exhibit a significant number of TLB shootdowns or significant amount of TLB shootdown penalties are not meaningful. Consequently, we chose a subset of PARSEC [78] applications and apache benchmark. As shown in the motivation, all these applications spend a considerable amount of time performing address translation coherence, especially in virtualized systems with higher overcommit ratios.

6.3.2 Simulation

We use a cycle accurate simulator that uses a heavily modified Ramulator. The simulator faithfully models the cache hierarchy of a Chip Multi-Processor (CMP). It models the Reorder Buffer (ROB) in which all instructions other than memory instructions are retired with a fixed schedule, as determined by the IPC of the benchmark on the real machine. We run each workload for 2 billion instructions. The front-end of our simulator uses the timed memory access traces collected from real system execution using the Pin tool. It also uses a trace of TLB shootdowns. Each entry in the TLB shootdown trace contains an approximation of instruction count at which the TLB shootdown occurred, the core which initiated the shootdown, and the number of cores that were affected by the shootdown. During playback, the simulator inspects the number of instructions that it has executed already, and injects the TLB shootdown at an appropriate time. The timing details of our simulator are summarized in Table 6.1. TLB shootdown penalty for each workload is computed using the average TLB shootdown latency and the average clock frequency during its execution. We use *turbostat* to measure the average clock frequency during execution of the benchmark. Table 6.3 lists the shootdown penalties for different benchmarks in the native setup and in the virtualized setup with different overcommit ratios. The performance improvement is calculated by using the ratio of improved IPC (geometric mean across all cores) over the baseline IPC (geometric mean across all cores), and thus, higher normalized performance improvement indicates a higher performing scheme.

Benchmark	Native	1:1	2:1	3:1	4:1
apache	7666	32219	660504	933364	1057079
dedup	22922	48660	155480	181460	180947
ferret	15683	530007	2206462	2492721	2583587
vips	9144	49167	368657	511626	696449

Table 6.3: TLB shootdown penalties (Cycles)

This section presents simulation results from a conventional native system employing TLB shootdowns, conventional virtualized systems employing kvmtlb, and various TCAT configurations. *Baseline-native* refers to the TLB shootdown baseline in native systems. *Baseline-kvmtlb* refers to the TLB shootdown baseline in virtualized systems. *TCAT-cotag* refers to the proposed scheme which employs POM-TLB co-tags to identify translations that need to be invalidated. *TCAT-cotagless* refers to the proposed scheme which employs computation of set-index to identify invalidation targets. *Ideal-sim* refers to the maximum performance that can be obtained without address translation coherence overheads in our simulator. *Ideal* refers to the maximum performance that can be obtained without address translation coherence overheads in the real system. *Ideal* performance is computed by assuming that the overhead of translation coherence in the real system, as measured in the motivational data, has been completely eliminated. We see that *Ideal-sim* tracks *Ideal* pretty reliably.

6.3.3 TCAT performance

We compare the performance (normalized IPC) of the baseline-kvmtlb, TCAT-cotag and TCAT-cotagless in this section. Figure 6.11 plots the per-

formance of these schemes for the 2:1 overcommit case. Note that we have normalized the performance of all schemes using the baseline-kvmtlb. Both TCAT-cotag and TCAT-cotagless gain over conventional system in every workload. Tracking guest page table updates by leveraging the POM-TLB cotag helps eliminate the expensive IPI overheads and busy wait overheads. This is confirmed in Figure 6.12 which plots the reduction in the busy-wait stall cycles after TCAT is added to the system. TCAT eliminates a large portion of the busy wait-cycles, with an average reduction of 99.52%. The performance difference from the ideal stems from the penalty of a Cache Line Flush and the penalty of a Read-Modify-Write (RMW). No prior work has explored the use of addressable TLBs to mitigate the address translation coherence overheads. Using addressable TLBs, we eliminate the imprecision associated with the invalidation in virtualized environments by allowing precise identification of the the invalidation target.

Both TCAT-cotag and TCAT-cotagless outperform the baseline, with an average performance improvement of 14% and 13% respectively. This highlights the need of a translation coherence scheme which allows precise identification of targets when the guest page table is updated in virtualized environments. In *apache*, by eliminating almost all of the 25% overhead of TLB shootdowns as seen in real systems, we obtain a 30% speedup. Similarly, in *dedup*, by eliminating almost all of the 20% overhead of TLB shootdowns, we obtain a 19% speedup. In *vips* and *ferret*, the shootdown overheads arise from individual shootdown latencies rather than the number of shootdowns. Due to

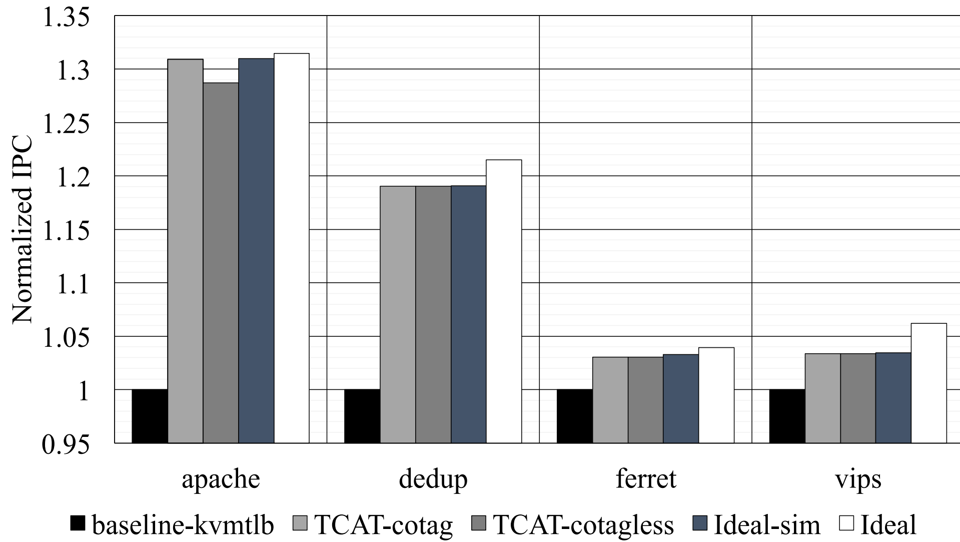


Figure 6.11: Performance improvement of TCAT

the small number of shootdowns, the penalty of RMW and Cache Line Flush is negligible. As a result, in *vips* and *ferret*, the stall cycles are reduced by almost 99.9%.

6.3.4 TCAT performance in Native Systems

While TCAT is motivated by the problem of imprecision of invalidation associated with virtualized environments, it is equally applicable to native environments. In native environments, the POM-TLB stores $VA - PA$ translations. Therefore, to enforce coherence upon update to a translation, the POM-TLB address corresponding to the virtual address (whose translation was

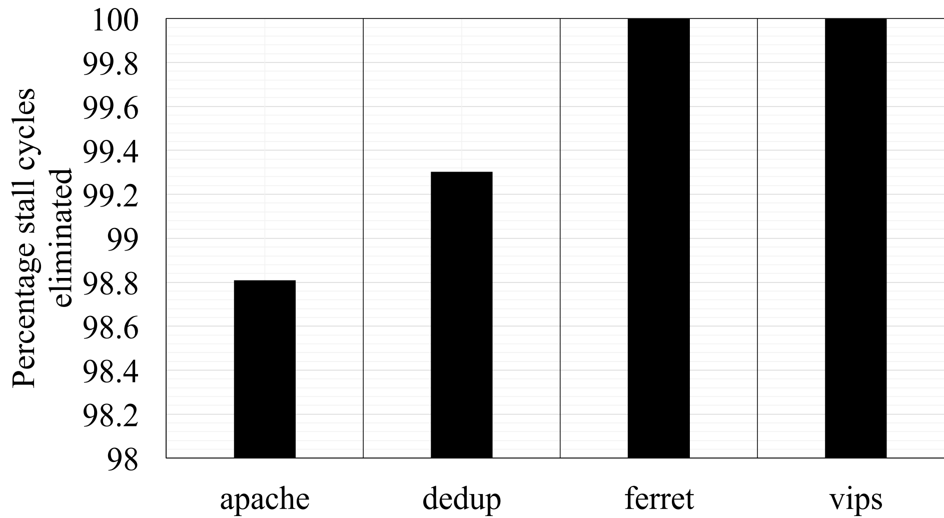


Figure 6.12: Percentage stall cycles eliminated

modified) can be written to, in a manner similar to virtualized environments. Figure 6.13 shows that TCAT achieves an average performance improvement of 1.4% (TCAT-cotag) and 1.2% (TCAT-cotagless) in native environments. The average performance gain is low because the percentage execution overhead of TLB shootdowns is comparatively low in native systems. However, with the advent of heterogeneous memories and page remapping between the slow and fast memories, shootdowns can consume a considerable portion of the application runtime [24]. Our approach will be able to eliminate all of those overheads.

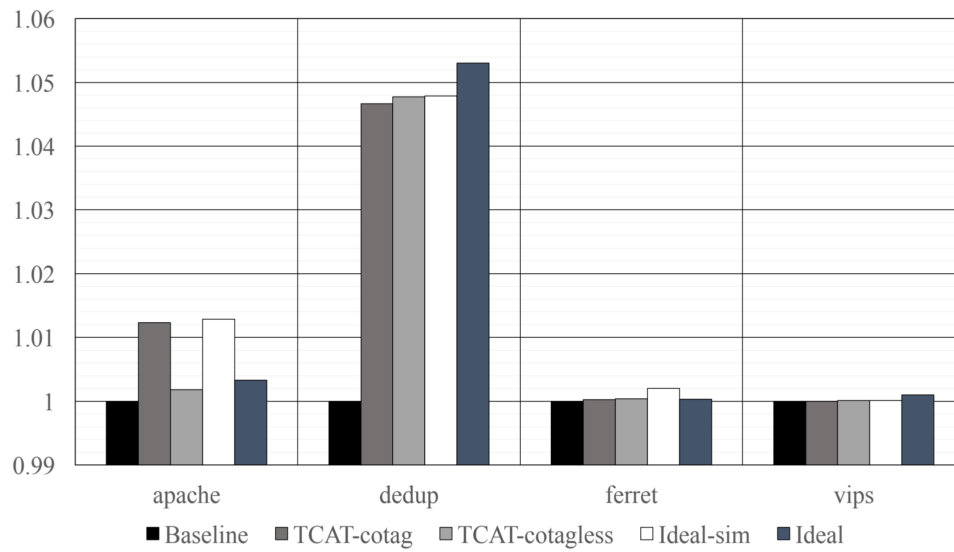


Figure 6.13: Performance improvement in Native systems

Chapter 7

Conclusion

In this work, we study the virtual memory overheads in virtualized environments, and propose two hardware solutions to mitigate those overheads.

First, we demonstrate the problem of TLB misses and cache contention caused by context-switching between virtual machines. We show that with just two contexts, L2 TLB MPKI goes up by a factor of 6X on average across a variety of large-footprint workloads. We presented CSALT - a dynamic partitioning scheme that adaptively partitions the L2-L3 data caches between data and TLB entries. CSALT achieves page walk reduction of over 97% by leveraging the large L3 TLB. By designing a TLB-aware dynamic cache management scheme in L2 and L3 data caches, CSALT is able to improve performance. CSALT-CD achieves a performance improvement of 85% on average over a conventional system with L1-L2 TLBs and 25% over the POM-TLB baseline. The partitioning techniques in CSALT are applicable for any designs that cache page table entries or TLB entries in L2-L3 caches.

Second, we demonstrate the overheads of TLB shootdowns in virtualized environments. We show that with higher overcommit ratios, the total time spent doing TLB shootdowns accounts for upto 50% of the application

runtime in a variety of multi-threaded workloads. We also show the inability of current hardware translation coherence schemes to track guest page table updates and achieve precise invalidations in virtualized environments. We presented TCAT - a hardware translation coherence scheme that overlays TLB coherence atop cache coherence and leverages the addressable POM-TLB to enable precise invalidations. TCAT eliminates almost all of the busy wait overheads (upto 99.8%) by leveraging the addressable TLB. It also enables precise invalidations on the slave core by allowing precise identification of the invalidation target. TCAT achieves a performance improvement of 13% on average over conventional virtualized systems with kvmtlb baseline.

Bibliography

[1] Amazon, "Amazon EC2 - Virtual Server Hosting," https://aws.amazon.com/ec2/.

[2] IBM, "SmartCloud Enterprise," https://www.ibm.com/cloud/.

[3] Microsoft, "Microsoft Azure," https://www.microsoft.com/en-us/cloud-platform/server-virtualization.

[4] Rackspace, "OPENSTACK - The Open Alternative To Cloud Lock-In," https://www.rackspace.com/en-us/cloud/openstack.

[5] HP, "HPE Cloud Solutions," https://www.hpe.com/us/en/solutions/cloud.html.

[6] H. Liu, "A Measurement Study of Server Utilization in Public Clouds," 2011, http://ieeexplore.ieee.org/document/6118751/media.

[7] K. Begnum, N. A. Lartey, and L. Xing, "Cloud-Oriented Virtual Machine Management with MLN," in *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*. Springer Berlin Heidelberg, 2009.

[8] A. W. Services, "High Performance Computing," https://aws.amazon.com/hpc/,.

[9] G. C. Platform, “Load Balancing and Scaling.” [Online]. Available: https://cloud.google.com

[10] Y. Mei, L. Liu, X. Pu, S. Sivathanu, and X. Dong, “Performance analysis of network I/O workloads in virtualized data centers,” *IEEE Trans. Services Computing*, vol. 6, no. 1, pp. 48–63, 2013. [Online]. Available: https://doi.org/10.1109/TSC.2011.36

[11] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, “Efficient resource provisioning in compute clouds via VM multiplexing,” in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC ’10. New York, NY, USA: ACM, 2010, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809052

[12] Z. A. Mann, “Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms,” *ACM Comput. Surv.*, vol. 48, no. 1, pp. 11:1–11:34, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2797211

[13] “ARM1136JF-S and ARM1136J-S,” http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/ddi0211k_arm1136_r1p5_trm.pdf.

[14] P. Jääskeläinen, P. Kellomäki, J. Takala, H. Kultala, and M. Lepistö, “Reducing context switch overhead with compiler-assisted threading,” in *Embedded and Ubiquitous Computing, 2008. EUC’08. IEEE/IFIP International Conference on*, vol. 2. IEEE, 2008, pp. 461–466.

[15] V. Vasudevan, D. G. Andersen, and M. Kaminsky, “The Case for VOS: The Vector Operating System,” in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS’13. Berkeley, CA, USA: USENIX Association, 2011, pp. 31–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991638

[16] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har’El, D. Marti, and V. Zolotarov, “Osv—Optimizing the Operating System for Virtual Machines,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[17] Intel. Intel(R) 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf

[18] ——, “5-Level Paging and 5-Level EPT,” 2016, https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf,.

[19] Oracle. Translation Storage Buffers. [Online]. Available: https://blogs.oracle.com/elowe/entry/translation_storage_buffers

[20] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, “Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB,” in *Computer Architecture, 2017 IEEE International*

Symposium on, ser. ISCA '17. ACM, 2017. [Online]. Available: http://lca.ece.utexas.edu/pubs/isca2017.pdf

[21] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 340–349.

[22] 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 *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[23] J. Ouyang, J. R. Lange, and H. Zheng, "Shoot4u: Using vmm assists to optimize tlb operations on preempted vcpus," *ACM SIGPLAN Notices*, vol. 51, no. 7, pp. 17–23, 2016.

[24] Z. Yan, J. Veselỳ, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 430–443.

[25] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *Proceedings of the Twenty-Third International Conference on Architectural*

Support for Programming Languages and Operating Systems. ACM, 2018, pp. 651–664.

[26] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *Proc. USENIX Ann. Conf*, 2017, pp. 27–39.

[27] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ACM, 2017, pp. 469–480.

[28] A. Kopytov, "Sysbench manual."

[29] C.-H. Yen, "SOLARIS OPERATING SYSTEM HARDWARE VIRTUALIZATION PRODUCT ARCHITECTURE," 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3F5AEF9CE2ABE7D1D7CC18DC5208A151?doi=10.1.1.110.9986&rep=rep1&type=pdf

[30] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346286

[31] J. Chang and G. S. Sohi, *Cooperative caching for chip multiprocessors*. IEEE Computer Society, 2006, vol. 34, no. 2.

[32] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors." in *HPCA*. IEEE Computer Society, 2011, pp. 62–63. [Online]. Available: http://dblp.uni-trier.de/db/conf/hpca/hpca2011.html#BhattacharjeeLM11

[33] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[34] X. Chang, H. Franke, Y. Ge, T. Liu, K. Wang, J. Xenidis, F. Chen, and Y. Zhang, "Improving virtualization in the presence of software managed translation lookaside buffers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 120–129.

[35] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A Mechanism for Speculative Address Translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 307–318. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000101

[36] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee, "Using TLB Speculation to Overcome Page Splintering in Virtual Machines," 2015.

[37] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes." in *USENIX Annual Technical Conference*, no. 98, 1998, pp. 91–104.

[38] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[39] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 210–222.

[40] G. B. Kandiraju and A. Sivasubramaniam, *Going the distance for TLB prefetching: an application-driven study*. IEEE Computer Society, 2002, vol. 30, no. 2.

[41] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 444–456.

[42] "bpoe8-eishi-arima," http://prof.ict.ac.cn/bpoe_8/wp-content/uploads/arima.pdf, (Accessed on 08/24/2017).

[43] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 381–391.

[44] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[45] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 430–441.

[46] C. L. Akanksha Jain, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," https://www.cs.utexas.edu/~lin/papers/isca16.pdf, 2016.

[47] D. S. Nathan Beckmann, "Maximizing Cache Performance Under Uncertainty," http://people.csail.mit.edu/sanchez/papers/2017.eva.hpca.pdf, 2017.

[48] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39.

Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.49

[49] R. Wang and L. Chen, "Futility Scaling: High-Associativity Cache Partitioning," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 356–367. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.46

[50] K. T. Sundararajan, T. M. Jones, and N. P. Topham, "Energy-efficient Cache Partitioning for Future CMPs," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 465–466. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370898

[51] C. Yu and P. Petrov, "Off-chip Memory Bandwidth Minimization Through Cache Partitioning for Multi-core Platforms," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 132–137. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837309

[52] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "Transactions on High-performance Embedded Architectures and Compilers III," P. Stenström, Ed. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Dynamic Cache Partitioning Based on the MLP of Cache Misses, pp.

3–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1980776.1980778

[53] W. Wang, P. Mishra, and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 948–953. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024935

[54] R. Kandemir, Mahmut a nd Prabhakar, M. Karakoy, and Y. Zhang, "Multilayer Cache Partitioning for Multiprogram Workloads," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 130–141. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033345.2033360

[55] P.-H. Wang, C.-H. Li, and C.-L. Yang, "Latency Sensitivity-based Cache Partitioning for Heterogeneous Multi-core Architecture," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898036

[56] W. Hasenplaugh, P. S. Ahuja, A. Jaleel, S. Steely Jr., and J. Emer, "The Gradient-based Cache Partitioning Algorithm," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 44:1–44:21, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086723

[57] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000073

[58] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware Partitioning and Replacement for Last-level Caches in Phase Change Main Memory Systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 53:1–53:21, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086732

[59] A. Pan and V. S. Pai, "Imbalanced Cache Partitioning for Balanced Data-parallel Programs," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 297–309. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540734

[60] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *ACM International Conference on Supercomputing 25th Anniversary Volume.* New York, NY, USA: ACM, 2014, pp. 402–412. [Online]. Available: http://doi.acm.org/10.1145/2591635.2667188

[61] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09.

New York, NY, USA: ACM, 2009, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555778

[62] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 428–439. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337208

[63] A. Bhattacharjee, "Preserving virtual memory by mitigating the address translation wall," *IEEE Micro*, vol. 37, no. 5, pp. 6–10, 2017.

[64] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on.* IEEE, 2015, pp. 188–200.

[65] "Intel(R) 64 and IA-32 Architectures Optimization Reference Manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[66] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970. [Online]. Available: http://dx.doi.org/10.1147/sj.92.0078

[67] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *Parallel Processing, 2009. ICPP'09. International Conference on.* IEEE, 2009, pp. 18–25.

[68] K. Kędzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-lru replacement policies," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–12.

[69] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[70] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[71] "Arm information center," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/BABJDBHI.html, (Accessed on 05/01/2018).

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[73] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1109/LCA.2015.2414456

[74] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[75] I. Habib, "Virtualization with KVM," *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217

[76] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, vol. 9, 2010.

[77] Intel, "Intel(R) Virtualization Technology," http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[78] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New

York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[79] The Graph500 List. [Online]. Available: Graph500:http://www.graph500.org/

[80] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in *Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012, pp. 31–46.

[81] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," 1999.

[82] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830773

[83] C. Li, C. Ding, and K. Shen, "Quantifying the Cost of Context Switch," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281702

[84] F. Liu and Y. Solihin, "Understanding the Behavior and Implications of Context Switch Misses," *ACM Trans. Archit. Code Optim.*,

vol. 7, no. 4, pp. 21:1–21:28, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1880043.1880048

[85] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250709

[86] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 383–394.

[87] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 48–59.

[88] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 237–248.

[89] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 223–234.

[90] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 457–468.

[91] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 258–269.

[92] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 558–567.

[93] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 435–448.

[94] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 1–12.

[95] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 307–318.

[96] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 297–308.

[97] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown, "Design tradeoffs for software-managed tlbs," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 3, pp. 175–205, 1994.

[98] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: exceeding the best of nested and shadow paging," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 707–718.

[99] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "Csalt: context switch aware large tlb," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 449–462.

[100] S. Phadke and S. Narayanasamy, "Mlp aware heterogeneous memory system," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[101] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 457–468.

[102] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-memory address translation," in *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. Ieee, 2017, pp. 303–317.

[103] Intel, "Skylake (microarchitecture)," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Skylake_(microarchitecture)

[104] Micron, "High-Performance On-Package Memory," https://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory.

[105] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[106] T. S. Karkhanis and J. E. Smith, "A First-Order Superscalar Processor Model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 338–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006729

[107] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for*

Programming Languages and Operating Systems, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346286

[108] T. W. Barr, A. L. Cox, and S. Rixner, “Translation Caching: Skip, Don’T Walk (the Page Table),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

[109] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, “Die Stacking (3D) Microarchitecture,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–479. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.18

[110] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[111] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth." in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE, 2014, pp. 38–50. [Online]. Available: http://dblp.uni-trier.de/db/conf/micro/micro2014.html#GulurMMG14

[112] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design." in *MICRO*. IEEE Computer Society, 2012, pp. 235–246. [Online]. Available: http://dblp.uni-trier.de/db/conf/micro/micro2012.html#QureshiL12

[113] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 404–415.

[114] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM Cache," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, 2014, pp. 25–37. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.51

[115] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 454–464. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155673

[116] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.56

[117] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628089

[118] SPEC, "SPEC CPU 2006," 2006.

[119] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB Preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 117–127. [Online]. Available: http://doi.acm.org/10.1145/339647.339666

[120] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: architectural and operating system support for reducing the impact of address translation," in *Proceedings of the 24th ACM International Conference on Supercomputing.* ACM, 2010, pp. 159–168.

[121] S. Srikantaiah and M. Kandemir, "Synergistic TLBs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2010, pp. 313–324.

[122] M. D. Hill, S. I. Kong, D. A. Patterson, and M. Talluri, "Tradeoffs in supporting two page sizes," 1993.

[123] SUN, "The SPARC Architecture Manual," http://www.sparc.org/standards/SPARCV9.pdf.

[124] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on.* IEEE, 2012, pp. 258–269.

[125] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on.* IEEE, 2014, pp. 558–567.

[126] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for performance analysis of VMware vSphere under Tier-1 applications," *VMware Technical Journal*, vol. 2, no. 1, 2013.

[127] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Proceedings of 23rd Hot Chips*, 2011.

[128] JEDEC, "High Bandwidth Memory (HBM) DRAM Gen 2 (JESD235A)," 2016. [Online]. Available: https://www.jedec.org

[129] Intel, "Knights Landing." [Online]. Available: http://www.realworldtech.com/knights-landing-details

[130] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked DRAM," in *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, 2015, pp. 188–200. [Online]. Available: http://dx.doi.org/10.1109/PACT.2015.30

[131] ARM, "Cortex-A72 Processor."

[132] Intel, "4th Generation Intel Core Processor."

[133] G. Gerzon, "Intel® virtualization technology processor virtualization extensions and intel® trusted execution technology," 2007.

[134] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th annual*

international symposium on Microarchitecture. IEEE Computer Society Press, 1995, pp. 93–103.

[135] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, "Using dead blocks as a virtual victim cache," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 489–500.

[136] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 430–441.

[137] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium® 4 processor," in *Intel Technology Journal*. Citeseer, 2001.

[138] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, Mar. 1995.

[139] G. H. Loh, N. Jayasena, K. Mcgrath, M. O'Connor, S. Reinhardt, and J. Chung, "Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems," in *the 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*, 2012.

[140] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," *IEEE Micro*, vol. 31, no. 1, 2011.

[141] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 198–210.

[142] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *Computer Design, 2007. ICCD 2007. 25th International Conference on*, 2007, pp. 55–62.

[143] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, 2012.

[144] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014.

[145] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014.

[146] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010.

[147] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[148] Oracle, "Translation Storage Buffer."

[149] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. ACM, 2004, p. 22.

[150] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 423–432.

[151] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on.* IEEE, 2013, pp. 639–650.

[152] L. Lamport, *LaTeX: A Document Preparation System*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1994.

[153] Oracle. (2016) Oracle SPARC Architecture 2015 Draft D1.0.9. [Online]. Available: https://community.oracle.com/docs/DOC-1005258

[154] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John, "A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems," in *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2010.5416654

[155] "AMD Nested Paging," http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[156] "Understanding the linux kernel 3rd edition," http://www.johnchukwuma.com/training/UnderstandingTheLinuxKernel3rdEdition.pdf.

[157] K. Kedzierski, M. Moretó, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-lru replacement policies." in

IPDPS. IEEE, 2010, pp. 1–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2010.html#KedzierskiMCV10