ATTC (@*C*) : Addressable-TLB based Translation Coherence

Harsh Gugale
University of Texas at Austin
Austin, TX
harsh.gugale@utexas.edu

Nagendra Gulur
University of North Texas
Denton, TX
nagendra.gulur@unt.edu

Yashwant Marathe
University of Texas at Austin
Austin, TX
marathe.yashwant@gmail.com

Lizy K. John
University of Texas at Austin
Austin, TX
ljohn@ece.utexas.edu

ABSTRACT

Heterogeneous memory systems are getting popular, however they face significant challenges from translation coherence overheads from page remappings. Translation coherence, which is typically implemented in software, can consume up to 50% of the runtime for some applications in virtualized platforms. In this paper, we propose *ATTC* – Addressable TLB-based Translation Coherence, a hardware translation coherence scheme which eliminates almost all of the overheads associated with software-based coherence mechanisms, and overcomes the challenges in existing hardware schemes. Unlike other proposals (HATRIC, UNITD) that require on-chip TLB tags to enforce coherence and are capable of tracking only the last level page table entries of either the guest or host page tables, *ATTC* tracks changes to both guest and host page tables without requiring any additional metadata in L1, L2 TLBs. *ATTC* enforces a "point of coherence" uniformly for both guest and host page table updates using an addressable TLB (*ATLB*) in the DRAM akin to the one in [41]. An inverse mapping table (*INVTBL* - present in DRAM) that maps host physical pages to *ATLB* locations helps to precisely track translations. We study the proposed *ATTC* scheme in detail for an emerging hybrid memory organization (a mix of DRAM and NVM) and show that *ATTC* practically eliminates all translation coherence overheads, yielding an average improvement of 35.7% over a baseline software coherence scheme in virtualized environment and 7.4% over the hardware HATRIC scheme.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Virtual memory**.

KEYWORDS

Virtualization, Translation coherence, TLB Shootdown

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414653

ACM Reference Format:
Harsh Gugale, Nagendra Gulur, Yashwant Marathe, and Lizy K. John. 2020. ATTC (@*C*) : Addressable-TLB based Translation Coherence. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414653

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies such as PCM [23], STTRAM [15], ReRAM [2] and 3DX Point [19] are promising enhancements to conventional DRAM for handling the needs of emerging workloads. Due to inherent drawbacks such as larger access times or low endurance, these emerging memories often cannot be used as a complete replacement. Researchers have shown that a hybrid memory subsystem comprising of a conventional DRAM coupled with a high capacity NVM or high bandwidth die-stacked DRAMs can provide the required performance, power and reliability benefits [37, 38]. Intel, AMD and other vendors are releasing systems with 3D Xpoint, High Bandwidth Memory (HBM), HBM2, Hybrid Memory Cube (HMC) etc.

In heterogeneous memory systems, pages are migrated frequently between the fast and slow memory devices, to match the demands of applications, necessitating operating systems to remap pages. During page remapping events, consistency has to be maintained across private TLBs in various cores for correctness; i.e. no core should access a stale memory translation. The OS must inform all the processors in a Chip Multi Processor (CMP) about the remapping and associated translation modification. 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 other cores (referred to as *victim cores*) which might potentially hold the affected address translations in their private TLBs. 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 victim cores jump to an interrupt handler and perform invalidations in their own private TLBs, a process typically referred to as *TLB shootdown*.

With increasing deployment of virtual machines for cloud services and server applications, maintaining efficient virtualization is important. An important challenge in virtualized systems is address translation and translation coherence. Translations in virtualized environments involve two levels of redirection (i) guest virtual address (gVA) to guest physical address (gPA), managed by the

guest OS, and done by the guest page tables; and (ii) guest physical address to system (host) physical address (sPA) - managed by the hypervisor (also called the Virtual Machine Manager or VMM) and done by the nested page tables. In many modern processors, both these page tables use a 4-level radix tree and modification to any set of mappings is expensive. In such systems, TLB shootdowns occur between virtual processors (vCPUs) instead of the physical cores. Virtualized systems suffer from overheads caused by virtual IPIs. The vCPUs generate virtual IPIs instead of physical IPIs. Delivery of virtual IPIs requires hypervisor (VMM) intervention to notify target vCPUs. Further, the target vCPUs may not be scheduled to immediately service virtual interrupts (or worse, may get pre-empted while processing virtual interrupts), delaying acknowledgment[29].

Ideally translation coherence mechanisms should invalidate only the TLB entries corresponding to the pages that were remapped, but in many systems precise invalidation cannot be accomplished. Current VMMs do not track the gVA of pages used by the guest. Since modern processors only permit invalidations of individual TLB entries when the gVA (for guest pages) is known, when the VMM updates the nested page table, translation structures are completely flushed [44]. Repopulating the flushed 2-dimensional page tables are very expensive. Additionally, VMMs track address translations at VM-granularity - VMMs track the subset of physical cores that a virtual CPU (vCPU) runs on, but do not track the subset of cores that a process on a vCPU runs on. As a result, in addition to the list of victim vCPUs approximated by the guest OS, the list of victim cores is approximated by the VMM. Combined with the flushing of translation structures upon a nested page table update, these approximations result in frequent needless evictions of unrelated translations [44].

TLB shootdown activity has been observed to be a significant bottleneck in prior studies [4, 6, 12, 25, 33, 40, 44] and is also confirmed by our own experiments. Our experiments with a heterogeneous memory hierarchy similar to recent work (SITE [6]) where a fast DRAM acts as a first level memory and an NVM is the second level memory illustrate that TLB shootdowns result in cycle counts increasing by 13.1% in the native case and 37% in the virtualized case (see Figure 1).

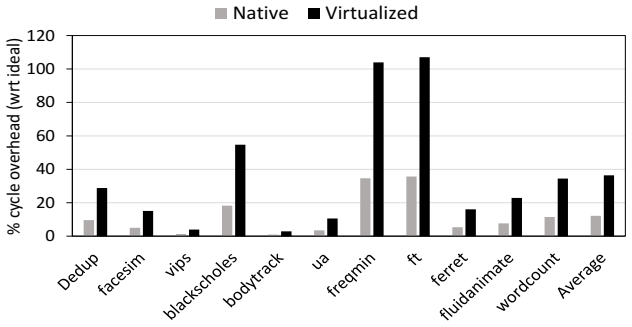


Figure 1: ***% cycle overhead introduced by TLB shootdowns with respect to an ideal case where shootdowns take zero cycles***

There have been industry and academic efforts to reduce TLB shootdown overheads. On the industry side, in x86-64 and ARM architectures, there are instructions to invalidate specific TLB entries (without flushing the entire TLB). Operating systems such as Linux have improved in how they track coherence targets [42] and as a result, reduce TLB shootdown overheads. However, many of these enhancements are limited to native execution [44] and TLB coherence remains a significant bottleneck in virtualized systems.

On the academic side, there have been various techniques proposed in both hardware and software to eliminate these overheads. Schemes like Lazy Translation Coherence (LATR) [22], ABIS [4], DIDI [42], UNITD [40], HATRIC [44] and SITE [6] aim to reduce the number of IPIs or eliminate them altogether. LATR solved several problems of software schemes by eliminating the requirement of acknowledgements from victim cores but is still imprecise. While DIDI eliminate expensive IPIs and the associated kernel interventions, they lack support for hardware virtualization, and do not eliminate the long waits on the initiator core. UNITD augments each TLB entry with the physical address of the last-level PTE for precise invalidations, but - it too does not address virtualized environments. Furthermore, as explained by Yan et al. [44], it is not straightforward to extend UNITD for virtualization.

HATRIC [44] was proposed by Yan et. al. to address shootdown overheads in virtualized environments. By augmenting TLB entries with co-tags constructed from the physical address of the last-level page table entry (PTE) of the host page table, HATRIC demonstrated that up to 30% performance improvement can be obtained in KVM-based virtualized systems, by providing hardware based translation coherence. However, HATRIC tracks changes to only the host page table, but not the guest page table. HATRIC can be made to track either the guest or the host, but tracking both at the same time involves additional on-chip hardware and is not straightforward [1]. Addressing only the host-initiated page table changes is insufficient, since many workloads have significant guest-level translation changes . As an example, in the dedup benchmark, our measurements on a state-of-the-art Intel Whiskey Lake system (Intel® Core™ i7-8565u CPU) indicated that guest side page table changes alone take upto 28% of the execution time (see Figure 2). Thus, it is highly desirable to offer efficient hardware support for both guest and host-initiated changes on both the initiator and victim cores.

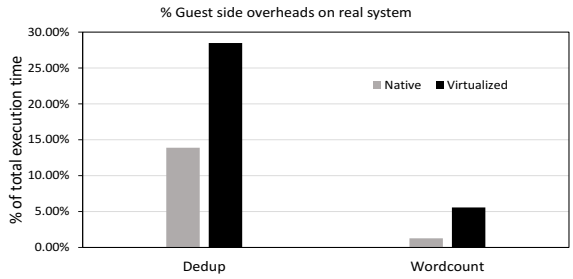


Figure 2: ***Guest-side shootdown overheads for dedup and wordcount running on Intel Whiskey Lake system***

We propose *ATTC* (pronounced *@C*), **a hardware translation coherence mechanism that goes beyond HATRIC and other works on TLB coherence, creating an efficient solution for dealing with guest and host level page table changes**. *ATTC* overlays translation coherence atop cache coherence. It leverages a very large addressable TLB (abbreviated *ATLB*) to enable precise

	UNITD	DiDi	SITE	HATRIC	ATTC
Handles Native System	Yes	Yes	Yes	Yes	Yes
Handles Virtualized System	No	No	No	Either Guest *or* Host	Both Guest *And* Host simultaneously
Hardware addition	32 bits per TLB entry (on-chip)	Requires Shared TLB	32 bits per TLB entry (on-chip)	16 bits per TLB entry (on-chip)	Inverse Table added but it is in DRAM. Minimal changes in on-chip control logic.

Table 1: Comparison of *ATTC* with other hardware coherence schemes. LATR, ABIS, Shoot4U are software schemes and do not provide virtualization support.

invalidations in virtualized environments. *ATTC* tracks changes to both guest and host page tables by tying each guest page table translation to a single address in the *ATLB* and maintaining an inverse mapping table (*INVTBL*) to precisely identify guest virtual pages that are mapped to a system physical page. By ensuring that both guest and host page table changes are reflected as *writes* to a common address space, *ATTC* achieves hardware translation coherence of virtualized environments without resorting to software IPIs or shootdowns. Unlike prior works (UNITD, HATRIC), *ATTC* does not introduce additional tag bits to area and latency sensitive TLBs. Table 1 briefly compares the main features of prior art and the proposed *ATTC* scheme. Succinctly, we make the following contributions:

- Propose the *ATTC* organization for hardware-based TLB coherence achieving coherence under both host and guest initiated page table changes, with minimal on-chip hardware changes. The major hardware addition, *INVTBL* is resident in the DRAM, where the area-overheads are more manageable.
- Demonstrate that *ATTC* eliminates nearly all of the IPI overheads by leveraging the addressable *ATLB* coupled with the proposed *INVTBL* to track both guest and host page table changes, resulting in approximately 40% improvement over kvmtlb-based schemes and 12% improvement over state-of-the-art hardware scheme, HATRIC.

This paper is structured as follows. In Section 2, we present our design. Section 3 presents the experimental setup and Section 4 presents the results. Section 5 presents a discussion of various design considerations. Section 6 offers a comparison with related works. We conclude with Section 7.

2 ATTC (@*C*) DESIGN

In this section, we describe our hardware translation coherence scheme *ATTC* which addresses both guest and system page table updates by overlaying TLB coherence atop cache coherence. It enforces coherence in hardware while requiring light changes to the guest and hypervisor code. Section 2.1 provides an overview of *ATTC* . The hardware additions for coherence-invalidation support at private TLBs and the L2 cache are described in Section 2.2. The major hardware addition to support host updates, the inverse table is described in Section 2.3. Section 2.4 describes the operation for handling guest and host updates and Section 2.5 describes the operating system support needed.

2.1 Overview

Figure 3 shows the high-level system organization of a multi-core system that integrates *ATTC*. As can be seen *ATTC* requires an addressable TLB (*ATLB*) and an inverse mapping table (*INVTBL*), both resident in main memory. Blocks outlined with the thick green border in Figure 3 are *ATTC* additions over the baseline system.

2.1.1 System-wide Point of Coherence. Prior hardware-based approaches (UNITD [40] and HATRIC [44]) have proposed to integrate TLBs into the cache coherence mechanism. However, the lack of a consistent system-wide (across guest OS and hypervisor) coherence-triggering mechanism limited their applicability to tracking either only the guest or only the hypervisor-initiated changes, but not both. *ATTC* addresses this fundamental issue using an addressable TLB (*ATLB*). Prior work [41] established the feasibility of using a large shared L3 TLB placed in addressable main memory for achieving very high TLB hit rates as well as low lookup latency. *ATTC* exploits such an addressable structure to design a TLB coherence mechanism that supports both guest and host-initiated page table changes. In prior work [41], the large TLB was made possible by die-stacked DRAM. But the mechanism would work even if the large TLB is in DRAM itself.

The *ATLB* is organized as a set associative cache. Each set is 64B in size, holding four 16B TLB entries, each entry holding one translation $gVA \rightarrow sPA$[1]. An *ATLB* comprising N sets is assigned an address range of $64 \times N$ bytes. The guest virtual address (gVA) of the L2 TLB miss is converted to an *ATLB* set index by extracting $log_2(N)$ bits of the gVA (after ignoring the page offset bits). These bits are XORed with VM_ID bits to ensure that identical guest addresses across VMs map to different *ATLB* sets. The memory address of the set that the gVA maps to is given by:

$$ADDR_{ATLB}(gVA) = BASE\_ADDR_{ATLB} + (((gVA >> 12) \oplus (VM\_ID << 14)) \& ((1 << log_2(N)) - 1)) * 64 \quad (1)$$

Since, the *ATLB* assigns a memory address to every set in the large TLB, this enables caching of these TLB sets in data caches in a manner identical to the caching of normal data. Any updates to an *ATLB* set (by writing to the respective *ATLB* set address) will trigger cache coherence actions *without requiring any special hardware support*. Snoop-based or directory-based hardware coherence schemes will ensure that changes to a memory location will result in appropriate cache-side actions to remove all stale copies of that

[1]Like other TLBs and caches, each *ATLB* entry also holds other bits such as the valid bit, dirty bit, page permissions, process ID, VM ID, recency bits etc.

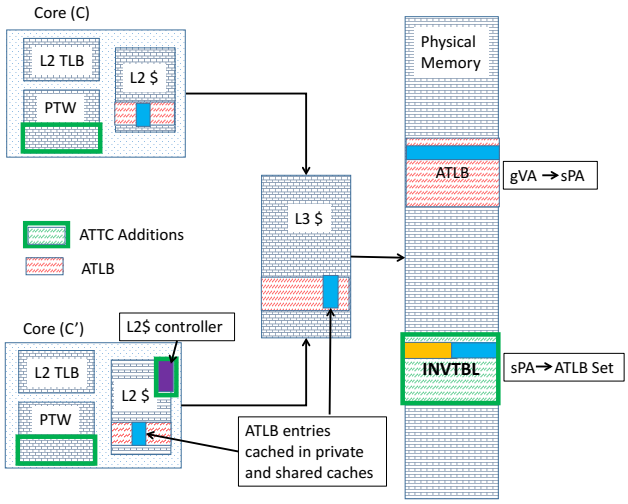


Figure 3: ATTC *Overview*

location, ensure a single owner for dirty data, and so on. Given the mapping scheme in equation 1, it is also possible to extract partial gVA from the *ATLB* set address ($ADDR_{ATLB}$) specified in the coherence message. Forwarding these messages to the private TLBs could be used to invalidate stale TLB entry through partial tag match.

This motivates the use of the *ATLB* as the *point of coherence*: *if all modifications to address translations are reflected as writes to the affected* ATLB *locations, the hardware cache coherence protocols could be used to invalidate stale translation entries from local TLBs.* It is important to emphasize that both guest-side and host-side translation changes should be reflected onto the *ATLB* writes in order to achieve an entirely hardware-based coherence scheme.

2.2 Support for ATTC in Private TLBs

As mentioned in the previous section, to enforce *ATLB* as point of coherence, invalidation messages must be sent to private TLBs with *ATLB* set address. *ATTC* uses a simple hardware extension to monitor and implement coherence-invalidation actions in private TLBs. Whenever a private TLB receives a coherence message with an *ATLB* set address, the TLB looks up the guest virtual tags of TLB entries that match the set index bits of the *ATLB* address and computes their corresponding *ATLB* set locations. If there is a match between the computed location and the incoming *ATLB* address in the coherence message, then the TLB entry is invalidated. This is similar to what happens with *invlpg* instruction in the x86 architecture. Unlike the mechanisms proposed in UNITD [40] and HATRIC [44], *ATTC* does not add any storage overhead to private TLBs.

For example, suppose the coherence message specifies an address A in a 16MB *ATLB*. From A, the 18-bit set index $ATLB[17:0]$ is extracted. Given the Skylake L1-dTLB and L2-sTLB, TLB indexing schemes [18], these TLB indices can be exactly reconstructed from the *ATLB* set index. The L1-dTLB index is the least four bits $ATLB[3:0]$. The L2-sTLB index, computed using the XOR-7 scheme (refer [18]), requires the least fourteen bits $ATLB[13:0]$. This reconstruction works because of our addressing scheme (refer Equation 1) that ensures that $ATLB[13:0] = gVA[25:12]$. If a different architecture used a different set of gVA bits for indexing into L1, L2 TLBs, then the *ATLB* set-indexing could be suitably modified to use the same gVA bits as part of *ATLB* indexing. Thus, *ATTC* can precisely identify the L1, L2 TLB sets to inspect and partially match the tag bits to identify TLB entries to invalidate. We show in our evaluation that proportion of false positives due to partial tag matching is extremely low.

2.3 The Inverse Mapping Table – *INVTBL*

Host-initiated translation changes require hardware and hypervisor support to track the affected guest addresses whenever the hypervisor makes a gPA to sPA translation change. As the *ATLB* is looked up using gVA, the host-side changes must be reflected in terms of the guest-virtual addresses whose 2D translations are affected. To address this, we propose an inverse mapping table (*INVTBL*) that is implemented as a large in-memory addressable cache.

2.3.1 INVTBL Organization. Figure 4 shows the organization of the *INVTBL* and its interaction with the *ATLB*. The *INVTBL* is an in-memory addressable cache of mappings. Each entry maps a host physical page to the corresponding entry in the *ATLB* that points to it. The organization of the *INVTBL* is similar to the *ATLB*. It is organized as a large set-associative cache indexed using sPA. In each set, inverse mapping entries are stored, each 4B entry containing the host physical address tag bits and the *ATLB* set index. We deliberately choose to store *ATLB* set indices rather than guest virtual addresses for two reasons: one, *ATLB* cache set indices require far fewer bits of storage as compared to storing guest virtual tags (18 bits versus 52), and two, directly storing the *ATLB* index avoids a conversion step for converting the guest virtual address into the *ATLB* set index. Like the *ATLB*, it is assumed that the *INVTBL* occupies a range of host physical address space: $[BASE_{INVTBL}, BASE_{INVTBL} + RANGE_{INVTBL}]$.

2.3.2 INVTBL Interaction with the ATLB. The *INVTBL* and the *ATLB* work in unison to track both guest-level and host-level page table updates. The *INVTBL* is installed with inverse mappings whenever 2D hardware page table walks occur. The hardware Page Table Walk (PTW) is augmented to add the inverse mapping whenever a page walk completes. Thus, the inverse table entry associated with the system (host) physical page contains a "pointer" to the *ATLB* set which contains the guest virtual page whose translation needs to be invalidated if the host makes a page table change that affects the host physical page of interest. Whenever the host makes a page table change that affects a set of host physical pages, then the host OS writes to the affected *INVTBL* entries to invalidate them. The hypervisor also uses the *ATLB* pointers stored in the affected *INVTBL* entries to issue invalidations to these *ATLB* entries.

For this scheme to work correctly, the system must guarantee one invariant: for every valid *ATLB* entry, there must be a pointer in the *INVTBL* that points to it. This invariant guarantees that whenever the host makes a change to the page table, it will be able to follow the pointer to the *ATLB* to invalidate the corresponding translation entry. Ensuring that this invariant is met requires that whenever the *INVTBL* suffers an eviction, then the *ATLB* entry that the evicted entry is pointing to is also evicted.

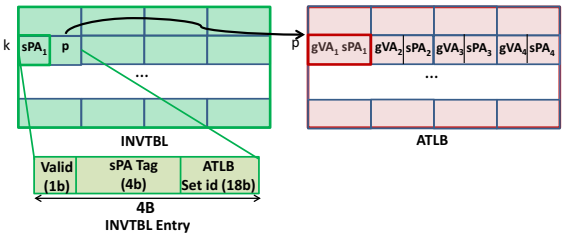


Figure 4: *INVTBL* ***and*** **ATLB** ***Interaction***

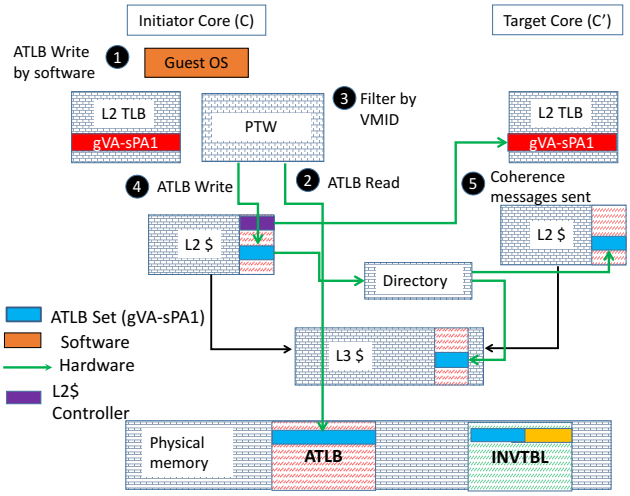


Figure 5: ATTC ***operation for guest side updates***

2.4 Handling Guest and Host Updates

The *ATLB* acts as point of coherence *if* all the address translation changes can be reflected as updates into the *ATLB*. In this section, we describe the hardware and software support needed to ensure this.

2.4.1 Handling Guest PTE Changes. Guest-initiated translation changes are straightforward to implement: the guest OS must invalidate the affected *ATLB* locations. Invalidating an entry will trigger coherence actions across the caches and upper-level TLBs resulting in removing all stale copies of the invalidated translations. A subsequent access to the affected gVA will result in a 2D page walk restoring the translation into the *ATLB*.

Consider a guest page table mapping T_g which maps guest virtual address gVA 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 sPA_1. In virtualized environments, TLBs store the mapping $gVA - sPA_1$, which we denote by T_{tlb}. When the guest OS initiates a guest page table update on Core C, it updates the $gVA - gPA_1$ mapping to $gVA - gPA_2$. This update has resulted in the TLB entry $gVA - sPA_1$ becoming stale. Suppose another core $C' \neq C$ has the translation $gVA - sPA_1$ resident on its private TLBs. Further, let us assume that this translation is also resident in private data caches of both C and C'. Translation coherence necessitates that all of these stale translations be invalidated.

Figure 5 provides an overview of the sequence of events that take place in order to achieve hardware coherence when a guest initiates page table changes. Below, we refer to the steps (indicated by numbered black circles) shown in the diagram.

- **Step 1: Guest OS issues a write to the *ATLB***: The OS calculates the *ATLB* address A corresponding to the affected gVA using Equation 1. Any mapping from gVA to any of one of the numerous host physical addresses must reside at this fixed location A owing to the property of the *ATLB* addressing scheme.
- **Step 2: Hardware Page Table Walker (PTW) issues a Read from address** A: The PTW intercepts the OS-issued write to A and issues a read of the *ATLB* set held at A.
- **Step 3: PTW filters read contents by** VM_ID: The PTW uses the VM_ID of the guest OS that issued the write in Step 1 to invalidate (clear the valid bit of) only those entries in the *ATLB* that belong to the issuing OS. This step enforces security: a guest OS can not accidentally or intentionally write to *ATLB* entries that do not belong to it.
- **Step 4: PTW issues a Write to address** A: The PTW performs a write to address A with the modified contents (with valid bit of entries that belong to the issuing guest cleared).
- **Step 5: TLB Coherence Actions are Taken**: The write to A issued by the PTW triggers coherence. As shown in Figure 5, the write to A is a hit in the private L2 data cache of the initiator core. By writing to A, any cached copies of A residing in other data caches are evicted through normal data cache coherence. Further, as A falls in the *ATLB* address range, the L2 cache controller generates coherence messages to private TLBs and they invalidate affected entries. See figure 6 for a summary of steps highlighting the interaction between the OS and the underlying hardware. More details of these steps are discussed in Section 2.5.

2.4.2 Handling Host PTE Changes. When the host initiates a page table update (due to page migration, page compaction etc), then a current host page table entry $gPA - sPA$ becomes stale[2]. The hypervisor writes to the *INVTBL* using the host physical page (sPA) to invalidate it. Following the "pointers" stored at that location (in our design, each *INVTBL* entry stores up to four pointers), the hardware PTW performs writes to the *ATLB* addresses to invalidate any translations that map to the now-stale sPA. As before, either imprecise (all contents in the matching set) or precise (entries that map to given sPA) invalidations could be carried out. In our evaluation, we used precise invalidations. As discussed above in Section 2.4.1, these writes to *ATLB* locations initiate coherence updates across the cache and TLB hierarchies.

2.5 OS Support

ATTC proposes to use the *ATLB* uniformly as the point of coherence for both guest and host-initiated page table updates. In order for this scheme to work, the guest OS and hypervisor must ensure the following:

(1) Whenever the guest OS makes a (guest) page table modification, it must perform a write to the corresponding *ATLB* location *even if the location does not presently contain the affected translation.* This is necessary to initiate coherence operations across the chip,

[2]More than one entry may become stale, and the inverse-table based *ATLB* updates must be performed for every modified host page table entry.

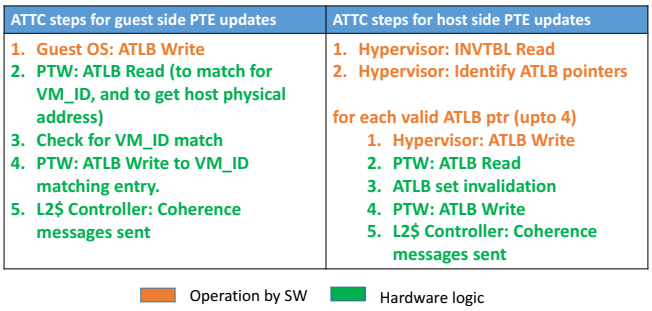


Figure 6: ATTC *Steps for guest/hypervisor initiated Changes*

as private TLBs and data caches may still be holding the affected translation. The *ATLB* write performed by the guest OS is intercepted by the hardware PTW to issue a DRAM read, filter the read response by VM_ID of the issuing guest, and to clear the matching entry. The column on the left in Figure 6 outlines this. This is a total of one DRAM read and one DRAM write.

(2) Whenever the host OS makes a (host) page table modification, it must invalidate the *ATLB* locations pointed to by the entry in the *INVTBL* that correspond to the old system physical address. In addition, the *INVTBL* entry itself must also be invalidated. The host performs a write to the *INVTBL* entry at the old host physical address. This triggers the hardware PTW to access and invalidate stale translations from the *ATLB*. The PTW uses the *ATLB* pointers to compute their addresses and to write to them. As the *INVTBL* maintains up to 4 pointers (in our 4-way associative design), up to 4 DRAM writes occur. These actions are outlined in the right hand column of Figure 6.

There is little performance overhead associated with these operating system initiated writes. Guest- and host-initiated page table changes result in up to 5 DRAM accesses. Thus, these operations take a few hundred CPU cycles to complete. As compared to software-IPI messages that initiate TLB shootdowns, these software operations are $10X$ to $100X$ cheaper.

In all scenarios, it may be observed that the resulting TLB invalidations are invisible to victim cores. There are no IPIs generated, and there is no need for software to track the occupancy of translation entries in virtual or physical cores.

2.6 Hardware Overhead

ATTC introduces the *INVTBL* as a DRAM-based cache of pointers to *ATLB* locations. Its organization and storage capacity are similar to the *ATLB* (in the range 4MB – 32MB, depending on anticipated workloads' footprints). Being DRAM-based, this storage overhead is negligible[3].

On the logic side, *ATTC* adds *ATLB* and *INVTBL* range checks to detect writes to *ATLB* locations and to issue invalidate messages to private TLBs. Private TLBs add logic to receive these messages and to use the *ATLB* set address from these messages to invalidate matching TLB entries. *ATTC* also extends the hardware page table walker to intercept writes to the *ATLB* in order to ensure VM-specific read-modify-writes.

[3] A 32MB *INVTBL* adds less than 0.2% overhead in a system with a physical memory of 16GB.

Processor	Values
Num Cores/Freq	8/4 GHz
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, 4 way
L1 TLB (2MB)	32 entry, 9 cycles, 4 way
L2 Unified TLB	1536 entry, 17 cycles, 12 way
DDR	Values
Type	DDR4-2133
tCAS-tRCD-tRP	14-14-14
Replacement policy	CLOCK Replacement
ATLB	16 MB, 4 way, 16B block
INV-TBL	4 MB, 4 way, 4B block
NVM (only in sim)	Values
Latency	2x DRAM
Migration Threshold	10
Shootdown Parameters	Values
Initiator Latency (Virtualized)	48000 cycles
Victim latency (Virtualized)	10300 cycles
Initiator Latency (Native)	16200 cycles
Victim latency (Native)	3500 cycles

Table 2: Simulation Parameters

3 EVALUATION

We evaluate *ATTC*ś performance on a cycle-accurate simulation framework that models the operation of a multicore Skylake processor. The simulation methodology, configuration parameters and workload details are provided below.

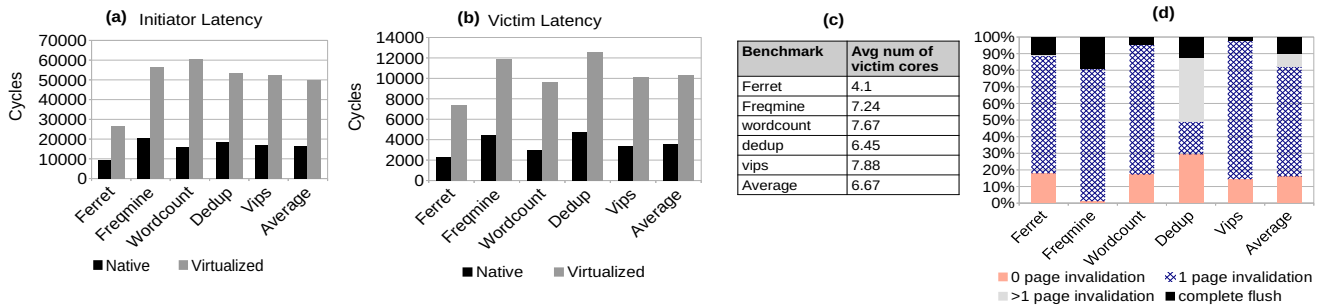
3.1 Simulation Methodology

In order to evaluate a scheme such as ours, we need a simulator model to capture system-level effects. Shootdown events are costly but infrequent and inherently require exploration over large timescales. Therefore, we use a combination of a detailed simulator coupled with memory access traces that also capture events pertaining to TLB accesses (TLB invalidations, page migrations etc.). This type of trace/ftrace/PIN-based simulation we use is similar to the approach in recent works involving TLBs/shootdowns [6, 27, 34, 41, 44].

We use a cycle accurate simulator modeling memory instructions in detail. The simulator models the cache and TLB hierarchy of a Chip MultiProcessor (CMP) using the parameters as described in Table 2. Simulation parameters for eg. average page walk cost, TLB shootdown cost (detailed discussion below) etc. are obtained from real world measurements using ftrace and performance counters. In order to obtain these we use QEMU [10] 3.1 as virtualization platform with KVM [21] support. The system also has Intel VT-x with support for Extended Page Tables. The guest system is Ubuntu 19.04 (Linux kernel v5.0.0.17) .

3.2 Calibrating TLB Shootdown Parameters based on Intel System

The shootdown latencies listed in Table 2 are determined with real system measurements on a state-of-the-art Intel processor with and without virtualization. To collect this data, we use the Linux



Benchmark	Avg num of victim cores
Ferret	4.1
Freqmine	7.24
wordcount	7.67
dedup	6.45
vips	7.88
Average	6.67

Figure 7: (a) Latency on initiator core (b) Latency on victim core (c) Avg. number of victim cores per shootdown (d) Page invalidations per shootdown

ftrace utility which provides a breakdown of the time spent in each shootdown activity on both the initiator and the victim core.

Figure 7(a) plots the average cycles spent by the initiator core in a TLB shootdown in native and virtualized modes. This includes the cost of IPI delivery, TLB invalidation, and the busy-wait state. On average, we observe that in the native case the initiator core spends 16.2K cycles servicing a TLB shootdown. In the virtualized case, the cost per shootdown on the initiator core is seen to be 48K cycles (about 3.2x compared to native). These average cycle numbers are used as the simulation parameters in Table 2 for the initiator overheads.

Figure 7(b) plots overheads for the victim core. This includes the cost of remote TLB invalidation and entry into and exit from the interrupt service routine. For the native case, victim core spends, on average, 3.5K cycles. For the virtualized case, we observe a latency of 10.3K cycles, approximately 2.9x compared to the native case. This experiment yielded the values for the simulation parameters in Table 2 for the victim core.

To further validate the chosen simulator overhead parameters, a few more measurements were done on the Intel SkyLake platform. We measured the average number of victim cores that receive and handle IPI requests in each of the workloads. The table in Figure 7(c) suggests that the workloads in which TLB shootdowns target a larger number of victim cores tend to show greater initiator-side overheads. This is not surprising as the initiator must wait for all the victim cores to acknowledge processing of the invalidation requests. Initially we were concerned about the difference in overhead cycles for Ferret in Figures 7(a) and 7(b) relative to other benchmarks. However as shown in Figure 7(c), in Ferret, an IPI goes to less than 4.1 cores on average as against $>$ 6.6 across other benchmarks. Naturally the overheads are less if fewer cores are receiving the IPIs.

We also validated our assumptions on imprecise invalidations using experiments on the SkyLake processor. Figure 7 (d) plots the fractions of different types of shootdown events that Linux issues depending on the number of PTE entries that are modified. Single page invalidations are the most common action ($>$ 66% on average), followed by other types such as multipage invalidation or complete TLB flushes. Zero page invalidations represent inefficiencies while issuing IPIs due to imprecise tracking of TLB entries. It is evident that there is potential to obtain performance benefits by optimizing the cost of the single-page and multi-page invalidations.

3.3 Simulation Details

A modified PIN tool is used to generate a trace of timed memory accesses to feed into the simulator. In addition to this, the PIN tool also generates a shootdown trace using the Linux pagemap, capturing page remaps and changes in permissions. The shootdown trace is used by the simulator to account for coherence related overheads. The first 10B instructions of each benchmark are skipped while collecting the PIN trace. Not all page remaps/permission changes will result in individual shootdown events (owing to several linux optimizations such as batching, lazy TLB shootdown etc.). Injecting shootdown activity (such as stalling the initiator/victim core) in the simulator by looking at the perf trace of the workload results in accurate simulations. Note that these Linux optimizations are accounted for in the overheads reported in Figure 1 and are also accounted for in our simulations.

For the front-end, the simulator models the Reorder Buffer (ROB) in which all instructions other than memory instructions are retired with a fixed schedule. Memory access instructions are simulated in detail to account for cache and TLB performance. *ATLB* and *INVTBL* are assumed to reside on DRAM (their system parameters are in Table 2). For shootdowns initiated due to host page migrations, we conservatively use a penalty of 500 cycles to perform the *INVTBL* and *ATLB* accesses, assuming that all the four pointers to *ATLB* entries (given that our *INVTBL* associativity is four) are followed and respective entries invalidated. For shootdowns due to guest changes, we simulate the required writes to *ATLB* addresses incurring DRAM cycles. In *ATTC*, since each page table remap is implemented as a guest OS write to the *ATLB* our simulator initiates coherence activity at each remap event seen in the input trace. The overall time taken by *ATTC* for coherence due to guest- or host-initiated page table changes is estimated as a sum of *ATLB* read + write and *INVTBL* lookup and write. This is a maximum of 5 DRAM accesses (when the host initiates page table updates) amounting to an estimated 500 cycles on average (sensitivity study on this parameter presented later).

The performance improvement is calculated by using normalized IPC (calculated as sum of instructions executed across cores divided by total number of clock ticks across cores) over the baseline IPC.

Name	Suite	input	threads	Data Footprint (MB)
Dedup	PARSEC	native	12	105
Facesim	PARSEC	native	16	99
Vips	PARSEC	native	12	32
Blackscholes	PARSEC	native	8	204
Bodytrack	PARSEC	native	12	11
Freqmine	PARSEC	native	12	636
Ferret	PARSEC	native	12	93
Fluidanimate	PARSEC	native	12	375
FT	NAS	Size C	12	583
UA	NAS	Size C	12	428
Wordcount	Phoenix	word_100MB	8	85

Table 3: Workload details

3.4 Memory Migration Policy

We investigate the benefits of *ATTC* on an emerging hybrid memory organization where a fast DRAM acts as a first level of physical memory and a large NVM acts as the second level. We use a paging policy similar to recent work [6]. All pages are initially assumed to reside in NVM. The OS (or the hypervisor) decides to migrate a page to DRAM when memory accesses to that page exceed a threshold value. Since, the guest and the hypervisor may decide to migrate pages independently, we use a static proportion of guest vs host migration shootdowns of 25% - 75% of all shootdowns for our analysis (given that most of the migrations are caused by the hypervisor updates [44]) and perform a sensitivity study over this ratio in section 4. DRAM capacity is assumed to be 95% of the memory footprint of a workload in order to create a realistic mildly congested scenario that causes migrations not just for cold misses in the DRAM but also for conflict misses that occur during the course of execution.

3.5 Workloads

We chose a subset of PARSEC [13], Phoenix [39] and NAS [7] parallel benchmark suites for our study. Workload details can be found in Table 3. Our workloads include programs with high guest side shootdown overheads (for eg. wordcount), those with high migration induced host side shootdown overheads (for eg. FT, Freqmin, blackscholes) and some which show overheads for both the cases (for eg. dedup). This allows us to draw a meaningful conclusion and bring out the effectiveness of *ATTC* in various scenarios.

4 RESULTS

This section presents simulation results for *ATTC* for a diverse set of benchmarks. We compare *ATTC* with three configurations: POMTLB [41], HATRIC [44] and *Ideal*. POMTLB refers to a system with a large addressable TLB as in [41] which has the addressable TLB component of *ATTC* but still incurs IPI based shootdown. *Ideal* refers to an ideal configuration wherein TLB shootdowns occur in zero time. Among other state-of-the-art hardware based-coherence schemes, only HATRIC supports virtualization and hence we compare *ATTC*'s performance only against HATRIC. All results are normalized with respect to the conventional kvmtlb baseline.

Performance in Virtualized Systems: Figure 8 shows the simulation results for the virtualized environment. *ATTC* outperforms the kvmtlb baseline, with average performance improvements of

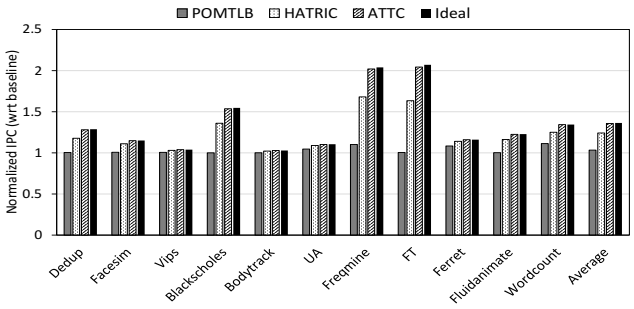


Figure 8: ATTC *Performance Improvement in virtualized system - (kvmtlb baseline)*

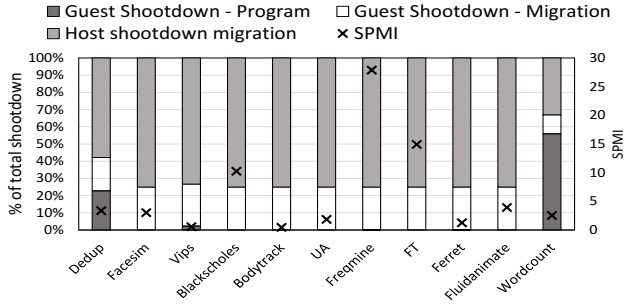


Figure 9: *Types of shootdowns incurred during simulation, and Shootdowns per million instructions (SPMI)*

35.7%. It also outperforms HATRIC by approximately 7.4%. Compared to HATRIC, *ATTC*'s gains are a result of eliminating guest-side shootdown overheads. In HATRIC, both application-inherent shootdowns (due to data sharing between threads) and guest-side page migrations result in IPIs. By avoiding these IPIs, *ATTC* eliminates more than 99% of the TLB shootdown overheads and remains within 1% of the ideal IPC.

It may also be observed that *ATTC* improves performance well above the gains achieved by an addressable TLB alone (as exemplified in the POMTLB case). While POMTLB brings about 4% improvement by reducing page walk latencies, it does not reduce/eliminate IPIs or their shootdown overheads.

Performance differences between the benchmarks can be explained using the breakdown of different types of shootdowns executed by the simulator and the shootdown intensity expressed in shootdowns per million instructions (SPMI) in Figure 9. Higher SPMI values indicate higher opportunities and are seen to result in higher improvements in both *ATTC*and HATRIC. For eg. Freqmine which accesses pages from a lot of different locations and has high off-chip BW requirements, has very high SPMI but *ATTC* is able to save more than 99% of the overhead and improve performance by 2.04x. *ATTC* is equally effective for both guest and host initiated shootdowns, whereas HATRIC can only handle one. For eg., wordcount which has many guest updates due to parallel map reduce operation has 60% guest shootdowns, and *ATTC* yields 7.4% improvement over HATRIC.

False invalidation due to partial tag match: As mentioned in section 2.2, writes to ATLB set address initiates invalidation requests to be issued to *both* data caches and local TLBs of the victim core. Stale *ATLB* entries are precisely invalidated from data caches (through cache coherencec protocols) but for local TLBs,

invalidations are performed by a partial tag match because of the reconstruction of partial gVA from *ATLB* set index. Our measurements indicate that occurrence of false invalidations is almost zero. A major reason for this is the large size of the *ATLB*. For eg. given our indexing scheme in equation 1, using a 16MB *ATLB* enables us to extract and compare 14 bits of partial gVA tag in local TLBs. For many workloads this granularity is enough to avoid any false invalidations.

***ATTC* Latency Sensitivity:** In our experiments, we used a penalty of 5 DRAM Read-Modify-Write operations for host-initiated DRAM accesses to the $INVTBL$ and the *ATLB*. In highly congested or bandwidth-limited systems, DRAM access latencies may be worse. It is also possible that the *ATLB* or the $INVTBL$ is organized differently (block size, associativity) requiring additional DRAM accesses to read/write these cache sets. To cover this large design space, we perform a sensitivity study of *ATTC* performance as invalidation penalties are set to 1000, 2000, and 5000 cycles (instead of the 500 cycles assumed in prior simulations). Results in Figure 10 indicate that *ATTC* continues to provide substantial improvements even at higher invalidation penalties. Intuitively, this is expected as the baseline IPI scheme incurs a very large overhead (over 48, 000 cycles on average, per shootdown as reported in Figure 7) and even the costlier *ATTC* memory accesses do not degrade its performance by a significant amount as compared to the baseline.

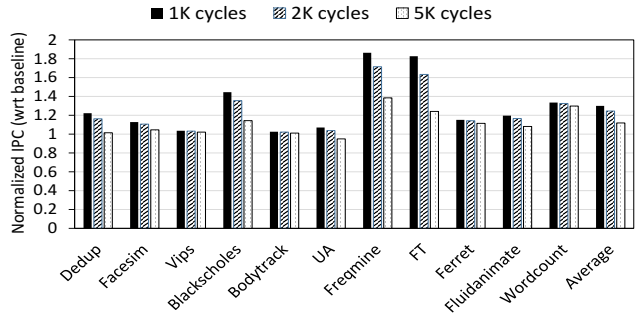


Figure 10: $INVTBL$ ***and* ATLB *access latency sensitivity.*** *We observe that* ATTC *continues to provide gains.*

***ATLB* Hit Ratio:** Next, we analyze the hit ratio for *ATLB* .It is important that *ATLB* exhibit high hit ratio since every miss in *ATLB* is accompanied by an insertion of (sPA – ATLB set ptr) inverse mapping in $INVTBL$, in addition to insertion of (gVA – sPA) mapping in *ATLB* by the Page Table Walker. High hit ratio would ensure that maintenance of *ATLB* and $INVTBL$ is not very costly. Intuitively, given the large size of *ATLB* translation requests should rarely incur misses after the initial application startup time. Our experiments confirm this intuition. As shown in figure 11, we observe that hit ratio is high, 93.1% on average. Several of the workloads show > 99% hit ratio. The low hit ratio for blackscholes was seen to be from cold start misses.

Performance in systems with only DRAM: Although, *ATTC* is motivated by prohibitively high shootdown overheads in hybrid memory systems, it also provides performance gains in systems with only one level of main memory i.e DRAM (no NVM). Figure 12 plots *ATTC* improvements for system with only DRAM. For this configuration, programs only incur guest-side shootdown overheads,

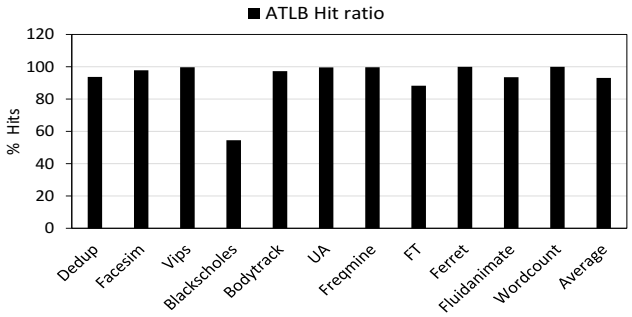


Figure 11: *ATLB* Hit ratio

due to page remaps by the guest OS. *ATTC* is able to effectively reduce this cost and improve performance by 5% on average.

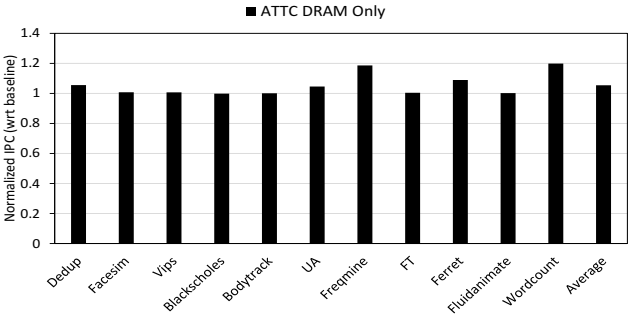


Figure 12: *ATTC* performance in systems with DRAM only (no NVM)

Performance in Native Systems: While *ATTC* is primarily motivated by the overheads of shootdowns in virtualized systems, it is equally applicable to native environments. In native environments, the *ATLB* stores $VA-PA$ translations. It should be noted that for native environments, there is no need of a separate $INVTBL$ and coherence may be enforced by simply using the *ATLB*. Figure 13 shows the performance of *ATTC* in native systems. We compare *ATTC*with three configurations: POMTLB, HATRIC/UNITD and Ideal. *ATTC* improves performance by 12.8% on average, over the baseline and performs similar to contemporary schemes - HATRIC/UNITD.

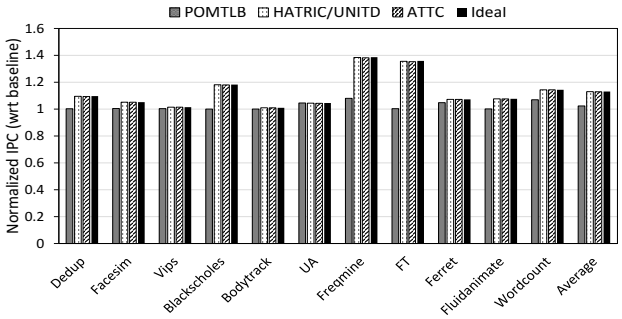


Figure 13: ATTC *Performance improvement - Native*

Sensitivity to varying proportion of guest side shootdowns: *ATTC* and HATRIC both support virtualization, however, fundamentally they differ in their support for simultaneous guest and host updates. We perform a sensitivity study varying the percentage of total guest side migration-induced shootdowns below. Figure 14

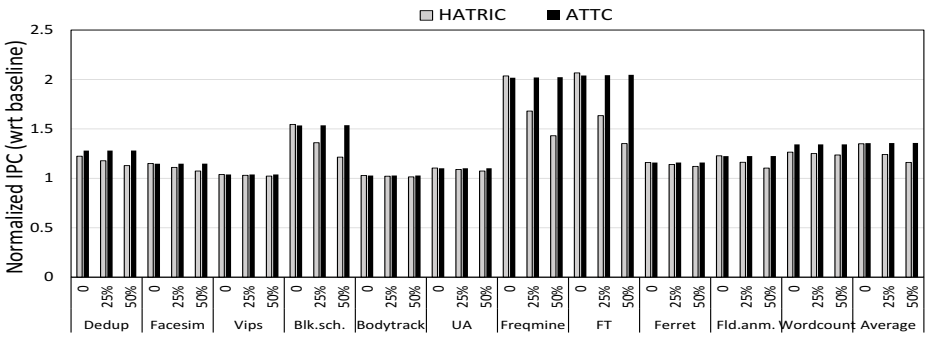


Figure 14: *When fraction of guest-initiated page table changes are small, HATRIC and* ATTC *have similar performance.* ATTC *improvements are more pronounced with increased guest updates.*

compares *ATTC* with HATRIC for three different fractions (0%, 25% and 50%) of page table changes initiated by the guest. As expected, HATRIC and *ATTC* behave comparably where there are no or very few guest-initiated page table changes. However, when the guest OS initiates non-trivial fractions of page table changes (such as for benchmarks wordcount or dedup),*ATTC* outperforms HATRIC due to its ability to address both guest and host initiated changes uniformly via the *ATLB*.

5 OTHER CONSIDERATIONS

Consistency, Memory Ordering: *ATTC* does not rely on any specific memory ordering mechanism and can be overlayed on top of existing software models with minimal changes. For e.g., in the baseline KVM system, all hypervisor updates to page tables are serialized under a single *kvm-mmu-lock*. *ATTC* uses the same locking mechanism to enforce ordering, but instead of sending IPIs, *ATLB* writes are issued. *ATTC* only needs to ensure that writes to the *ATLB* initiated by the hypervisor are architecturally committed before releasing the lock. If a system does not guarantee this, writes to *ATLB* can be followed by a suitable fence instruction. Moreover, across VMs, simultaneous writes to the same set of the *ATLB* might, at worst, result in issuing multiple invalidation requests. Given that shootdowns are infrequent, this scenario is extremely unlikely.

Coherence when Dirty/Access bits are Updated: This is handled similar to guest translation changes. If properties of a guest page are modified (such as changes to dirty/permissions bits), the guest OS should perform a write to the corresponding *ATLB* location whenever it updates the guest page table.

Handling Updates to Intermediate Levels of Page Tables: Similar to HATRIC and UNITD which require that intermediate level updates are reflected as writes to the last level PTEs, *ATTC* requires that the OS (or hypervisor) "reflect" intermediate level updates by writing to corresponding *ATLB* entries of all guest pages whose translations are modified. The cost of this update is similar to prior schemes.

Support for TLB Flush Operations: In some cases such as when a process is terminated, Linux issues a TLB *flush* in order to efficiently invalidate all TLB entries. While this can be efficiently implemented for small TLBs, it poses a challenge for the large *ATLB* (time duration for executing the primitive, as well as issue of invalidating useful entries belonging to other VMs). We identify three possible solutions that we outline below: (i) using background DMA[4] operations, the OS could wipe out the *ATLB* contents (ii) the hardware page-table walker could be enhanced to perform read-modify-writes to shoot-down VM-specific entries from the *ATLB* (iii) the *ATLB* entries could be enhanced to keep a "time-stamp" that could be used to lazily invalidate entries.

Synonyms: Multiple *gVA*s can map to the same *sPA* (e.g., shared libraries). If the *sPA* changes, then all the affected *gVA*s should be invalidated. This is supported in *ATTC* by using a set associative organization of the *INVTBL*. The maximum synonym count is limited to the associativity of the *INVTBL* organization (4 in our case). By increasing the associativity, a higher number of synonyms can be supported. At run-time, if synonyms exceed this limit, inserting a new entry to *INVTBL* is similar to inserting a new cache block to a set and will result in evicting an older entry (and this eviction would trigger coherence by writing to the corresponding ATLB entry following the pointers). For some pages (such as the zero page), the synonym count can be very high and it can be expensive to maintain inverse mappings for such pages. In such cases, the hypervisor may decide to track only those pages in the *INVTBL* that are anticipated to be migrated. Pages that are expected to be accessed frequently may be allocated on DRAM to begin with thereby not requiring any migration.

Support for Large Pages: While the discussion so far focused on small (4KB) pages, the scheme can be extended to support large pages. *ATLB* supports large pages by using a second addressable cache. Thus, *ATTC* can be extended to issue coherence messages triggered by writes to both the small and large page *ATLB* caches. The *INVTBL* will require an extra bit per entry indicating whether the *ATLB* pointer is to the small pages cache or the large one.

Security: In general, the security considerations for *ATTC* need to be studied in the context of virtualization and address translation hardware support (L1, L2 TLBs, MMU caches etc). Recent works such as [20] have proposed various forms of isolation between guests, the hypervisor and the hardware. Here, we discuss how we ensure security of the *ATLB* and the *INVTBL*. Since the *ATLB* is a shared structure, an untrusted VM could access and modify contents that belong to other VMs, breaching confidentiality and integrity.

[4]Direct Memory Access

	***ATLB* Reads**	***ATLB* Write**	*INVTBL* **Reads**	*INVTBL* **Writes**
Guest Processes	No	No	No	No
Guest OSes	No	Filtered by VM_ID in HW	No	No
Hypervisor	No	No	Allowed	Allowed

Table 4: Access Permissions

Table 4 summarizes the access permissions granted to guest processes, guest OSes and the hypervisor for accessing *ATLB* and *INVTBL*. *ATTC* enforces security by disallowing all reads and writes from guest OS to the *ATLB* and the *INVTBL*. Guest OS-initiated writes to the *ATLB* are intercepted by hardware (based on simple address range checks) and appropriate updates are performed by the Page Table Walker. Thus, software writes to *ATLB* serve only to trigger *ATTC* hardware actions and not to actually directly modify memory contents. *ATTC* ensures isolation between guests and isolation of hypervisor from guests: even though the *ATLB* is shared, different guests are confined to updating just their own entries (enforced by hardware using VM_ID). Further, no guest can access the hypervisor-controlled *INVTBL*. The host physical address range allocated to the *INVTBL* is not mapped to any guest.

The guest VM can write to the *ATLB* in order to trigger invalidations. These writes are filtered by *VM_ID* in hardware in order to ensure that an untrusted VM can not write to other entries. This protection mechanism (filtering by a context ID and preventing writes to select bit-fields) is commonly implemented in pin-muxing and other memory-mapped IO device accesses (see, for e.g., [5] which documents access types such as *RAZ* – Read-as-zero and *WI* – Write Ignore). The hypervisor is assumed to be trusted and hence the hypervisor has read and write privileges to *INVTBL*.

Cache side-channel attacks based on knowing the *ATLB* or the *INVTBL* address range could be thwarted by encryption-based (such as [36, 43]) or random mapping techniques (such as [24]).

6 RELATED WORK

Virtualization and reducing virtual memory overheads in virtualized environments have been the subject of significant research activity [3, 6, 8, 9, 11, 14, 16, 17, 26, 30–35, 45]. In order to improve the translation coherence process, various techniques have been proposed in both software and hardware. SITE [6] proposes to mitigate the overhead of TLB shootdowns by letting TLB entries expire and invalidate themselves. Oskin et al. [28] propose a scheme for hardware-assisted TLB shootdown. They introduce a special form of IPI, called the *REMOTE_INVLPG*, and an associated microcode change. The CPU handles a TLB shootdown process entirely in microcode. This approach eliminates OS interaction on the victim cores, but still incurs initiator overheads. Amit et al. [4] introduce Access Based Invalidation System (ABIS). Using access bits in the PTE, ABIS determines if the entry is cached in a TLB in the CMP. This information is used to avoid sending IPIs to cores that do not hold this translation. ABIS requires extensive hardware support for direct TLB insertion and complex software infrastructure like a secondary page-hierarchy.

Kumar et al. [22] 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.

Ouyang et al. [29] 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.

Villavieja et al. [42] proposed DIDI that couples shared TLB directory with load/store queue support for lightweight TLB invalidation to eliminate unnecessary IPIs.

Although, all the above mentioned schemes try to eliminate TLB shootdown penalty to some degree, they still leave TLB shootdowns in virtualized environments a high-overhead affair. While some schemes do not address virtualized environments, some eliminate only portion of TLB shootdown overhead and others are very imprecise or add a significant area and power penalty.

7 CONCLUSION

Translation coherence operations are generally expensive. In particular, when pages are migrated in a heterogenous memory system, TLB shootdowns constitute a significant performance penalty. To solve this problem, we present *ATTC*, a hardware-based translation coherence scheme leveraging an addressable TLB, which provides a system-wide point of coherence. Using a novel inverse table mechanism, *INVTBL*, *ATTC* enables hardware-based TLB coherence for both guest- and host-initiated page table changes. *ATTC* eliminates almost all of the busy wait overheads (upto 99%) seen in conventional systems. *ATTC* achieves a performance improvement of 35.7% on average over virtualized systems that employ an aggressive *k*vmtlb baseline without any TLB area penalty. Compared to the state-of-the-art hardware coherence scheme HATRIC, *ATTC* yields 7.4% improvement on average. *ATTC* will be beneficial for native and homogeneous memory systems as well, but the benefits will be most pronounced for virtualized systems and for heterogeneous memory systems.

8 ACKNOWLEDGEMENTS

The researchers are supported in part by National Science Foundation grants 1745813, 1725743 and 1763848. The authors would also like to thank Texas Advanced Computing Center (TACC) at UT Austin for providing compute resources. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, or any other sponsors. We are grateful for the feedback from Abhishek Bhattacharjee, Jayneel Gandhi, Karthik Ganesan and Jim Mattson.

REFERENCES

[1] Bhattacharjee Abhishek. [n.d.]. Personal Communication.
[2] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830
[3] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2018. Do-It-Yourself Virtual Memory Translation. *Operating Systems Review* 52, 1 (2018), 1–12. https://doi.org/10.1145/3273982.3273984
[4] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX

Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[5] ARM. 2015. ARM Glossary. (2015).

[6] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. https://doi.org/10.1109/PACT.2017.38

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks: Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. 2013. Efficient virtual memory for big memory servers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 237–248.

[10] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[12] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors.. In *HPCA*. IEEE Computer Society, 62–63. http://dblp.uni-trier.de/db/conf/hpca/hpca2011.html#BhattacharjeeLM11

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[14] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. 2013. Improving virtualization in the presence of software managed translation lookaside buffers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 120–129.

[15] P. Chi, S. Li, S. H. Kang, and Y. Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 109–114. https://doi.org/10.1109/ASPDAC.2016.7427997

[16] Guilherme Cox and Abhishek Bhattacharjee. 2017. 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, 435–448.

[17] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile paging: exceeding the best of nested and shadow paging. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 707–718.

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*. USENIX Association, 955–972.

[19] F. T. Hady, A. Foong, B. Veal, and D. Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (Sep. 2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

[20] David Kaplan. 2019. Upcoming x86 Technologies for Malicious Hypervisor Protection, SEV-SNP Slide deck, AMD. (2019). https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. 225–230.

[22] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselỳ, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. Latr: Lazy Translation Coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 651–664.

[23] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[24] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. 203–215. https://doi.org/10.1109/MICRO.2014.28

[25] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. https://doi.org/10.1145/2445572.2445574

[26] Zoltán Ádám Mann. 2015. Allocation of Virtual Machines in Cloud Data Centers&Mdash;A Survey of Problem Models and Optimization Algorithms. *ACM Comput. Surv.* 48, 1, Article 11 (Aug. 2015), 34 pages. https://doi.org/10.1145/2797211

[27] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K John. 2017. CSALT: context switch aware large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 449–462.

[28] Mark Oskin and Gabriel H Loh. 2015. A software-managed approach to die-stacked DRAM. In *Parallel Architecture and Compilation hatricPACT), 2015 International Conference on*. IEEE, 188–200.

[29] Jiannan Ouyang, John R Lange, and Haoqiang Zheng. 2016. Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs. *ACM SIGPLAN Notices* 51, 7 (2016), 17–23.

[30] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. 2015. Prediction-based superpage-friendly TLB designs. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 210–222.

[31] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 444–456.

[32] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 558–567.

[33] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain. 2018. TLB Shootdown Mitigation for Low-Power Many-Core Servers with L1 Virtual Caches. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 17–20. https://doi.org/10.1109/LCA.2017.2712140

[34] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 258–269.

[35] Binh Pham, Ján Veselỳ, Gabriel H Loh, and Abhishek Bhattacharjee. 2015. 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, 1–12.

[36] Moinuddin K. Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 360–371. https://doi.org/10.1145/3307650.3322246

[37] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[38] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[39] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 13–24. https://doi.org/10.1109/HPCA.2007.346181

[40] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. 2010. 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, 1–12.

[41] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K John. 2017. 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, 469–480.

[42] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. 2011. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 340–349.

[43] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[44] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 430–443.

[45] Chien-Hua Yen. 2007. SOLARIS OPERATING SYSTEM HARDWARE VIRTUALIZATION PRODUCT ARCHITECTURE. (2007).