# ShareMMU: Supporting Secure Address Translation Sharing among Untrusted Accelerators


Faiz Alam*
*Granular Labs*
USA
faiz@granular-labs.com

Mohamed Tarek Ibn Ziad
*NVIDIA*
USA
mtarek@nvidia.com

Yuanchao Xu*
*University of California, Santa Cruz*
Santa Cruz, CA, USA
yxu314@ucsc.edu

Abdulrahman Mahmoud*
*MBZUAI*
Abu Dhabi, UAE
abdulrahman.mahmoud@mbzuai.ac.ae

Greg Byrd
*North Carolina State University*
Raleigh, NC, USA
gbyrd@ncsu.edu

Aamer Jaleel
*NVIDIA*
USA
ajaleel@nvidia.com



***Abstract*—The growing demand for accelerated computing is driving widespread deployment of multi-accelerator systems in the cloud and at the edge. These systems are temporally shared across processes that may not mutually trust one another and often include large pools of third-party, untrusted accelerators operating within a shared virtual address space. Due to the limited capability of the Input-Output Memory Management Unit (IOMMU) in supporting address translation, SoC designers are integrating large Shared Translation Lookaside Buffers (TLBs) that serve many accelerators and processes. While effective for performance, shared TLBs introduce new security risks that require urgent and careful consideration. This paper presents ShareMMU, an IOMMU design that securely enables reuse of pre-translated addresses across multiple untrusted accelerators while also mitigating potential side channels in the shared TLB. Compared to state-of-the-art mechanisms like Border Control and CryptoMMU, which incur 32% and 17% performance overhead, respectively, ShareMMU incurs a negligible 1.66% performance overhead relative to an unsecured baseline. Our side-channel resilient variant, ShareMMU-SP, incurs roughly 2.82% overhead.**



***Index Terms*—IOMMU, accelerators, access control, address translation, shared TLB, side-channel attacks, hardware security, memory isolation**


## I. Introduction

Multi-accelerator systems, such as NVIDIA's SuperPOD [1] and Google TPU Pods [2], are now central to modern accelerated computing. They target applications with high computational demands and large memory footprints. Such systems often integrate several specialized accelerators for distinct data-processing tasks or multiple instances of the same accelerator to exploit parallelism [3]–[9]. Many of these systems may deploy a variety of third-party accelerators to reduce development time, access novel capabilities, and improve cost efficiency. These off-the-shelf black-box IPs have become common to accelerate the design cycle, but they cannot be fully trusted [10]–[12].

*Part of this work was done while Faiz Alam and Yuanchao Xu were PhD students at North Carolina State University, and Abdulrahman Mahmoud was a Postdoctoral Researcher at Harvard University.

Parallel execution naturally creates page sharing across accelerators. Since a significant portion of an accelerator's performance depends on efficient address translation [13]–[18], a shared TLB across multiple accelerator instances provides substantial performance benefits [14], [19]–[21]. A shared TLB eliminates duplicate page walks by allowing accelerators with the same permissions to reuse translations fetched by peers. In pipelined designs, where multiple accelerators sequentially process a shared data structure, a shared TLB enables later stages to reuse earlier translations [14], [22]. In multi-context systems, shared TLBs also avoid frequent flushes on context switches and enable dynamic capacity sharing, allowing heavier workloads to access more TLB entries than lighter ones [14], [23], [24].

While a shared TLB enables translation reuse, the operating system (OS) must often update page permissions for the accelerators to coordinate shared resources. Pages may transition between no-access, read-only, and read-write at different stages to prevent data races and allow safe collaboration [25]–[28]. On a permission change, the OS issues TLB shootdowns to flush private TLBs [12]. Untrusted accelerators may maliciously ignore these requests and continue using stale translations (or even fabricate physical addresses), leading to correctness and security violations [10], [12], [29]–[36]. Because an untrusted accelerator can cache physical addresses and issue requests with escalated permissions, every accelerator access must be validated. This validation must incur low overhead to justify the adoption of third-party designs or the integration of multiple accelerators.

Additionally, shared TLBs often serve multiple distrusting processes. In this case, a malicious process can execute timing side-channel attacks on the shared TLB (e.g., contention or occupancy-based attacks [37]). Such attacks can leak secrets, reveal a TLB's structure and replacement policy, create covert communication channels, and compromise other defenses [38]–[41]. Therefore, protecting the shared TLB from becoming an attack vector is critical.

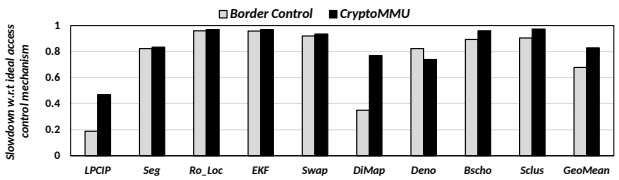


Fig. 1: Slowdown in Border Control and CryptoMMU relative to an unsecured baseline.

The state-of-the-art access control mechanism, **CryptoMMU** [12], validates each pre-translated access by verifying a cryptographic hash derived from the Page Table Entry (PTE). A pre-translated address is a cached physical address obtained from a prior translation of a virtual address. CryptoMMU requires sending the Virtual Page Number (VPN) and a hash on every TLB hit, demanding significant modifications to the TLB interface and hit datapath. The hash generation uses per-accelerator keys; therefore, the same translation yields different hashes across devices, reducing shared TLB coverage and preventing reuse. CryptoMMU also enforces TLB shootdowns by changing the device key, which invalidates all cached translations. **Border Control** [36], in contrast, permits translation sharing but maintains large per-accelerator permission bitmaps ("Protection Table") and a cache ("Border Control Cache", BCC) to verify accesses from pre-translated addresses. Maintaining these structures requires complex software support and incurs substantial overhead to keep them coherent with page table updates. Figure 1 shows slowdowns of 32% (up to 40%) and 17% (up to 23%) for Border Control and CryptoMMU, respectively, relative to an unsecured baseline. The unsecured baseline uses an IOMMU that only provides Address Translation Services (ATS) to accelerators. The untrusted domain can then cache and access physical memory without subsequent verification. Furthermore, neither approach provides side-channel resilience; in fact, CryptoMMU's "Read Acceleration" optimization can facilitate leakage by speculatively permitting reads before permissions are fully verified, enabling adversaries to collect traces for side-channel attacks. To guarantee system security, defenses must ensure an accelerator accesses only specific physical addresses (spatial safety) and only within permitted time windows (temporal safety).

We introduce **ShareMMU** to address the limitations of previous IOMMU designs. ShareMMU's design philosophy aligns with established principles in accelerator system design that bound the number of supported accelerators and guide the sizing of hardware components such as caches and TLBs. Extending this philosophy, ShareMMU introduces a single change to the system stack: a specialized hardware cache named the ***Permission Tracker*** to support a fixed number of third-party accelerators. Each Permission Tracker entry, called a ***Secure Vector***, is tagged with a Physical Page Number (PPN) and records per-accelerator permissions and the VPNs mapped to that PPN.

TABLE I: Comparison of Border Control (BC), CryptoMMU, and ShareMMU.

| **Metric** | **BC** | **CryptoMMU** | **ShareMMU** |
|---|---|---|---|
| Performance | Very Low | Low | High |
| TLB Sharing | Yes | No | Yes |
| Hardware Changes | Minimal | Significant | Minimal |
| Software Changes | Significant | Minimal | Minimal |
| Side-Channel Resilience | No | Worsens | Yes |

ShareMMU's security policy considers any privately cached translation in the untrusted accelerator to be manufactured or suspicious until proven otherwise. Thus, it verifies every incoming memory access request that uses a pre-translated address. The Permission Tracker enables such verification by recording the physical address and permissions provided to the untrusted accelerator and checking future requests against this record. Any request not found in the Permission Tracker is treated as invalid or suspicious. To ensure that the Permission Tracker can always verify accesses from a privately cached translation, the Permission Tracker is inclusive of the private TLBs. Inclusion is enforced by sending a back-invalidation to the accelerator whenever a Secure Vector is evicted. To prevent fairness issues and avoid accelerator monopolization of the Permission Tracker, we manage it using an *Accelerator-Aware LRU (AA-LRU)* policy. We also present **ShareMMU-SP**, a side-channel–resilient variant that mitigates conflict- and occupancy-based attacks on shared TLB and Permission Tracker entries. We evaluate ShareMMU using the gem5 PARADE simulator [42], [43] following the same methodology as prior work [12], [14]. ShareMMU has an overhead of 1.66%, while ShareMMU-SP has 2.82% overhead compared to the unsecured baseline. Table I summarizes how ShareMMU compares against prior work.

## II. Background

### A. Multi Accelerator SoCs

In multi-accelerator System-on-Chips (SoCs) with a pipelined architecture, data flows sequentially from one accelerator to another, with each accelerator performing a specialized computation on shared data. Memory access permissions must be dynamically managed to ensure data consistency of shared pages. This is typically achieved by granting and revoking access to specific memory pages as needed, ensuring that only the relevant accelerator has the necessary permissions at any given time. Fine-grained access control prevents races, stale data, and unauthorized access, enabling efficient and secure processing. Figure 2 illustrates a typical data flow involving multiple accelerators in a System on Chip (SoC) where various components interact to process data with varying permission levels. The input data stream generated by the camera feed is processed by the CPU and stored in the insecure DRAM. This data is then forwarded to the Crypto accelerator, which is granted permission to access (indicated as 'Perm-yes') for encryption or decryption tasks. After crypto processing, the permission of the Crypto accelerator is revoked

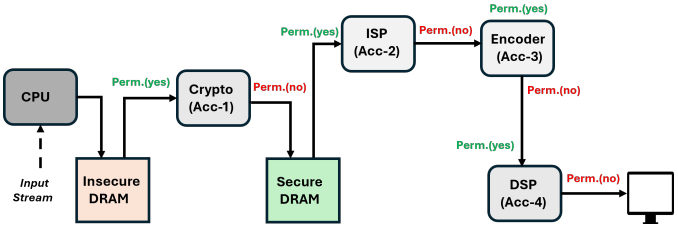


Fig. 2: A typical Multi-Accelerator SoC.

(indicated as 'Perm-no'). The data is routed to the Image Signal Processor (ISP), Encoder, and Digital Signal Processor (DSP), each performing their specific tasks before forwarding the data. While these accelerators are processing, they have permission to access that data, which is immediately revoked as they are done, highlighting a controlled access strategy within the SoC architecture [44].

### B. Address Translation in DSAs

Address translation in domain-specific accelerators (DSAs) is facilitated by frameworks like unified virtual memory (UVM). UVM enables the CPU and accelerator to access shared data using the same virtual pointers, eliminating explicit data copying between host and accelerator memories [45]–[48]. The Input-Output Memory Management Unit (IOMMU) translates the accelerator's virtual addresses into the system's physical addresses.

The IOMMU features a translation cache, i.e., Input-Output translation lookaside buffer (IOTLB), as well as Page Table Walker (IOPTW) logic to traverse the accelerator/device page table [49]. A Page Table Walk Cache (PTWC) stores recent PTEs from different page-table levels. The OS-managed page table maps IO virtual addresses (IOVA) to physical addresses and encodes per-accelerator permissions. Because DSAs often comprise many application-specific accelerators collaborating with the host CPU, a shared page table lets the IOMMU operate directly and coherently on the host's tables. Further, a shared page table eliminates redundant per-accelerator I/O page tables [50]. Therefore, this paper focuses on the ***Shared Page Table*** model, noting that ShareMMU can also be built for a private page table model.

### C. Memory Access Patterns of Accelerators

Although some accelerators target applications with regular memory access patterns, such as string matching, image processing, and graphics, emerging accelerators target applications with irregular memory access patterns (e.g., pointer-chasing), such as databases, key-value stores, and graphs. For example, the latest Intel processor has introduced the In-Memory Analytics Accelerator (IAA) to accelerate key-value store applications with irregular accesses [51], [52]. Numerous other prior works have studied different irregular access patterns [53], [54]. In addition, the accelerator's physical memory access patterns depend on a range of factors, including runtime state, the operating system's scheduling policy, page allocation strategies, and the overall memory footprint of the applications. This complexity highlights the adaptive nature of accelerator performance, which must be carefully managed to optimize efficiency across regular and irregular access patterns.

### D. Secure Cache Design

Time-Secure Caches [55] utilize traditional set-associative caches but add a keyed function for indexing, which combines the cache line address and Process ID (PID) as inputs. This eliminates obvious cache congruences across different processes, reducing cache interference. However, the indexing function is relatively weak and requires frequent re-keying, which flushes the cache and harms performance. ScatterCache expands on this concept by introducing a higher entropy cryptographic indexing function [56]. Similarly, CEASER [57] randomizes cache indexing using encrypted addresses and periodic re-keying, while CEASER-S [58] partitions cache ways with distinct keys and selective re-keying to improve security and performance.

Despite these advancements, the ScatterCache design [56] has been shown vulnerable [37]. Frequent re-keying to guarantee security, as in [57], [58], is ill-suited for a shared TLB and may incur significant performance penalties. SassCache [37] addresses these challenges by using a skewed set-associative cache design and a low-latency keyed index-derivative function per security domain, providing pseudorandom isolation among domains. ShareMMU draws inspiration from these principles to secure the shared TLB.

## III. Motivation

### A. Fundamental Needs for Shared TLBs in SoCs

IOMMUs have become a performance bottleneck when providing Address Translation Services (ATS) for modern multi-accelerator SoC systems [15]–[18]. A private TLB per accelerator instance can improve performance by serving hits locally. Recent studies [14], [19]–[21] confirm the performance benefits of adding a shared TLB. When multiple accelerators operate on cache blocks within the same page, a shared TLB avoids duplicate page walks by allowing reuse of existing translations. Because many pages are shared across accelerators, they benefit from translations already fetched into the shared TLB. We systematically evaluated page sharing across 16 accelerators in our system. Figure 3 shows the percentage of pages simultaneously shared by multiple accelerators. On average, 46% of pages have no sharers, 44% of pages are shared by 2 to 5 accelerators, and 10% are shared by more than 5 accelerators across our benchmarks. The fact that over half the pages are shared emphasizes the importance of implementing a shared TLB.

We also evaluated the impact of adding a 1024-entry shared TLB to a system with 16 accelerators, each with a 16-entry private TLB. Figure 4 shows the normalized performance improvement across our benchmarks when using only private TLBs per accelerator compared to a setup with both private TLBs and a shared TLB. We observed an average performance

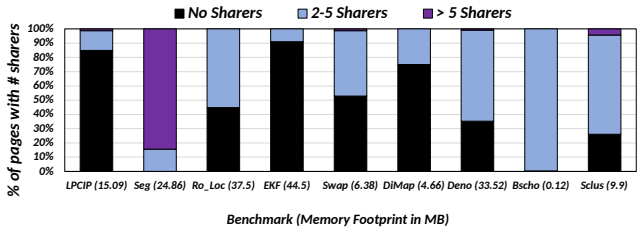


Fig. 3: Percentage of pages shared by different accelerators.

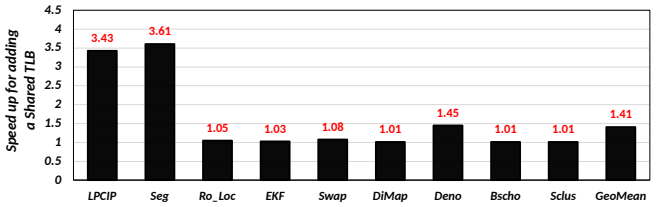


Fig. 4: Speedup from shared TLB compared to private-only.

improvement of 41%. More details on our architecture and benchmarks are presented in Section VI. This performance gain is primarily driven by (1) dynamic partitioning of shared TLB entries to adapt to varying workload demands among accelerators and (2) the sharing of translations across accelerators. These results suggest that a shared TLB can significantly enhance address translation performance.

### *B. Limitations of Prior Studies*

Prior work, such as Border Control [36], addresses access control by introducing a physically tagged bitmap structure called a "protection table", stored per accelerator in memory to record permissions for each memory page. This protection table can be further cached in the IOMMU's Border Control Cache, enabling faster verification of access permissions. Similarly, CryptoMMU [12] enforces restricted access by cryptographically ensuring that only designated memory can be accessed by each accelerator by embedding a cryptographic hash generated from the PTE, VPN, and device key in every translation. Accelerators must provide this cryptographic evidence on accesses using a pre-translated address. The IOMMU regenerates the cryptographic hash and matches it with the provided hash to allow this access. Despite these efforts, both approaches face significant security and scalability limitations in multi-context, multi-accelerator environments.

**(1) Poor performance.** CryptoMMU does not support scalable TLB sharing among untrusted accelerators. As discussed earlier, a shared TLB is critical for multi-accelerator performance. By design, CryptoMMU prohibits sharing: when an accelerator-specific PTE (i.e., a hash generated with device keys) is installed in private and shared TLBs, another accelerator cannot reuse this translation because it cannot prove ownership to the IOMMU. Hence, CryptoMMU would incorrectly flag those accesses or redundantly walk the page table to fetch the translation and generate a new hash for the requester. Moreover, device-specific hashes create duplicate entries for the same translation in the shared TLB, reducing effective coverage. Additionally, CryptoMMU requires transmission of the VPN and a large cryptographic hash on every TLB hit, which would compete for valuable interconnect bandwidth, further impacting system performance. On the other hand, Border Control does not inherently prevent sharing; however, its performance significantly suffers due to the need to maintain coherence between page tables, protection tables stored in memory, and the BCC in the IOMMU. The problem worsens as the number of accelerators increases because more protection tables and BCC state must be updated.

**(2) Complex hardware and software modifications.** CryptoMMU requires changes to the TLB interface and hit datapath. While the TLB hit datapath usually requires the accelerator to send the physical address to access memory, CryptoMMU requires sending the PTE, virtual address, and cryptographic hash on every TLB hit. Modifying the TLB interface and datapath is non-trivial and requires substantial hardware changes and verification effort. Border Control, in contrast, requires a per-accelerator protection table in memory. These tables can become extremely large as main memory scales to TBs [12]. Further, the per-accelerator Border Control cache does not scale with the number of accelerators. Finally, Border Control requires complex OS modifications to manage multiple large protection tables and keep them consistent with the page table.

**(3) Inadequate handling of frequent permission changes.** Border Control and CryptoMMU handle frequent permission changes poorly and incur substantial overheads. The OS may frequently modify page permissions to avoid data races, particularly when the CPU or an accelerator alternates reads and writes to a shared page [26]–[28]. Figure 5 shows the frequency of permission changes per 1,000 instructions (PCPKI) in our benchmarks. StreamCluster and Disparity Map exhibit high rates (0.016 and 0.31, respectively) due to multiple iterations using various accelerators over the same data block. BlackScholes and Segmentation have PCPKI of 0.048 and 0.019, respectively, due to alternating reads and writes between the CPU and accelerators. All other benchmarks have moderate yet significant permission changes.[1]

In such a scenario, an untrusted accelerator might ignore permission changes or TLB shootdown requests, continuing to use outdated translations. This behavior can lead to severe correctness and security issues. To address these problems, CryptoMMU changes the cryptographic key, invalidating all cached cryptographic hashes and effectively flushing all translations stored in private and shared TLBs. Border Control, in contrast, requires multiple steps to enforce these permission changes. After issuing a TLB shootdown request, it must search the entire Border Control Cache for entries with the corresponding PPN and invalidate them. Then, the OS must

[1]The graph only contains dynamic permission upgrades/downgrades on pages shared among CPU and accelerators after permissions are set and physical pages are mapped. It does not include other permission updates/management, such as copy-on-write, swapping, etc.

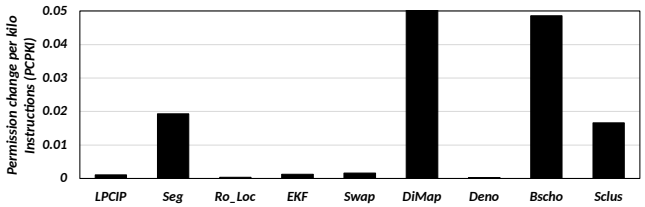


Fig. 5: Frequency of page permission changes.

update the protection tables of all accelerators in memory, leading to considerable performance overhead.

**(4) Susceptibility to side channels.** Both Border Control and CryptoMMU are susceptible to side-channel attacks via shared TLBs. Many studies have shown that timing-based side-channel attacks can reveal TLB structure and replacement policies, establish covert communication channels, and compromise other defenses through shared TLB attacks [38]–[41]. CryptoMMU's "Read Acceleration" optimization facilitates such attacks by speculatively allowing accelerators to read data while permissions are being checked. Malicious processes can exploit this optimization to generate architectural traces that may leak sensitive information.

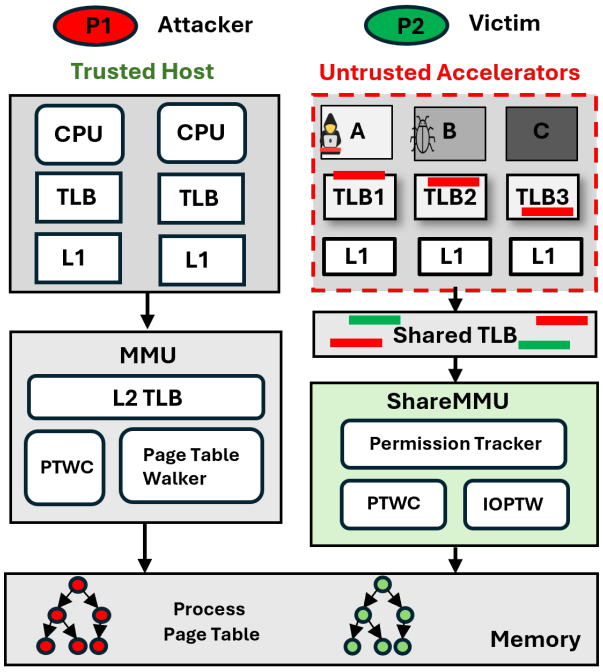


Fig. 6: Our assumed threat model.

## IV. Threat Model

Figure 6 illustrates our assumed threat model, which is closely aligned with prior work [12], [36]. The accelerator subsystem and its private TLBs are untrusted and lie outside the Trusted Computing Base (TCB). In contrast, the shared TLB is trusted and may hold translations for multiple accelerators and processes. The OS is trusted and sets legitimate permissions. The on-chip IOMMU (e.g., ShareMMU) is trusted and enforces OS-defined access permissions stored in per-process page tables, which reside in trusted memory [59]–[63]. Accelerators are granted read, write, or no-access permissions on a per-page basis. When the OS modifies a page's permissions, the IOMMU must enforce the updated access controls.

Our threat model does not assume the untrusted accelerator corrupts its own computational task or leaks its data. Instead, we trust the accelerator to operate only on authorized data within a strict time window. For example, the accelerator may access data $D_1$ at time $t_1$, but must be prevented from accessing $D_1$ at a later time $t_2$, or from accessing unrelated data $D_2$ elsewhere in memory. The threat is a rogue or faulty accelerator that attempts to bypass domain isolation. It may issue unauthorized DMA requests to exfiltrate data or corrupt host memory, or leverage forged physical addresses and stale permissions to break access-control policies.

The ShareMMU threat model also includes side-channel attacks from a malicious process that exploits shared hardware structures, such as the TLB or Permission Tracker, to infer information about co-located processes. We assume an unprivileged attacker process co-scheduled with the victim and co-resident in the shared TLB can launch conflict- or occupancy-based attacks [64], [65] consistent with prior threat models [58]. All other types of side-channel attacks are considered beyond the scope of this work. Processes P1 (attacker) and P2 (victim) utilize accelerators A, B, and C in a time-shared manner. As depicted in the figure, the private TLB holds translations exclusively for P1 (red), while the shared TLB contains translations for both P1 (red) and P2 (green). Because the shared TLB includes translations for both processes, it is vulnerable to side-channel leakage [38]–[41].

## V. ShareMMU Design

A fundamental limitation of prior designs is that they were never architected for multi-accelerator systems that share address translations, imposing structural barriers rather than mere optimization bottlenecks. CryptoMMU ties validation keys to individual devices, which prohibits translation sharing and bloats the shared TLB with duplicate entries, while Border Control relies on large in-memory tables that serialize permission synchronization across devices.

### A. ShareMMU Overview

ShareMMU is a realistic IOMMU designed for modern multi-context, multi-accelerator systems. It allows untrusted accelerators to securely share cached translations and efficiently manage dynamic permission updates.

The key idea of ShareMMU is to track the permissions of all translations sent to the private TLBs. Each address translation is first verified (using a page table walk) before being installed in the shared TLB. After installation, this translation can be reused. Any translation fetched in the private TLB can be tampered with; hence, it is potentially illegal and requires permission verification. To realize the above functionality, ShareMMU uses a hardware cache called the

*Permission Tracker* to track private TLB translations for safe reuse and memory access.

We now explore the details of ShareMMU design. We first examine the design of the Permission Tracker in Section V-B and the operation of ShareMMU to provide access control, translation reuse, and management of dynamic permission changes in Section V-C. Finally, we discuss the ShareMMU strategy to protect shared TLB and Permission Tracker from side-channel attacks in Section V-D.

### B. Permission Tracker

The Permission Tracker is designed to contain permissions for all devices that can access a particular physical page. We implement the Permission Tracker as a hardware-managed cache within the ShareMMU. Each entry of the Permission Tracker contains a *Secure Vector*, which is tagged with the physical address and stores the permissions of all accelerators for that page and all the virtual pages mapped to it, as shown in Figure 7. It also contains a bitmap "Access Mask" and Least Recently Used (LRU) bits to implement our *Accelerator-Aware LRU* (AA-LRU) replacement policy described in Section V-C2. The Access Mask contains one bit per accelerator and is set whenever an accelerator accesses that Secure Vector. Depending on the implementation, device permission bits can reside either in a shared or separate page table.[2] Our design adopts a shared page table across accelerators, as demonstrated in prior works [14], [21], [66], [67]. The leaf nodes of the shared page table store the permission bits for all accelerators accessing the page. Whenever a page table walk services a new translation request, these bits are updated in the shared TLB and Permission Tracker. In designs using separate page tables, permission updates in the Permission Tracker will follow an approach similar to shared TLB updates, as discussed in prior works [13], [68], [69]. If multiple virtual pages are mapped to the same physical page, the Secure Vector contains a pointer to a special table named *Alias Table* in memory, which contains all mapped virtual pages as discussed in Section V-C5.

The Permission Tracker is inclusive of all the private TLBs in accelerators. Since the number of accelerators a host must support has an upper bound, the Permission Tracker size is also bounded, similar to caches, the shared TLB, and the page table walk cache. The Permission Tracker's inclusion policy ensures that any physical memory access request originating from a private TLB will always find a corresponding Secure Vector in the Permission Tracker for verification. Any verification request that is not found is deemed suspicious. ShareMMU ensures inclusivity by sending back-invalidation to the private TLB whenever an entry is evicted from the Tracker.

The Permission Tracker maintains a VPN with each PPN in its Secure Vector. VPN enables reuse of existing TLB shootdown mechanisms for sending back-invalidation requests to private TLBs. The Access Mask bitmap contains set bits for accelerators that have already accessed a translation; hence, the Access Mask can be used to send targeted back-invalidations to the corresponding accelerator. If a malicious accelerator refuses to honor a back-invalidation, the Permission Tracker will reject any subsequent accesses to the invalidated address. Consequently, accelerators are compelled to honor back-invalidations to maintain correct operation. If multiple virtual addresses map to the same physical page, we retrieve all such virtual addresses from the Alias Table and send back-invalidations to maintain inclusion.

[2]We assume that permission bits for each accelerator are embedded in the leaf nodes of the process page table. In other implementations, they can be fetched and updated similarly to a shared TLB without modifications.

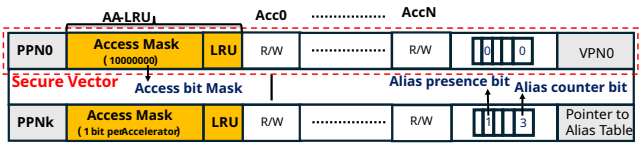


Fig. 7: Secure Vector. An entry in Permission Tracker.

### C. Access Control and Translation Reuse

In this section, we describe how access control and translation reuse are enabled in ShareMMU. Figure 8 describes the operation of ShareMMU on a miss in the private TLB. Let us consider an example with three accelerators, `A`, `B`, and `C`, each with its own private TLB and L1 data cache. Shared TLB and private TLBs are inclusive, and page tables are stored in memory. ShareMMU has a Permission Tracker that can hold one Secure Vector. `B` and `C` are operating on the same page but on different data blocks, `B1` and `Bn`. Initially, `C` has the translation for page `VPN0` in the private TLB and operates with read-only permission on block `Bn`. The Permission Tracker has a Secure Vector created with the tag `PPN0` when `C` fetched that translation. The Secure Vector also has the latest permissions for "A(no access)", "B(write)", and "C(read)", which were fetched during C's page walk.

The miss-handling process works as follows. ① `B` requests an address translation for `VPN0`, which misses in private TLB but hits in the shared TLB. ShareMMU checks the Permission Tracker, finds a Secure Vector tagged with `PPN0` with write access to `B`, and supplies the translation to its private TLB. This step enables `B` to reuse the translation brought by `C` since it has appropriate permissions. ② `A` requests a translation of `VPN1` and misses in both private and shared TLB. ShareMMU walks the page table and fetches the translation `PPN1` and the permissions of all the accelerators on that page. ③ ShareMMU updates the shared TLB and searches the permission tracker for a Secure Vector for `PPN1`. Since there is no entry, it creates one before sending the translation to the private TLB. ④ While creating a Secure Vector for `PPN1`, `PPN0` needs to be evicted because of insufficient space in the Permission Tracker. ⑤ Since `PPN0` is evicted, no accelerator can now access this page. The Access Mask would be (011) signifying that `B` and `C` have previously used this translation. Therefore, `VPN0` is extracted, and a back-invalidation is sent to the private TLBs of `B` and `C`. ⑥ Since `B` has write access, it might have a dirty block `B1`, which needs to be written back before flushing the translation. The back-invalidation guarantees that

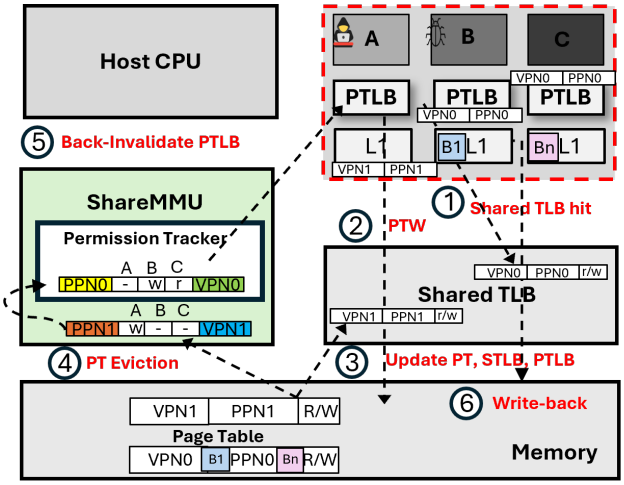


Fig. 8: ShareMMU miss operation.

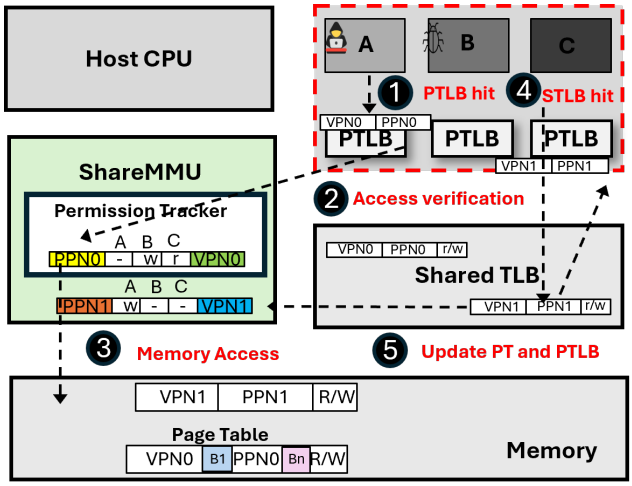


Fig. 9: ShareMMU hit operation.

the Permission Tracker can track every legitimate access request that should come from the accelerator. If an accelerator refuses to flush its translation on a back-invalidation request, it can never use the stale translation since it will not be in the Permission Tracker.

Figure 9 describes the access control mechanism of ShareMMU on TLB hits. ❶ We start with a private TLB hit for accelerator A. ❷ The accelerator sends the physical address for access verification. Thus, this design requires no interface changes (e.g., sending additional metadata or virtual addresses as in CryptoMMU). ❸ If this request hits in the Permission Tracker and access permissions are correct, the access is allowed. Otherwise, this request would be flagged as a malicious access that needs to be handled by the OS. ❹ When another accelerator misses in the private TLB but hits in the shared TLB, the Permission Tracker is checked before sending the translation. Nothing needs to be done if the Secure Vector is present with all the permissions, implying that another accelerator wants to reuse the previously fetched translation. ❺ A Secure Vector is created if it does not exist. This case can occur since the Permission Tracker only maintains inclusivity with untrusted private TLBs, not trusted shared TLBs. So, there can be translations in shared TLB that do not have a Secure Vector entry in Permission Tracker.

*1) Enforcing Frequent Permission Updates via TLB Shootdowns:* Page mapping updates that result in TLB shootdowns can occur for various reasons, such as context switches, page migration, memory compaction, and copy-on-write, all of which can unmap physical pages [36]. However, many accelerator models rely on permission changes to maintain consistency across shared data structures. In these models, the physical page mapping remains unchanged, but page permissions are upgraded or downgraded. This prevents data races and allows the CPU and multiple accelerators to process shared data coherently and consistently. The frequent change in page permissions of different accelerators incurs significant overhead in prior work, as discussed in Section III-B.

ShareMMU does not rely on costly or complicated mechanisms to enforce permission changes with untrusted accelerators. It uses existing TLB shootdown mechanisms without modification. Whenever the OS updates the PTE, ShareMMU issues a TLB shootdown to the trusted shared TLB, Permission Tracker, the PTWC, and the untrusted private TLB. On a permission downgrade, the Permission Tracker need not flush the Secure Vector if other accelerators still use it; it simply clears the affected accelerator's permission bit (e.g., to no-access) while leaving the shared entry intact. Additionally, a targeted back-invalidation is sent to the private TLB on Permission Tracker eviction [70]–[74] by extracting the VPN and Access Mask bits from the Secure Vector. No additional steps are needed to ensure that the untrusted accelerator respects TLB shootdown or back-invalidation requests. However, the system still waits for accelerators to complete write-backs of dirty cache lines, flush their private TLBs, and send flush acknowledgment to the host.

*2) Accelerator-Aware LRU replacement policy:* A malicious accelerator can attempt to monopolize the Permission Tracker by repeatedly inserting translations it exclusively uses, evicting entries actively shared by other accelerators. We propose Accelerator-Aware LRU (AA-LRU) that uses the Permission Tracker *Access Mask* to avoid monopolization. The Access Mask is a bitmap with one bit per accelerator, set whenever an accelerator successfully verifies a physical page, as shown in Figure 7. As explained in Algorithm 1, on a Permission Tracker hit, the corresponding device's Access Mask bit is set (whether it was previously unset or already set), and the LRU state is updated. On a miss, if no free entry is available and an existing Secure Vector must be evicted, a *victim score* is computed for each way in the set to select the replacement. The victim score equals the number of set bits in the Access Mask (i.e., the number of accelerators reusing the translation) combined with the entry's LRU priority. Entries reused by many accelerators receive a higher score and are preserved, while entries used by only one accelerator (a single

bit set) receive a low score and are preferentially evicted. The AA-LRU policy prevents monopolization, as entries inserted by malicious accelerators would have only their bit set and no reuse from others; they always receive the lowest victim score and are evicted first. Additional weighting between the LRU component and the Access Mask count can be applied depending on the use case. Moreover, victim-way selection can be occasionally randomized to further reduce any potential gamification, similar to the policy used in Section V-D1. At some intermittent timestamps, random ways may be evicted to maintain fairness and improve robustness.

**Algorithm 1** AA-LRU Replacement Policy

**Require:** AccessMask, LRUbits, Request(PPN, deviceID)
**if** PPN is present in Permission Tracker **then** ▷ PT hit
    update LRUbits;
    AccessMask[deviceID] = 1; ▷ mark access
**else** ▷ PT miss
    **for** each entry $x$ in the set **do** ▷ needs eviction
        VictimScore = (Way − LRUbits) + count(AccessMask);
    **end for**
    victim = argmin(VictimScore);
    victim = random(way) ▷ at timestamp
    replace victim;
**end if**

*3) Handling Silent Evictions:* There may be cases where the Permission Tracker still contains a Secure Vector for a translation that has already been silently evicted from a private TLB. In such situations, we expect Permission Tracker's replacement policy to manage these stale entries. Regardless of the replacement policy used by private TLBs[3], the Permission Tracker always applies AA-LRU. We assume that an honest accelerator will not generate verification requests for entries it has already evicted. Therefore, AA-LRU's LRU component naturally deprioritizes stale vectors and prevents them from displacing actively used ones. This ensures that valid, frequently used translations in private TLBs are not needlessly back-invalidated.

*4) Handling Disjoint Accelerator Working Sets:* The Permission Tracker is provisioned to hold the combined capacity of all private TLBs, which can naturally reside in the Permission Tracker without conflict. As with any shared structure, there may be instances where multiple accelerators frequently access disjoint working sets that map to the same set, resulting in Permission Tracker thrashing. Such collisions between disjoint working sets occur only if their physical pages hash to the same Permission Tracker set, which is a property of the indexing function rather than the access pattern. If such collisions arise, the corresponding Secure Vectors can be quickly reconstructed from the shared TLB or the PTWC, preventing sustained thrashing or performance loss.

*5) Handling Aliases in ShareMMU:* Aliasing occurs when multiple virtual addresses map to the same physical page. Although aliases are uncommon, they are non-zero [75]. Since the Secure Vector in the Permission Tracker is tagged with a physical page number, the corresponding virtual address is required to back-invalidate entries in the private TLB during a Secure Vector eviction. To enable this, the Secure Vector's virtual-page field stores a pointer to an "Alias Table" in a reserved memory region, which lists all aliases for that physical page, as shown in Figure 10. Since the system usually maintains an inverted page table or an alias tracking mechanism to support the virtual caches, we can reuse it without needing additional changes [76], [77].

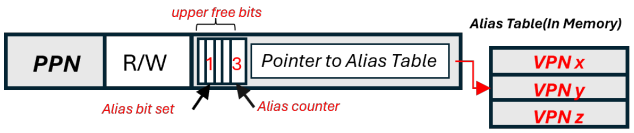


Fig. 10: Alias Handling.

We use two fields in the upper free bits of the virtual-page field in the Permission Tracker: an "alias presence" bit and an "alias counter". The alias presence bit is set when aliases exist, indicating that multiple virtual addresses map to the physical page. The alias counter records the number of virtual pages in the Alias Table. The Alias Table is updated via the inverted page table or during Permission Tracker updates.

When an entry with the alias presence bit set is evicted from the Permission Tracker, all virtual addresses are fetched and back-invalidations are sent to the corresponding private TLBs.

*6) Coalescing and Huge Pages of Permission Tracker:* The Permission Tracker operates at the same granularity as the page table, enabling ShareMMU to handle huge pages without any modifications. Since huge pages can be used in accelerator workloads to improve TLB coverage, they also enhance the coverage of the Permission Tracker. ShareMMU can seamlessly cache permissions for huge pages, ensuring efficient operation without additional adjustments. For instance, a 1 GB huge page may encapsulate permissions for 262,144 (4 KB) pages, all of which can be represented within a single entry in the Permission Tracker.

*7) Transient States and Deadlock Freedom:* A potential race condition can occur during ShareMMU back-invalidations caused by PT entry evictions or regular TLB shootdowns due to permission updates. The global permission update should not finish until the accelerator writes back dirty cache lines and drains all in-flight memory accesses. This transient state protocol is illustrated in Figure 11 and traverses the following sequence: ❶ `S_VALID`: the accelerator holds valid page permissions. ❷ The host intends to change page permissions, or a PT entry needs eviction. Before issuing any request, the host flushes and locks the trusted shared TLB (no flush is required on a PT eviction), then sends a targeted back-invalidation to the accelerators indicated by the Access Mask. ❸ While the back-invalidation is in transit, the entry is in `S_BCKInV` (pending-flush state). ❹ Meanwhile, a private-TLB hit may proceed safely after PT verification, whereas a private-TLB miss simply stalls. ❺ Once the invalidation reaches the accelerator, it writes back dirty cache lines and

[3]ShareMMU imposes no constraints on inclusivity between private and shared TLBs; they may be exclusive, inclusive, or mostly inclusive.

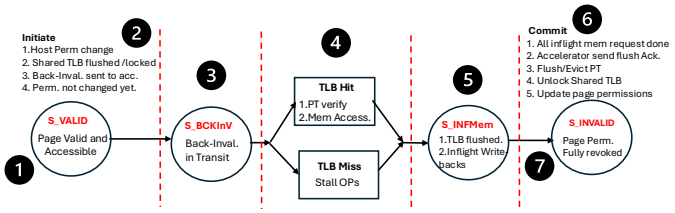


Fig. 11: Transient States during Back-Invalidation.

completes all in-flight memory requests. This is `S_INFMem` (accelerator draining memory requests). ❻ After the in-flight requests drain, the host receives the flush acknowledgment, and the PT entry is safely evicted, or the page permissions are updated. ❼ The shared TLB is unlocked and the entry enters `S_INVALID`, with the previous permissions fully revoked.

The protocol is deadlock-free because it avoids circular dependencies during the race window (❹). Stalled TLB misses wait passively at the locked shared TLB and do not consume the accelerator's internal execution resources, while in-flight TLB hits complete in bounded hardware time because physical-memory permissions remain valid until the host commits the update in step (❻). Since stalled misses cannot block active hits, the execution pipeline deterministically drains (`S_INFMem`), guaranteeing that the flush acknowledgment is eventually sent and permissions propagate globally.

*8) Permission Tracker Lifecycle:* ShareMMU invalidations are not permanent; an address can be invalidated and revalidated frequently, as described in Section II-A. When an address is invalidated, the invalidation is sent to both the PT and the accelerator. The accelerator must flush that entry, and the corresponding per-accelerator permission bit for that physical page in the PT (i.e., the Secure Vector) is cleared to no-access. As noted in Section V-C1, the Secure Vector itself is not flushed if other accelerators still use it. When the address is later revalidated for an accelerator, e.g., after an OS permission update, the accelerator's next access triggers a translation miss because it has already flushed that entry. A standard hardware page-table walk then retrieves the updated permissions, refills the private and shared TLBs, and updates the Permission Tracker entry, safely restoring access. Because revalidation reuses the normal page-walk path, the address space never shrinks and no "dead" entries accumulate.

*9) Toward Supporting Trusted Execution Environments (TEEs):* ShareMMU could also be extended to systems that mix Trusted Execution Environments (TEEs) with untrusted accelerators, which would require strict domain isolation in the memory-management layer. A promising direction is to tag each request with its process ID to identify the requesting security domain, and to isolate domains using either distinct Permission Tracker entries or nested two-stage translation tables. Such an approach could allow trusted and untrusted devices to coexist within a shared virtual address space while keeping secure memory regions accessible only to authorized TEE components. We leave a full design and evaluation of TEE-aware ShareMMU to future work.

### D. Mitigating Side-Channel Attacks

Since the ShareMMU design supports multi-context accelerators, translations from multiple distrusting processes can be co-located in the shared TLB. This creates potential vulnerabilities where an adversarial process could launch timing side-channel attacks, such as prime-probe, flush-reload, or occupancy-based attacks [38]–[41].

**Shared-TLB Prime+Probe attack scenario.** We describe an end-to-end attack on a shared TLB running mutually distrustful processes, where attacker P1 targets a victim P2's secret key. In the *Prime* stage, P1 constructs an eviction set and accesses the corresponding pages to fill the TLB. On a context switch, P2's data-dependent accesses run; processing a secret bit '1' maps to the primed set (loading P2's translations and evicting P1's entries due to the conflict), whereas a '0' maps elsewhere. In the *Probe* stage, P1 is rescheduled and times its primed pages: a high latency signals a miss, revealing a '1'. Repeating this cycle leaks P2's access pattern, a severe vulnerability when the shared TLB lacks isolation.

Our system also includes another shared structure, the Permission Tracker, which presents an additional attack target. To mitigate these threats, we propose ShareMMU-SP, which provides resiliency against such attacks.

*1) Securing Shared TLB:* ShareMMU-SP is inspired by SassCache [37], which secures shared LLCs. We introduce two steps to secure shared TLBs: (i) randomizing TLB-set selection with the *crypto-randomization* stage and (ii) limiting the TLB sets accessible to a process with the *crypto-partition* stage. We use an Index Derivation Function (IDF) built on the tweakable block cipher QARTA, similar to SassCache. The crypto-randomization step cryptographically selects a fixed number of random ways in the shared TLB (a TLB set), obfuscating the mapping between a virtual address and a TLB set. The crypto-partition step prevents a malicious process from gaining access to the entire shared TLB by cryptographically limiting the TLB sets available to that process, making occupancy-based attacks difficult. The replacement policy being used is random. A *coverage* parameter tunes a process's share of TLB entries and is an input to the IDF to manage per-process TLB allocation (Section V-D3).

As shown in Figure 12, the shared TLB is set-associative with four sets and four ways per set, where processes P1 and P2 are co-scheduled. Unique session keys, "key1" and "key2", are assigned to P1 and P2, respectively. The IDF takes the virtual page (with index), process key, and coverage parameter and performs crypto-randomization. This mapping makes it extremely difficult for an adversary to profile the TLBs and evict entries from another process. However, there is a non-zero probability of overlap between the two processes in a given way (marked in blue), but the probability is extremely low and insufficient for a meaningful attack (Section V-E).

*2) Securing Permission Tracker:* Since multiple processes also share the Permission Tracker, we design a mechanism that leverages the cryptographic hash function for shared TLB set selection to eventually secure the Permission Tracker against side-channel attacks. Since low-latency private TLBs

are typically flushed during context switches, while shared TLBs retain translations from multiple processes using process ID tagging [69], [70], [78], we leverage this behavior to protect the Permission Tracker by flushing it during a context switch. The Permission Tracker can be reconstructed using retained entries in the shared TLB and Alias Table. If this information is incomplete, the required permissions can be retrieved from the PTWC or the page table, similar to how private TLBs are refilled, and thus no additional complexity is introduced.

Figure 12 illustrates our mechanism, where Process P1 and Process P2 are running on the multi-accelerator system with accelerators `A` and `B`. Initially P1 is scheduled. The private TLBs and Permission Tracker have entries of P1. The shared TLB is secured using our crypto-randomization and crypto-partition stages. ❶ The OS switches the context; ❷ P1 is switched out, P2 is scheduled, the accelerators' private TLBs are flushed, and ShareMMU flushes the Permission Tracker. ❸ When P2 requests address translation, it misses in the private TLB but hits in the shared TLB. Since the shared TLB is protected against side channels, P2's access pattern cannot be leaked. ❹ ShareMMU creates a Permission Tracker entry for every incoming translation request from the accelerator. The ability to recreate the Secure Vector from a side-channel-resilient shared TLB enables flushing the Permission Tracker on context switches to ensure process isolation.

*3) Handling Occupancy Attacks:* An occupancy attack is a timing side-channel attack that exploits changes in the usage of shared hardware resources, such as a shared TLB. These attacks do not require precise address knowledge and instead rely on observing aggregate resource usage [56]–[58]. Figure 12 illustrates the partial overlap of TLB entries between two untrusted processes (highlighted in blue), which can be exploited for such attacks. The coverage represents the proportion of total TLB entries accessible to a process (its security domain). By controlling the coverage, a trade-off can be achieved between the security margin (less overlap) and resource utilization (more overlap). The impact of coverage on security is analyzed in detail in Section V-E.

The *Coverage* parameter ($t$) probabilistically determines the extent of overlap between processes. Coverage can be modeled as the probability of accessing at least one TLB set more than once when making $x + t$ accesses, where $x$ represents the total number of TLB sets, and $t$ is the coverage parameter. Assuming accesses are independent and with replacement, the expected coverage $C$ reflects the likelihood of overlapping entries across the $x+t$ accesses. Thus, the expected coverage $C$ is

$$C = 1 - \left(\frac{x-1}{x}\right)^{x+t}$$

Assuming a TLB configuration with 1024 entries and 8 ways, the IDF maps a 48-bit virtual address to 7-bit indices for set selection ($x$ = 128). The coverage can be computed as $C = 0.39, 0.63, 0.86$ for $t \in \{-64, 0, 64\}$. The parameter $t$ can be tuned to determine the share of TLB per process. As the number of processes increases, the fairness of TLB share

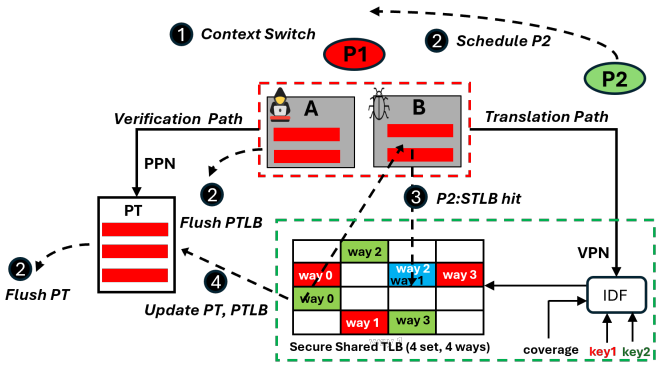


Fig. 12: ShareMMU-SP TLB set randomization.

improves, as multiple processes already limit each other's TLB share, and overlaps between multiple processes decrease, improving the security margin.

### *E. Security Analysis*

This section evaluates ShareMMU's security guarantees for access control and side-channel resiliency.

**Access Control.** ShareMMU identifies any access request not present in the Permission Tracker as malicious, effectively reducing the probability of unauthorized access to zero. In contrast, CryptoMMU, as reported in [12], demonstrated a probability of malicious access to be $2.98 \times 10^{-8}$, which is a significantly weaker security margin.

**Side Channel Security.** We evaluate the side-channel resiliency of a shared TLB with 1024 entries 8-way set associative. The victim's TLB entry must fall within the attacker's coverage across all $W$ ways to form an eviction set. The probability of such overlap, $P_{\text{overlap}}$, for *coverage* ($C = 0.39$) can be computed as:

$$P_{\text{overlap}} = C^W = (0.39)^8 \approx 5.3 \times 10^{-4}.$$

This implies the attacker can form an eviction set for approximately 1 in 53,000 sets. Even if the attacker achieves sufficient coverage to form an eviction set, the probability $P_{\text{VA}}$ of a virtual address mapping to the same TLB (128 set, 8-way) entry can be calculated as

$$P_{\text{VA}} = 2^{-(7\times 8)} = 2^{-56} \approx 1.39 \times 10^{-17}.$$

This probability is exceedingly small ($\approx 1.39 \times 10^{-17}$), making it effectively outside the virtual address space of modern systems and practically impossible to exploit. The combined probability of an attacker forming an eviction set and finding another virtual address that maps to the same TLB entry is:

$$P_{\text{combined}} = P_{\text{overlap}} \cdot P_{\text{VA}} \approx 7.35 \times 10^{-21}.$$

This security margin is well above the threshold of commercially available solutions [12], [79]. The Permission Tracker, on the other hand, is flushed on the context switch and refilled on demand through a secure shared TLB, making it side-channel resilient.

TABLE II: Simulation configuration

| | |
|---|---|
| **Processor** | |
| CPU | x86, OoO, 2.00GHz, 2 processes, context switch interval: 5 ms [36] |
| L1 Cache | 2-way, 32KB, 64B block, access latency: 2 cycles |
| L2 Cache | 256KB per slice, 32 slices, 8-way, 64B block, access latency: 20 cycles |
| **Accelerator** | |
| AFU | n = 4, 8, 16, 32 |
| Local TLB | 16-entry, 2-way, access latency: 1 cycle |
| Shared TLB | 1024-entry, 8-way, access latency: 3 cycles |
| **Security Schemes** | |
| Border Control | 1 KB BCC per AFU [36] |
| CryptoMMU | 20 cycles for hash generation [12] |
| ShareMMU | PT(256-entry, 16-way); access latency: 5 cycles |
| ShareMMU-SP | QARTA latency: 2.2 ns; coverage: 0.86 [37] |
| **IOMMU** | |
| Page size | 4KB |
| PTWC | 32-entry (8 per level), 5-cycles hit latency, 120-cycles miss latency (per level) |
| Mem. latency | 120 cycles [80] |

TABLE III: Description of workloads

| Domain | Apps (short name) | FUs |
|---|---|---|
| Medical Imaging | Denoise (*Deno*) | 2 |
| | Segmentation(*Seg*) | 1 |
| PARSEC | BlackScholes (*Bscho*) | 1 |
| | StreamCluster (*SClus*) | 5 |
| | Swaptions (*Swap*) | 4 |
| Computer Vision | Disparity Map (*DiMap*) | 4 |
| | LPCIP Desc (*LPCIP*) | 1 |
| Computer Navigation | EKF SLAM (*EKF*) | 2 |
| | Robot Localization (*Ro_Loc*) | 1 |

## VI. METHODOLOGY

**Infrastructure.** To evaluate ShareMMU, we use gem5 PARADE, a cycle-accurate full-system simulator with high-level synthesis support [42], [81]. As shown in Table II, we simulate an 8-issue out-of-order x86 CPU core at 2GHz running 2 processes with a context switch interval of 5 ms. The private TLBs are flushed, while the shared TLB is preserved on context switch. Each accelerator contains multiple Function Units (FUs) that accelerate different algorithms within an application. A collection of FUs composes an Accelerator Functional Unit (AFU) with a private TLB. Multiple AFU instances exploit parallelism and share a trusted 1024-entry, 8-way TLB. Each AFU has a 16-entry, 2-way private TLB. The Page Table Walk Cache (PTWC) is 32 entries (8 per level) [14].

**Configurations.** Our baseline design is unsecured and relies on caching pre-translated addresses within private (untrusted) and shared TLBs (trusted). There is no access control; accelerators can access memory with a pre-translated address. We model TLB shootdowns where the accelerator must write back dirty cache lines before flushing translations. We evaluate ShareMMU alongside CryptoMMU and Border Control, which we faithfully model in gem5 PARADE. The original Border Control design includes a shared TLB, whereas CryptoMMU does not. For a fair comparison, we model a shared TLB for CryptoMMU using the same organization as ShareMMU, but storing device-keyed entries. Because each CryptoMMU translation carries a device-specific hash, a translation shared by $k$ devices appears as $k$ duplicate entries, reducing effective shared-TLB coverage. On a permission update, the device key changes, invalidating all cached entries associated with that key; the corresponding hashes are then regenerated from the shared TLB or PTWC and re-sent to the private TLB. For Border Control, we model a physically tagged 1 KB BCC per accelerator and a protection table in memory (64B protection block size). We also model the latency of permission propagation to the BCC, protection table, and page table. The Permission Tracker and BCC are flushed on each context switch.

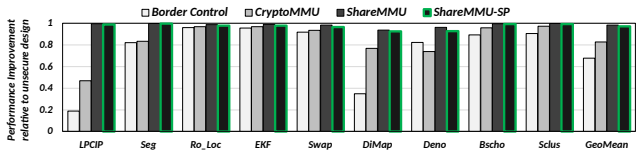


Fig. 13: Performance relative to unsecured baseline.

In addition to ShareMMU, which provides access control, we evaluate ShareMMU-SP, which provides both access control and side-channel resiliency. We use a 256-entry, 16-way set-associative Permission Tracker inclusive of 16 accelerators, each with a 16-entry private TLB. ShareMMU-SP uses a coverage of 0.86 with two processes running.

**Workloads.** We evaluate all approaches using accelerator benchmarks from PARADE [43]. Benchmarks span four domains: Medical Imaging, Computer Navigation, Computer Vision, and PARSEC, with varying numbers of Function Units (Table III). We use short names for each benchmark. Our benchmarks span multiple domains to capture the diverse memory demands of modern SoC systems. They cover a comprehensive spectrum of memory behaviors. The Medical Imaging domain has a large footprint with irregular access patterns. The Computer Navigation domain generates large but regular memory streams. Computer Vision imposes strict latency requirements, and PARSEC applications feature highly varied access patterns. Therefore, these workloads stress the address-translation and memory subsystems.

## VII. EVALUATION

### A. ShareMMU Performance

We compare ShareMMU, ShareMMU-SP, CryptoMMU, and Border Control against an unsecured baseline.

As shown in Figure 13, ShareMMU outperforms all compared designs that at least guarantee access control, achieving an average slowdown of 1.66% compared to the unsecured

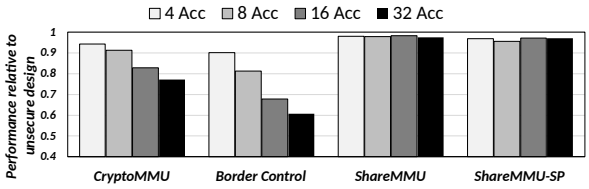


Fig. 14: Geometric mean sensitivity to varying accelerator count (higher is better).

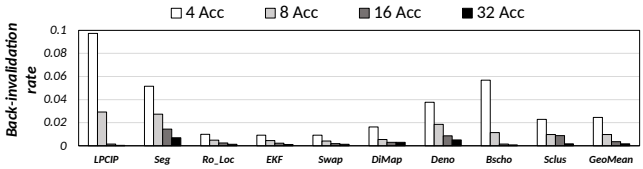


Fig. 15: Back-invalidation rate versus accelerator count (lower is better).

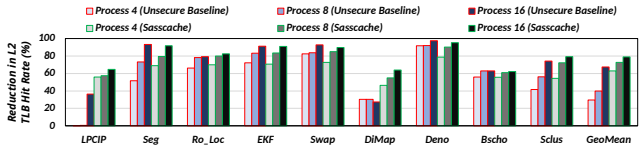


Fig. 16: Shared TLB hit rate under increasing concurrency.

baseline. ShareMMU-SP, which adds side-channel protection, has an average slowdown of 2.82%. Other approaches exhibit significant slowdowns. For example, CryptoMMU incurs a 17.21% slowdown due to reduced shared TLB coverage from duplicate translations and forced flushing of cached translations on permission updates. Similarly, Border Control slows down by 32.19%, primarily because coherent updates across the BCC, in-memory protection tables, PTWC, and TLBs introduce substantial overhead. ShareMMU remains very close to the unsecured baseline (typically 95–99% of baseline performance), indicating that PT lookups, access-mask updates, and back-invalidations add negligible overhead even under high accelerator concurrency.

The LPCIP and DiMap workloads are highly latency-sensitive, making them vulnerable to per-access overheads [14]. Border Control experiences severe slowdowns of 81.19% and 65.07%, while CryptoMMU incurs 53.01% and 23.09% degradation on these benchmarks. In contrast, ShareMMU introduces negligible overhead on LPCIP and only 6.27% slowdown on DiMap. The modest slowdown in DiMap arises from its very high rate of permission changes, which triggers frequent creation and updates of Secure Vectors. The shared TLB greatly benefits benchmarks like Denoise and Segmentation; accordingly, CryptoMMU degrades on these due to duplicate translations. For Segmentation and Denoise, Border Control slows down by 17.74% and 21.59%, and CryptoMMU by 16.13% and 35.19%, respectively. ShareMMU remains close to baseline with only 0.09% slowdown on Segmentation and 3.79% on Denoise. ShareMMU-SP shows similarly low overheads (0.23% and 7.29%). Other benchmarks, such as BlackScholes and Swaptions, have small memory footprints and show minimal differences across schemes; any slowdown is primarily tied to permission handling. Finally, benchmarks like Robot Localization and EKF SLAM, which are highly regular computational navigation workloads, also perform similarly across all approaches. Across all benchmarks, ShareMMU maintains near-baseline performance, handling permission churn without significant latency penalties.

### *B. Sensitivity Analysis*

We now conduct a sensitivity study of our design.

**Sensitivity of varying number of accelerators.** Figure 14 shows the geometric mean performance relative to the unsecured baseline across CryptoMMU, Border Control, ShareMMU, and ShareMMU-SP, for systems with 4, 8, 16, and 32 accelerators. CryptoMMU performance degrades as the number of accelerators increases, with slowdown rising from 5.79% (4 accelerators) to 22.82% (32), due to duplication from device-specific hashes and more translation flushes. Border Control performs worse, with slowdown ranging from 9.92% (4 accelerators) to 39.23% (32). While more accelerators improve parallelism, Border Control needs extra memory accesses to update more protection tables, hurting performance.

In contrast, both ShareMMU and ShareMMU-SP maintain consistently high performance, with slowdowns of approximately 2.03% and 3.17% at 4 accelerators and 2.50% and 2.93% at 32 accelerators, respectively. ShareMMU is largely insensitive to accelerator count. Although more accelerators increase request volume and contention at the Permission Tracker, this is offset by cross-accelerator sharing, which reduces redundant Secure Vector creation and lowers per-accelerator back-invalidation traffic. As a result, performance peaks at 16 accelerators: it improves over 8 because more accelerators collaborate on shared data chunks. It is still better than 32, where the available data parallelism in the workloads is saturated, and additional accelerators increase contention at the Permission Tracker. The average per-accelerator back-invalidation rates are 2.45%, 0.97%, 0.34%, and 0.16% for 4, 8, 16, and 32 accelerators, respectively, as shown in Figure 15. This declining rate reflects increased page sharing, letting a single Secure Vector represent more accelerators' permissions and expand Permission Tracker coverage.

**Multi-Process Sensitivity.** Figure 16 shows the percentage reduction in shared-TLB hit rate as the number of concurrent processes grows from 4 to 16, relative to a single-process baseline. As expected, increasing the number of processes degrades hit rates across all workloads because of the shared TLB's limited capacity. The unsecured baseline experiences an average reduction of 29.6% at 4 processes, worsening to 67.4% at 16 processes. ShareMMU-SP, with its SassCache-style protection, follows the same capacity-bound trajectory, degrading by 62.9% at 4 processes and 79.1% at 16 processes.

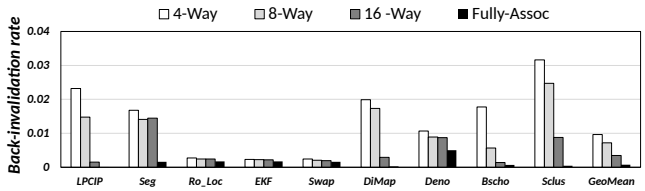


Fig. 17: Back-Invalidation sensitivity to Permission Tracker associativity.

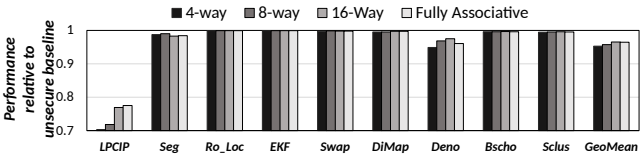


Fig. 18: Sensitivity to Permission Tracker associativity.

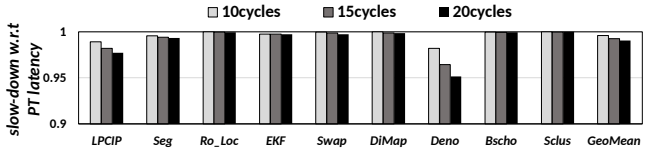


Fig. 19: Sensitivity to Permission Tracker latency.

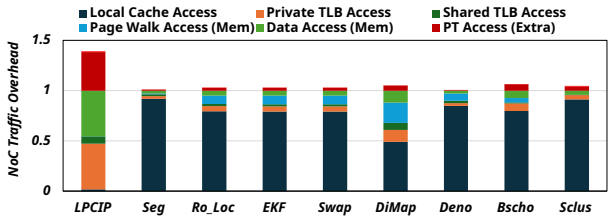


Fig. 20: Additional access control traffic on NoC.

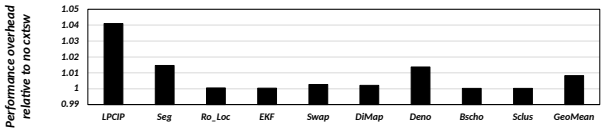


Fig. 21: Performance overhead of ShareMMU-SP for multi-context system.

The SassCache randomization causes a notably larger degradation than the unsecured baseline for LPCIP; conversely, for Denoise, Swaptions, and BlackScholes, it performs better than the baseline, while the remaining benchmarks are comparable. Although ShareMMU-SP's shared-TLB hit rate incurs an additional isolation cost at low process counts, this gap narrows sharply as processes increase (from roughly 33% at 4 processes to only 12% at 16), as capacity pressure rather than isolation becomes dominant. Thus, its protection does not bottleneck scalability, converging toward the unprotected baseline.

**Sensitivity to Permission Tracker associativity.** We evaluated the sensitivity of ShareMMU to varying Permission Tracker associativity in a system with 16 accelerators. While a fully associative design offers maximum flexibility, it is power-hungry and may not be practical. Figure 17 shows the private TLB back-invalidation rate due to Secure Vector eviction. The average rates are 0.07%, 0.34%, 0.71%, and 0.96% for fully associative, 16-way, 8-way, and 4-way designs, respectively. As expected, lower associativity increases back-invalidations.

Figure 18 presents the relative performance of ShareMMU under different associativities. The overheads are 5.04%, 1.66%, 5.89%, and 8.05% for fully associative, 16-way, 8-way, and 4-way designs, respectively. Higher associativity reduces back-invalidations but increases access latency, with 16-way associativity serving as the sweet spot. Segmentation and Denoise suffer from lower associativity due to their larger memory footprints, which lead to more conflicts and frequent evictions. Full associativity raises lookup latency, degrading performance. Other benchmarks are largely unaffected.

### C. ShareMMU overheads

We now discuss ShareMMU and ShareMMU-SP overheads.

**Stress testing critical path latency.** Figure 19 presents the performance overhead observed in ShareMMU when the latency of the Permission Tracker, which lies on the critical path, is artificially increased. Latencies of 10 cycles (2×), 15 cycles (3×), and 20 cycles (4×) yield slowdowns of 0.41%, 0.74%, and 0.96%, respectively. LPCIP and Denoise show the highest impact due to latency sensitivity and higher request rates, while other benchmarks show minimal impact.

**Overhead of frequent context switches.** We evaluate the overhead of flushing the private TLB and Permission Tracker on every context switch, assuming an aggressive 2 ms interval [82]. The PT is reconstructed from the shared TLB and alias table upon each switch, avoiding a full page table walk. As shown in Figure 21, LPCIP (due to latency sensitivity), Denoise, and Segmentation (due to high request rates and large memory footprints) exhibit the highest overheads of 4.1%, 1.4%, and 1.3%, respectively. All other benchmarks show negligible impact, with a geometric mean overhead of only 0.82%. The low overhead is because the shared TLB primarily handles the reconstruction of the Permission Tracker and the refill of private TLBs.

**Additional traffic on the NoC.** We evaluate the additional traffic introduced by the access-control mechanism in the accelerator subsystem. The baseline traffic comprises private-TLB, local-cache, shared-TLB, page-walk, and data accesses. The only additional traffic ShareMMU introduces is the Permission Tracker (PT) validation accompanying requests that miss in the local cache and reach memory, together with the back-invalidations triggered on PT eviction. As shown in Figure 20, this additional NoC traffic averages only 3.23% relative to the baseline (3.09% for PT validation and 0.14% for back-invalidation), confirming that ShareMMU does not double NoC traffic.

## VIII. Discussion

### A. Hardware/Software changes in ShareMMU

ShareMMU requires minimal hardware and software modifications compared to CryptoMMU and Border Control.

**Hardware Requirements.** ShareMMU's only hardware addition is a physically tagged Permission Tracker. It requires no changes to the TLB interface or TLB hit datapath (unlike CryptoMMU), making it compatible with existing accelerator designs. The Permission Tracker also maintains the VPN and Access Mask bits to support targeted back-invalidations using existing TLB shootdown mechanisms [70]–[73].

**Software Requirements.** When aliases are present, each secure vector references the Alias Table and includes an alias presence bit and an alias counter. Alias tracking is standard in systems with virtual caches or inverted page tables [83], [84]. ShareMMU leverages existing OS support to update these fields with minimal or no additional software changes. In contrast, Border Control requires significant OS modifications to manage and synchronize multiple large protection tables.

### B. Power and Storage Overheads

The Permission Tracker is a 256-entry, 16-way set-associative structure sized for 16 accelerators. Each entry includes a 36-bit PPN, a 16-bit permission vector, a 16-bit access mask, another 36-bit field, and status flags—about 13 bytes per entry—totaling approximately 3.25 KB of SRAM. Read and write accesses consume on the order of 1–2 picojoules, keeping dynamic power in the sub-milliwatt range. Overall, the Permission Tracker is lightweight, adding negligible storage and power overhead.

### C. Scalability of ShareMMU

ShareMMU requires negligible storage overhead that scales linearly with the number of accelerators in the system. The Permission Tracker can be provisioned for the maximum accelerator count required by the product. For large-scale SoCs (e.g., 64-128 accelerators), the overhead remains practical.

## IX. Related Work

**Intel VT-d Host Permission Table.** Intel VT-d now supports the Host Permission Table (HPT) to enforce access control for device-issued requests that use pre-translated addresses [85]. HPT is a four-level, software-managed structure indexed by device physical address; Page Permission (PPi) bits specify read, write, and atomic access. Access is validated via a hierarchical HPT walk and supports multiple page sizes. While secure, HPT introduces storage, lookup, and synchronization overhead because it must remain coherent with the page table, similar to Border Control.

**Advancing Secure TLBs.** Secure TLBs propose static and random fill mechanisms [86]. Static fill statistically partitions TLBs among security domains to limit interference, but suffers from low utilization. Random fill introduces nondeterminism by filling a random TLB entry while the correct translation is sent to the processor, adding overhead. ShareMMU-SP addresses these issues with high-entropy, low-overhead cryptographic randomization and security-domain partitioning. While SassCache [37] provides side-channel protection for shared caches against malicious processes, ShareMMU-SP offers a more comprehensive solution by securing both the shared TLB and the Permission Tracker.

**IOMMU Technology: Beyond Default Configurations.** The IOMMU has traditionally been utilized to augment system security and ensure device compatibility, providing essential features such as device isolation and DMA remapping [87]. Lately, its use has expanded, facilitating address translation for domain-specific accelerators [88]. These accelerators are built for a broad spectrum of emerging applications, each with unique demands, including stringent energy and power constraints [89], time-critical operations [90], extensive computational intensity, and security margins. This diversification has stimulated innovations in IOMMU technology, pushing it beyond standard configurations. Modifications to the IOMMU are achievable at both the hardware and software levels, allowing for flexible adaptation to these evolving requirements.

The integration of IOMMU in Linux systems is straightforward. It is enabled via kernel parameters, such as intel_iommu=on and amd_iommu=on in the GRUB configuration [91]. Innovations such as CryptoMMU have emerged, leveraging cryptographic methods to enhance access control within third-party accelerators. This approach involves purely hardware-based alterations to the IOMMU, facilitating the authentication of memory access from pre-translated addresses [12]. On the other hand, Border Control introduces a hybrid strategy of hardware and software modifications to the IOMMU to enable access control for third-party accelerators. This is achieved through a hardware-based cache, the "Border Control Cache", and a software-managed protection table [36]. Conversely, rIOMMU [49] represents a software-centric solution designed to optimize performance, particularly in IO-intensive workloads. It amortizes the cost of TLB invalidation, trading security for performance.

## X. Conclusion

Modern SoCs increasingly deploy third-party accelerators alongside trusted hosts. This paper introduced **ShareMMU**, a practical IOMMU that securely enables translation reuse across untrusted accelerators while preserving performance and scalability. ShareMMU tracks permissions for translations sent to private TLBs via a lightweight Permission Tracker, enabling: (1) secure reuse of pre-translated addresses without modifying TLB interfaces or hit datapaths, (2) strong access control by validating every accelerator request against cached permissions, (3) low-overhead permission updates via inclusive tracking and targeted back-invalidations, and (4) side-channel resiliency through ShareMMU-SP, which hardens shared TLBs and isolates the Permission Tracker. ShareMMU incurs only 1.66% average slowdown over an unsecured baseline; ShareMMU-SP adds side-channel protection at 2.82% slowdown with minimal hardware and software changes.

## Acknowledgment

We sincerely thank the anonymous reviewers from MICRO 2026 for their insightful feedback. This work is supported in part by the National Science Foundation (NSF) under Grant No. CNS-2350230. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

## References

[1] NVIDIA Corporation, *NVIDIA DGX SuperPOD Reference Architecture DGX H100*, 2024, accessed: 2026-09-10. [Online]. Available: https://docs.nvidia.com/dgx-superpod-reference-architecture-dgx-h100.pdf

[2] Google Cloud, *Google TPU Pods Architecture and Performance*, 2024, accessed: 2026-09-10. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[3] L. Wan, W. Zheng, and X. Yuan, "Efficient inter-device task scheduling schemes for multi-device co-processing of data-parallel kernels on heterogeneous systems," *IEEE Access*, vol. 9, pp. 59 968–59 978, 2021.

[4] C. Sau and F. Palumbo, "Automatic generation of dataflow-based reconfigurable co-processing units," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, 2014.

[5] V. Leon, C. Bezaitis, G. Lentaris, D. Soudris, D. Reisis, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU co-processing in space applications: Development and testing with DSP/AI benchmarks," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021, pp. 1–5.

[6] P. Blinzer, "The heterogeneous system architecture: It's beyond the gpu," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. iii–iii.

[7] J. Lee, Z. Liu, X. Tian, D. H. Woo, W. Shi, and D. Boumber, "Acceleration of bulk memory operations in a heterogeneous multicore architecture," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 423–424.

[8] Y. Zhou, X. Dong, T. Meng, M. Tan, B. Akin, D. Peng, A. Yazdanbakhsh, D. Huang, R. Narayanaswami, and J. Laudon, "Towards the co-design of neural networks and accelerators," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 141–152.

[9] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High performance binary neural networks on the Xeon+FPGA™ platform," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[10] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security implications of third-party accelerators," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 50–53, 2016.

[11] S. K. K., S. Sahoo, A. Mahapatra, A. K. Swain, and K. Mahapatra, "A flexible pay-per-device licensing scheme for fpga ip cores," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 677–682.

[12] F. Alam, H. Lee, A. Bhattacharjee, and A. Awad, "Cryptommu: Enabling scalable and secure access control of third-party accelerators," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 32–48. [Online]. Available: https://doi.org/10.1145/3613424.3614311

[13] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, "NeuMMU: Architectural support for efficient address translations in neural processing units," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1109–1124. [Online]. Available: https://doi.org/10.1145/3373376.3378494

[14] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting address translation for accelerator-centric architectures," in , 02 2017.

[15] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular GPU applications," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 180–192.

[16] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU memory manager with application-transparent support for multiple page sizes," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 136–150.

[17] S. Shin, M. LeBeane, Y. Solihin, and A. Basu, "Neighborhood-aware address translation for irregular GPU applications," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 352–363.

[18] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "SnakeByte: A TLB design with adaptive and recursive page merging in GPUs," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1195–1207.

[19] B. Li, J. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-GPUs via sharing and spilling aware TLB design," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1154–1168. [Online]. Available: https://doi.org/10.1145/3466752.3480083

[20] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, "Scalable distributed last-level TLBs using low-latency interconnects," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 271–284. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00030

[21] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu, and L. Benini, "Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, sep 2017. [Online]. Available: https://doi.org/10.1145/3126560

[22] Z. Fang, F. Javadi, J. Cong, and G. Reinman, "Understanding performance gains of accelerator-rich architectures," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, 2019, pp. 239–246.

[23] B. Li, Y. Wang, and X. Tang, "Orchestrated scheduling and partitioning for improved address translation in GPUs," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[24] D. Channon and D. Koch, "Performance analysis of re-configurable partitioned TLBs," in *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, vol. 5, 1997, pp. 168–177 vol.5.

[25] A. Basu, S. Puthoor, S. Che, and B. M. Beckmann, "Software assisted hardware cache coherence for heterogeneous processors," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 279–288. [Online]. Available: https://doi.org/10.1145/2989081.2989092

[26] M. Harris, "Unified memory for cuda beginners: A note on concurrency," 2017, accessed: 2026-09-10. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/#:~:text=On%20Pascal%20and%20later%20GPUs%2C%20the%20CPU%20and%20the%20GPU,conditions%20caused%20by%20simultaneous%20accesses.

[27] C. F. Schimmel, "Method for managing concurrent access to virtual memory data structure," US Patent OO6 496 909B1, 2002.

[28] Y. Li, R. Melhem, and A. K. Jones, "Ps-tlb: Leveraging page classification information for fast, scalable and efficient translation for future cmps," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, jan 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400687

[29] B. Luo, Y. Li, L. Wei, and Q. Xu, "On functional test generation for deep neural network ips," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1010–1015.

[30] S. Gundabolu and X. Wang, "On-chip data security against untrustworthy software and hardware ips in embedded systems," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 644–649.

[31] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[32] S. Zeitouni, G. Dessouky, and A.-R. Sadeghi, "SoK: On the security challenges and risks of multi-tenant FPGAs in the cloud," 2020.

[33] N. Pundir, F. Rahman, F. Farahmandi, and M. M. Tehranipoor, "What is all the FaaS about? - remote exploitation of FPGA-as-a-Service platforms," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 746, 2021.

[34] Y. Luo, C. Gongye, Y. Fei, and X. Xu, "DeepStrike: Remotely-guided fault injection attacks on DNN accelerator in cloud-FPGA," 12 2021, pp. 295–300.

[35] F. Hategekimana, J. Mandebi Mbongue, M. J. H. Pantho, and C. Bobda, "Secure hardware kernels execution in CPU+FPGA heterogeneous cloud," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 182–189.

[36] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 470–481.

[37] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2273–2287.

[38] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. SEC'18. USA: USENIX Association, 2018, p. 955–972.

[39] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 989–1007. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/tatar

[40] T. Kim, H. Park, S. Lee, S. Shin, J. Hur, and Y. Shin, "Devious: Device-driven side-channel attacks on the IOMMU," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2288–2305.

[41] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "Tunnels for bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 960–974. [Online]. Available: https://doi.org/10.1145/3576915.3616672

[42] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[43] J. Cong, Z. Fang, M. Gill, and G. Reinman, "PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 380–387.

[44] M. Hill and V. Janapa Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 317–330.

[45] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for GPU accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3480855

[46] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in CPU-GPU unified memory," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1212–1217.

[47] W. Zhu, G. Cox, J. Vesely, M. Hairgrove, A. L. Cox, and S. Rixner, "UVM discard: Eliminating redundant memory transfers for accelerators," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, 2022, pp. 27–38.

[48] Z. Ma, Y. Tan, H. Jiang, Z. Yan, D. Liu, X. Chen, Q. Zhuge, E. H.-M. Sha, and C. Wang, "Unified-TP: A unified TLB and page table cache structure for efficient address translation," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 255–262.

[49] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "RIOMMU: Efficient IOMMU for I/O devices that employ ring buffers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2015, p. 355–368.

[50] P. Vogel, A. Marongiu, and L. Benini, "Exploring shared virtual memory for FPGA accelerators with a configurable IOMMU," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 510–525, 2019.

[51] Binuraj Ravindran,, "Intel® In-Memory Analytics Accelerator Plugin for RocksDB Storage Engine," Intel, Tech. Rep., 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/788607/intel-in-memory-analytics-accelerator-plugin-for-rocksdb-storage-engine-intel-iaa-plugin-for-rocksdb-storage-engine.html

[52] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, "Hardware-based address-centric acceleration of key-value store," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 736–748.

[53] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in gpus for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370. [Online]. Available: https://doi.org/10.1145/3373376.3378529

[54] S. Shin, G. Cox, M. Oskin, G. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular gpu applications," 06 2018, pp. 180–192.

[55] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3196003

[56] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[57] M. K. Qureshi, "Ceaser: mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 775–787. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00068

[58] ——, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 360–371.

[59] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A low-overhead access obfuscation for trusted memories," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 107–119.

[60] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 94–106.

[61] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[62] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2019, p. 157–168.

[63] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 403–415.

[64] Z. Allaf, M. Adda, and A. E. Gegov, "A comparison study on flush+reload and prime+probe attacks on aes using machine learning approaches," in *Advances in Computational Intelligence Systems - Contributions Presented at the 16th UK Workshop on Computational Intelligence, September 7-9, 2016, Lancaster, UK*, ser. Advances in Intelligent Systems and Computing, P. Angelov, A. E. Gegov, C. Jayne, and Q. Shen, Eds., vol. 513. Springer, 2017, pp. 203–213. [Online]. Available: https://doi.org/10.1007/978-3-319-66939-7_17

[65] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 987–1002.

[66] P. Vogel, A. Marongiu, and L. Benini, "Exploring shared virtual memory for FPGA accelerators with a configurable iommu," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 510–525, 2019.

[67] G. Schieffer, J. Wahlgren, J. Ren, J. Faj, and I. B. Peng, "Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper," in *International Conference on Parallel Processing*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271088381

[68] B. Gao, Q. Kang, H.-W. Tee, K. T. N. Chu, A. Sanaee, and D. Jevdjic, "Scalable and effective page-table and tlb management on numa

systems," in *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'24. USA: USENIX Association, 2024.
[69] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 62–63.
[70] S. Maass, M. K. Kumar, T. Kim, T. Krishna, and A. Bhattacharjee, "Ecotlb: Eventually consistent tlbs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Sep. 2020. [Online]. Available: https://doi.org/10.1145/3409454
[71] I. Corporation, *Remote Action Request*, Intel Corporation, Santa Clara, CA, 2021. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf
[72] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 27–39.
[73] 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 *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 340–349.
[74] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain, "Tlb shootdown mitigation for low-power many-core servers with l1 virtual caches," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 17–20, 2018.
[75] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 90–102.
[76] K. Elphinstone, S. Russell, and G. Heiser, "Issues in implementing virtual memory," University of New South Wales, School of Computer Science and Engineering, Tech. Rep. UNSW–CSE–TR–9411. [Online]. Available: https://cgi.cse.unsw.edu.au/~reports/papers/9411.pdf
[77] B. Jacob, "Segmented addressing solves the virtual cache synonym problem," University of Maryland, College Park, Dept. of Electrical & Computer Engineering, Tech. Rep. UMD-SCA-1997-01. [Online]. Available: https://user.eng.umd.edu/~blj/papers/UMD-SCA-97-01.pdf
[78] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John, "Csalt: Context switch aware large tlb," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 449–462.
[79] Qualcomm Inc., "Pointer Authentication on ARMv8.3," Qualcomm Inc., Tech. Rep., 2017.
[80] S. Dublish, V. Nagarajan, and N. Topham, "Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in gpus," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 239–248.
[81] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.
[82] N. Jammula, M. Qureshi, A. Gavrilovska, and J. Kim, "Balancing context switch penalty and response time with elastic time slicing," in *2014 21st International Conference on High Performance Computing (HiPC)*, 2014, pp. 1–10.
[83] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 90–102.
[84] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 297–308.
[85] I. Corporation, "Intel® virtualization technology for directed i/o architecture specification," Intel Corporation, Tech. Rep., August 2024.
[86] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 346–359. [Online]. Available: https://doi.org/10.1145/3307650.3322238
[87] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU scalably," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 549–562. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/peleg
[88] H.-C. Fu, P.-H. Wang, and C.-L. Yang, "Active forwarding: Eliminate IOMMU address translation for accelerator-rich architectures," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.
[89] T. Tang, S. Li, L. Nai, N. Jouppi, and Y. Xie, "NeuroMeter: An integrated power, area, and timing modeling framework for machine learning accelerators (industry track paper)," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 841–853.
[90] "IEEE Draft Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications," *IEEE P802.1AS-Rev/D6.0 December 2017*, pp. 1–496, 2018.
[91] Red Hat, Inc, "Configuring a host for PCI passthrough," 2022, accessed: 2026-09-10. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_virtualization/4.1/html/installation_guide/appe-configuring_a_hypervisor_host_for_pci_passthrough

## APPENDIX

### A. Abstract

This artifact provides gem5/PARADE models, full-system checkpoints, nine accelerator workloads, a containerized legacy toolchain, experiment drivers, metric extractors, and SVG plotting tools for ShareMMU. The scoped workflow reproduces Figures 1, 13, and 20 from simulator outputs for Unsecure, Border Control, CryptoMMU, ShareMMU, and ShareMMU-SP.

### B. Artifact check-list (meta-information)

- **Program:** gem5/PARADE full-system simulator and nine accelerator workloads.
- **Artifact version:** 3.0.0.
- **Compilation:** GCC 4.8.4, Python 2.7, SCons 1.3.1, SWIG 2.0.9, and Protocol Buffers 2.5.0 inside Docker.
- **Binary:** locally built `gem5.opt` for five variants.
- **Model:** x86 full-system SoC with shared/private TLBs, accelerators, and the ShareMMU Permission Tracker.
- **Data set:** checkpoint, disk images, boot scripts, and nine supplied workloads.
- **Run-time environment:** x86-64 Linux and Docker Engine.
- **Hardware:** no GPU, FPGA, or KVM; 8 GB RAM for smoke, 32 GB or more recommended for concurrent reproduction.
- **Execution:** Docker-first Make workflows.
- **Metrics:** cycles, normalized performance, TLB/PTrack statistics, back-invalidations, and normalized NoC traffic.
- **Output:** logs, gem5 statistics, CSV summaries, and SVGs.
- **Disk space:** 20 GB smoke; 60 GB minimum and 100 GB recommended for full reproduction.
- **Preparation time:** approximately 20–30 minutes on a large server for image, assets, and initial build.
- **Experiment time:** approximately 30–60 minutes for `kick_the_tires`; 12–18 hours for a highly parallel full run.
- **Publicly available:** Zenodo record.
- **License:** Apache-2.0 for artifact-authored material; bundled components retain upstream terms.
- **Workflow automation:** Docker, Make, Bash, and Python.
- **Archived:** DOI 10.5281/zenodo.21543532.

### C. Description

*1) How to access:* Download and unpack the archived artifact, then select a writable directory:

```
export SHAREMMU_WORKDIR=/path/with/space
```

*2) Hardware dependencies:* An x86-64 Docker host is required. Each simulation is primarily single-threaded; additional cores and memory permit workload/variant concurrency. No special accelerator hardware is needed.

*3) Software dependencies:* Only Docker, Make, Bash, and standard Linux utilities are required on the host. Docker supplies the pinned legacy compiler and simulator toolchain.

*4) Data sets and models:* The workflow downloads and verifies a 224 MiB full-system archive, expanding to approximately 4.1 GB. The package contains five simulator variants, checkpoints, and workload boot scripts. Assets and results remain under the caller-selected work directory. The bundled design/update guide documents the request path, model boundaries, and update procedure.

### D. Installation

Run the verbose setup test:

```
make kick_the_tires
```

It builds one Docker image and all five variants, runs BlackScholes, prints cycle/normalized values, and generates a four-bar smoke Figure 13. Success ends with `KICK_THE_TIRES_PASS`; failures identify the stage and produce a diagnostics bundle.

### E. Experiment workflow

The automated sequence builds the image and gem5 variants; checksums assets; restores the checkpoint; runs workloads; verifies normal exits; extracts cycles and NoC statistics; and renders SVGs. Each result records the exact command, artifact version, wall time, configuration, log, and statistics.

### F. Evaluation and expected results

*a) Full campaign.:*

```
make docker-reproduce-fast
```

The command requires 45 normal completions (five variants times nine workloads), then emits cycle, normalized-performance, and geometric-mean CSVs. Figure 1 plots Border Control and CryptoMMU; Figure 13 plots Border Control, CryptoMMU, ShareMMU, and ShareMMU-SP. All values are normalized to Unsecure. The SP model uses randomized shared-TLB replacement. Swap, DiMap, and Deno may be up to approximately four percentage points higher than the paper; the geometric mean remains within approximately one percentage point.

*b) NoC traffic.:* Figure 20 contains nine normalized stacks for local cache, private/shared TLBs, page walks, data memory, PTrack, and back-invalidations. The expected headline result is approximately 3.23% geometric-mean additional traffic.

*c) Generated figures.:*

```
results/figures/figure01_prior_perf.svg
results/figures/figure13_performance.svg
results/figures/figure20_noc_traffic.svg
```

### G. Experiment customization

`MAX_JOBS` controls workload concurrency and `VARIANT_JOBS` controls variant concurrency. `VARIANTS` and `BENCHMARKS` restrict execution. Set `SKIP_EXISTING=1` to reuse only normally completed runs with matching experiment metadata.

### H. Notes

CryptoMMU/LPCIP is a known long-running workload. One variant can produce several GB of logs. The package performs structural completeness checks and reports generated metrics and figures directly.

### I. Methodology

Submission, reviewing, and badging methodology:

- ACM artifact review and badging policy
- cTuning AE methodology