Beyond the Bermuda Triangle of Contention: IOMMU Interference in Mixed Criticality Systems

1st Diogo Costa
Centro ALGORITMI / LASI
Universidade do Minho
id10560@alunos.uminho.pt

2nd José Martins
Centro ALGORITMI / LASI
Universidade do Minho
jose.martins@dei.uminho.pt

3rd Sandro Pinto
Centro ALGORITMI / LASI
Universidade do Minho
sandro.pinto@dei.uminho.pt

***Abstract*—As Mixed Criticality Systems (MCSs) evolve, they increasingly integrate heterogeneous computing platforms that combine general-purpose processors with specialized accelerators such as AI engines, Graphics Processing Units (GPUs), and high-speed networking interfaces. This heterogeneity introduces challenges, as Direct Memory Access (DMA)-capable devices act as independent bus masters that directly access shared memory. Ensuring both security and timing predictability in such environments is critical. The Input-Output Memory Management Unit (IOMMU) addresses these concerns by enforcing memory isolation and access control. While prior work has examined the IOMMU from a security perspective, highlighting side-channel vulnerabilities rooted in shared caching structures, these same architectural resources can also undermine system safety by introducing timing unpredictability due to shared resources contention. In time-sensitive workloads, such interference can hinder the system's ability to meet real-time guarantees. In this work, we analyze IOMMU contention using the Xilinx UltraScale+ ZCU104 platform, which integrates an Arm SMMUv2-compliant IOMMU. We reverse-engineer the IOMMU's internal translation structures and use microbenchmarking to demonstrate how shared components, particularly the IOTLB, introduce measurable latency under concurrent DMA activity. Our results show that translation can induce DMA transaction delays that can arise up to 1.79×. These findings emphasize the need to consider IOMMU-induced contention when designing real-time MCS workloads on modern embedded platforms.**

***Index Terms*—IOMMU, Interference, Contention, Mixed-Criticality Systems, Virtualization**

I. Introduction

In recent decades, the trend toward digitization [1], [2] has reshaped numerous industries, including automotive, robotics, and aerospace. This evolution has rapidly increased in system complexity, with high-end embedded platforms evolving from simple, single-core Microcontroller Units (MCUs) into complex, heterogeneous architectures [3]. These modern systems integrate multi-core CPUs and specialized hardware accelerators such as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), Neural Processing Units (NPUs), and Field-Programmable Arrays (FPGAs) [4], [5], enabling unprecedented levels of performance and functionality. At the same time, the demand for integration and efficiency has driven the consolidation of multiple workloads onto a single hardware platform, as integrating peripherals and computational tasks helps meet stringent Size, Weight, Power, and Cost (SWaP-C) requirements. This shift gave rise to Mixed Criticality Systems (MCSs), where systems with different criticality levels co-exist on the same hardware [6].

Consolidating mixed-criticality workloads onto a shared platform presents significant challenges in ensuring safety, security, and isolation [3], [7]. To address these challenges, virtualization has emerged as a key technology. Static Partitioning Hypervisors (SPH) [8]–[11] represent the zeitgeist of MCSs hypervisors, as they leverage static resource allocation to ensure isolation and determinism [12], [13]. While SPHs effectively enforce spatial isolation by partitioning architectural resources (e.g. memory regions and devices) among Virtual Machines (VMs), they cannot fully guarantee temporal isolation, as microarchitectural resources, including the Last-Level Cache (LLC), main memory, and system bus, remain inherently shared. The resulting resource contention introduces two key challenges: (i) increased execution time and (ii) reduced determinism, caused by unpredictable delays [14]–[18]. These timing variations pose a significant challenge in hard real-time systems, where strict Worst-Case Execution Time (WCET) guarantees are essential to ensuring system correctness and safety [17], [19].

Over the past decade, considerable research has been conducted to identify and mitigate interference caused by shared micro-architectural resources. The majority of this work has focused on three primary sources of contention: (i) the LLC [20]–[22], (ii) the system bus [23]–[26], and (iii) the main memory [23], [27]–[31], [31] - forming what we refer to as the *'Bermuda Triangle of Contention'*. While these sources have been well studied, modern platforms incorporate a broader array of shared resources (e.g., interrupt controllers [3]), whose effects on contention remain insufficiently explored.

Among these overlooked components, the Input/Output Memory Management Unit (IOMMU) plays a critical role in enforcing memory isolation for non-CPU bus masters, preventing unauthorized memory access and enhancing system security [32], [33]. While IOMMUs are primarily recognized for these security functions, existing research has focused on a narrow set of concerns, mainly their susceptibility to side-channel attacks such as Input/Output Translation Lookaside Buffer (IOTLB)-based timing attacks [32], [34]. In contrast, the implications of IOMMU design and usage on timing predictability in consolidated MCSs remain largely unexplored. In such systems, where multiple devices may simultaneously

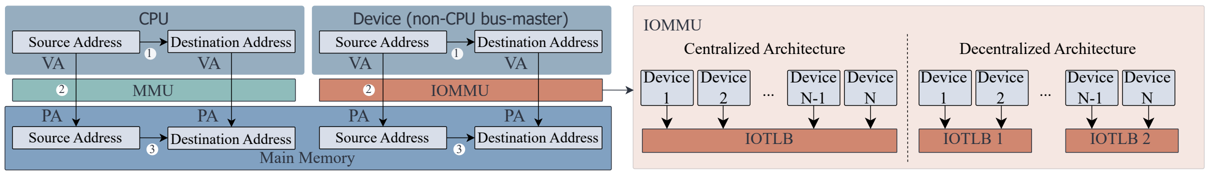


Fig. 1. Conceptual overview of address translation for CPU and non-CPU bus-master devices, comparing MMU and IOMMU operations.

share IOMMU resources, contention in translation structures such as address translation overheads and IOTLB cache pressure can lead to unpredictable latency, undermining the determinism required for real-time applications.

This oversight is concerning, as such contention not only impacts system behavior but can also undermine timing determinism in real-time workloads. Despite the critical role of the IOMMU in MCSs, there has been little investigation into how contention for its internal resources may affect execution latency and predictability. In this paper, we aim to advance the understanding of these effects through an empirical study of contention in a representative Commercial Off-The-Shelf (COTS) platform: the Xilinx UltraScale+ ZCU104. This platform integrates an IOMMU compliant with the ARM SMMU-500 (SMMUv2) specification. a decentralized architecture in which multiple Translation Buffer Units (TBUs) handle address translation for device clusters. We adapt established microbenchmarking techniques to the IOMMU domain to quantify the impact of shared translation structures under concurrent DMA traffic. Our contributions are as follows:

1) **Reverse-Engineering IOMMU Structures for Contention Analysis** – In this work, we reverse-engineer the translation structures of the IOMMU implementation found in the ZCU104 platform. By uncovering the microarchitectural details of the IOMMU, we gain valuable insights into how translation caches, like the IOTLB, contribute to contention and DMA performance degradation. This reverse-engineering effort enhances our understanding of IOMMU internals and provides procedures that can be applied in future research to analyze IOMMU contention across other platforms and architectures.
2) **Characterizing Contention Effects on Translation Latency** – Our experimental results show that IOMMU contention can introduce measurable latency increases (up to 1.79×) in DMA transactions. While moderate in magnitude, these delays may affect fine-grained timing behavior in real-time systems. We discuss these effects from a microarchitectural perspective and defer broader system-level implications to future work.
3) **Open-Source Artifacts for Independent Validation and Further Exploration** – To foster further research and validation, we make all experimental artifacts, including code, configurations, and data, publicly available to the research community. enabling independent validation and serving as a foundation for further research on IOMMU behavior across other hardware and system architectures.

II. Background

Efficient Direct Memory Access (DMA) has become essential in modern computing, allowing high-speed data transfers and improving overall system performance. Devices with DMA capabilities, such as hardware accelerators, rely on direct access to system memory, improving data processing speed and operational efficiency [35]. However, this direct access raises concerns about security and data integrity, particularly in consolidated computing environments where multiple systems share the same hardware. Ensuring both workload-specific isolation and system-wide protection requires robust mechanisms to control and restrict memory access. IOMMUs are pivotal to protect against such threats.

As shown on the left side of Figure 1, when a CPU issues a memory request, it uses a Virtual Address (VA) (1), which the MMU translates to a Physical Address (PA) (2) by consulting its page tables, typically via a Page Table Walk (PTW). This PA is then used to access the appropriate location in main memory (3). Similarly, on the right side, when a non-CPU bus master (e.g., a DMA-capable device) initiates a memory transaction, it uses a virtual address (1) within its assigned I/O address space. This address is intercepted by the IOMMU, which performs the translation to a physical address (2), accessing the I/O page tables via a PTW to enforce isolation and access control policies. The resulting PA is used to access the corresponding memory location (3). To perform these translations efficiently, the IOMMU maintains I/O page tables, analogous to those used by the MMU. Additionally, it uses IOTLBs to cache recently translated address mappings and reduce translation latency. The right side of Figure 1 highlights two architectural models: (i) a centralized design, where a single shared IOTLB serves all devices; and (ii) a decentralized approach, where multiple IOTLBs are distributed across device clusters. These designs offer trade-offs in latency, scalability, and isolation granularity. By enforcing strict access control and isolating memory domains across devices, the IOMMU significantly enhances system reliability and security.

While the security benefits of IOMMUs are well-documented, their impact on performance remains an underexplored area, particularly in MCSs. The overhead introduced by address translation, permission enforcement, and IOTLB management can create latency and contention when multiple devices share the same IOMMU. This contention is especially concerning for real-time and safety-critical applications, where predictable timing is crucial. To provide context for our study, we briefly review the ARM System Memory Management Unit (SMMU) architecture, which serves as the reference IOMMU implementation in our evaluation. While other specifications exist, including Intel VT-d [36], AMD-Vi [37], and the RISC-V IOMMU [38], this work focuses specifically on ARM's specification. The ARM SMMUv2 [39] is the standard IOMMU implementation for arm64 systems, providing secure memory isolation and efficient address translation. It employs a stream-based model, using StreamIDs and optional SubstreamIDs for fine-grained device identification. SMMUv2 supports both stage-1 and stage-2 address translations and integrates tightly with ARM's memory architecture. To improve lookup efficiency, it integrates IOTLBs at the interface between devices and the SMMU, reducing memory access overhead. Additionally, it defines three circular buffer queues for translation updates, fault handling, and page requests.

IOTLBs play a crucial role in accelerating address translation and reducing memory overhead, but their organization varies across IOMMU implementations. Centralized designs, such as those in Intel VT-d and AMD-Vi, simplify caching but can become bottlenecks under high traffic. In contrast, architectures like Arm's CoreLink MMU-500 and MMU-600, as well as the RISC-V IOMMU open-source implementation from Zero-Day Labs [40], support more flexible or fully distributed IOTLB configurations. While decentralized designs improve scalability and reduce lookup latency, they also introduce challenges in maintaining coherence and avoiding contention. These issues are not limited to specialized systems but extend to a wide range of COTS platforms. With growing adoption of distributed IOMMUs (e.g., Arm forecasts over 330 million MMU-600-based systems [41]), contention due to frequent invalidations, synchronization overhead, and I/O-intensive workloads becomes a critical concern. This is particularly relevant in MCSs, where timing predictability is essential, and in virtualized environments where devices assigned to different VMs compete for translation resources. Although this work focuses on a specific platform, our findings offer broader insight into the performance and scalability trade-offs posed by modern IOMMU designs.

III. Methodology and Experimental Setup

This section presents our methodology for evaluating IOMMU contention. Our goal is to determine whether IOTLB resources are a significant source of contention and how their capacity affects translation overhead under varying workloads. To do this, we use micro-benchmarking and performance monitoring to analyze cache behavior and identify performance limitations in modern IOMMU designs.

Hardware Platform. Experiments were carried out on the Xilinx ZCU104 evaluation board, which integrates a Zynq UltraScale+ System on Chip (SoC) and serves as a representative platform for studying IOMMU architectures. It features a quad-core Arm Cortex-A53 processor (1.2 GHz) and the Arm CoreLink MMU-500, an implementation of SMMUv2 running at 525 MHz. The SoC also includes an FPGA (100 MHz) and two DMA engines: one in the Full-Power Domain (FPD) (600 MHz) and one in the Low-Power Domain (LPD) (500 MHz). While our analysis targets this specific platform, the methodology is broadly applicable to systems that include: (i) a programmable device capable of issuing DMA requests, (ii) an IOMMU with accessible performance monitoring events, and (iii) a system configuration where the programmable interface and the interfering DMA share the same IOTLB.

Measurement Tools. To analyze the architectural behavior of the SMMU implemented in the Xilinx UltraScale+ ZCU104 platform, we leveraged the Arm SMMUv2 Performance Monitor Unit (PMU) to collect micro-architectural events, which enables the measurement of (i) cycle counts, (ii) the number of Translation Lookaside Buffer (TLB) allocations, and (iii) the number of SMMU transactions processed. These measurements are used exclusively for characterizing the IOTLB structures and do not impact the execution timing of benchmarks. To evaluate the performance impact of SMMU contention, we deployed a DMA unit within the FPGA and leveraged the Integrated Logic Analyzer (ILA) to obtain cycle-accurate execution times of DMA transactions, capturing detailed hardware behavior without adding runtime overhead.

Arm SMMUv2 Implementation on the ZCU104. The micro-architectural details of the SMMU on the ZCU104 platform are not publicly documented. To address this gap, we conducted reverse-engineering experiments to uncover key aspects of its implementation. According to the Arm SMMUv2 specification, the architecture features a hierarchical caching system designed to optimize address translation and memory management. The primary cache structures include:

1) **Micro-TLB**: Caches PTW results returned by the Translation Control Unit (TCU).
2) **Macro-TLB and PTW Cache**: Caches PTW results within the TCU, with the PTW cache specifically storing partial PTWs to reduce the frequency of full PTWs.
3) **Prefetch Buffer (IPA to PA Cache)**: Prefetches pages to minimize future PTWs.

While the specification outlines general guidelines, such as a fully associative micro-TLB and four-way set associative macro-TLB, while the exact depth (number of entries) of these caches is vendor-specific. This configurability introduces challenges in evaluating the impact of shared structures within the SMMU. To address these challenges, we developed a set of micro-benchmarks to empirically determine the cache depths of the MMU-500 implemented on this platform. According to the MMU-500 specification, the micro-TLB can support up to 128 entries, backed by a TCU cache with up to 2000 entries.

Cache Depth Measurements. To understand the impact of IOTLB contention, it is essential to investigate the cache depths within the Arm SMMU. The number of cache entries directly influences how quickly these caches will be evicted during high-frequency DMA workloads. Specifically, a smaller number of cache entries results in more frequent cache misses, which increases the overhead of address translation. Moreover, determining the precise cache depth allows us to quantify the number of memory pages that can be accessed while the

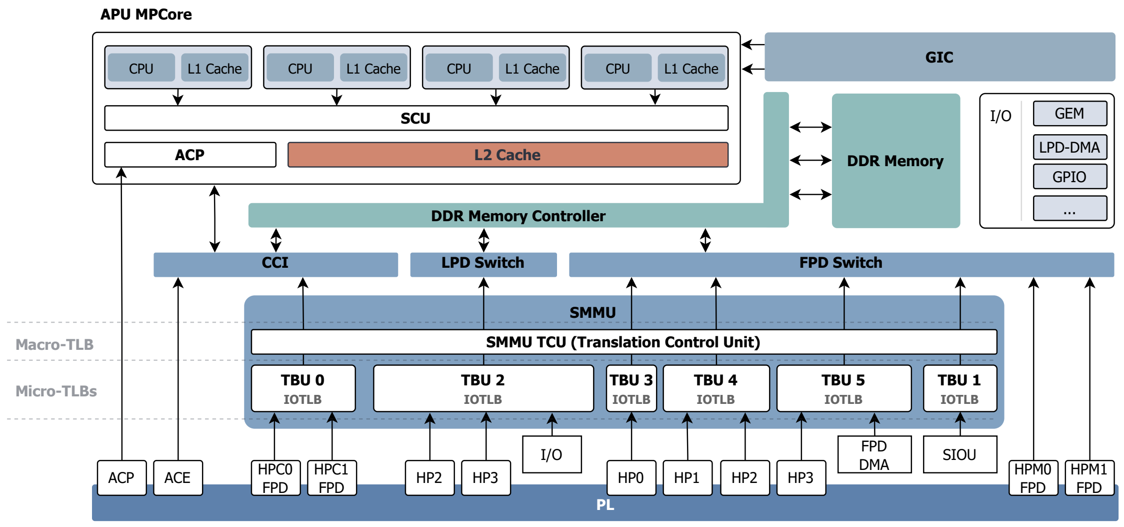


Fig. 2. ZCU104 Platform Overview. The IOMMU implementation on this platform follows a decentralized architecture in which multiple TBUs, functionally equivalent to IOTLBs, are associated with clusters of devices. These TBUs, also referred to as micro-TLBs, are part of the SMMU's Translation Control Unit (TCU) caching subsystem, which also includes a centralized macro-TLB.

translations from VAs to PAs remain cached. This information is crucial for understanding the scalability limitations of the IOTLB and identifying the point at which performance degradation begins to occur. While techniques for analyzing cache and TLB behavior via microbenchmarking are well-established in the real-time research community [42], particularly for uncovering WCET, our work extends these methods to the IOMMU context. Specifically, we focus on the previously underexplored IOMMU components, such as the micro-TLB, in the context of modern heterogeneous platforms. To explore the behavior of the IOTLB and infer the available translation resources, we conducted a series of experiments to probe the cache depths of the SMMU on the ZCU104 platform. Specifically, we targeted the micro-TLB and the macro-TLB, that includes the PTW cache, by leveraging the SMMU PMU to monitor the number of translations processed and the associated allocations to the micro- and macro-TLBs. However, there is a limitation in the current setup: the SMMU's PMU only allows us to measure micro-architectural events related to the micro-TLBs, preventing a complete analysis of the macro-TLB configuration. As a result, we could only measure the depth of the micro-TLB, while for the macro-TLB, we relied on the depth of 2000 entries defined in the SMMUv2 specification. To probe the micro-TLB depth, we leveraged the PMU available in each TBU of the SMMU. Specifically, we configured TBU5's PMU to monitor the following events:

1) **Number of DMA transactions** processed by TBU5 (read+write operations) — $SMMU\_access\ [T]$.
2) **Number of TLB entry allocations** triggered for DMA reads — $TLB\_entry\_alloc\ [R]$.
3) **Number of TLB entry allocations** triggered for DMA writes — $TLB\_entry\_alloc\ [W]$.

To induce SMMU translations and probe the behavior of the TBU, we deployed a baremetal VM atop the Bao hypervisor. The baremetal's workload consists of a micro-benchmarking designed to leverage DMA channels to perform memory transactions across different virtual memory pages. The micro-benchmarking runs (i) read and (ii) write operations: reads fetch small data chunks (e.g., 16B) from N memory pages into a buffer, while writes store the buffer's data into N memory pages. These micro-benchmark leverages the FPD-DMA unit connected to TBU5, as depicted in Figure 2.

Empirical results, presented in Figure 3, reveal a linear correlation between the number of DMA transactions and the corresponding SMMU and TLB accesses. For both read and write transactions, $SMMU\_access\ [T]$ increases proportionally with the number of DMA transfers, demonstrating consistent handling of translation requests as the workload scales. A key observation emerges when analyzing TLB entry allocations, $TLB\_entry\_alloc\ [T]$. Before collecting micro-architectural events, the micro-benchmark runs multiple times to warm up the caches. This ensures that the required translations are already cached when measurements begin, resulting in zero TLB allocations for smaller values of N. However, as the number of DMA transactions increases, so does the number of TLB allocations. This increase occurs when the number of transactions exceeds the capacity of the micro-TLB. The results from both read and write micro-benchmarking confirm that the micro-TLB has a depth of 64 entries. However, the observed increase in $TLB\_entry\_alloc\ [T]$ occurs slightly earlier than expected—around the 64th transaction. This behavior is explained by the additional translation required to complete the DMA transfer. For example, in the read benchmark, an extra translation is needed for the write oper-

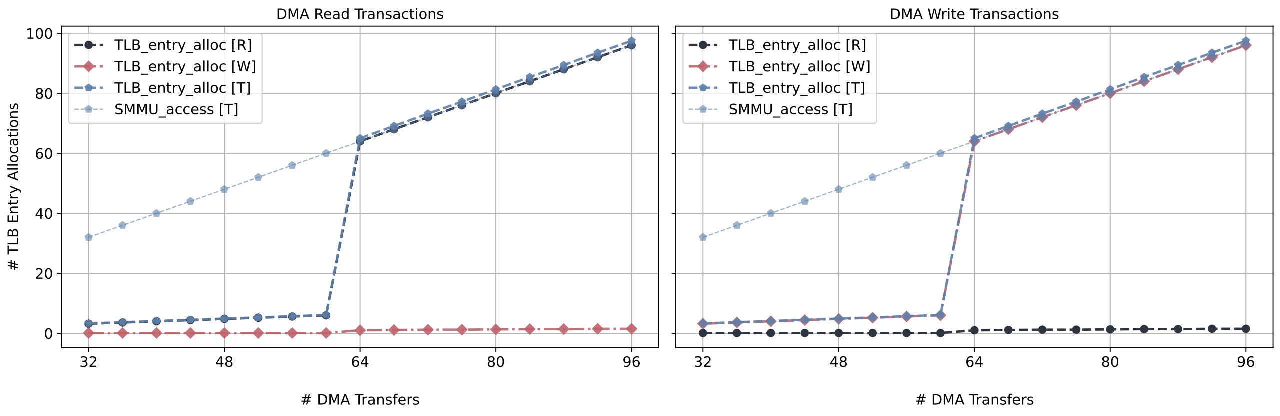


Fig. 3. SMMU TBU Micro-Benchmarking

ation that stores the fetched data into the destination buffer. When performing 64 transactions, a total of 65 translations are needed, exceeding the micro-TLB capacity and triggering allocations to the macro-TLB for subsequent transactions. In contrast, with 63 transactions, the number of required translations precisely matches the cache size, resulting in no misses and no additional TLB entry allocations. Our measurements show no significant difference in timing behavior between read and write operations in their interaction with the IOMMU structures. This is because each DMA transaction consists of both a read and a write phase, and the IOTLB allocates translation entries in the same manner for both. As a result, our methodology primarily focuses on determining the number of available translation entries, or the cache depth.

IV. IOMMU Contention and Performance Impact

The IOMMU plays a critical role in modern SoC platforms by ensuring secure and controlled access to memory for DMA-capable devices, thereby enhancing system security and reliability. However, the caching mechanisms in the IOMMU introduce challenges related to both performance and predictability, particularly when multiple devices compete for shared resources. In the case of the Arm SMMUv2 used in our target platform, we identify two primary sources of contention: (i) the TBUs (IOTLBs) distributed across the platform and (ii) the cache hierarchy within the SMMU's TCU.

As depicted in Figure 4, when a DMA device initiates a memory transfer (*DMA iteration*), the process begins with an *Address Read (AR)* handshake. The SMMU then checks the address translation. If the translation is **not cached**, the SMMU incurs a delay to perform a PTW (if necessary) and fetch the translation into the TBU. Conversely, if the translation is already **cached**, the TBU simply verifies the StreamID of the DMA device and the VAs, and resumes the transaction with the correct PAs, with negligible overhead. This distinction is highlighted in Figure 4, where the latency differences between cached and non-cached translations are depicted. Further insights into the operation of the SMMU are provided in Figure 5, which illustrates the detailed interaction between the TBU and TCU. When a translation is **not cached**, the process can be divided into two main stages: (i) the TBU stage and (ii) the TCU stage. In the TBU Stage: The TBU first checks the *micro-TLB* to determine if the required translation is cached locally. If it is not, the TBU sends a request to the TCU for further action. In the TCU Stage, upon receiving the request, the TCU verifies if the translation is cached in the *macro-TLB*. If it does, the translation is loaded into the *micro-*

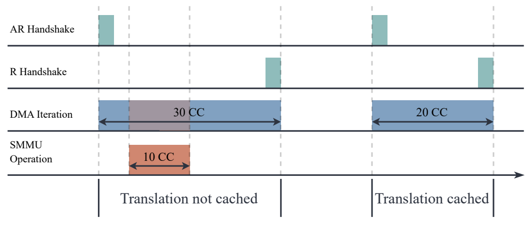


Fig. 4. Toy example illustrating two DMA transactions in clock cycles (CC). The first corresponds to scenario (iii), where the translation is not cached, leading to higher latency. The second represents scenario (i), where the translation is cached, significantly reducing translation overhead.

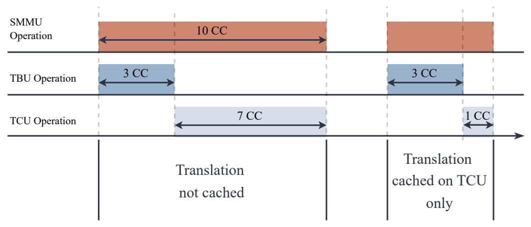


Fig. 5. Toy example showing the impact of TBU and TCU caching effects on two DMA transactions. The first represents scenario (iii), where the translation is not cached in either the micro- or macro-TLB, leading to the highest latency. The second corresponds to scenario (ii), where the translation is absent from the micro-TLB but cached in the macro-TLB, reducing translation overhead.

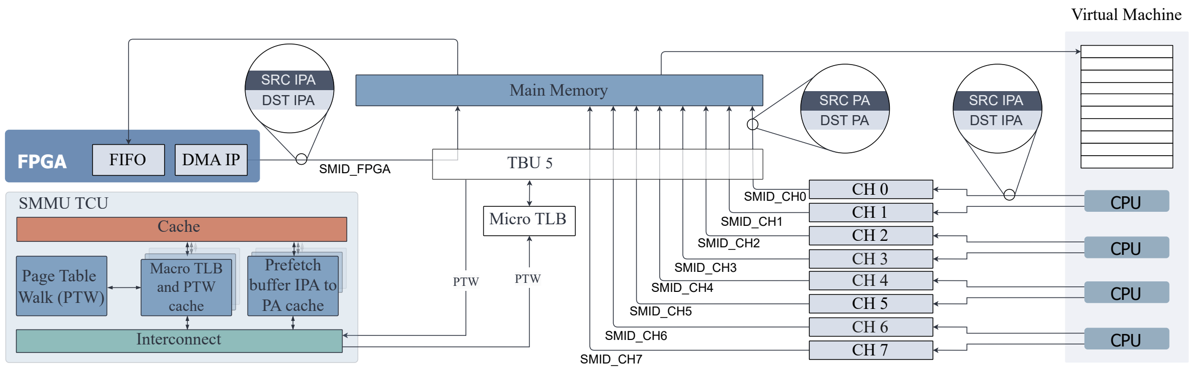


Fig. 6. IOMMU Interference: Evaluation Setup

TLB to complete the process. If not, the SMMU performs a PTW, fetches the required data, and updates the caches accordingly. As depicted in Figure 5, three distinct scenarios arise depending on the caching status of the translation:

(i) **Micro-TLB hit** – The translation is cached directly in the *micro-TLB*, allowing for near-instant translation and minimal latency.

(ii) **Micro-TLB miss and Macro-TLB hit** – The translation is cached in the *macro-TLB* but not in the *micro-TLB*. Here, the overhead is lower, as the translation only needs to be fetched into the *micro-TLB*.

(iii) **Micro-TLB and Macro-TLB Miss** – The translation is missing from both the *micro-TLB* and *macro-TLB*, necessitating a complete PTW, incurring higher latencies.

As demonstrated in both Figure 4 and Figure 5, the performance penalty increases progressively from scenario (iii) to scenario (i). The most significant delay occurs when translations are missing in both TLB levels, requiring a full PTW to proceed with the DMA transaction.

Methodology. To assess the performance impact of shared SMMU structures, we implemented a synthetic setup that emulates real-world scenarios using the Bao hypervisor. In this setup, an FPGA-based device performs DMA transactions as the benchmark, where we measure execution times. Meanwhile, a VM generates contention by issuing memory transactions through separate FPD-DMA channels. Specifically, the FPGA-based device accesses DDR memory via a DMA-based IP connected through the S_AXI_HP3_FPD interface, while the VM concurrently issues read operations from DDR memory using the FPD-DMA. As shown in Figure 6, both devices share the same TBU, specifically $TBU\ 5$.

While adding more channels to the FPGA's DMA controller could simulate contention, this approach risks introducing interference at the controller level, as multiple channels would compete for the same internal resources. Instead, by leveraging a separate FPD-DMA channel controlled by the VM, we create controlled contention without affecting the FPGA DMA controller itself. This allows us to isolate the impact on the SMMU structures, ensuring a clearer understanding of performance degradation at both the TBU and TCU levels.

To further evaluate the impact of varying operational conditions, we adjust the FPGA clock frequency in our experiments. The FPGA DMA operates at 100MHz in the base case, but this frequency is lower compared to other DMA devices integrated into the SoC (e.g., the FPD-DMA, which operates at 600MHz). By varying the FPGA clock frequency, we aim to simulate more realistic scenarios. Specifically, we increased the frequency to 150MHz and 300MHz to model the behavior of high-throughput DMA engines (e.g., PCIe or FPD-DMA) and observe how performance scales with increasing translation request rates. This change brings our controlled setup closer to real-world scenarios, while still isolating the effects within the IOMMU's internal structures.

The evaluation consists of two configurations designed to explore the performance implications of shared SMMU structures under the interference scenarios introduced earlier. These configurations directly assess contention at the micro-TLB (TBU) and maco-TLB (TCU) levels (e.g., by varying payload sizes), corresponding to the scenarios *interf_tbu* and *interf_tcu*, respectively. In the first configuration (*interf_tbu*), we assess contention at the TBU, as both the FPD-DMA and the S_AXI_HP3_FPD interface are routed through TBU5, as shown in Figure 2. This setup reflects real-world scenarios in which multiple devices rely on the same TBU for address translations, leading to interference at the IOTLB. By analyzing this scenario, we aim to quantify the performance impact when DMA devices contend for the same TBU resources, especially when translations are not cached locally and require further interaction with the SMMU. The second configuration (*interf_tcu*) targets contention at the SMMU's TCU, specifically its caching hierarchy. In this setup, concurrent access patterns from multiple devices stress the SMMU caches, creating scenarios where frequent PTW operations are required to perform address translations. Such conditions are common in use cases where various devices or VMs issue frequent and competing translation requests. This configuration allows us to evaluate how the TCU handles high contention, focusing on the latency differences when translations are either cached in the *macro-TLB* or entirely absent, generating PTW request.

Benchmarking Workload. We deployed an IP core on the FPGA designed to issue DMA transactions in a circular pattern, enabling precise control over workload parameters. The core allows the configuration of (i) the payload size for each DMA transaction, (ii) the number of memory pages accessed in sequence before wrapping around, and (iii) the base address from which transactions begin. To ensure consistency across experiments, we used a fixed page size of 4KB. Although we considered using superpages (e.g., 2 MB pages) for potential performance gains, this configuration was discarded due to the complexities they introduce, such as reduced TLB efficiency, higher miss rates, and less flexible memory allocation. Moreover, superpages precludes the use of cache coloring, which helps mitigate interference at the LLC. For our experiments, the IP core was configured to issue DMA transactions in a circular manner, accessing the first address of 63 consecutive memory pages before wrapping around to the initial position. Each entry in this circular pattern corresponds to a DMA transaction rather than an explicit read operation. This results in 63 entries allocated in the micro-TLB for read operations and one entry for write operations. The payload size for each transaction was set to 16 bytes, ensuring a lightweight yet consistent workload. This setup was designed to focus exclusively on measuring the peak performance of DMA transfers while minimizing interference from other shared hardware resources (e.g., DDR memory).

Interference Workload. To evaluate the impact of contention on the SMMU, we deployed a baremetal VM running on top of the Bao hypervisor, configured to use eight DMA channels for exerting stress on both the TBU and TCU caches. To maximize the volume of transaction requests, all four available CPUs on the platform are allocated to the VM. Each CPU manages two distinct DMA channels, with each channel accessing $N/4$ unique pages, where N is the total number of pages targeted by the application. A key characteristic of this workload is that each of the eight DMA channels operates with a unique Stream-ID. This differs from the FPGA IP workload, where only a single translation entry is needed for the page being accessed. In contrast, the interference workload's use of eight distinct Stream-IDs requires eight separate translation entries, even when all DMA channels write to the same memory page. This design amplifies contention at the TBU, as the independent translations for each Stream-ID add additional pressure on its caching resources. To analyze the impact of this contention, we configured two scenarios. First, with $N = 56$, the workload targets the TBU (*interf_tbu*), using 64 total entries, a configuration to stress the micro-TLB. Second, with $N = 1992$, the workload shifts focus to the TCU cache (*interf_tcu*), enabling the assessment of the SMMU's behavior under significantly higher contention levels. These values were selected based on the IOMMU's architectural details (as discussed in Section III), to induce contention directly within the IOMMU structure. To isolate IOMMU contention effects, a fixed and small payload size was used across all configurations. This approach minimizes interference from main memory and system bus activity, keeping performance degradation primarily due to translation-related contention.

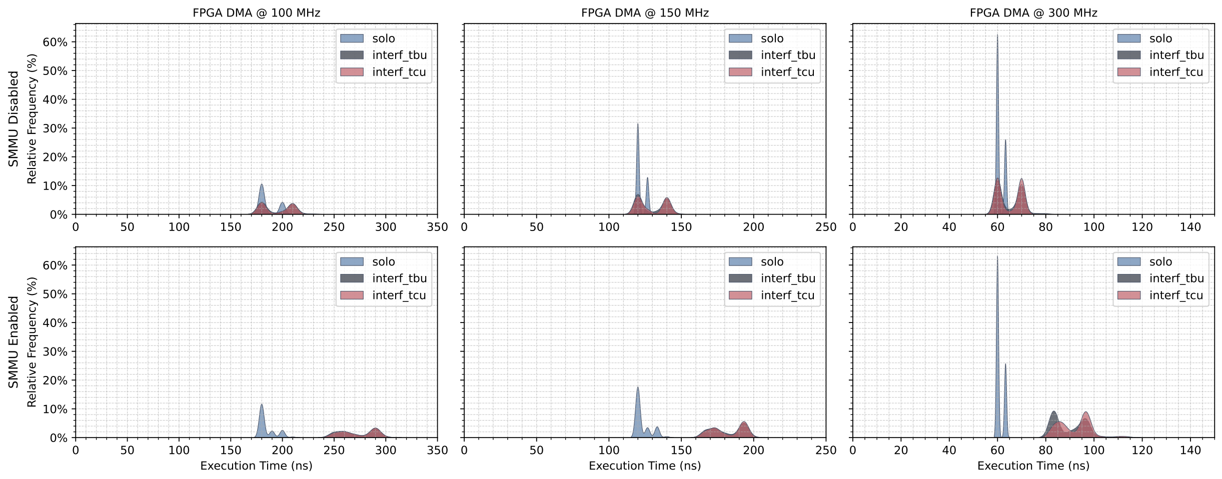
A. Latency of DMA Transactions with SMMU Disabled

Solo Execution. When the SMMU is disabled, the solo execution case serves as a baseline measurement for pure DMA performance without address translation overhead. The FPGA-based DMA engine directly accesses physical memory without intervention from the SMMU, ensuring that latency is only influenced by factors such as system interconnect and memory access times. At a 100MHz FPGA frequency, the average execution time per DMA transaction is 185.71ns. This time scales linearly with the FPGA clock frequency, meaning that at 150MHz, the execution time reduces to 123.18ns, and at 300MHz, it reaches 60.96ns. The near-perfect scaling confirms that in a solo environment, execution time is tightly coupled with the clock frequency, with minimal external sources of variation. The peak relative frequency (i.e., the most commonly observed execution time) aligns closely with the average execution time across all tested frequencies, reinforcing the predictability of solo execution. Furthermore, the minimum execution time remains unchanged, indicating a highly deterministic operation.

Interference Scenarios. Under interference conditions (*interf_tbu* and *interf_tcu*), execution times increase slightly. At 100MHz, the average execution time rises to 194.79ns for *interf_tbu* and 195.25ns for *interf_tcu*, reflecting a 1.05x overhead compared to the solo case. A similar pattern is observed at higher FPGA frequencies, where interference induces only minor variations in the average execution time. However, the maximum execution time fluctuates, with the highest overhead occurring in the ***interf_tbu*** scenario at 300MHz, reaching 126.67ns (1.26x higher when compared to the baseline). This suggests occasional contention at the memory subsystem and system bus, as further analyzed in Figure 7. Importantly, these effects primarily impact the latency of individual transactions rather than the overall throughput, indicating that while interference causes higher delays in specific transactions, it does not significantly affect the overall bandwidth.

B. Latency of DMA Transactions with SMMU Enabled

Solo Execution. When the SMMU is enabled, the solo execution scenario assesses the impact of address translation overhead in an environment free from contention. Unlike the SMMU-disabled case, each DMA transaction must now pass through the IOMMU translation process, introducing potential latency due to TLB lookups, PTW, and cache accesses. Despite these additional steps, solo execution latency remains nearly identical to the SMMU-disabled case. At 100MHz, the average execution time is 184.82ns, almost matching the 185.71ns observed without the SMMU. The execution time scales linearly with frequency, reducing to 123.18ns at 150MHz and 60.96ns at 300MHz, showing that in the absence of contention, address



SMMU Enabled	FPGA Frequency	Average Execution Time (ns)			Peak Density Time (ns)			Max Execution Time (ns)			Min Execution Time (ns)		
		solo	*interf_tbu*	*interf_tcu*	*solo*	*interf_tbu*	*interf_tcu*	*solo*	*interf_tbu*	*interf_tcu*	*solo*	*interf_tbu*	*interf_tcu*
Disabled	100MHz	185.71 **(baseline)**	194.79 **(1.05x)**	195.25 **(1.05x)**	180.00 **(baseline)**	180.10 **(1.00x)**	180.05 **(1.00x)**	200.00 **(baseline)**	230.00 **(1.15x)**	230.00 **(1.15x)**	180.00 **(baseline)**	180.00 **(1.00x)**	180.00 **(1.00x)**
	150MHz	121.93 **(baseline)**	129.10 **(1.06x)**	129.66 **(1.06x)**	120.00 **(baseline)**	120.05 **(1.00x)**	120.11 **(1.00x)**	126.67 **(baseline)**	146.67 **(1.16x)**	146.67 **(1.16x)**	120.00 **(baseline)**	120.00 **(1.00x)**	120.00 **(1.00x)**
	300MHz	60.98 **(baseline)**	64.98 **(1.07x)**	65.04 **(1.07x)**	60.00 **(baseline)**	60.04 **(1.00x)**	60.01 **(1.00x)**	63.33 **(baseline)**	80.00 **(1.26x)**	73.33 **(1.16x)**	60.00 **(baseline)**	60.00 **(1.00x)**	60.00 **(1.00x)**
Enabled	100MHz	184.82 **(baseline)**	272.29 **(1.47x)**	273.35 **(1.48x)**	180.00 **(baseline)**	289.89 **(1.61x)**	289.94 **(1.61x)**	210.00 **(baseline)**	300.00 **(1.43x)**	300.00 **(1.43x)**	180.00 **(baseline)**	250.00 **(1.39x)**	250.00 **(1.39x)**
	150MHz	123.18 **(baseline)**	182.07 **(1.48x)**	182.44 **(1.48x)**	120.00 **(baseline)**	193.26 **(1.61x)**	193.28 **(1.61x)**	140.00 **(baseline)**	200.00 **(1.43x)**	193.33 **(1.38x)**	120.00 **(baseline)**	166.67 **(1.39x)**	166.67 **(1.39x)**
	300MHz	60.96 **(baseline)**	89.77 **(1.47x)**	92.27 **(1.51x)**	60.00 **(baseline)**	83.36 **(1.39x)**	96.61 **(1.61x)**	63.33 **(baseline)**	110.00 **(1.74x)**	113.33 **(1.79x)**	60.00 **(baseline)**	83.33 **(1.39x)**	83.33 **(1.39x)**

Fig. 7. Performance impact of SMMU contention.

translations are efficiently handled by the SMMU's cache hierarchy, avoiding costly PTW operations. Like in the SMMU-disabled case, the peak relative frequency remains stable across frequencies, confirming a highly predictable execution time in solo mode. Additionally, the minimum execution time is identical to the previous scenario, suggesting that best-case transactions are not affected by SMMU involvement.

Interference Scenarios. With the SMMU enabled, the impact of interference becomes significantly more pronounced. At 100MHz, the average execution time increases to 272.29ns for *interf_tbu* and 273.85ns for interf tcu, reflecting an overhead of 1.47x compared to *solo* execution. Similarly, at 300MHz, interference results in a 1.51x overhead, highlighting the substantial performance cost introduced by contention at the SMMU. Unlike the SMMU-disabled case, minimum execution times are also affected, increasing by at least 1.39x across all frequencies. This suggests that even memory and system bus transactions, which are typically free from interference, experience contention under SMMU load, demonstrating that the SMMU consistently introduces a bottleneck in concurrent scenarios. The maximum execution time reveals even more severe degradation. At 100MHz, the maximum execution time increases from 210ns (*solo*) to 300ns in both interference scenarios, resulting in a 1.43x overhead. At 300MHz, the worst-case latency overhead reaches 1.79x in the *interf_tcu* scenario. This increase is attributed to contention at the SMMU's TCU and TBU caches, which exacerbates transaction delays under high-load conditions. Figure 7 further illustrates these findings, showing that SMMU contention primarily impacts worst-case latency, while average execution times remain more predictable. This highlights a key challenge: when the SMMU is heavily utilized, maintaining low-latency, deterministic transactions becomes increasingly difficult, posing significant challenges for real-time systems.

C. Impact of SMMU contention on DMA Throughput

Methodology. To evaluate the impact of SMMU contention on DMA throughput, we configured the FPGA IP to transfer varying payload sizes, ranging from the system bus width (128 bits, or 16B) to a memory page size (4KB). The DMA transfers were performed under both *solo* execution and interference scenarios (*interf_tbu* and *interf_tcu*).

Solo Execution. In the solo configuration, the DMA throughput scales predictably with the payload size, as shown in Figure 8. For small payloads (e.g., 16B), the bandwidth reaches approximately 84.6KB/s. As the payload size increases, the bandwidth improves significantly, reaching 1440KB/s for a 4KB payload. Empirical results show the linear relationship between payload size and bandwidth for smaller transfers.

Interference Scenarios. Under interference conditions (*interf_tbu* and *interf_tcu*), the DMA throughput is notably affected, particularly for smaller payloads: for a payload of 16B, the bandwidth decreases to from 84.6KB/s to 57.4KB/s and 56.7KB/s, respectively, corresponding to a reduction of

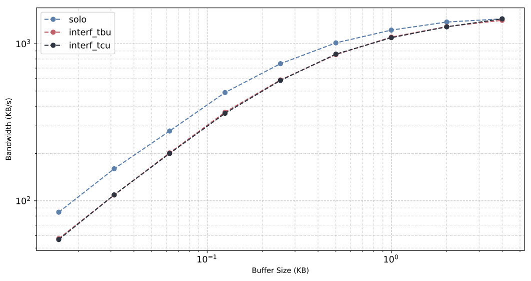


Fig. 8. DMA transactions throughput

approximately 32% compared to the solo case. As the payload size increases, the impact of interference diminishes. For instance, at 1KB, the bandwidth for *interf_tbu* and *interf_tcu* reaches approximately 1102KB/s and 1094KB/s, respectively, which is only about 10% lower than the solo configuration. At the maximum payload size of 4KB, the throughput for all configurations converges, with interference scenarios achieving bandwidths of up to 1410KB/s, closely matching the baseline.

Key Observations. The reduction in throughput due to interference is primarily caused by the overhead introduced by the SMMU during each DMA transaction. Empirical results show that this overhead remains constant at approximately 100ns, regardless of the payload size. Consequently, for smaller payloads, the overhead represents a significant portion of the total transaction time, leading to a substantial reduction in throughput. However, as the payload size increases, the total transaction time becomes dominated by the transfer of the payload itself (on the order of milliseconds), rendering the 100ns overhead relatively insignificant.

V. Discussion and Future Directions

This section examines the impact of IOMMU contention, emphasizing its effects on transaction latency and overall system performance. Additionally, we discuss key architectural factors influencing contention and outline future research to further quantify this issue across different platforms.

A. IOMMU as a Source of Contention

Our findings confirm that the IOMMU can act as a significant source of contention, introducing delays in memory transactions, particularly for smaller payload sizes. Experimental results indicate that lower-sized transactions can experience delays up to 1.79x due to contention in the IOMMU. This occurs because the overhead introduced by address translation and access control mechanisms within the IOMMU disproportionately affects transactions with shorter execution times. As a result, the IOMMU plays a non-negligible role in performance degradation for latency-sensitive applications, particularly those relying on frequent small transfers. These latency increases are especially relevant in the context of real-time systems, as they may influence the ability of such systems to meet timing constraints. The observed delays could affect worst-case execution time (WCET) estimates, potentially requiring adjustments in system models to account for DMA contention. While a detailed exploration of real-time implications is beyond the scope of this work, the measured contention-induced latency increases can serve as a foundation for future real-time analysis.

B. IOMMU Contention in Real-World Systems

Analyzing contention in a real-world scenario, our experiments demonstrate that interference from TBU and TCU exhibit similar performance impacts on the evaluated platform. This equivalence is primarily attributed to the highly optimized caching structures present in the Arm SMMU, such as prefetch buffers and multi-level TLB hierarchies. The architectural refinements of SMMUv3, including the presence of Micro-TLBs, Macro-TLBs, and PTW caches, effectively mitigate excessive translation overhead by caching frequent translations and prefetching pages proactively. Consequently, contention effects are significantly absorbed, leading to comparable interference levels between the TBU and TCU. Furthermore, our findings suggest that the impact of IOMMU contention diminishes when considering larger payload transfers. The primary reason is that memory transaction latencies overshadow IOMMU processing times by orders of magnitude, causing the introduced overhead to be diluted over the total execution time. Thus, while smaller transfers suffer from substantial performance penalties, higher payload transactions experience relatively negligible impact from IOMMU contention.

C. Mitigation Strategies for IOMMU Contention

While the primary focus of this work was to characterize and quantify the impact of IOMMU contention, particularly in relation to DMA transaction latency, we recognize the importance of exploring mitigation strategies to alleviate such performance degradation. Although a full experimental evaluation of these techniques is outside the scope of this study, several promising approaches can be considered to reduce the impact of IOMMU-induced latency spikes. One promising direction is the use of QoS regulators to manage bandwidth allocation across DMA stream-IDs. Recent IOMMU designs—such as those in RISC-V and Arm SMMUv3—integrate support for stream-ID-aware Quality-of-Service controls. These regulators can throttle or prioritize memory traffic on a per-stream basis, reducing interference and helping enforce bandwidth or latency constraints for critical tasks. Another complementary approach involves static memory mapping and software-assisted page coloring. By mapping critical DMA workloads to predetermined page sets and avoiding page reuse across contending devices, designers can minimize pressure on IOMMU caching structures. Page coloring can also be used to partition IOMMU's caches to reduce interference between non-CPU bus masters, partitioning translation entries and improve locality.

D. Future Directions

The observations in this study highlight the need to further investigate IOMMU contention across different architectures.

Since IOTLB distribution significantly affects contention, a broader experimental study on diverse architectures would be valuable. In systems where multiple devices compete for shared address translation resources, the extent of contention may vary depending on the architectural design choices, such as the number and placement of IOTLBs. Future research should also explore methods to mitigate IOMMU contention, such as dynamic translation prefetching strategies (e.g., preemptively fetching translation entries based on memory access patterns), adaptive TLB partitioning (e.g., allocating TLB resources differently based on workload characteristics), and hardware enhancements like hardware-managed TLB eviction (e.g., using specialized hardware to optimize TLB entry replacements). Additionally, evaluating contention effects in multi-tenant cloud environments or heterogeneous computing platforms can provide insights into how IOMMU behavior scales with an increasing number of concurrent memory-intensive workloads. By extending this study to a wider range of use cases, we can better understand and optimize IOMMU designs to improve system performance and efficiency.

VI. Related Work

This section reviews prior work on IOMMU-related side channels and performance, focusing on security risks and architectural challenges relevant to contention and scalability.

Side-channel attacks on IOMMU. While IOMMUs aim to enforce isolation and protection for DMA transactions, recent studies show they also introduce new opportunities for side-channel attacks. Several works demonstrate how adversaries can exploit the IOTLB and peripheral interactions to bypass existing defenses. Kim et al. introduced DevIOus [43], a side-channel attack that leverages the IOTLB to extract sensitive information using DMA-capable PCIe devices such as GPUs and RDMA-enabled NICs. Unlike CPU-centric attacks, DevIOus operates entirely within the IOMMU domain, avoiding interference with CPU caches or TLBs. Similarly, Markettos et al. presented Thunderclap [44], which investigates vulnerabilities in OS-level IOMMU protections against DMA-based attacks. Despite widespread IOMMU deployment, they show how malicious peripherals can exploit shared memory regions and kernel interactions to hijack execution and extract data. Their findings are validated using a custom FPGA-based platform. Expanding on these studies, Tiemann et al. [45] introduced IOTLB-SC, identifying the IOTLB as a key leakage source in cloud systems. Using an FPGA accelerator, they uncover covert channels enabling unauthorized communication between peripherals, potentially leaking GPU-accelerated workloads. Their work highlights the growing relevance of IOTLB-based channels with emerging interconnects such as CXL and PCIe 5.0. Lastly, Kim et al. [46] analyzed the IOMMU's role in PCIe-based side channels, focusing on RDMA-enabled network cards. By examining IOMMU translation behavior, they demonstrate how PCIe interfaces can serve as practical side-channel vectors in diverse systems.

IOMMU performance. Prior work has explored both architectural enhancements and empirical evaluations of IOMMU behavior [47]. Hur et al. [48] investigated the use of hashed page tables in the ARM SMMU, showing that this approach simplifies translation lookups and improves scalability in heterogeneous SoCs. By reducing lookup complexity, it can enhance performance for workloads with frequent DMA activity. Paraskevas et al. [49] performed an experimental analysis of the ARM SMMU on a COTS platform, evaluating its behavior under real accelerator workloads. Their work provides useful insights into SMMU overheads but does not explore microarchitectural aspects such as micro- and macro-TLBs, or contention under co-allocated workloads. Complementary to IOMMU-induced contention, Zini et al. [50] studied memory contention caused by I/O devices on COTS platforms. They propose regulating bus traffic using COTS mechanisms like the Arm QoS-400 to improve performance predictability in MCSs. These studies highlight both the security risks and performance challenges introduced by the IOMMU, reinforcing the need for further investigation into contention, caching, and scalability in modern IOMMU designs.

VII. Conclusion

The IOMMU is a critical component for memory protection and translation in modern computing systems, yet its impact on performance, particularly in mixed-criticality systems, has been largely overlooked. In this work, we analyzed contention effects within IOMMU structures, demonstrating that its shared nature can introduce interference that degrades the performance of time-sensitive workloads. Our findings show that this contention primarily affects small memory transactions, where translation delays are not diluted over long execution times. Additionally, based on the shared design principles of TBUs and TCUs across different architectures, we hypothesize that interference from these components would follow similar patterns, as their caching structures (e.g., prefetching and IOTLB designs) are based on common design principles, although further research is needed to confirm this.

These results highlight that the IOMMU is not only a security-relevant component but also a performance-sensitive element that must be carefully managed in heterogeneous MCSs. As system complexity increases, understanding and mitigating IOMMU-induced contention becomes essential to ensuring predictable performance. Future work should extend this analysis across different architectures, considering factors such as IOTLB distribution, resource sharing, and workload co-location. Addressing these challenges will be key to moving beyond the Bermuda Triangle of contention, toward a deeper and more comprehensive understanding of IOMMU behavior in complex systems.

Acknowledgment

This work has been supported by FCT - Fundação para a Ciência e Tecnologia grants 2022.13378.BD, SFRH/BD/138660/2018.

References

[1] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, "Multi-core devices for safety-critical systems: A survey," *ACM CSUR*, vol. 53, no. 4, 2020.

[2] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," *29th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 76, pp. 4:1–4:22, 2017.

[3] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, F. Tronci, A. Biasci, and S. Pinto, "IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms," *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, vol. 108, pp. 2:1–2:13, 2023.

[4] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," *31st Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 133, pp. 27:1–27:25, 2019.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, " Real-time cache management framework for multi-core architectures ," *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 45–54, 2013.

[6] T. A. Henzinger and J. Sifakis, ""The Embedded Systems Design Challenge"," *FM 2006: Formal Methods*, pp. 1–15, 2006.

[7] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Future Generation Computer Systems*, vol. 129, pp. 315–330, 2022.

[8] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, vol. 77, pp. 3:1–3:14, 2020.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009.

[10] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," *5th IEEE Consumer Communications and Networking Conference*, pp. 257–261, 2008.

[11] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look Mum, no VM Exits! (Almost)," 2017.

[12] J. Martins and S. Pinto, "Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems," *IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 40–53, 2023.

[13] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The Omnivisor: A Real-Time Static Partitioning Hypervisor Extension for Heterogeneous Core Virtualization over MPSoCs," *36th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 298, pp. 7:1–7:27, 2024.

[14] D. Oliveira, W. Chen, S. Pinto, and R. Mancuso, "Shared Resource Contention in MCUs: A Reality Check and the Quest for Timeliness," *36th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 298, pp. 5:1–5:25, 2024.

[15] H. Yun, "Understanding and mitigating hardware interference channels on heterogeneous multicore," *IEEE 3rd Real-Time and Intelligent Edge Computing Workshop (RAGE)*, pp. 1–3, 2024.

[16] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, A. Biasci, and S. Pinto, "IRQ Coloring and the Subtle Art of Mitigating Interrupt-Generated Interference," *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 47–56, 2023.

[17] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega, "Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey," *ACM CSUR*, vol. 52, no. 1, 2019.

[18] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.

[19] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10, 2015.

[20] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, 2015.

[21] A. Oliveira, G. Moreira, D. Costa, S. Pinto, and T. Gomes, "IA&AI: Interference Analysis in Multi-core Embedded AI Systems," *International Conference on Data Science and Artificial Intelligence*, pp. 181–193, 2024.

[22] M. Bechtel and H. Yun, " Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention ," *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 357–367, 2019.

[23] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mempol: polling-based microsecond-scale per-core memory bandwidth regulation," *Real-Time Systems*, pp. 1–44, 2024.

[24] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

[25] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in cots-based multicore systems," *8th IEEE international symposium on industrial embedded systems (SIES)*, pp. 39–48, 2013.

[26] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–14, 2019.

[27] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo, and R. Mancuso, "Coherence-aided memory bandwidth regulation," *IEEE Real-Time Systems Symposium (RTSS)*, pp. 322–335, 2024.

[28] A. Löfwenmark and S. Nadjm-Tehrani, "Understanding Shared Memory Bank Access Interference in Multi-Core Avionics," *16th International Workshop on Worst-Case Execution Time Analysis (WCET)*, vol. 55, pp. 12:1–12:11, 2016.

[29] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 55–64, 2013.

[30] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms," *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, 2014.

[31] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," *24th Euromicro Conference on Real-Time Systems*, pp. 299–308, 2012.

[32] B. Morgan, E. Alata, V. Nicomette, and M. Kaâniche, "IOMMU protection against I/O attacks: a vulnerability and a proof of concept," *Journal of the Brazilian Computer Society*, vol. 24, pp. 1–11, 2018.

[33] H. Tang, Q. Li, S. Feng, X. Zhao, and Y. Jin, "Iommu para-virtualization for efficient and secure dma in virtual machines," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 10, no. 12, pp. 5375–5400, 2016.

[34] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete GPUs," 2017, pp. 1–11.

[35] D. Rossi, I. Loi, G. Haugou, and L. Benini, "Ultra-low-latency lightweight DMA for tightly coupled multi-core clusters," *Proceedings of the 11th ACM Conference on Computing Frontiers*, pp. 1–10, 2014.

[36] Intel Corporation, *Intel Virtualization Technology for Directed I/O (VT-d): Architecture Specification*, 2021.

[37] Advanced Micro Devices, Inc., *AMD I/O Virtualization Technology (IOMMU) Specification*, 2022.

[38] RISC-V International, *RISC-V IOMMU Specification*, 2023.

[39] Arm Ltd., *ARM System Memory Management Unit Architecture Specification (SMMU v2)*, 2023.

[40] Z.-D. Labs, "Risc-v iommu," 2023. [Online]. Available: https://github.com/zero-day-labs/riscv-iommu

[41] F. Menarini, ""Arm CoreLink MMU-600 saves >$1Billion in premium content protection systems"," 2019, [online] Available at: https://community.arm.com/arm-community-blogs/b/mobile-graphics-and-gaming-blog/posts/arm-corelink-mmu-600-saves-1billion-in-premium-content-protection-systems.

[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 2008.

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

[44] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, “Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals,” 2019.

[45] T. Tiemann, Z. Weissman, T. Eisenbarth, and B. Sunar, “IOTLB-SC: An Accelerator-Independent Leakage Source in Modern Cloud Systems,” *ACM Asia Conference on Computer and Communications Security*, p. 827–840, 2023.

[46] H. Kim and J. Hur, “PCIe Side-Channel attack on I/O device via RDMA-enabled network card,” *13th International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1468–1470, 2022.

[47] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, “The price of safety: Evaluating iommu performance,” *The Ottawa Linux Symposium*, pp. 9–20, 2007.

[48] W. Hur and W. W. Ro, “Performance Implication of Hashed Page Tables on ARM System Memory Management Unit Design,” *IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pp. 1–4, 2024.

[49] K. Paraskevas, K. Iordanou, M. Luján, and J. Goodacre, “Analysis of the usage models of system memory management unit in accelerator-attached translation units,” *Proceedings of the International Symposium on Memory Systems*, pp. 86–96, 2020.

[50] M. Zini, G. Cicero, D. Casini, and A. Biondi, “Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms,” *Software: Practice and Experience*, vol. 52, no. 5, pp. 1095–1113, 2022.