Fast & Safe IO Memory Protection

Benny Rubin
Cornell University

Saksham Agarwal
UIUC

Qizhe Cai
Cornell University

Rachit Agarwal
Cornell University

Abstract

IO Memory protection mechanisms prevent malicious and/or buggy IO devices from executing errant transfers into memory. Modern servers achieve this using an IOMMU—IO devices operate on virtual addresses, and IOMMU translates virtual addresses to physical addresses (potentially speeding up translations using a cache called IOTLB) before executing memory transfers. Despite their importance, design of memory protection mechanisms that can provide strong safety properties while achieving high performance has remained elusive. Indeed, recent studies from production datacenters demonstrate that inefficiencies within state-of-the-art memory protection mechanisms result in significant throughput degradation, orders-of-magnitude tail latency inflation, and violation of isolation guarantees.

We present Fast & Safe (F&S), a simple modification to existing memory protection mechanisms that enables them to provide the strongest safety property, and yet, near-completely eliminates their overheads. The key insight in F&S design is that, rather than solely focusing on minimizing IOTLB miss rates, we should focus on reducing the cost of each IOTLB miss. We demonstrate that this change of perspective enables a simple F&S design that requires no modifications in host hardware and minimal modifications within the operating system.

CCS Concepts: • **Security and privacy** → **Operating systems security**; • **Software and its engineering** → **Memory management**.

Keywords: Memory protection, IOMMU

ACM Reference Format:
Benny Rubin, Saksham Agarwal, Qizhe Cai, and Rachit Agarwal. 2024. Fast & Safe IO Memory Protection. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages.

1 Introduction

IO Memory protection mechanisms prevent malicious and/or buggy IO devices like network interface cards (NICs) from executing errant transfers into memory. Modern servers achieve IO memory protection using an Input-Output Memory Management Unit (IOMMU) [28, 43]. An IOMMU operates very similar to memory management units within processors; we describe the IOMMU-enabled datapath from the NIC to the application in §2.1, but briefly: with IOMMU enabled, the operating system assigns NICs so-called IO virtual addresses (IOVAs), that are used by the NIC to initiate direct memory access (DMA) to the host memory; for every DMA initiated by the NIC, the IOMMU uses a host memory resident IO page table to translate NIC-visible IOVAs to the host physical address that is used for the final data transfer. These address translations are sped up using a special IOTLB cache that caches frequently used address translation entries. Upon an IOTLB hit, the address translation incurs near-zero cost; however, upon an IOTLB miss, an IO page table walk is initiated to perform the address translation.

Memory protection using IOMMU can have significant impact on application-layer performance. Consider a server with 100Gbps NICs and 128Gbps PCIe. To provide the strongest safety property (referred to as the strict mode in modern operating systems), each IOVA must be unmapped, and the corresponding IOTLB entry invalidated immediately after the DMA (§2.1); thus, one must incur at least one IOTLB miss every (huge)page worth of data transfer and perform 4 sequential memory accesses per miss for IO page table access in the worst case. Even with 100ns memory access latency, this will incur $\sim 4\times 100$ns of IOMMU overhead. Thus, intuitively, with modern PCIe devices that allow buffering up to ~100 cachelines at the processor-side end of the PCIe device [2, 37, 44], enabling IOMMU in the strongest safety mode essentially pushes PCIe to its limits—using Little's Law [14], the maximum sustainable rate would be $100 \times 64\text{bytes}/400\text{ns} = 128\text{Gbps}$. In practice, the situation is much worse: memory access latencies can be much higher [2, 12, 13, 30, 44], PCIe has its own overheads [37], IOTLB can incur multiple misses during the huge(page) worth of data transmission [1, 7], etc. Indeed, recent studies from large-scale production clusters [1] have shown that inefficiencies within state-of-the-art memory protection mechanisms can result in significant throughput degradation, orders-of-magnitude tail latency inflation, and violation of isolation guarantees.

Sadly, the above analysis is so fundamental that it has left the community with a bleak outlook about memory protection—without rearchitecting memory management in operating systems and/or without significant improvements in server hardware (*e.g.*, lower memory access latencies), it appears impossible to provide the strongest safety property while maintaining high performance. Unfortunately, technology trends suggest that server hardware improvements along these directions are unlikely [1]; thus, most recent works on high-performance memory protection target a weaker safety property [9, 16, 39], re-architect memory management within the operating system [33, 34], and/or redesign IOMMU hardware [7, 18, 29, 31]. Unfortunately, such weaker safety properties and clean-slate modifications themselves introduce new security vulnerabilities [4, 27, 32, 33, 35]. This state-of-affair leaves organizations with two equally expensive and painful options: (1) high-performance but unsafe systems; or, (2) safe systems but with suboptimal performance.

Motivated by building an in-depth understanding of the above impasse, we set out to explore inefficiencies in modern memory protection mechanisms (§2). While reproducing the overheads of memory protection mechanisms were straightforward, our preliminary experiments during this exploration led to some baffling results: we found that, in many cases, the total number of memory accesses for IO page table walks were less than 4× the number of IOTLB misses! Diving deeper into this inexplicable and seemingly impossible phenomenon, we discovered an aspect of IOMMU hardware that has been ignored in all previous studies: IO page table caches [24] (in hindsight, this should have been obvious, given that processor MMUs have had such caches for a long time now [10]). Specifically, IOMMU has hardware caches that store most frequently accessed entries from level1, level2, and level3 of the IO page table. Upon an IOTLB miss, IOMMU first checks the cache for the corresponding IO page table level1 page—if a hit, the corresponding level2 page can be identified without an IO page table walk (thus, avoiding one memory access), and IOMMU now checks the cache for the corresponding IO page table level2 page, and so on (to be precise, these checks happen in parallel). Indeed, in the worst-case scenario, IOMMU still needs to perform 4 memory accesses per IOTLB miss; however, in the best-case scenario, the IO page table caches enable address translation using just 1 memory access per IOTLB miss (for the corresponding level4 entry)! This paper is the outcome of this deep dive into understanding the overheads of memory protection.

We present F&S, a simple modification to existing memory protection mechanisms that provides the strongest safety property, and yet, near-completely eliminates the overheads of IO memory protection. The key insight in F&S design is that, while IOTLB misses are unavoidable with the strongest safety property, it is still possible to *reduce the cost of address translation upon an IOTLB miss* using IO page table caches within the IOMMU hardware. Indeed, we demonstrate that, by carefully allocating IOVAs (to maximize IO page table cache hit rate) and by carefully managing IO page table cache invalidations (to ensure safety), F&S is able to provide the strongest safety property with minimal overheads of memory protection. For instance, we find that F&S results in 0 level1 misses, 0 level2 misses, and at most 0.054 level3 misses per page worth of data across all evaluated workloads (except for an extreme case we discuss in §4.4); F&S, thus, near-completely eliminates the overheads of IO memory protection across all evaluated workloads. F&S provides such powerful results without any modifications in host hardware and without any modifications in operating systems memory management; it needs merely 630 lines of code changes within the IOMMU and network drivers.

F&S implementation, along with the documentation to reproduce our results, is available at https://github.com/host-architecture/Fast-and-Safe-IO-Memory-Protection/.

2 Inefficiencies in Memory Protection

We start with background on the NIC-to-memory datapath when using memory protection (§2.1). We then evaluate state-of-the-art memory protection mechanisms with the goal of understanding the root causes underlying their inefficiencies (§2.2). Our key findings are:

- State-of-the-art memory protection mechanisms can result in as much as 65% degradation in throughput and multiple orders of magnitude inflation in tail latency. Increasing number of flows/connections, access link bandwidths, data transfer sizes, number of cores, and/or memory contention result in increasing overheads for existing memory protection mechanisms. Similar observations have been made in several recent studies, including those from large-scale production clusters [1, 7, 16, 39].
- The core reason for application-level performance degradation is the inflation in per-DMA latency due to address translation required for memory protection [1]. Specifically, to achieve the strict safety property, existing memory protection mechanisms necessitate *at least* one IOTLB miss per (huge)page worth of data; we find that the actual number of IOTLB misses can be much larger in practice—even for basic workloads as in our setup, we observe from 1.30 to 2.20 IOTLB misses per page worth of data. Since each IOTLB miss requires 4 memory accesses in the worst case, memory protection can incur hundreds of nanoseconds or even microseconds of inflation in per-DMA latency. Given that only a small number of fixed-size DMA transactions can be in flight at any given point of time [37, 44], the increase in per-DMA latency reduces the effective PCIe utilization [1, 44]. Consequently, NIC buffers build up resulting in eventual packet drops and throughput degradation.
- We find that modern memory protection hardware has IO page table caches that store most frequently accessed entries from each level of the IO page table. Our evaluation suggests that, for some workloads, these caches can reduce

the number of memory accesses required per IOTLB miss from 4 to ~1.76. However, modern memory protection mechanisms are not designed to exploit these caches—even with moderate number of flows, link bandwidths, data transfer sizes, number of cores, etc., we observe large number of misses in IO page table caches. These IO page table cache misses bring the overheads of memory protection mechanisms closer to their worst-case overheads; indeed, application-level performance degradation near-perfectly matches the high miss rates on these IO page table caches. We find that these IO page table cache misses are rooted in poor locality of IO virtual addresses and (unnecessary) IO page table cache invalidations.

2.1 NIC-to-memory DMA datapath

Figure 1 illustrates the NIC-to-memory datapath when memory protection is enabled. We focus primarily on the receive-side (Rx) datapath where packets are DMA'd from the NIC to the host memory since the receive-side datapath is usually the performance bottleneck [1, 2, 44]. The sender-side (Tx) datapath is nearly identical, with the datapath steps simply performed in the reverse order; there are some minor differences between the Rx and the Tx datapaths, such as size of socket buffers associated with the ring descriptors which we discuss in §3. We describe the datapaths for CX-5 Mellanox NICs with Intel CPUs; the high-level datapath for other NICs and CPUs is similar [40].

The key steps in the Rx datapath are as follows:

1. The NIC driver sets up a per-core ring buffer of Rx descriptors, each of which contains virtual memory addresses to be used to DMA data. Modern NICs support each descriptor to have addresses worth one or more pages (*e.g.*, default 64-page descriptors in Mellanox CX-5 NICs), allowing multiple packets to be DMA'd using the same descriptor. To prepare a single descriptor, for each page, the NIC driver allocates a physical frame and passes it to the IOMMU driver. The IOMMU driver then allocates a page-sized IO Virtual Address (IOVA) from the IOVA allocator (we discuss the IOVA allocator below); an IOVA itself is simply a range of addresses and the IOVA allocator keeps track of free address ranges. The IOMMU driver then maps the IOVA to the physical page in the IO page table, and returns the IOVA to the NIC driver. The NIC driver inserts these IOVAs into the descriptor. The driver will periodically replenish these descriptors when the number of remaining descriptors in a ring buffer falls below a threshold.
2. Upon receiving a packet, the NIC first enqueues the packet into its input buffer. The NIC then uses the IOVAs in the Rx ring buffer descriptor (associated with the corresponding application core) to DMA the packet. Each DMA request may be executed as multiple PCIe transactions based on the request and the transaction size.

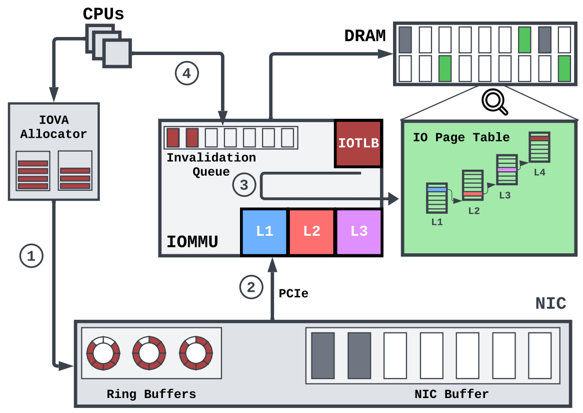


Figure 1. Illustration of NIC-to-memory DMA datapath at the receiver when memory protection is enabled. Discussion in §2.1.

3. Upon arriving at the root complex—the other end of the PCIe interconnect, also known as the Integrated IO Controller in the Intel architecture—each transaction's IOVA must be translated to a physical address. The IOMMU performs these translations as follows: first, it looks up the IOVA in the IOTLB, which caches recently used IOVA-to-physical address mappings. Upon an IOTLB hit, the address is immediately translated. Otherwise, the IOMMU performs an IO page table walk. The IO page table has four levels (referred to from now on as PT-L1, PT-L2, PT-L3 and PT-L4) and each page has 512 page table entries, with a size of 64 bits each; PT-L1 entries map from the 9 MS bits of the IOVA to a PT-L2 page, PT-L2 entries map from the next 9 bits of the IOVA to a PT-L3 page, etc. The PT-L4 entries contain direct mappings to physical addresses. Modern IOMMU hardware also has caches for the IO page table that can speed up translation [24]. Specifically, there is a cache for each of the first three levels of the IO page table, which we refer to as PTcache-L1, PTcache-L2, PTcache-L3; each entry in the cache points from an IOVA to the address of the corresponding PT-L2/L3/L4 page in the IO page table. Thus, in the worst-case (an IOTLB miss followed by a miss in all PTcache-L1/L2/L3), address translation would require exactly four memory accesses (one for each level of the IO page table) as is assumed in most prior works on IOMMU performance enhancements [6, 7, 31, 39]; however, in the best case, address translation requires one (unavoidable) memory read from a PT-L4 page, which contains the requisite physical address. Since translations occur at the granularity of PCIe transactions, it may take multiple address translations to complete the DMA of a single packet.

④ Once the NIC has performed DMAs for all available pages in a descriptor—to ensure that the NIC can no longer access the buffers of the DMAed packets—the IOMMU driver unmaps the IOVA-to-physical page mapping for each page in the descriptor, invalidates the entries for the unmapped IOVA in all IOMMU caches (IOTLB and PTcache-L1/L2/L3), and finally frees the unmapped IOVAs back to the IOVA allocator. Modern operating systems typically provide two safety modes [7]: strict and deferred. In the strict mode, IOVA invalidation happens immediately after the IOVA is unmapped (hence, memory protection is guaranteed at a per-descriptor granularity). In the deferred mode, invalidations are deferred until a fixed threshold number of IOVAs have been unmapped leaving stale entries in the IOTLB that can be used by the device.

⑤ Upon completion of DMA requests, the network driver constructs packets from the descriptor buffers and passes the packets to the transport layer. While processing received packets, the transport layer may generate and transmit acknowledgement (ACK) packets. Similar to the Rx datapath, to transmit an ACK packet, an IOVA is allocated and then mapped to the physical address of the ACK packet in the IO page table. After the ACK is transmitted, the IOVA is unmapped from the IO page table, and its IOTLB and page table cache entries are invalidated. This step is not shown in Figure 1, since it follows the Tx datapath.

IOVA Allocator. The IOVA allocator provides two operations: allocation and freeing of IOVAs. It also ensures that IOVAs are only re-allocated once they are freed. To accomplish this, allocated IOVA ranges are managed with a globally locked red-black tree, where each node in the tree contains a high and a low IOVA address [39]. In the worst case, operations on the red-black tree can incur linear time searches, leading to high CPU overheads [39]. To avoid such overheads, modern IOVA allocators keep all operations constant time using a stack-based cache of IOVAs. To avoid synchronization overheads of multiple cores accessing the IOVA allocator, the OS maintains a per-core IOVA cache, allowing IOVAs to be recycled on a per-core basis.

2.2 Understanding Memory Protection Inefficiencies

In this subsection, we build an in-depth understanding of the impact and root causes of inefficiencies in modern memory protection mechanisms.

Measurement setup. We use two servers directly connected via a switch to ensure that bottlenecks are at the host. Our 4-socket Intel Cascadelake servers use Xeon Gold 6234 3.3GHz processors with 8 cores per socket, DDR4 DIMMs with 2 memory channels (with maximum theoretical memory bandwidth of 46.9GBps), a 100Gbps Mellanox CX-5 NIC and 128 Gbps PCIe 3.0. Both servers use Ubuntu 20.04 and Linux kernel v6.0. We use the widely-deployed and open-sourced DCTCP [5] transport protocol with 4K bytes MTU size, 256 MTU-sized packets worth of ring buffer size and all hardware offloads (*e.g.*, TSO, GRO, aRFS, and Dynamically-Tuned Interrupt Moderation (DIM) [11]) enabled to maximize network throughput with minimum latency. For better explicability of our observations, we disable Direct Cache Access (Intel's Data Direct IO, or DDIO [23]). Unless mentioned otherwise, we use five cores and one flow per core since this is the minimum required to saturate the 100Gbps link; we also study the impact of varying number of flows, ring buffer size, MTU size, number of cores, and enabling/disabling DDIO on IOMMU performance. We use Iperf [15] to generate traffic, with one flow per core. We use PCM counters [25] to measure the number of misses experienced in IOMMU caches per page worth of packet data.

[Figures 2a, 2b, 3a & 3b] Modern memory protection mechanisms result in significant reduction in application-level throughput and increase in data drop rates at the host. Figure 2a and Figure 3a show the impact of memory protection on application-level throughput for increasing number of flows and increasing ring buffer sizes, respectively, with all other parameters fixed to default. With the IOMMU disabled, the application saturates 100Gbps link bandwidth; a slight degradation in throughput with larger number of flows is due to CPU bottlenecks since larger number of flows reduce opportunities for batched processing of data packets within the network stack [11]. Enabling IOMMU results in 20−65% lower throughput. For the IOMMU enabled cases, CPU was far from utilized. Figures 2b and 3b show that memory protection overheads can result in a large fraction of packets—as much as 4%—being dropped at the host.

Figure 9 in §4 shows that enabling memory protection can also result in orders-of-magnitude inflation in tail latency for latency-sensitive applications, when colocated with throughput-bound applications (*e.g.*, in multi-tenant deployments). The reason for such a latency inflation is the same as studied in [2]: latency inflation at P99 is dominated by queueing delay at the NIC buffer, and latency inflation at P99.9 is dominated by retransmission timeouts after packet drops.

Understanding the root cause of inefficiency in memory protection mechanisms: latency inflation due to address translation. The root cause of application-layer performance degradation when enabling memory protection is increase in address translation latency due to two compounding factors: IOTLB misses requiring page table walks, and the poor IO PTcache performance (i.e., PTcache-L1/L2/L3 miss rates) leading to higher latency per page table walk. Intuitively, this increase in latency results in buffers at the processor-side end of the PCIe device starting to fill up. Once buffers fill up to their limit (modern PCIe devices allow buffering up to ~100 cachelines [2, 37, 44]), PCIe can no longer keep enough requests in flight resulting in PCIe link underutilization. This

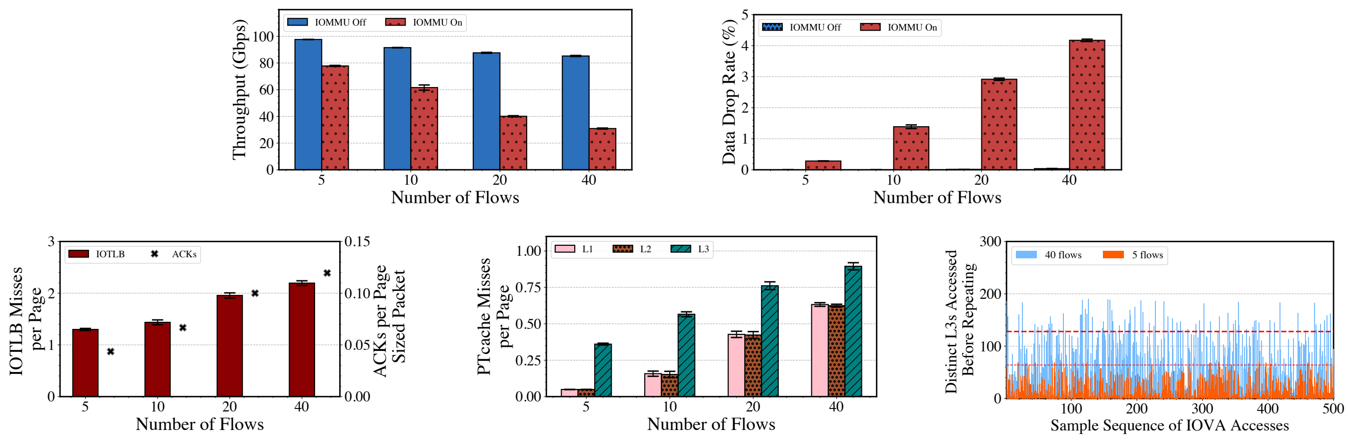


Figure 2. Modern memory protection mechanisms have high overheads due to large number of IOTLB misses and IO page table cache misses, resulting in $20-65\%$ application-layer throughput degradation. With IOMMU enabled and increasing number of flows, we observe: (a) higher throughput degradation; (b) larger packet drop rates; (c) larger number of IOTLB misses; (d) higher PTcache misses; and (e) poorer locality in PTcache-L3. See §2.2.

ultimately results in NIC buffers building up, eventual packet drops at the NIC, and throughput degradation.

A simple model[1] provides more insights on the above. Let p be the size of packet (4KB in our default setup), l_0 be the average latency for DMA'ing a packet in absence of memory protection, M be the average number of memory reads due to IOTLB and PTcache misses per packet, and l_m be the average IOMMU-to-memory read latency per packet. In this model, both l_0 and l_m incorporate any form of parallelism (*e.g.*, DMA engine issuing multiple parallel requests, IOMMU page table walkers issuing multiple parallel IOMMU-to-memory reads, etc.) while computing the average values. Then, the maximum achievable PCIe throughput is given by $T = p/(l_0 + M \cdot l_m)$. If the end-to-end application throughput is bottlenecked by the PCIe throughput, this model gives us an estimate of the end-to-end application throughput.

For any given experiment, we can calculate the number of memory reads per packet using the IOTLB and PTcache-L1/L2/L3 misses. Specifically, as discussed in §2.1, a miss at any level of the PTcache leads to one additional memory read; thus, the average number of memory reads for address translation for a packet worth of data is given by $m_{\text{IOTLB}} + m_1 + m_2 + m_3$, where m_{IOTLB} is the average number of IOTLB misses (for a packet worth of data) and m_i is the average number of misses at level i that also led to misses in all levels $> i$ (again, for a packet worth of data). Figures 2c, 2d, 3c and 3d show m_{IOTLB} and m_i for the respective experiments. For instance, for the 5 and the 40 flow cases in Figure 2, we observe 0.05 and 0.63 PTcache-L1 misses, 0.05 and 0.63 PTcache-L2 misses, 0.36 and 0.90 PTcache-L3 misses, and 1.3 and 2.20 IOTLB misses per 4KB worth of data, respectively. Overall, for the 5 and the 40 flow case, these misses result in 1.76 and 4.36 memory reads per 4KB worth of data.

Even with modern servers, it is hard to measure l_0 and l_m that depend on the parallelism with DMA and IOMMU hardware. To that end, as an approximation, we use the two datapoints from our 5 flow and 10 flow experiments to estimate these values. Fitting the aforementioned simple model along with measured throughput and memory reads for individual experiments, we get $l_0 = 65ns$ and $l_m = 197ns$.

Given the number of IOTLB, PTcache-L1/L2/L3 misses, and the packet size, the above simple model allows us to estimate the application-layer throughput within 10% of the measured throughput across most of the experiments in this paper.

[Figures 2c & 3c] Reasons for IOTLB misses: safety properties and Rx/Tx interference. Figure 2c and Figure 3c show the number of IOTLB misses with increasing number of flows and with increasing ring buffer sizes, respectively. We make two observations. First, the number of IOTLB misses is always greater than 1—this is fundamental—as discussed in §2.1, existing memory protection mechanisms require unmapping IOVA-to-physical page mappings immediately upon every DMA request completion. Thus, an IOTLB miss per (huge)page worth of data is unavoidable: since IOVAs are unmapped and corresponding IOTLB entries are immediately invalidated after use, IOTLB entries cannot be re-used by another subsequent DMA request, even if it re-uses an IOVA. Thus, the first PCIe transaction (belonging to a DMA request to a single page) will always experience an IOTLB miss.

The second observation is that, in many cases, IOTLB misses can be larger than 1. This is because PCIe transactions after the first transaction may also observe an IOTLB miss depending on various factors including IOTLB size and interference due to simultaneous PCIe transactions (for concurrent

[1]This model is similar to that in [1], with the main difference that m in our model depends on both IOTLB and PTcache-L1/L2/L3 miss rates.

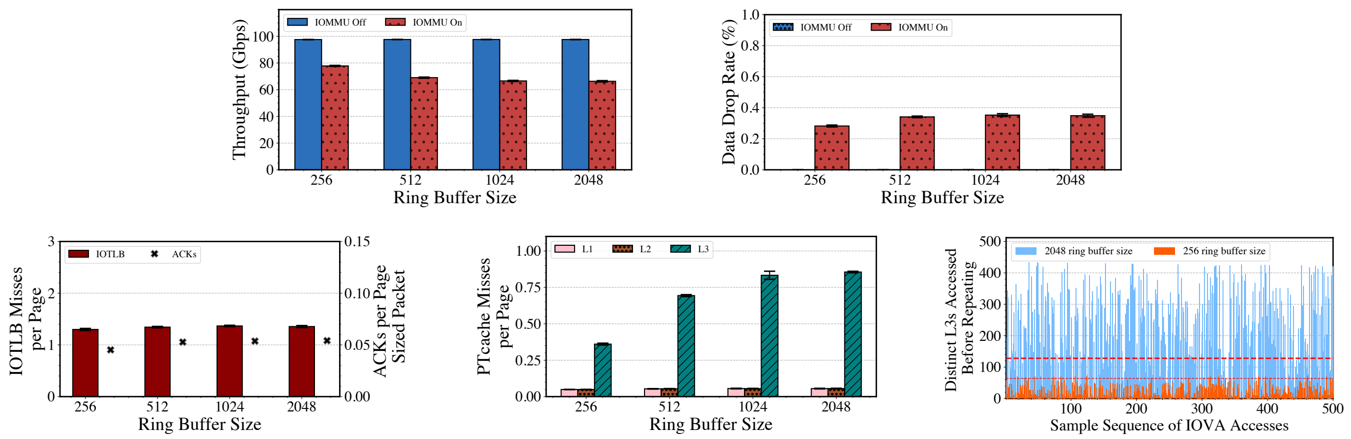


Figure 3. Large number of PTcache-L3 misses result in as much as 15% higher application-layer throughput degradation due to memory protection mechanisms. With IOMMU enabled and increasing ring buffer size, we observe: (a) higher throughput degradation; (b) slightly increasing packet drop rates; (c) relatively constant IOTLB misses; (d) higher PTcache-L3 misses; and (e) poorer locality in PTcache-L3. See §2.2.

DMAs, for accessing descriptors, Tx packets, acknowledgement packets in transport protocols) that also require address translation and contend for the same IOTLB [1]. As a concrete example, Figure 2c shows increase in IOTLB miss rates with increasing number of flows. This phenomenon is mainly caused due to increase in the number of ACKs generated per Rx packet with increasing number of flows, resulting in even greater contention for IOTLB. The increase in number of ACKs sent with increasing flow count is primarily due to underlying transport behavior. It is a well-known phenomenon that AIMD-based congestion control protocols like DCTCP used in our setup lead to larger drop rates with increasing number of flows [2, 3, 36] (we also observe similar trend in Figure 2b). Such an inflated drop rate may also lead to larger number of out-of-order packets at the transport layer and hence an increase in number of ACKs sent [11]; Figure 2c provides experimental evidence for the increase in Tx packets with increase in number of flows.

[Figures 2d & 3d] Reasons for PTcache-L1/L2 misses: cache invalidations. Figure 2d and Figure 3d show that there are non-zero misses in PTcache-L1/L2, and in some cases (*e.g.*, with number of flows = 40) there can be a PTcache-L1/L2 miss for every other packet arriving to the NIC! This may be surprising at first glance because, in both of these experiments, only a single entry is accessed for IO page table levels PT-L1 and PT-L2. The reasoning is as follows: IOVAs are 48bits long, and thus they span an address space of size 2^{48}; since each PT-L1 and PT-L2 page contains $512 = 2^9$ entries, each PTcache-L1 and PTcache-L2 entry covers 2^{39} and 2^{30} bytes of address space, respectively. On the other hand, the active IOVA address space size is given by $2 \times$CPU cores$\times$MTU bytes$\times$ring buffer size, where MTU size is rounded down to the nearest power of two[2]. For our evaluated scenarios, this turns out to be at most 2^{27} (using 2048 ring buffer size, 4KB MTU size and 5 cores). Existing IOVA allocators ensure that the set of active IOVAs *across all descriptors* are allocated compactly from the top of the address space [39]; thus, the 2^{27} bytes of the IOVA address space require only a single PTcache-L1/L2 entry and we would never expect PTcache-L1/L2 misses.

Digging deeper, we found that these PTcache-L1/L2 misses stem from invalidations of previously DMAed IOVAs that share the same PTcache-L1/L2 entry. This is because, as discussed in §2.1, modern memory protection mechanism in Linux invalidates both IOTLB entries and IO PTcaches upon an IOVA unmap. These invalidations can lead to IO PTcache misses despite good access locality of the IOVAs, especially in levels PT-L1 and PT-L2 of the page table.

The impact of these invalidations becomes more pronounced in presence of Rx/Tx interference. For example, Figure 2d shows that the PTcache-L1/L2 misses significantly increase with increase in number of flows. This is because concurrent Tx DMAs for sending ACK packets lead to PTcache-L1/L2 misses by invalidating page table caches used by IOVAs for Rx DMAs (we focus on ACKs since they are the only Tx packets in our evaluation setup). Since Rx and Tx packets share the same IOVA address space (IOVA address space is per device in non-virtualized scenarios like our evaluation setup), their respective address translations use the same page table cache entries. As a result, Tx packets interfere with Rx packets: a page table cache invalidation for a Tx IOVA can cause PTcache-L1/L2 miss for an Rx IOVA. This correlation between increase in Tx packets and increase in PTcache-L1/L2 misses can be seen in Figure 2c (increase

[2]Intuitively, this is because our NIC is allocated enough pages to handle twice as many MTU size packets as specified by the ring buffer size (we are unaware of the reasons for the extra factor of 2).

in ACK packets) and Figure 2d (increase in PTcache-L1/L2 misses); from Figure 3c and Figure 3d, we get another confirmation for this—with the same rate of ACKs, we observe the same rate of PTcache-L1/L2 misses.

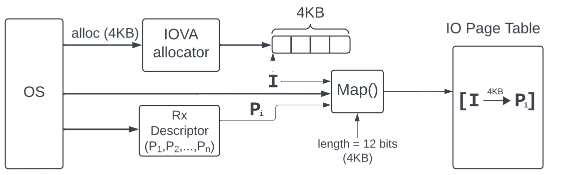
[Figures 2d, 2e, 3d & 3e] Reasons for PTcache-L3 misses: cache invalidations and poor locality. Figure 2d and Figure 3d show that PTcache-L3 observes much larger number of misses per page worth of data compared to PTcache-L1/L2. This is because of two reasons. First, for exactly the same reasons as PTcache-L1/L2 misses discussed above, PTcache invalidations also contribute to PTcache-L3 misses.

The second reason is that, unlike PT-L1 and PT-L2 where only a single page table entry is required to span the entire IOVA working set, the number of entries required for PT-L3 is over 64 for our setup. The larger working set size combined with poor locality between allocated IOVAs leads to an increased number of PTcache-L3 misses, as entries are evicted before they can be reused.

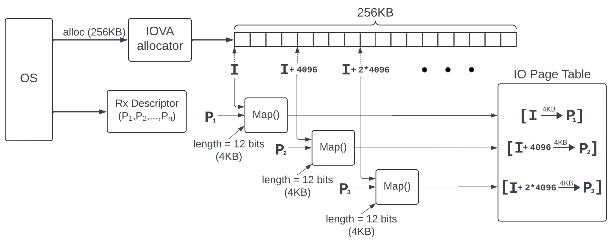
To understand the above, recall from §2.1 that descriptors are filled with IOVAs by calls to the IOVA allocator. The IOVA allocation pattern directly corresponds to the IOMMU access pattern when packets are DMA'd using IOVAs within a descriptor ring buffer. Unfortunately, as studied in depth in [32], existing IOVA allocators fail to provide good locality due to a hard trade-off between CPU efficiency and good locality inherent within their design. Specifically, the per-core caches allow IOVA allocators to achieve high CPU efficiency by avoiding expensive operations on red-black trees (as discussed in §2.1) but can lead to poor locality: allocation and free calls by different cores (each of which uses its own ring buffer) or even by the same core but for the Rx and the Tx datapaths (that also use their respective ring buffers) can result in degradation of locality within the caches over time. On the other hand, always executing allocation and free calls from the red-black tree can lead to improved locality, but requires high CPU overheads.

Figure 2e and Figure 3e provide evidence that existing IOVA allocators fail to provide good locality. The X axis represents the index for each subsequent IOVA allocation; and the Y axis represents the number of unique PTcache-L3 page table entries used before that same PTcache-L3 entry is used by an allocated IOVA again. The red line thresholds represent potential PTcache-L3 cache sizes of 64 and 128[3]. A value above the threshold means it is more likely that an IOVA will experience an L3 miss when translated by the IOMMU. Figure 2e shows that the locality of PTcache-L3 entries degrades with increasing number of flows due to increased interference between Rx and Tx datapaths (as in the IOTLB case, increasing the number of flows results in larger interference between Rx and Tx packets). The worsening locality observed in Figure 3e is due to increase in IOVA working set size. In this experiment, larger ring buffer sizes do not result in increase in the number of ACKs and much fewer invalidations contribute to PTcache-L3 misses; instead, the IOVA working set size (and corresponding PTcache-L3 working set size) increases by 8× with an 8× increase in ring buffer size, resulting in poor IOVA locality.

[3]The IO page table cache sizes are not public; these numbers represent a likely range of cache sizes based on our measurements.



(a) Linux IOVA allocation and IO page table mapping.



(b) F&S IOVA allocation and IO page table mapping.

Figure 4. F&S enables contiguous IOVA allocation without requiring changes to IOMMU hardware, IOVA allocator and IOMMU driver. Discussion in §3.

3 Fast & Safe Memory Protection

We now present Fast & Safe, a simple modification to existing memory protection mechanisms that near-completely eliminates the memory protection overheads while providing the strict safety guarantees as in modern operating systems.

The strict safety memory protection model in Linux necessitates at least one IOTLB miss per page-worth DMA: each IOVA must be unmapped, and the corresponding IOTLB and IO page table cache entries must be invalidated upon every DMA completion. Thus, F&S focuses on reducing the cost of each IOTLB miss.

F&S design minimizes the cost of each IOTLB miss using three key ideas applied to both the Rx and the Tx datapath: (1) contiguous IOVA allocations to improve IOVA locality and reduce PTcache-L3 misses; (2) preserving PTcache during invalidations to retain benefits of IOVA locality; and (3) exploiting contiguity in IOVAs to perform invalidations in a CPU-efficient manner.

Contiguous IOVA allocations for improved locality. Figure 4a shows the default IOVA allocation mechanism in Linux for Rx datapath (we discuss some minor differences in Tx datapath later in this section): for each page, the OS allocates a 4KB IOVA via a call to the IOVA allocator. The page is then mapped to the IOVA in the IO page table; the map function requires specifying the starting address of the

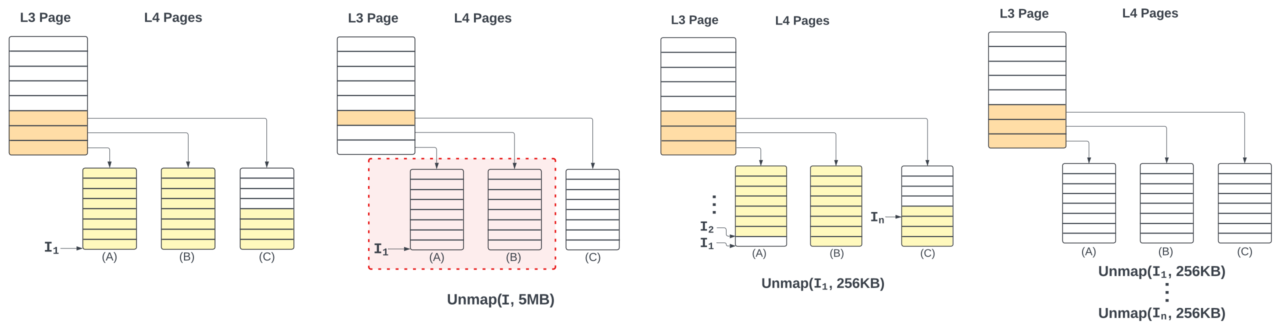


Figure 5. Linux IO page reclamation. For simplification, pages (2MB) are shown with 8 entries; yellow highlight represents a mapped region and orange highlight represents a valid PT-L3 entry. (a) Initial page table state: 5MB of physical pages are mapped to IOVA with starting address I_1; (b) Large unmap of I_1 in a single call results in pages (A) and (B) being reclaimed (red zone)—the PT-L3 entries that pointed to them are now invalid; (c) An unmap call of 256KB does not lead to page reclamation, since the unmap call does not cover the entire range of a page; (d) Many consecutive 256KB unmap calls also do not lead to page reclamation even when the entire 5MB address range is unmapped since no single call covers the entire range of a page.

IOVA, the starting address of the page, and a 4KB length. As discussed in §2.2, existing IOVA allocators do not guarantee good locality resulting in a large number of PTcache misses.

The key insight in F&S is that unlike the context of conventional Linux memory management (where the applications themselves dictate the access pattern for virtual addresses), the IOVA access pattern for networking applications is known a priori: IOVAs within a single descriptor are accessed in a sequential manner. F&S exploits this insight to achieve improved locality by allocating a large, contiguous, IOVA and mapping it to consecutive pages within a descriptor (e.g., descriptors used by recent Mellanox drivers consist of 64 pages per descriptor). Specifically, as shown in Figure 4b, F&S allocates a contiguous descriptor-sized chunk (256KB for 64 pages in Rx descriptor) of the IOVA while maintaining the mapping granularity—each physical page in the descriptor is still mapped to a page-sized segment of the IOVA range.

F&S's technique of allocating a large IOVA but mapping individual physical pages has two benefits: (1) it does not require changes to the underlying hardware; and (2) it ensures good locality within a descriptor. Such improved IOVA locality allows F&S to reduce the number of PTcache-L3 entries per descriptor: existing mechanisms may have as many as 64 PTcache-L3 entries per descriptor worth of data (i.e. each IOVA has a unique PTcache-L3 entry), leading to potentially 64 PTcache-L3 misses per descriptor (or one miss per page). On the other hand F&S ensures that there are no more than 2 unique PTcache-L3 entries per descriptor[4], leading to potentially $2/64 = 0.031$ PTcache-L3 misses per page (of course, there may be more misses in the presence of PTcache-L3 contention, such as from Tx datapath).

[4]one for the beginning of the IOVA range, mapped to the first page in the descriptor (**I** in Figure 4b), and another one if the IOVA range spans multiple PT-L4 pages. For example, if **I** is the last mapping in a PT-L4 page, then **I**+4096 will be on another page and will have a different PTcache-L3 entry.

There is an implementation difference between the Tx and the Rx datapaths in terms of the way the descriptors are set up by the NIC driver. Unlike the Rx datapath where there is a constant 64 pages per descriptor, there is no fixed number of pages per Tx descriptor (we find that in practice it is often less than the 64 pages used for Rx descriptors). This discrepancy exists because for Tx packets, the physical pages pointed to by descriptors are furnished by the Linux networking stack (depending upon the rate of generation of Tx packets from the applications, and/or ACKs from the transport). This is unlike the Rx datapath where pages are allocated beforehand by the NIC driver. Each Tx packet, independent of its size, must be mapped to a separate 4KB IOVA, since a page is the smallest supported mapping granularity in the IOMMU.

We generalize F&S's technique to the Tx datapath as follows: in order to allocate a 256KB IOVA (same granularity as Rx datapath), the 4KB chunks of IOVA space are mapped contiguously to pages *across* descriptors, in the same order that they will be accessed by the NIC; this is identical to the illustration in Figure 4b, except rather than coming from a single descriptor, each page, P, can possibly come from multiple adjacent descriptors. Once the entirety of a given 256KB IOVA has been mapped, a new IOVA is allocated.

Realizing F&S's technique of contiguous IOVA allocation requires minimal modifications within the operating system. We utilize pre-existing structures in the ring buffer to keep track of metadata (e.g. when a new IOVA must be allocated).

Preserving IO PTcaches during invalidations to retain benefits of IOVA locality. Modern memory protection mechanism in Linux opts to invalidate both IOTLB entries and IO PTcaches upon an IOVA unmap. While IOTLB invalidation is necessary to ensure correctness/safety guarantees, IO PTcache invalidation is necessary only if an IO page table page is reclaimed during an IOVA unmap operation. For example, if a PT-L4 page is reclaimed, then the PTcache-L3 entry pointing to that page would become stale and require an invalidation. Despite the above, Linux takes a safe (but potentially sub-optimal) approach of always invalidating the PTcache-L1/L2/L3 whenever an IOVA is unmapped.

F&S makes the observation that IO page table page reclamation is extremely rare in the context of networking applications. To understand why, consider the current reclamation strategy employed by Linux: an IO page table page is reclaimed only if there is a single operation unmapping the entire address range that it covers [39][5]. Concretely, in order to reclaim a PT-L4 page (thus invalidating the PTcache-L3 entry that points to it) the entire 2MB IOVA range on that page must be unmapped in a single operation (as shown in 5b). To invalidate a PTcache-L2 entry and reclaim a PT-L3 page, the entire 1GB range of IOVAs pointed to by that PTcache-L2 entry must be unmapped in a single operation.

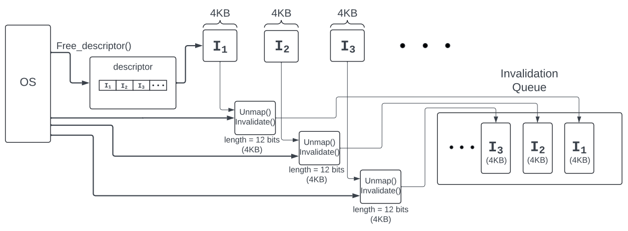
F&S exploits the above observation to avoid memory protection inefficiencies due to unnecessary PTcache invalidations. Specifically, the strict safety guarantees in modern memory protection mechanisms require performing unmap operations at the granularity of an individual descriptor (at most 64 pages)—far from large enough to cause reclamation of an IO page table page; hence, IOMMU page table cache entries will become stale extremely rarely (as shown in 5d) and it is safe to preserve PTcaches in the common case. Thus, upon each IOVA unmap operation, F&S only invalidates the corresponding IOTLB entry while preserving the PTcache-L1/L2/L3 entries. To keep F&S general and prevent stale IOMMU page table cache entries from being accessed in the case of page table page reclamation, F&S ensures that page table caches are invalidated whenever an unmap operation leads to page table page reclamation.

This simple change allows the page table caches to re-use the cached entries, even after IOVA invalidations. This enables F&S to effectively take advantage of improved IOVA locality: successive IOVA accesses are much more likely to share the same PTcache-L1/L2/L3 entries under good locality.

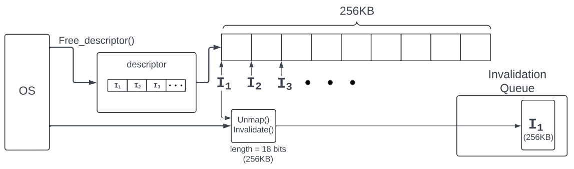
Realizing F&S design idea of selectively preserving IO PTcaches requires no modifications within the memory protection datapath. IOMMU hardware provides an invalidation queue as the interface to the IOMMU driver to request IOVA invalidations; the invalidation queue interface specifies an option to only invalidate the IOTLB entry while preserving the IO page table cache entries. F&S uses this interface to set the option to prevent the invalidation of associated page table caches when invalidating IOTLB entries.

Exploiting contiguity in IOVAs to perform invalidations in a CPU-efficient manner. In addition to a higher latency-cost for each IOTLB miss, invalidations also lead to a higher CPU-cost per IOTLB miss [7, 32, 34, 39, 42], since upon each invalidation request, the initiating CPU core waits until successful invalidation of IOTLB and/or PTcache-L1/L2/L3 entries by the IOMMU hardware [39].

[5]This reduces synchronization costs associated with reclaiming page table pages. In particular, as studied in [39], safely reclaiming a page table page requires mechanisms to check that the page has no entries and to remove its reference from the parent page, all while ensuring that no new mappings are created. Such mechanisms usually incur high CPU overheads.



(a) Linux invalidations.



(b) F&S invalidations.

Figure 6. F&S, by exploiting its contiguous IOVA allocation mechanism, enables a single entry in the invalidation queue to invalidate an entire descriptor's worth of IOVA. Discussion in §3.

The invalidation queue interface specifies an IOVA starting address and a size, which correspond to the IOVA range that will be invalidated in the IOMMU caches. Existing memory protection mechanisms are unable to efficiently use the IOMMU invalidation queue interface. This is because existing IOVA allocation mechanism do not guarantee contiguous IOVA allocations (§2.2); thus, the OS must make a request to the invalidation queue for each 4KB IOVA in a descriptor. Figure 6a illustrates this process: for each IOVA in the descriptor, the OS requires a call to unmap and invalidate, leading to an invalidation queue entry for each IOVA in the descriptor. Therefore, increased invalidation requests also results in a degradation of per-core application throughput (results in [42]).

The final design idea of F&S is to leverage the contiguous IOVA allocation: with perfect contiguity, the IOVA range mapped to a descriptors worth of pages can be invalidated with a single request to the invalidation queue (figure 6b). F&S, using these batched invalidations, amortizes the CPU cost of invalidating all IOVAs within a single descriptor.

Generality of F&S techniques. We have primarily focused on F&S design for the case of multi-page descriptors. However, F&S techniques may also be useful for network devices that do not support multi-page descriptors (*e.g.*, Intel ICE [26]). Specifically, F&S contiguous IOVA allocation and page table cache preservation apply directly even to the extreme case of single-page descriptors (as discussed for the Tx datapath above). The F&S technique of batched invalidations will not

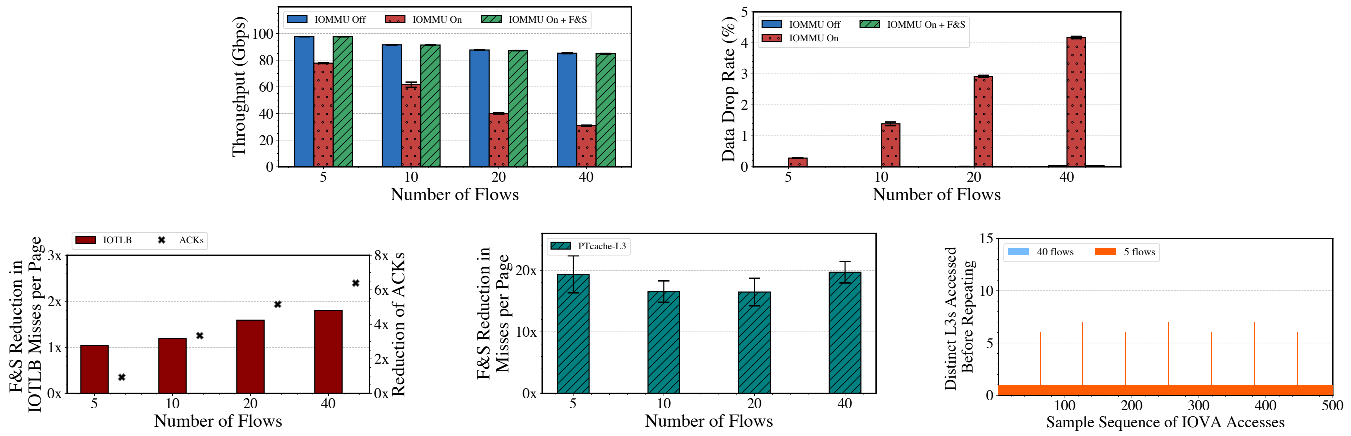


Figure 7. F&S near-completely eliminates memory protection overheads. (a) F&S achieves throughput close to IOMMU off; (b) eliminates packet drops due to memory protection; (c) decreases IOTLB miss rates by 2× due to reduced number of ACKs (due to lower drop rates); (d) reduces PTcache-L1/L2 misses to zero and reduces PTcache-L3 misses by an order of magnitude, resulting in significantly lower cost of IOTLB misses. (e) achieves good locality for IOVAs. See §4.1 for more discussion.

work as effectively for single-page descriptors—while F&S can allocate (contiguous) IOVAs across multiple descriptors, invalidations will need to be performed at descriptor granularity for maintaining the strict safety guarantees. We leave it to future work to evaluate F&S benefits for devices that use single-page descriptors. Nevertheless, our evaluation (§4) suggests that multi-page descriptors have significant benefits in terms of memory protection, providing motivation for a wider-scale adoption of multi-page descriptors.

F&S safety guarantees. As discussed in §2.1, modern memory protection mechanisms typically provide one of the two safety properties: strict and deferred. As defined in [34], when the strict mode is enabled, a malicious and/or faulty IO device can no longer access physical addresses corresponding to an IOVA after the IOVA has been unmapped from the IO page table; on the other hand, when the deferred mode is enabled, a malicious and/or faulty IO device may still be able to access physical addresses corresponding to an IOVA after the IOVA has been unmapped from the IO page table. In [42], we show that F&S enables the same safety properties as the strict mode under very mild assumptions that hold for most operating systems (*e.g.*, Linux).

4 F&S Evaluation

We implement F&S within the Linux kernel v6.0.3. F&S implementation requires no hardware modifications, no interface changes to the IOVA allocator and minimal changes to the IOMMU and NIC drivers. F&S changes ~630 LOC, primarily to establish a new datapath in the IOMMU driver for contiguous IOVA allocation. In this section, we use this implementation along with Mellanox NICs (that use 64-page descriptors by default) to demonstrate that F&S significantly reduces the cost of each IOTLB miss, thus eliminating memory protection overheads for almost all evaluated scenarios while providing strict safety properties. Unless mentioned otherwise, we use the same setup as in §2.2.

4.1 Understanding F&S Benefits

Figures 7, 8, and 9 demonstrate that F&S enables applications to achieve the same throughput as when IOMMU is disabled for almost all evaluated scenarios (we discuss later in §4.4 the reason for tiny gap between F&S and IOMMU disabled throughput for large ring buffer sizes in Figure 8a). F&S also near-completely eliminates packet drops (the 40 flows scenario has 0.036% packet drops, but this matches the drop rate when IOMMU is disabled) for all evaluated scenarios. F&S achieves similar benefits with varying data transfer sizes, core counts, and modern direct cache access mechanisms [42]. Additionally, F&S enables the latency-sensitive RPC application to achieve tail latencies within 1.17× the IOMMU disabled case across all evaluated percentiles and RPC sizes (except P99.99, which is ~1.42×), as shown in Figure 9.

F&S achieves its performance benefits by reducing PTcache-L1/L2/L3 miss rates—bringing PTcache-L1/L2 cache misses to 0, and >10× reduction in PTcache-L3 misses per page in all cases (at most 0.045 and 0.053 PTcache-L3 misses per page in Figures 7d and 8d, respectively). In addition, even though F&S design does not explicitly target reducing IOTLB misses, it does reduce IOTLB misses in many cases (*e.g.*, almost by 2× in the 40 flow case).

We now dive deeper into F&S techniques that enable reduction in PTcache-L1/L2/L3 misses (and also IOTLB misses, albeit indirectly). For brevity we focus on the case of increasing flow sizes (Figure 7); the same insights also apply to all other experiments. By eliminating unnecessary invalidations of IO page table caches, F&S minimizes PTcache-L1/L2/L3 miss rates. Recall from §2.2, each PTcache-L1 and PTcache-L2 entry covers 2^{39} and 2^{30} bytes, respectively. In our evaluation setup, the IOVA working set size is 2^{27} bytes; thus we get 0

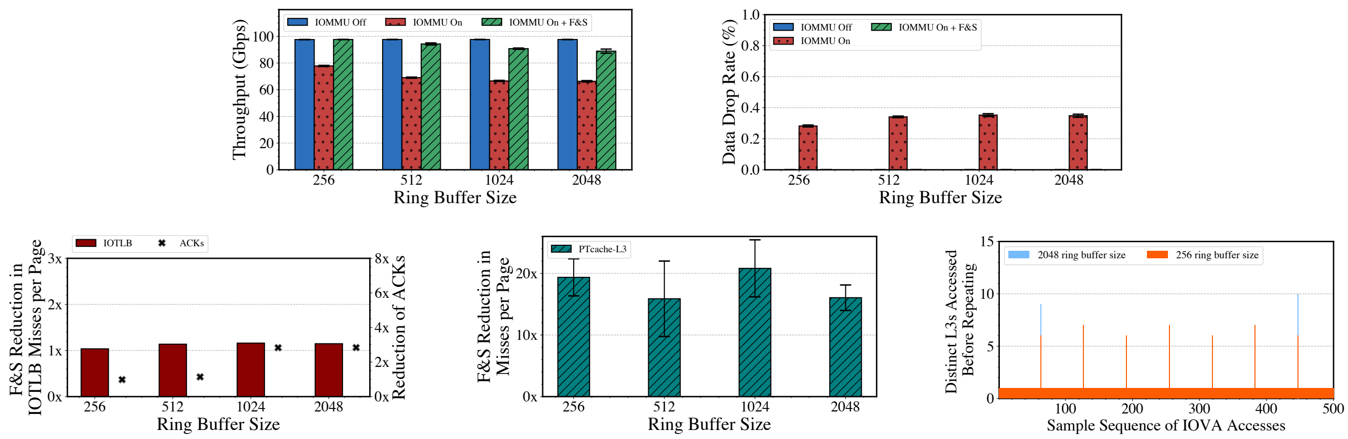


Figure 8. F&S maintains good locality, resulting in low L3 cache misses even as the I/O working set sizes increase. With IOMMU enabled and increasing ring buffer sizes, we observe that F&S achieves several key performance metrics: (a) nearly-matches the performance of systems with IOMMU disabled (reason for slight degradation is described in [42]); (b) experiences no packet drops; (c) maintains a consistent number of IOTLB misses; (d) incurs low IOMMU cache misses; and (e) demonstrates good locality in L3 page caches. See §4.1 for more discussion.

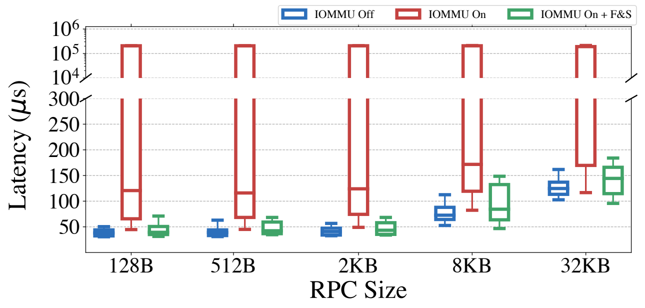


Figure 9. F&S enables latency-sensitive applications to achieve tail latency within a factor of 1.42 of the IOMMU off case. The figure shows P50, P90, P99, P99.9, P99.99 latencies (represented by whiskers). We run a latency-sensitive RPC application (running concurrently with our throughput-bound application–iperf); we use netperf [19] to generate RPCs with sizes varying from 128B-32KB between the sender and receiver hosts (similar to [2]). The RPC application is run on a separate core from throughput-bound application on both the hosts to avoid any CPU interference. See §4.1 for more discussion.

PTcache-L1/L2 misses. Qualitatively, such low PTcache-L1/L2 miss rates are not an artifact of our workloads; in fact, even with much larger number of cores, ring buffer sizes, and network bandwidths, the IOVA working set size suggest that F&S will still lead to 0 PTcache-L1 misses and near-zero PTcache-L2 misses. Specifically, assuming a cache size of 32, PTcache-L1 working set would need to cover over 2^{44} bytes and PTcache-L2 working set cover over 2^{35} bytes to see any PTcache-L1/L2 misses. Using the same formula from §2.2 to calculate the IOVA working set size, we see that even with 9K MTUs, 4096 ring buffer size and up to 512 cores (resulting in 2^{35} bytes) there would be 0 PTcache-L1/L2 misses; for larger MTU sizes, ring buffer sizes and/or number of cores, F&S contiguous IOVA allocation would help to reduce the number of misses, but they could not be completely avoided.

Figure 7d shows that F&S results in a 20× reduction in the number of PTcache-L3 misses, compared to existing memory protection mechanisms. This is due to F&S's contiguous IOVA allocation technique that leads to improved locality independent of the number of flows (as shown in Figure 7e). As discussed in §3, contiguous IOVA allocation using F&S guarantees that every descriptor will contain at most two unique PTcache-L3 entries for our evaluation setup (in the majority of cases, all IOVAs within a descriptor will in fact share the same PTcache-L3 entry). The few spikes in Figure 7e occur at descriptor boundaries: since IOVAs are allocated at descriptor granularity, subsequent descriptor's IOVAs can possibly have different PTcache-L3 entries. For all our evaluated scenarios, these spikes do not lead to any performance degradation. Furthermore, F&S ensures that PTcache-L3 performance no longer depends on IOVA working set size, since locality is now guaranteed at per descriptor granularity (2 PTcache-L3 entries in the worst case). Thus, future increase in ring buffer size and/or MTU size would not affect F&S's performance in terms of PTcache-L3 misses. However, PTcache-L3 contention (*e.g.*, due to Rx/Tx interference and/or larger number of cores creating larger number of concurrent translations) can lead to additional PTcache-L3 misses.

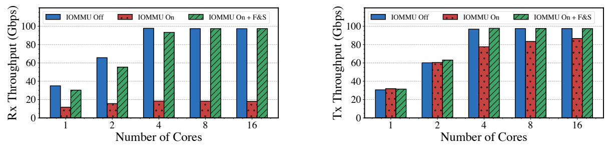
While F&S does not explicitly try to reduce the IOTLB miss rate, we still observe its reduction as an indirect outcome of the F&S performance benefits discussed earlier. Specifically, by reducing packet drop rates, F&S leads to a reduction in the number of ACKs per page-worth data (illustrated by crosses in Figure 7c). Consequently, the decrease in interference from ACKs results in fewer IOTLB misses (Figure 7c).

Finally, F&S helps resolve tail latency inflation in Figure 9 by reducing delays due to packet queueing and retransmissions: the reduced address translation cost due to F&S alleviates the PCIe link underutilization problem (discussed in §2.2), and thus leads to reduction in NIC queue build-up and subsequent packet drops.

F&S performance with concurrent Rx and Tx traffic. To understand how F&S performs when pushed to extreme scenarios with respect to IOMMU cache contention, we run an additional experiment where the servers have (1) both Rx and Tx data packets at the same time (unlike our default setup where the receive-side server only had ACK packets for Tx); and (2) have larger number of CPU cores, and hence run larger number of concurrent flows (without CPU being the bottleneck). As described in §2.2, increasing amount of Rx/Tx interference leads to larger number of IOTLB misses (due to increased IOTLB contention) and larger cost for each IOTLB miss (due to increased PTcache-L1/L2/L3 misses). Further, Rx/Tx interference worsens with increasing number of flows due to increasing amount of drops and ACKs (see discussion for Figure 2).

For this experiment, we use a different server setup with a larger CPU core count per socket. We use Intel Icelake servers, with Xeon Platinum 8362 processors and 32 cores per socket. Each socket has 8 3200MHz DDR4 memory channels with one DIMM each. The rest of the server configuration and evaluated applications are the same as our default setup, except that DDIO cannot be disabled on these servers. We present results comparing DDIO on and off in [42]: we find that enabling DDIO only improves average CPU utilization, while having negligible impact on IOMMU cache performance and F&S benefits.

Figure 10 shows the results of this experiment. We create separate flows for performing Rx and Tx data transfers on both the servers, with each flow running on a separate core. As the number of cores increases, so does the amount of Tx/Rx interference, until the link is saturated in both directions (with 4 cores each for Tx/Rx). As expected, since the flows are run on different cores, increasing the Tx throughput does not affect the Rx throughput when IOMMU is disabled. With IOMMU enabled, increased IOTLB misses and cost per IOTLB miss due to Rx/Tx interference leads to up to $\sim 80\%$ degradation in throughput even with 4 flows (compared to $\sim 20\%$ without Tx data traffic with 5 flows). On the other hand, despite an inflated IOTLB miss rate, F&S is still able to match IOMMU disabled performance by reducing the cost of each IOTLB miss for all evaluated scenarios (except for Rx throughput when number of cores <4, where we have a small gap between F&S and IOMMU disabled performance; we discuss the reason in §4.4). Note that Tx throughput observes smaller degradation compared to Rx throughput with IOMMU enabled due to a known result that PCIe read transactions can tolerate larger inflation in per transaction latency (eg., latency inflation due to address translation overheads in this context) than write transactions before the PCIe link becomes underutilized [44].



(a) Rx Throughput **(b)** Tx Throughput

Figure 10. F&S provides significant benefits even in the case of extreme Rx/Tx interference. (a) Rx throughput; and **(b)** Tx throughput. See discussion in §4.1.

4.2 Real World Applications with F&S

We now demonstrate that F&S provides above benefits over real-world applications. We use three applications (i) Redis, a widely deployed in-memory key-value store [41]; (ii) Nginx, a widely used web server [38]; and (iii) SPDK, a high-performance userspace storage stack that can work with Linux TCP stack for remote/disaggregated storage [20, 21, 45].

Applications have their own CPU overheads. To focus on scenarios where the IOMMU (and not CPU) is the primary bottleneck, we use the same two-server setup as in §2.2 but with two minor exceptions: we use 8 cores and 9K MTU sizes; this allows each applications to saturate 100Gbps link. We focus on the Rx datapath, that is, the hosts running server instances for Redis and Nginx, and the host running client instances for SPDK.

Redis. We use one of the hosts to run one Redis server instance per core, and the other host to run client threads. We find that Redis is able to saturate 100Gbps link bandwidth using 32 requests in-flight at any point of time (referred to as pipelining/parallelism in Redis). We use the 100% SET request workload—each request has 4B keys and value sizes from $4 - 128$KB. We vary the number of clients for each configuration and report the optimal aggregate throughput.

Figure 11a shows that enabling default memory protection results in $38-70\%$ throughput degradation in Redis; smaller value sizes worsen throughput degradation. Enabling memory protection with F&S alleviates performance degradation while providing strongest memory protection. There is a small gap between IOMMU disabled and F&S performance at 4KB value size, which we discuss in §4.4.

Nginx. We run a single Nginx server instance per core on one host and run client threads on the other host. We use the wrk HTTP benchmark [17], with 128KB-2MB web page sizes (average web page size today is around 2MB [22]). We vary the number of client threads and report the optimal throughput for each web page size.

Figure 11b shows that Nginx running without memory protection achieves a maximum of 90Gbps application-level

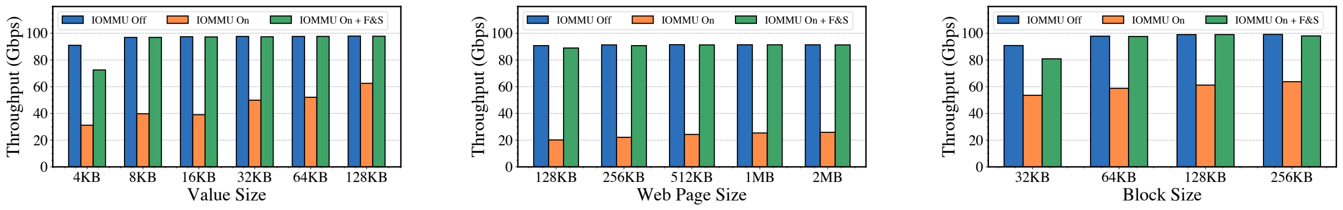


Figure 11. F&S enables real-world applications to achieve memory protection with near-zero overheads. (a) Redis; **(b)** nginx; and **(c)** SPDK. See discussion in §4.2.

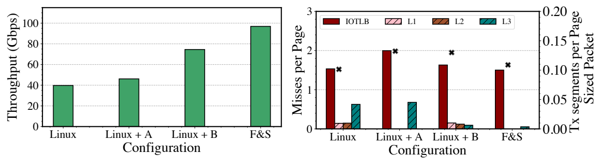


Figure 12. Contribution of each key idea of F&S. A is preserving page table caches; B is contiguous IOVA allocation and batched invalidations; discussion in §4.3.

throughput due to its application layer overheads (we observe the link bandwidth is fully utilized); enabling default memory protection results in 65–70% reduction in throughput across all web page sizes. Enabling memory protection using F&S completely eliminates these overheads for all web page sizes while providing the strongest memory protection.

SPDK. We run an SPDK server instance on each core on one host and distribute client threads equally among the cores on the other host. The client threads send SPDK read requests of varying block sizes (32KB-256KB) to the server. We use an IO-depth of 8 requests because that achieves the highest throughput (similar results observed in [21]). Figure 11c shows the aggregate throughput measured at the client host.

SPDK without memory protection saturates the link bandwidth; and, enabling memory protection results in SPDK performance dropping to a maximum of ~60Gbps across all experiments. F&S is able to provide the strongest memory protection while matching IOMMU disabled performance. There is a small gap between IOMMU disabled and F&S performance at 32KB value size, which we discuss in §4.4.

4.3 Necessity of each F&S Idea

Figure 12 shows that each of three key F&S design ideas—contiguous IOVA allocations, perserving IO page table cache invalidations, and batching invalidations—are necessary to achieve its performance benefits.

We run the same setup as the Redis experiment (Figure 11a). We focus on 8KB value size, for four F&S configurations: (i) default Linux, (ii) Linux + A (preserve IO page table caches), (iii) Linux + B (contiguous IOVA allocation + batching invalidations) and (iv) Linux + F&S. Since we cannot batch invalidations without contiguous IOVA allocations, we club both ideas in Linux + B configuration. We observe that simply preserving PTcaches is not sufficient due to large number of PTcache-L3 misses. Recall that PTcache-L3 misses occur due to PTcache invalidations and poor locality—preserving PTcaches only fixes the first issue. Likewise, while contiguous IOVA allocation + batched invalidations improves the PTcache-L3 locality, and reduces the total number of invalidations, the remaining PTcache-L1/L2/L3 misses significantly reduce the application throughput. F&S, using all its techniques, is able to achieve high throughput.

4.4 Understanding the few scenarios where F&S doesn't match IOMMU disabled performance

F&S enables matching the IOMMU disabled throughput for almost all evaluated settings. However, in a few scenarios, F&S observes a tiny gap compared to IOMMU disabled performance. This is primarily due to two reasons: CPU overheads due to memory protection operations (mapping/unmapping and invalidations) and Rx/Tx interference leading to inflated IOTLB misses. We discuss both these scenarios below.

In Figure 8a, despite F&S resulting in similar reduction in IOTLB and PTcache-L1/L2/L3 miss rates for both 256 and 2048 sized ring buffers, we observe slightly lower throughput with larger ring buffer sizes compared to IOMMU disabled case (eg., ~10Gbps for 2048 size ring buffer). This is because F&S becomes CPU-bottlenecked for the 2048 ring buffer size; digging deeper, we found that larger ring buffer sizes result in larger CPU cache miss rates for data reads due to decreased efficiency of hardware prefetching [42]. In comparison to IOMMU disabled case, F&S does have tiny CPU overheads for memory protection operations; these overheads result in the gap between F&S and IOMMU disabled for this specific case. Enabling DDIO, increasing MTU sizes, and/or using more cores alleviates this bottleneck, allowing F&S to achieve the same throughput as IOMMU disabled case even with 2048 ring buffer size [42]. The additional CPU overheads of memory protection operations are also the reason for the slight gap between F&S and the IOMMU disabled case in Figure 10.

The second reason for F&S throughput degradation compared to IOMMU disabled is inflated IOTLB misses. Recall that F&S does not directly attempt to reduce the number of IOTLB misses, only the cost of each miss. In the Redis experiment in

Figure 11a, the gap between F&S and IOMMU off for 4KB value size exists because of a larger rate of IOTLB misses using Redis with small value sizes; IOTLB misses per page increase by $\sim 2\times$ at 4KB value sizes (when compared to 128KB value sizes). This happens because Redis sends an application-level reply for every request (similar to ACK packets for each segment received in microbenchmarks); and, smaller value sizes increase the Tx sending rate, resulting in increased IOVA translations and thus IOTLB contention. SPDK application performance at smaller block sizes (Figure 11c) is similar to the Redis result at 4KB value size: we observe ~ 1.5x increase in IOTLB misses compared to 256KB block size due to larger rates of Tx packets (SPDK clients sends small-sized request packets for each read response from server). We discuss potential complementary approaches to F&S to reduce IOTLB misses in §5.

5 Related Work

We are not aware of any other work that utilizes IO page table caches in modern servers to design high-performance IO memory protection mechanisms that guarantee strict safety. Below, we discuss the most closely related work.

Software-based approaches. There have been several software based approaches to reduce address translation overheads [9, 16, 32–34, 39]. Some of these approaches specifically focus on reducing the CPU overheads in address translation [32]; others explore reducing the cost of address translation in terms of memory accesses by resorting to weaker security properties [9, 16, 39].

F&S design reduces the address translation overheads by lowering the cost of each IOTLB miss, but not explicitly attempting to reduce the IOTLB misses themselves. Hugepages can potentially be used to reduce the number of IOTLB misses by increasing the reach of each IOTLB cache entry. A recent work uses hugepages to reduce IOTLB contention [16]; however, they target a weaker safety property because they keep IOVAs permanently mapped to pages in the IO page table and not invalidating them after each DMA. Integrating hugepages with F&S to further reduce memory protection overheads (fewer IOTLB misses) while providing strict memory protection is an interesting direction for future work.

DAMN [34], building upon its predecessor [33], claims to achieve strong safety properties while achieving high performance. Unfortunately, we could not confirm either of these claims. DAMN has two implementations: compatible with the Linux kernel v4.7 and v5.1. We could not get the former kernel image to boot. We were able to boot into the latter, but found that it suffers from the same performance degradation as the default Linux strict mode. When we inquired with the authors, they suggested that their implementation in v5.1 is incomplete. We provide more details in [42].

Conceptually, DAMN cannot avoid performance degradation of modern memory protection mechanisms under the strict mode due to the following reason. The key idea in DAMN is to keep persistent mappings between IOVA to physical addresses until the page is explicitly freed, and recycle pre-mapped pages within descriptors. This avoids unmapping and invalidation overheads under the (implicit) assumption that applications are reading data faster than the rate at which the NIC is DMAing the data. When this assumption does not hold true, DMA'd pages can be overwritten (since they are never unmapped, the NIC still has access to them); to ensure that the data is never overwritten, after all the pre-mapped pages are consumed, DAMN continuously allocates new pages and creates new mappings, resulting in large active IOVA address space. Without any additional mechanisms, DAMN will thus suffer from the same IOTLB and PTcache-L1/L2/L3 miss rates and subsequent performance degradation as in modern memory protection mechanisms.

Irrespective of the above, DAMN requires intrusive modification in the operating system memory management mechanisms. F&S, on the other hand, guarantees strict safety properties while alleviating the overheads of memory protection mechanisms with minimal modifications.

Hardware-based approaches. There has been significant work on rearchitecting IOMMU hardware to reduce the cost of address translation, *e.g.*, by reducing the number of IOTLB misses [7] and/or invalidation overheads [8, 31]. While F&S focuses on reducing the cost of address translation with minimal changes within the operating systems, it would be interesting to integrate these hardware approaches with F&S to further reduce memory protection overheads.

6 Conclusion

We have presented Fast & Safe, a simple modification to existing memory protection mechanisms that enables them to provide strongest safety properties, and yet, near-completely eliminates their overheads. Fast & Safe builds upon the observation that modern memory protection hardware has IO page table caches that store most frequently accessed entries from each level of the IO page table, and that these caches can be used to minimize the overheads of IO memory protection. That is, rather than solely focusing on minimizing IOTLB misses, Fast & Safe focuses on minimizing the cost of each IOTLB miss. We demonstrate that this change of perspective enables a simple F&S design that requires no modifications in host hardware, minimal modifications within the operating system, and yet, near-completely eliminates the overheads of existing memory protection mechanism.

Acknowledgments

We would like to thank our shepherd and SOSP reviewers for insightful feedback. We would also like to thank Midhul Vuppalapati for useful discussions. This research was in part supported by NSF grant CNS-2047283, and a Sloan fellowship.

References

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding host interconnect congestion. In *ACM HotNets*.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *ACM SIGCOMM*.

[3] A. Aggarwal, S. Savage, and T. Anderson. 2000. Understanding the performance of TCP pacing. In *IEEE INFOCOM*.

[4] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *ACM EUROSYS*.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM*.

[6] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *USENIX ATC*.

[7] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *ISCA*.

[8] Arkaprava Basu, Mark D Hill, and Michael M Swift. 2017. I/O memory management unit providing self invalidated mapping. (2017). US Patent 9,547,603.

[9] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, Leendert van Doorn, and Doorn Amd. 2007. The price of safety: Evaluating IOMMU performance. In *OLS*.

[10] Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches. In *IEEE/ACM MICRO*.

[11] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[12] Kevin K. Chang. 2017. Understanding and Improving the Latency of DRAM-Based Memory Systems. *CoRR* (2017).

[13] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. *ACM SIGMETRICS* (2016).

[14] Dilip Chhajed and Timothy Lowe. 2008. *Building Intuition: Insights From Basic Operations Management Models and Principles.* Vol. 115.

[15] Jon Dugan, John Estabrook, Jim Ferbuson, Andrew Gallatin, Mark Gates, Kevin Gibbs, Stephen Hemminger, Nathan Jones, Gerrit Renker Feng Qin, Ajay Tirumala, and Alex Warshavsky. 2021. iPerf. https://iperf.fr/. (2021).

[16] Alireza Farshin, Luigi Rizzo, Khaled Elmeleegy, and Dejan Kostić. 2023. Overcoming the IOTLB wall for multi-100-Gbps Linux-based networking. *PeerJ Comput. Sci.* (2023).

[17] Will Glozer. 2021. wrk HTTP benchmark. https://github.com/wg/wrk. (2021).

[18] Yuchen Hao, Zhenman Fang, Glenn Reinman, and Jason Cong. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*.

[19] HewlettPackard. 2021. Netperf. https://github.com/HewlettPackard/netperf. (2021).

[20] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *NSDI*.

[21] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *USENIX OSDI*.

[22] Jamie Indigo, Dave Smart, Chris Steele, and Danielle Rohe. 2022. Web Almanac 2022: Page Weight. (2022). https://almanac.httparchive.org/en/2022/page-weight#request-bytes

[23] Intel. 2012. Intel® Data Direct I/O Technology. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf. (2012).

[24] Intel. 2023. Intel Virtualization Technology for Directed I/O. https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html. (2023).

[25] Intel. 2024. Intel Performance Counter Monitor . https://github.com/intel/pcm. (2024).

[26] Intel. 2024. Intel® Network Adapter Driver for E810 Series Devices under Linux. https://www.intel.com/content/www/us/en/download/19630/intel-network-adapter-driver-for-e810-series-devices-under-linux.html. (2024).

[27] Taehun Kim, Hyeongjin Park, Seokmin Lee, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. 2023. DevIOus: Device-Driven Side-Channel Attacks on the IOMMU. In *IEEE Symposium on Security and Privacy*.

[28] Sanjay Kumar. 2008. *New abstractions and mechanisms for virtualizing future many-core systems.* Georgia Institute of Technology.

[29] Alexey Lavrov and David Wentzlaff. 2020. HyperTRIO: hyper-tenant translation of I/O addresses. In *ACM/IEEE ISCA*.

[30] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin Chang, and Onur Mutlu. 2015. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *IEEE HPCA*.

[31] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *ACM ASPLOS*.

[32] Moshe Malka, Nadav Amit, and Dan Tsafrir. 2015. Efficient intra-operating system protection against harmful DMAs. In *USENIX FAST*.

[33] Alex Markuze, Adam Morrison, and Dan Tsafrir. 2016. True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy. In *ACM ASPLOS*.

[34] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. 2018. DAMN: Overhead-Free IOMMU Protection for Networking. In *ACM ASPLOS*.

[35] Benoît Morgan, Éric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2016. Bypassing IOMMU Protection against I/O Attacks. In *LADC*.

[36] Robert Tappan Morris. 1997. TCP behavior with many flows. In *IEEE ICNP*.

[37] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *ACM SIGCOMM*.

[38] nginx. 2024. nginx. https://nginx.org/en/. (2024).

[39] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU scalably. In *USENIX ATC*.

[40] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *USENIX OSDI*.

[41] Redis. 2024. Redis. https://redis.io. (2024).

[42] Benny Rubin, Saksham Agarwal, Qizhe Cai, and Rachit Agarwal. 2024. Fast & Safe IO Memory Protection. In *tech report*.

[43] Sami Vaarala and Jukka Manner. 2006. Security considerations of commodity x86 virtualization. *Helsinki University of Technology, Telecommunications Software and Multimedia Laboratory* (2006).

[44] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the host network. In *ACM SIGCOMM*.

[45] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *IEEE CLOUDCOM*.