HyperTRIO: Hyper-Tenant Translation of I/O Addresses

Alexey Lavrov
Department of Electrical Engineering
Princeton University
Princeton, NJ, USA
alavrov@princeton.edu

David Wentzlaff
Department of Electrical Engineering
Princeton University
Princeton, NJ, USA
wentzlaf@princeton.edu

***Abstract*—Hardware resource sharing has proven to be an efficient way to increase resource utilization, save energy, and decrease operational cost. Modern-day servers accommodate hundreds of Virtual Machines (VMs) running concurrently, and lightweight software abstractions like containers enable the consolidation of an even larger number of independent tenants per server. The increasing number of hardware accelerators along with growing interconnection bandwidth creates a new class of devices available for sharing. To fully utilize the potential of these devices, I/O architecture needs to be carefully designed for both processors and devices.**

This paper presents the design and analysis of scalable Hyper-tenant TRanslation of I/O addresses (HyperTRIO) for shared devices. HyperTRIO provides isolation and performance guarantees at low hardware cost by supporting multiple in-flight address translations, partitioning translation caches, and utilizing both inter- and intra-tenant access patterns for translation prefetching. This work also constructs a Hyper-tenant Simulator of I/O address accesses (HyperSIO) for 1000-tenant systems which we open-sourced. This work characterizes tenant access patterns and uses these insights to address identified challenges. Overall, the HyperTRIO design enables the system to utilize full available I/O bandwidth in a hyper-tenant environment.

***Index Terms*—I/O subsystem, virtualization, address translation**

I. Introduction

Massive server consolidation in data centers drives higher server utilization, reduced energy usage, and lower operating cost. This consolidation is enabled by collocating on the same physical server multiple *tenants* in the form of Virtual Machines (VMs) [8], [14], machine containers [11], or application processes [20]. All these tenants have to be isolated from each other but still share computational, storage, and I/O resources.

The ever-increasing number of cores per server enables the consolidation of even more tenants, which can reach the order of thousands. A single server using eight Xeon scalable processors [22] has 448 cores in total. Assuming a 4:1 CPU oversubscription ratio [27], such a server would have up to 1792 concurrent tenants. On top of the large number of cores, commodity servers have an ample amount of the main memory (up to 1TB) [4]. As an example, Firecracker can run hundreds or thousands of lightweight MicroVMs per server, depending on their configuration [4]. Drawing an analogy with hyperscale systems, we name a platform with such a large number of tenants "a *hyper-tenant* platform."

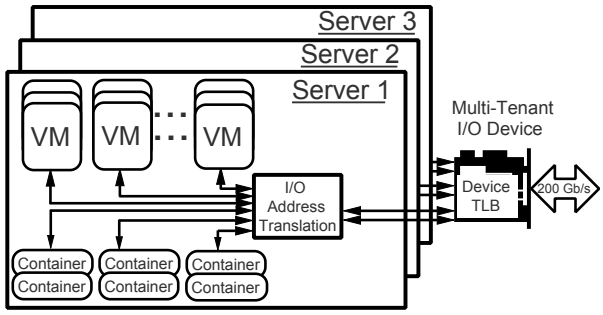


Fig. 1: High-bandwidth I/O device shared by multiple tenants.

A separate but related trend is that the available I/O bandwidth continues to grow [1]. Network Interface Cards (NICs) with 100Gb/s ports are readily available [31], and a standard for 400Gb/s Ethernet has started to be used. In addition, some NICs have technology enabling them to be shared between different servers, making the total number of tenants per device to be even higher [31]. Given this high degree of sharing and extreme available bandwidth, it becomes challenging to architect a system which can fully utilize all available resources, in particular, available I/O bandwidth.

The challenge that this paper addresses is the severe under-utilization of available I/O bandwidth as the number of tenants approaches the hyper-tenant regime (Figure 1). We identify that this severe I/O bandwidth underutilization is caused by the lack of scalability in the I/O address translation subsystem - IOMMU design, device design, and software structures. HyperTRIO solves these challenges by supporting many outstanding I/O address memory translations, partitioning the Device Translation Lookaside Buffer (DevTLB) to enforce performance isolation between tenants, and the introduction of an intelligent, hyper-tenant-aware, address translation prefetcher which utilizes both inter- and intra-tenant information. Further, we show that simply scaling up the size of the DevTLB is insufficient to enable Hyper-tenant I/O.

The key contributions of this work are:

- Architectural design and evaluation of Hyper-tenant TRanslation of I/O addresses (HyperTRIO). We investigate the interaction between I/O device and system

memory and use cloud benchmarks as a case study.
- The creation and open source release of a Hyper-tenant Simulator of I/O address accesses (HyperSIO) used for analysis and performance evaluation[1].
- Detailed analysis of inter- and intra-tenant interactions during I/O address translation.
- Study of IOTLB replacement policies, partitioning, parallel address translation, and prefetching of I/O address translations.

II. Background and Motivation

This section provides background on how I/O device sharing currently performs and sets up the challenges involved with translating I/O address mappings for high-bandwidth devices with large tenant count. We include a case study of how utilization of I/O bandwidth scales with present day translation schemes and how current designs are sub-optimal as the number of concurrent connections increases.

A. Device Sharing and Address Translation

Single Root I/O Virtualization (SR-IOV) provides a way for one physical device to be shared between multiple independent tenants [38]. Such I/O devices can be viewed as a number of separate PCIe devices, each of them represented by a Virtual Function (VF). For example, some NICs [32], [36] and GPUs [34] have up to one thousand VFs. In addition, multi-host technology allows the sharing of one physical device between up to four hosts [30], [35], [36], further increasing the total number of tenants. Each VF can be independently used by a tenant, and it provides isolation, and low virtualization overhead while efficiently using available hardware resources.

To further decrease the involvement of the tenant's CPU when moving data between main memory and I/O device, modern-day processors use direct memory access (DMA). When communicating with the tenant's main memory via DMA, the device uses guest I/O virtual addresses (gIOVA) to read/write data from/to it. These addresses are generated by a tenant's OS, and they provide a flexible manner for accessing a shared device by multiple isolated units at the same time. However, all of the gIOVAs must be translated to host physical addresses (hPAs) before reading/writing data in the memory. I/O Memory Management Units (IOMMUs) implement this functionality for both non-virtualized and virtualized environments. In the latter case, translation takes the form of a two-dimensional page-table walk [12], shown in Figure 2. Every access to a guest page table (labeled as the first-level walk) incurs a walk through a host page table (second-level walk). This two-dimensional walk is expensive and requires 24 or 35 memory accesses for 4-level or 5-level page tables [21], [23] respectively on the x86-64 architecture. In order to reduce the number of memory accesses, IOMMUs can have translation caches (L[1-4]TLBs in Figure 2) or nested TLBs [12], which store translations from guest physical to host physical addresses.

[1]http://parallel.princeton.edu/hypersio.html

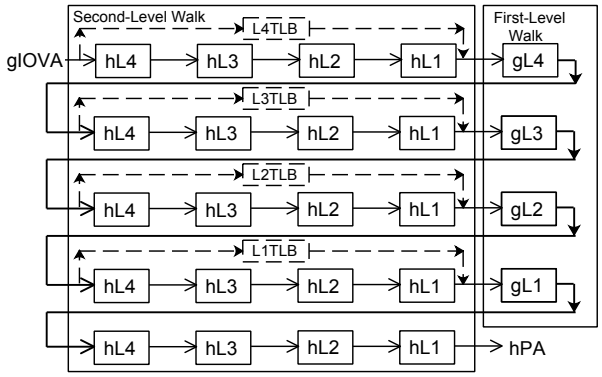


Fig. 2: Two-dimensional 4-level page table walk with 4 KB pages. gIOVA - guest I/O Virtual Address. hPA - host Physical Address. gLx, hLx - page table entries corresponding to level 'x' of the page table. L[1-4]TLB - Translation Caches.

As an example in Figure 3, we show the translation steps performed for each incoming packet. First, a tenant places a gIOVA into its corresponding ring buffer, located inside of the packet handling logic, which is later read upon the arrival of a packet. After identifying the source ID (SID) (e.g. PCIe Bus/Device/Function) for a request, the device looks up in Context Cache (CC in Figure 3) ① to find a corresponding Context Entry (CE) which contains a pointer to the base of the second-level paging entries and Device ID (DID) ② configured by the host. To accelerate translation from gIOVA to hPA, the device can have a cache to store the most recent translations shown as the Device Translation Lookaside Buffer (DevTLB). It is checked for a request ③, and the device sends a request to the system over PCIe ⑤ with a translated address in the case of a hit ④, and with an untranslated address otherwise. The translation subsystem (IOMMU), located in the chipset in Figure 3, translates the gIOVA ⑥, performing a two-dimensional Page Table Walk (PTW) when there is a miss in the DevTLB. There can be multiple hardware structures for caching page-table entries (L[1-4]TLBs) and for caching translations from gIOVA to hPA (IOTLB) to accelerate the Page Table Walk. When an entry is not found in a corresponding caching structure, the IOMMU accesses main memory ⑦ to retrieve a page-table entry (PTE). After the page table walk is finished, the hPA is returned to the device ⑧, and it finishes the requested read/write operation.

Throughout this work, we focus on hyper-tenant environments where every tenant requires a two-dimensional page table walk to translate its guest I/O Virtual Address (gIOVA) to a host Physical Address (hPA). This typically happens when a tenant has a form of a VM, while containers do not require the long walk [40]. However, the isolation of the latter ones usually raises lots of concerns in hyper-tenant setups, and lightweight MicroVMs are used instead [4].

B. Case-Study of a SR-IOV NIC

As a motivating case study, we use two real-world systems - one with a server running on an AMD Ryzen 9 3900X

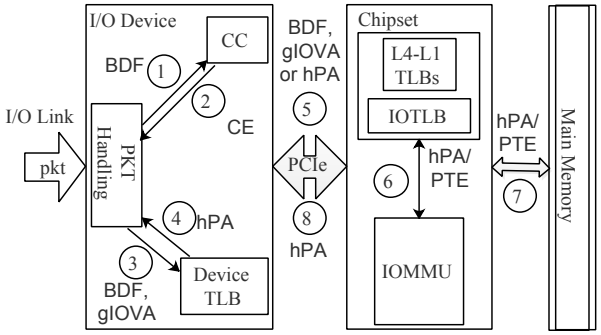


Fig. 3: Translation steps performed for gIOVA triggered by an incoming packet to a device. BDF - PCIe Bus/Device/Function triplet. CC - Context Cache. CE - Context Entry. gIOVA - guest I/O Virtual Address. hPA - host Physical Address. IOMMU - I/O Memory Management Unit. IOTLB - I/O Translation Lookaside Buffer. PTE - Page Table Entry.

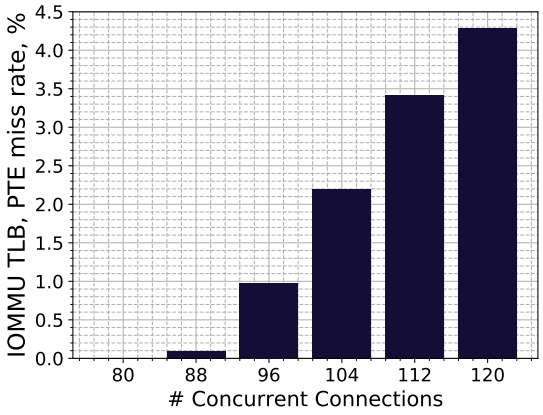


Fig. 4: IOMMU TLB PTE miss rate versus number of parallel *iperf3* connections between 80 and 120 on an AMD system (Server Host 1).

CPU, and another one with a server running on an Intel Xeon E7-4870 (Table I). Using these systems, we measure the bottlenecks for IOMMU translation for a multi-tenant use case. For each machine we use a dual-port Intel X540-T2 NIC as an instance of a shared device, supporting a maximum of 63 VFs per port [19]. As a workload, we chose *iperf3 v3.1.3* [3] to create connection pairs between two machines over a 10Gb/s link. On both client and server, core affinity was set to a different CPU for every connection to avoid resource contention and improve single stream locality.

AMD – Firstly, we perform experiments using the AMD machine for running *iperf* servers and the Client Host for *iperf* clients. Using IOMMU performance counters available on the server, we record the number of IOMMU TLB PTE hits/misses and the number of nested IOMMU page reads as we vary number of parallel connections between 2 and 120. We interleave VFs between two available PFs. Every *iperf* server was running inside a VM with one of the NIC's VFs directly assigned to it. For all the experiments, the total bandwidth was around 12.1Gb/s, which was the same as when using non-virtualized configuration (it is less than the expected 20Gb/s due to the NIC design, which was also found in other studies [2]). Using the recorded statistics, we calculate the TLB PTE miss rate, which is less than 0.1% when there are less than 80 connections. However, for a larger number of connections we observe increasing miss rate for up to 4.3% for 120 connections (shown in Figure 4). In addition to increasing IOMMU TLB PTE miss rate, the number of nested page table reads increases more than 400 times for 120 connections compared to 80 connections. These results indicate that a large number of tenants cause contention for cached translations and it will be even more challenging in a hyper-tenant setup.

Intel - We perform a second study using Intel CPUs and a single 10Gb/s link. In this study we compared I/O bandwidth when running native versus virtualized (using VF's) network interfaces.

In one case (native), the server (Server Host 2) and the client were running directly on the host and natively sharing a single network interface. In the other case, every *iperf* server was run inside a VM with a directly assigned VF to it. The results of our experiments are shown in Figure 5. We observe that a single connection using a host interface can utilize up to 8.7Gb/s, which is less than the 9.49Gb/s of useful bandwidth possible for 1500B packets on a 10Gb/s link. This behavior is caused by a CPU bottleneck on the server side and can be removed by using faster cores. At the same time, the maximum achievable bandwidth for the connection using a VF is only 6.7Gb/s, which is lower than the direct (non-virtualized) host interface speed. This can be explained by virtualization overhead.

When the number of client-server pairs increases, bandwidth per connection goes down, therefore removing the CPU bottleneck (since in our configuration, every client and server is pinned to a different core). As the number of connections is varied between two and eight, the physical link is utilized 99% in both experiments. However, when the number of connection pairs exceeds eight, the total bandwidth for configurations using VFs starts to decrease, flattening out at around 0.5Gb/s for more than 16 pairs. In contrast, increasing the number of client-server pairs does not affect total bandwidth in the case of running on the host directly.

From the above experiments, we conclude that it is contention for a shared IOVA translation resource in the virtualized setup which ultimately limits the utilization of available I/O bandwidth. Every tenant's OS allocates a number of pages for use by its device independently, and translations of gIOVAs for every tenant start thrashing the DevTLB, L[1-4]TLBs, and overloads the IOMMU with a large number of requests, as the experiment with the AMD host showed. Unfortunately, the Intel system does not provide the same level of performance counter visibility as the AMD system for IOMMU, therefore we measured the effect indirectly. Using the Hyper-tenant Simulator of I/O described in Section IV, we show that contention for the shared translation resources causes I/O link

TABLE I: System parameters for case-study of SR-IOV NIC.

	Server Host 1	Server Host 2	Client Host
CPU	AMD Ryzen 9 3900X 1 socket, 24 threads	Xeon E7-4870, 4 sockets, 80 threads	Xeon E3-1231 v3, 1 socket, 8 threads
Chipset	x570	Intel 7500	Intel C224
Memory	64GB, 400 MB/VM	256GB, 2 GB/VM	16GB
NIC	Intel X540-T2, Driver 5.1.0-k		
Linux Kernel	Host - 5.0.0, VMs - 4.15.0	Host/VMs - 4.15.0	4.4.73
Test Duration	300s		

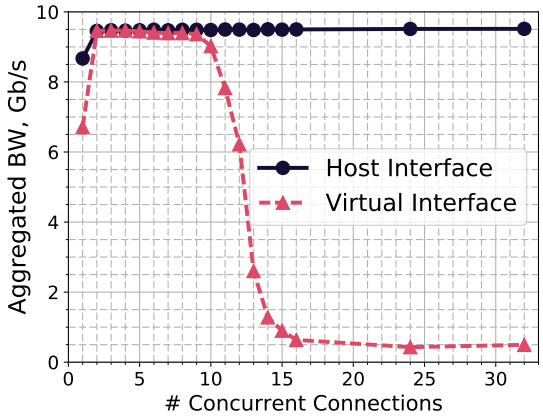


Fig. 5: Cumulative I/O bandwidth for different number of concurrent connections on an Intel Server Host 2.

utilization to go down with the increasing number of tenants in the same way as in Figure 5.

III. HyperTRIO Architecture

To remove the guest I/O Virtual Address (gIOVA) translation bottleneck in a hyper-tenant environment and enable full utilization of available I/O device bandwidth, we propose the *HyperTRIO* architecture - *Hyper-tenant TRanslation of I/O addresses*. Below we describe three main innovations augmenting the base design covered in Section II:

- *Pending Translation Buffer (PTB)* - The PTB is located inside of each a device and supports multiple in-flight translations from different tenants.
- *Partitioned Device-TLB (P-DevTLB)* - The P-DevTLB provides architectural support for translation isolation by assigning a unique tenant's tag per row of the Device-TLB.
- *Translation Prefetching Scheme* - The Prefetch Unit (PU) initiates translations of the most recent gIOVAs stored from previous tenant's accesses to IOMMU using inter-tenant information.

The HyperTRIO architecture is presented in Figure 6 with our newly added blocks shown in light gray. We analyze HyperTRIO performance in Section V.

Pending Translation Buffer (PTB). Every packet coming from an I/O link generates several gIOVA translation requests to determine the physical addresses of the corresponding ring buffer, data buffer, and address for the interrupt mailbox. For a 200Gb/s link, a 1500B packet arrives every 62ns, leaving a device working at 1.2GHz [33] only 74 cycles for all translations to be completed. This amount of available processing time between requests is even less for real-world applications. For example, in a key-value store application [7] most of the keys are under 60B, and values are under 1000B.

HyperTRIO keeps track of all in-flight gIOVA to hPA translations in a *Pending Translation Buffer - PTB*. To avoid head-of-line blocking when an IOMMU performs a two-dimensional page table walk, PTB supports out-of-order translation completion. If a new packet arriving from the I/O link cannot allocate an entry into the PTB, it is dropped. A larger buffer can prevent the dropping of a packet at additional hardware cost. However, to keep PTB size reasonable, we look for optimization in other parts of the design. For example, in the case of performing a full 4-level two-dimensional page table walk when doing a translation for 1500B packets, PTB has to keep track of 112 outstanding requests. Having such a large number of outstanding requests in hardware becomes expensive and not scalable with increasing link bandwidth and the growing number of tenants.

Partitioned Device-TLB (P-DevTLB). The Device-TLB stores translations from gIOVAs to hPAs. Since it is located on a device, it provides the fastest translation in the case of a hit and allows us to avoid expensive communication over PCIe with a chipset. In a hyper-tenant environment, every tenant will try to allocate their most recent translations in the DevTLB, causing the eviction of translations for another tenant. To exacerbate the problem, independent tenants can use the same VF driver and OS which allocates the same virtual addresses for a device (see Section IV-D). The more tenants that share a system, the higher the chance that two translations from different tenants will use the same page address and corresponding entries in the DevTLB will conflict. Since address allocation by a driver cannot be controlled from a host machine, we propose a partitioning scheme for the DevTLB which improves isolation between tenants and helps to increase utilization of available I/O bandwidth.

Every translation request received by a DevTLB contains a Source ID (SID) and/or Process Address Space Identifier (PASID) [20]. SID assignment is controlled by a hypervisor, it is known after a VF is allocated to a tenant, and it does not depend on a tenant's type. Therefore, we can use it to isolate

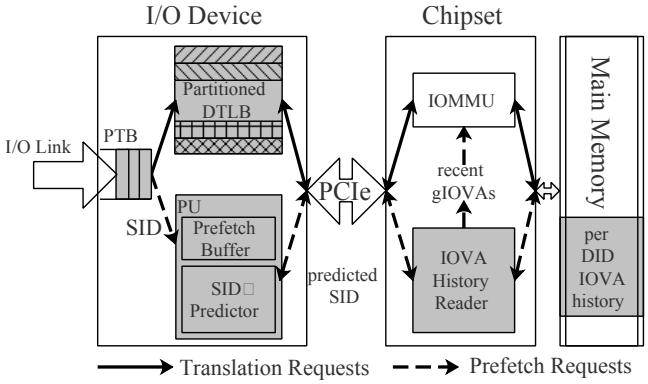


Fig. 6: HyperTRIO architecture. PTB - *Pending Translation Buffer*. *PU* - Prefetch Unit. SID - Source ID. gIOVA - guest I/O Virtual Address.

translations of independent tenants at the DevTLB.

We add a partition tag (PTag) to every row in the DevTLB, which should match with a SID in order for translation to be cached. Depending on the design, the PTag allows caching of translations only from a single tenant (complete match between SID and PTag) or for a group of tenants (matching lower bits of SID and PTag). In Figure 6, rows tagged with different PTags are cross-hatched differently. This partitioning of a single DevTLB enables performance isolation, e.g., it prevents a low-bandwidth tenant from evicting translations for high-bandwidth tenants.

Translation Prefetching Scheme. The maximum bandwidth of a tenant can be specified at configuration time just after adding it to a system. While the tenant is active, hardware load balancing and queue management enforce configured bandwidth to meet QoS, therefore providing a regular number of accesses to the translation subsystem for all tenants. We use this insight to predict the next SID based on the previously observed history. We also found that each tenant's page is accessed more than a thousand times (see Section IV-D).

We introduce a Prefetch Unit (PU) on each HyperTRIO device, which is accessed concurrently with the DevTLB (see Figure 6). The Prefetch Unit has two parts - a Prefetch Buffer (PB), and Source ID predictor (SID-predictor). The PB is a small fully-associative cache, which keeps translations from gIOVAs to hPAs and it is shared by all tenants in the system. It is populated after the completion of every prefetch request generated by a PU. The SID-predictor contains a table which directly maps from the currently accessed SID to a predicted SID and a history-length register. The latter is configured by the host and can be updated whenever bandwidth allocation per tenant changes and/or when a tenant is added/removed.

The PU is checked along with DevTLB to see if it contains a valid translation for a current translation request in a PB. If there is one in a PB, it is returned, and no further requests are generated. In the case of a miss in the PB, the SID-predictor is checked for a corresponding entry to a currently accessed SID. In the presence of a valid entry, the PU sends a prefetch request with a predicted SID to the chipset. The latter contains a gIOVA history reader, which uses a predicted SID to read the most recent translations from main memory. The IOVA history reader fetches two previously accessed gIOVAs. Instead of keeping translations from gIOVA to hPA, it issues translation requests for predicted gIOVAs to an IOMMU. This enables fetching the most recent translations from the memory when previous ones were invalidated, and at the same time gives a chance to cache intermediate translations in the L[1-4]TLBs. Later requests can benefit from having a hit to these translation caches in the case of a miss in the DevTLB and PB.

We keep only the Prefetch Buffer and SID-predictor on a device to provide low-latency in the case of a hit in the PB and to able to update the SID-predictor on every incoming packet. IOVA history reader has only a state machine for fetching the most recently accessed gIOVAs from a tenant, keeping its hardware cost independent of the number of tenants. Since prefetching is done in advance, memory access latency can be hidden by issuing translation prefetching earlier through configuring history length register in a Prefetch Unit.

IV. Hyper-tenant Simulator of I/O

To be able to study I/O devices in a hyper-tenant environment running real-world workloads, we created a Hyper-tenant Simulator of I/O - HyperSIO. It consists of three main parts:

- *Log Collector* - derived from QEMU, it models up to 24 tenants at the same time with Network Interface Cards directly assigned to them. It records all operations performed by an IOMMU while translating addresses for tenants' devices.
- *Trace Constructor* - using multiple collected logs, HyperSIO constructs translation information for every tenant and its sequence of accesses to an IOMMU. Using this information, it generates a *Hyper-Trace* which is used by the hyper-tenant performance model.
- *Trace-Driven Device-System Performance Model* - a fully custom trace-driven performance model written in C++. It incorporates detailed interaction between an I/O device, translation subsystem and host main memory using real-world latencies to compute I/O utilization.

We created HyperSIO because modern-day servers have hundreds of cores [22] and can potentially run workloads with large numbers of VFs supported by I/O devices [4], [32]. To enable analysis and experimentation with I/O sharing for varying number of tenants, all accesses between a device, a server, and an I/O link have to be recorded. Below we discuss different options which were considered for studying and evaluating such hyper-tenant systems.

Even though there exist many tools for CPU and memory instrumentation [26], they typically do not give any visibility into the chipset and do not allow recording translations of guest I/O Virtual Addresses (gIOVAs). One option is to modify the OS or IOMMU drivers and record the information about page accesses. However, modifying the OS solves only part of the problem, since some translation requests can be handled by a hardware translation cache on a device without accessing

the IOMMU, thereby preventing a researcher from seeing all memory accesses. The second option would be to record translation requests directly on a device, but this functionality is not provided by hardware vendors and would significantly impact performance of a high-bandwidth I/O device. The third option is to use available full-system architecture simulators. Yet they model processor architecture thoroughly, they lack detailed I/O simulation. On top of that, it is hardly feasible to simulate hyper-tenant systems with reasonable performance for real-world workloads when using detailed or cycle-accurate software simulation. As a result, we decided to create HyperSIO which runs real-world applications, saves I/O device logs, constructs data-structures and traces for a hyper-tenant system, and uses a fully custom performance model for studying and performance evaluation. The source code for HyperSIO can be found at *http://parallel.princeton.edu/hypersio.html*.

A. HyperSIO: Log Collector

HyperSIO's *Log Collector* is used to record accesses to an IOMMU from independent tenants running real-world workloads. As we discussed above, we need to be able to model a full system, including I/O device, system chipset, main memory, and processor. QEMU 3.0.0 can emulate all these pieces, including an IOMMU [5], which is a part of its Q35 chipset model. However, QEMU lacks a model of an I/O device supporting SR-IOV for sharing. To get around this problem, we emulate a system with multiple independent Network Interface Cards (NICs) where every NIC is directly assigned to a separate VM. Since modeling a system with thousands of tenants at the same time would require thousands of VMs, it was not feasible to do it on our server. Another QEMU limitation was that in order to directly assign a device to a VM, the former has to be connected to a PCIe root complex, which supports only 24 slots at the same time. So, instead of emulating a system with a thousand tenants, we run a configuration with fewer tenants multiple times, we record all the logs separately and use a *Trace Constructor* to generate a single trace representing all translations from a hyper-tenant system.

Figure 7 shows a detailed diagram of the emulated system used by the *Log Collector*. We model a server with 64 x86_64 cores, 208GB of main memory, Q35 chipset with an IOMMU, and 24 e1000 NICs. Since this system is emulated using QEMU, we name it a Level-1 VM (L1VM). Inside of L1VM, we run nested Level-2 VMs (L2VM) representing separate tenants, where each of them is using one NIC directly assigned to it using PCIe passthrough. As a result, every NIC is allocated into a separate IOMMU group in the L1VM, guaranteeing its isolation from other NICs.

After we assign the devices to the tenants, we send traffic through the NICs. To emulate I/O link connections for the devices we use tun/tap interfaces available as an option for QEMU. Every tenant is running a server part of real-world workload inside of L2VM, and it is connected to the client through an I/O device managed by an IOMMU. The client is connected to a tap interface on the host side, and it communicates with the server as if they were connected through a real network. The *Trace Collector* runs the same workload for every client-server pair, starting it in parallel for all the pairs and recording all translation requests received by the IOMMU.

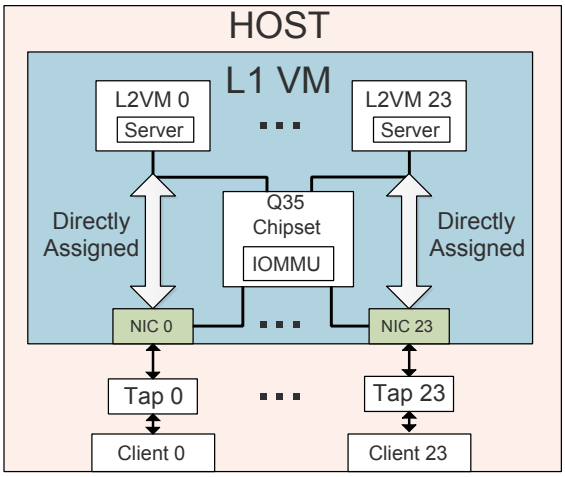


Fig. 7: Architecture of an emulated system with up to 24 NICs directly assigned to VMs.

B. HyperSIO: Trace Constructor

HyperSIO's *Hyper-Trace Constructor* produces a single trace from the logs generated by a *Log Collector* to model a hyper-tenant system. The Constructor parses accesses to a Context Cache, IOMMU, and page table entries from each trace, creates separate data-structures for translation requests, context cache entries, and page table entries, and saves them in a format supported by the Performance Model described later.

Since the *Trace Collector* records translation requests from up to 24 I/O devices for a single run, the *Hyper-Trace Constructor* has to read results from multiple runs when the number of modeled tenants is larger than the number of emulated devices. Consequently, there stems a question on how to interleave requests between tenants. Since we study scalability of I/O bandwidth utilization with the increasing number of tenants, we model the same bandwidth for all tenants in a system. The number of consecutive requests sent by every tenant can be configured through a command-line option to model bursty traffic.

In addition to configurable number of requests from a tenant, the *Trace constructor* supports two options for inter-tenant interleaving. The first one is `Round-Robin (RR)`, which is used as an arbitration scheme between queues and is found in a real NIC [19]. This scheme is efficient for hardware implementation, and models the case when tenants have steady long-lived connections providing data with an arbiter that selects the next stream. The second one is `random (RAND)`, which represents a scenario for tenants sending separate requests instead of generating a steady data stream.

TABLE II: System parameters used by performance simulator.

One-way PCIe latency [33]	450ns
DRAM latency	50ns
IOTLB hit	2ns
# memory accesses during PTW [21]	24
Packet size at I/O link	1542B (Eth Pkt + IPG)
I/O link bandwidth	200Gb/s
L2 Page Cache	512 entries, 16-ways
L3 Page Cache	1024 entries, 16-ways

C. HyperSIO: Performance Model

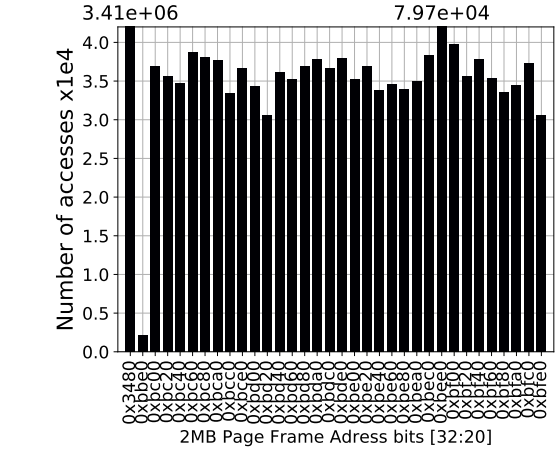
The *HyperSIO Performance Model* is a fully custom device-system model written in C++. It reads traces generated by the *Trace Constructor* which includes gIOVA translation requests, Context Cache, and page-table entries. HyperSIO models the system described in Section III in order to evaluate different architectural aspects of an I/O address translation scheme and their impact on achievable device utilization in a hyper-tenant environment. The main parameters used for modeling are listed in Table II.

HyperSIO calculates the next packet arrival time based on provided I/O link bandwidth and packet size, therefore modeling a fully utilized link. When a new packet arrives, it is placed in a Pending Translation Buffer (see Figure 6) when there is available buffer space, and it is dropped otherwise. At the next arrival time, a dropped packet is retried. Every accepted packet generates three translation requests corresponding to a translation of the ring buffer pointer, accessing the data buffer, and sending a notification to the system about a newly arrived packet. Depending on whether a miss/hit occurs in every translation structure, a request is either completed and the packet is considered processed, or a new event is scheduled in a queue for a corresponding structure. For example, if a translation request misses in the Device TLB (DevTLB), it is scheduled to access an IOMMU in the future after the PCIe traversal time. When a request hits in the DevTLB, its completion is scheduled after the hit time. Information about page table entries read from a *Hyper-Trace* is used to populate page table caches (L[1-4]TLB in Figure 6) when the IOMMU performs a page-table walk.

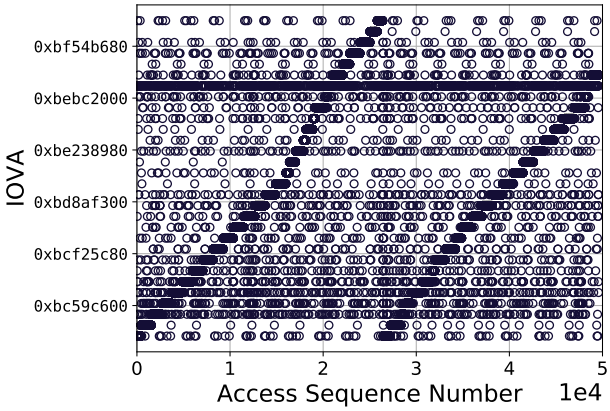
At the end of a simulation, *HyperSIO* calculates the total amount of data processed by a device by multiplying the total number of processed packets from all tenants by an average packet size. To get an average bandwidth, *HyperSIO* divides the total number of bytes by the time spent to translate all the requests. Since packets are dropped when there is no available space in a PTB, average link bandwidth is lower than nominal when the translation subsystem becomes a bottleneck. Otherwise, if all translations for a packet are finished before the next one arrives on the link, the total bandwidth is limited by I/O link throughput.

D. Single- and Multi-Tenant Characterization

Using logs recorded by the *HyperSIO Log Collector* we analyze page access patterns for single- and multi-tenant setups. From a single-tenant analysis, we make two conclusions. The first is that all accessed page frames can be split into three groups based on their frequency (Figure 8a). The second is that accesses to certain pages have a periodic pattern and every page is accessed around 1500 times in a row (Figure 8b). From the analysis of a multi-tenant trace, we observe different tenants are using the same page frame addresses, increasing the chance of evicting each other's translations from caching structures.



(a) Access frequencies.



(b) Access pattern.

Fig. 8: I/O virtual page access for a single tenant.

Single tenant characterization. To characterize single-tenant translation requests, we ran the *mediastream* benchmark and recorded around 4.6×10^6 IOMMU translation requests from a NIC. All requests were from 104 pages assigned to a tenant's I/O device by its OS. We analyzed the frequency of accesses to these page frames and found that all of them can be split into three groups based on the total number of accesses. Figure 8a shows the frequency of accesses to page frames in the first and the second groups.

The first group contains accesses to a single gIOVA page at address `0x34800000`. These accesses are to a page containing addresses of ring buffers allocated to a tenant and therefore addresses inside that page are translated for every packet arriving at the I/O link. The second group has 32 page

frames in the range from `0xbbe00000` to `0xbfe00000` which are used for data buffers. During the test, the L1VM had huge pages enabled, and therefore all page frames in the second group are for 2MB pages where each of it is accessed almost the same number of times. Since a page frame in the first group is accessed for every incoming packet, it is seen around 30 times more frequently than page frames for data buffers. The third group contains 70 page frames between `0xf0000000` and `0xffffffff`. They are 4KB in size and used only after NIC initialization with the total number of accesses less than 100 times per each page frame (not shown in Figure 8a due to limited space). So, even though there are around 5×10^6 packets, they access only around 31 pages most of the time.

Figure 8b shows the order of page frame accesses for the second group described above. It has a periodic structure since the NIC is using a ring buffer to store them. Each 2MB page is accessed around 1500 times sequentially until the driver unmaps it and starts using buffers located in the next page. The same pattern of IOVAs was observed previously [6], but during our experiments the working page set was larger.

Analysis of a single-tenant stream provides two insights into IOVA access patterns. First, page frames can be grouped based on their access frequency. When caching translations in the IOTLB, this fact can be used to decide which translation to evict in the case of a conflict. The second observation is that each 2MB page used for data buffers is accessed many times sequentially, exhibiting high temporal locality. Periodic access to those pages can be used by a prefetcher to load the next page. It also shows that switching to a new page frame happens less frequently than accesses to the same page.

Multi-Tenant Characterization. To model a multi-tenant system, we run several copies of the same workload used for a single-tenant setup. All L2VM servers use the same L1VM host and therefore share the same IOMMU. Though all VMs are independent of each other, they use the same subset of page frames for data buffers in the address range from `0xbc000000` to `0xbf000000`. This can be attributed to the fact that all VMs in our experiment run the same OS with the same version of the device driver. In a virtualized environment where a tenant has a virtual device directly assigned to it, this can often be the case since all VFs are identical to each other and the hypervisor has no control over guest virtual address assignment. As a result, gIOVA distribution can cause certain rows in the DevTLB or L[1-4]TLBs to be used more frequently than others, leading to conflicts between different tenants.

In order to study how I/O link utilization depends on the number of tenants, we used the *HyperSIO Performance Model* and ran different numbers of copies of the same workload. We assume that DevTLB has 64 entries, which is the same as the number of IOTLB entries in Intel's design [33], and use a 200Gb/s I/O link. Figure 9 presents the results of performance simulation indicating that the maximum achievable aggregated I/O bandwidth depends on the number of connections in the same way as our motivational study shown in Figure 5. Since the DevTLB is a shared resource, it becomes a bottleneck

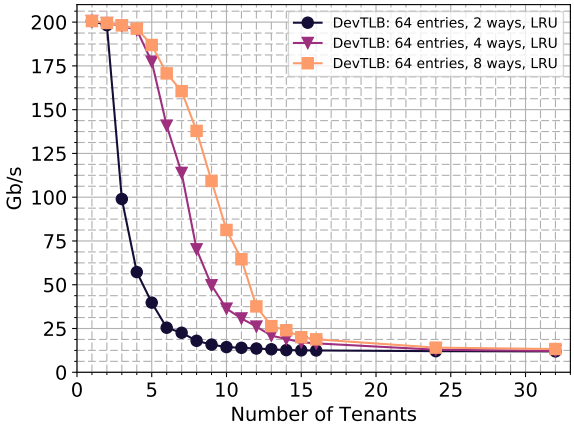


Fig. 9: Modeled I/O bandwidth depending on IOTLB configuration and number of concurrent connections.

when utilized by a large number of tenants. For an 8-way set-associate DevTLB, more than four concurrent connections start evicting entries of other tenants, which eventually leads to thrashing and significantly increases translation time for every request. Finally, the system becomes limited by the performance of the gIOVA translation subsystem, which involves traversing the PCIe bus, doing a two-dimensional page-table walk, and accessing DRAM.

V. Evaluation

In this section we evaluate the HyperTRIO architecture using HyperSIO to see how it addresses the challenges appearing in hyper-tenant environments and how insights about intra- and inter-tenant interaction can be used to remove guest I/O Virtual Address (gIOVA) to host Physical Address (hPA) translation bottleneck.

First, we describe real-world workloads used to model a hyper-tenant environment. After that, we compare HyperTRIO's performance with a base design and show that our architecture efficiently utilizes I/O device bandwidth independent of the number of tenants in a system and workload parameters. Then we study if changing parameters of a default translation subsystem like the DevTLB replacement strategy and its size can significantly improve I/O link utilization. Finally, we perform multiple sensitivity studies looking at how separate blocks of HyperTRIO - *Pending Translation Buffer*, *Partitioned Device-TLB*, and *Translation Prefetching* - affect utilization of a device.

A. Benchmarks

We used three I/O intensive benchmarks for evaluation listed in Table III. Every benchmark consists of a server and a client. The client sends a stream of requests or data to the server. Each client-server pair communicates through a separate network interface, which prevents the network interface from becoming a bottleneck. HyperSIO runs the same benchmark for all the tenants in its emulated system, records translation requests and related information, constructs a single trace modeling a hyper-tenant system, and uses HyperSIO's *Performance Model* to get

TABLE III: Maximum, minimum, and total number of translation requests recorded for every benchmark. *Tnt* - Tenant(s).

Benchmark	Max # Transl/Tnt	Min # Transl/Tnt	Total Transl for 1024 Tnt
iperf3	108,510	68,079	69,712,894
mediastream	73,657	5,520	5,652,477
websearch	108,513	43,362	44,402,679

the link utilization for every configuration (see Section IV). The benchmarks include:

- *iperf3* [3] - throughput oriented benchmark stressing network stack. It generates a steady stream of packets with a maximum size specified as a parameter. We used a maximum size of 1500B and restricted *iperf3* to use only IPv4 packets. Each client was run for 60 seconds to have enough time to record the interval when all tenants were active.
- *mediastream* - a benchmark from Cloudsuite 3 [37], serving videos of different length and qualities to a client. We used default options, except for the number of connections per host which was set to eight to stress the I/O link.
- *websearch* - another benchmark from Cloudsuite 3, where a client sends requests to multiple server index nodes. This benchmark was run for 360s in a steady state instead of the default 60s to collect longer translation traces. It also required 12GB of memory per index server which reduced the maximum number of simultaneously running tenants to twelve due to the main memory constraints of the host machine.

As described in Section IV, HyperSIO stops generating a trace when any tenant runs out of requests in order to avoid the "edge effect" when only a subset of all tenants is active. Therefore, the number of requests per tenant depends on which logs were read by HyperSIO to generate a trace. Table III lists minimum and maximum number of translation requests per tenant along with the total number for the 1024-tenant setup.

We also constructed multiple traces for each benchmark using different inter-tenant interleaving - `round-robin (RR)` and `random (RAND)`. The used interleavings are RR1, RR4, and RAND1, where the number at the end indicates consecutive number of packets sent to a tenant. For example, for a bursty traffic pattern, more packets can arrive at a given time interval compared to a non-bursty traffic. In that case, RR4 would better capture this parameter compared to RR1.

B. HyperTRIO Scalability

One of the main goals of the HyperTRIO architecture described in Section III is to enable full I/O link utilization for devices in hyper-tenant environments by removing the guest I/O Virtual Address Translation (gIOVA) to host Physical Address (hPA) bottleneck. Figure 10 shows maximum achievable link bandwidth of HyperTRIO compared to a Base architecture

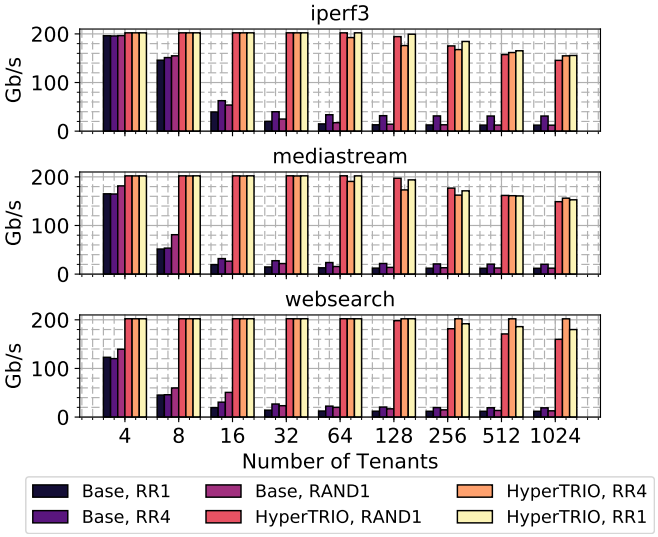


Fig. 10: Scalability of I/O bandwidth for HyperTRIO and Base designs.

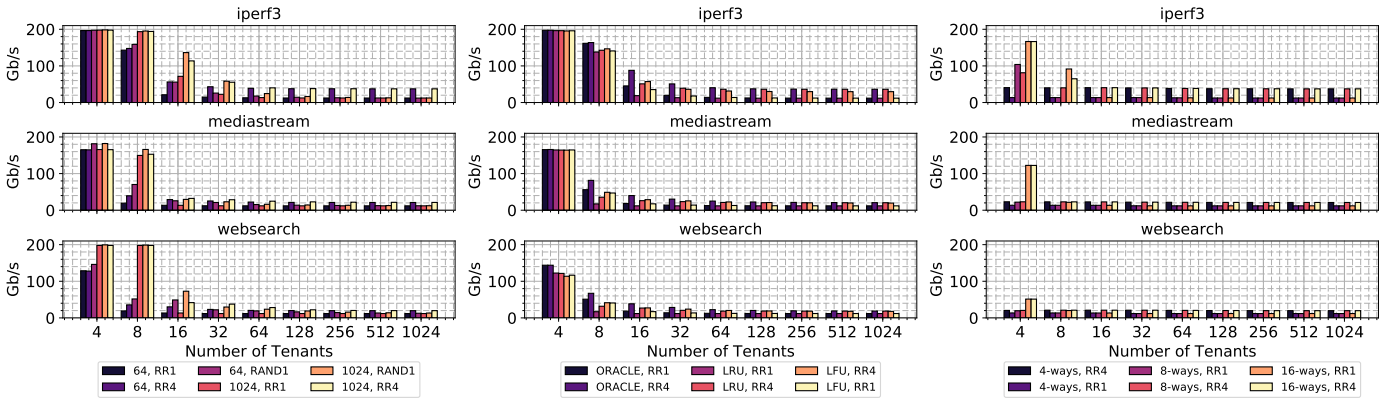
TABLE IV: Architectural parameters of HyperTRIO and Base configurations used for evaluation.

Parameter	Base	HyperTRIO
PTB	1 entry	32 entries
DevTLB	64-entries	
	8-ways, LFU	
	1 partition	8 partitions
L2TLB	512-entries	
	16-ways, LFU	
	1 partition	32 partitions
L3TLB	1024-entries	
	16-ways, LFU	
	1 partition	64 partitions
Prefetching Scheme	No	8-entry buffer, 48-access stride 2 pages history/tenant

using parameters listed in Table IV. We vary number of tenants from 4 to 1024 and evaluate different inter-tenant interleavings.

For the Base configuration, the maximum achievable I/O bandwidth does not scale with increasing number of tenants independent of their interleaving. It scales slightly better for *iperf3* than for two other benchmarks due to regular accesses of the former one, but for any number of tenants greater than 32, link utilization is between 12 and 30Gb/s, which is at most 15% of the nominal 200Gb/s I/O bandwidth. For small number of tenants, interleaving causes eviction of different translations from DevTLB. With larger burst size of 4 packets, more evictions are caused by RR4 than by RR1 (*mediastream, websearch*), reducing link utilization. In contrast, for large number of tenants, translations used to access the ring buffer pointer can be reused inside a burst, therefore RR4 has higher bandwidth compared to RR1.

In contrast, the HyperTRIO architecture enables the use of up to 100% of the total link bandwidth in an environment with 1024 tenants. Its partitioning of DevTLB and Translation Caches allows the use of these structures more uniformly



(a) Different 8-way DevTLB sizes (b) Effect of replacing policies (c) Fully associate DevTLB with oracle replacement

Fig. 11: I/O link utilization for the Base design with different DevTLB sizes and replacement policies.

in a hyper-tenant setup, and it also provides enough entries for caching of translations for low tenant-count environments. The *Prefetching Scheme* captures inter- and intra-tenant information supplying a valid translation from a *Prefetch Buffer* for 45% of requests for *websearch* benchmark in 1024-tenant setup. The Pending Translation buffer provides support for hiding misses to the DevTLB and *Prefetch Buffer* by keeping track of in-flight translations and hiding the latency caused by performing a two-dimensional page table walk. Tenant interleaving has little effect on HyperTRIO, and in the case of the least predictable RAND1 order it achieves up to 80% link utilization with 1024 tenants.

C. Base Configuration Study

In this sensitivity study we want to answer the question of whether changing some parameters of the Base configuration can significantly affect utilization of an I/O link.

Scaling DevTLB Size

Figure 11a shows results using the Base configuration with two different DevTLB sizes. A 1024 entry DevTLB enables reaching higher bandwidth for up to 64 tenants. However, it depends on the tenants' order. For example, a 64-entry size DevTLB for 16 tenants allows the utilization of the I/O link 3 times more efficiently for RR4 interleaving than a 1024-entry DevTLB with RR1 interleaving for *websearch* benchmark.

When the number of tenants exceeds 128, configurations with both sizes provide the same link utilization for RR1 and RAND1. RR4 gives higher bandwidth due to reuse of translations inside of a burst as described above. Overall, in a hyper-tenant setup, when many tenants use the same IOVAs, a simple increase of DevTLB size does not improve utilization of available bandwidth due to conflicts in frequently used sets.

Studying DevTLB Replacement Policies

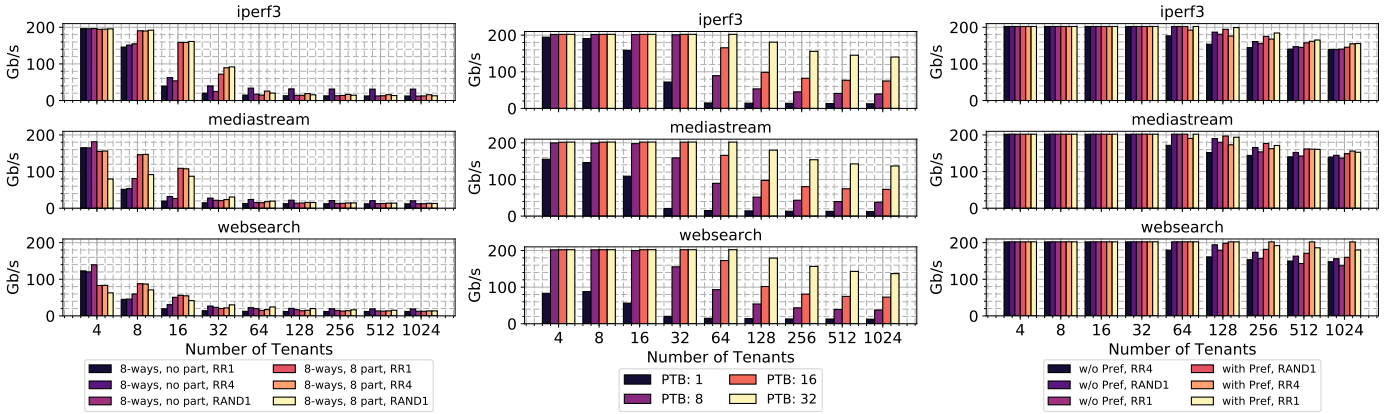
Analysis of a single-tenant trace (see Section IV-D, Figure 8a) shows that translation requests to some pages are seen more frequently than to other pages, and motivates us to implement a Least Frequently Used (LFU) replacement scheme. We use a 4-bit counter to track the number of accesses per cache entry, and all the counters in a row are divided by two when any of them saturates [24]. Having a full translation trace allows us to build an *oracle* scheme [9], evicting in the case of a conflict with the entry which will be used furthest in the future from the current access.

Figure 11b shows results for the described policies. For a small number of tenants, all translations fit into DevTLB without conflicts, therefore allowing the system to fully utilize available I/O bandwidth. With increasing number of tenants, total bandwidth starts to decrease, and for more than 64 of them the translation cache becomes completely thrashed by requests from different tenants, making the translation subsystem a bottleneck. In-between we see that LFU outperforms the LRU scheme, potentially improving achievable bandwidth by up to two times for *iperf3* benchmark in a 16-tenant setup. This is attributed to the fact that evicting a translation corresponding to the most frequently used page will more likely cause a miss some time in the feature than evicting one of the translations for a data buffer. Compared to *oracle*, LFU performs slightly worse, but even a perfect strategy does not allow the translation scheme to scale in a hyper-tenant setup due to long reuse distance of the same page belonging to a single tenant.

Scalability with Fully Associative DevTLB

Next, we study the effect of the total number of available entries in the DevTLB on I/O utilization. Once again, we use insights from Section IV-D, but this time focusing on the distribution of IOVA accesses in time (see Figure 8b). Since all pages are accessed periodically, the number of translations which should be kept in a DevTLB can be small: including only translation to a page with ring buffers, data buffers, and interrupt mailbox.

We define *active translation set size* as the minimum number of entries in a fully-associate DevTLB required to achieve the full utilization of an I/O link. Among three benchmarks, *iperf3* has the most regular access streaming pattern, and its active translation set is 8. *mediastream* and *websearch* benchmarks



(a) Effect of DevTLB and L[1-4]TLBs partitioning for HyperTRIO configuration in Table IV.

(b) Effect of Pending Translation Buffer (PTB) size on achievable bandwidth.

(c) Improvements with *Translation Prefetching Scheme* compared to a design with only PTB and Partitioned DevTLB.

Fig. 12: Contribution of *Pending Translation Buffer*, *Partitioned DevTLB*, and *Translation Prefetching Scheme* to I/O link bandwidth in HyperTRIO design.

have a much higher number of active pages for a single tenant - hypothetical 32- and 36-way fully associative caches are needed respectively. Compared to *iperf3*, they have less regular access patterns, which requires more pages.

Figure 11c shows how I/O link utilization scales for full-associative DevTLBs with increasing number of tenants. In this study we use the *oracle* replacement scheme to remove the effects of evicting a wrong entry and focus only on tenant interaction in an ideal scenario. As can be seen from Figure 11c, in all benchmarks, using more than eight tenants produces low bandwidth utilization. Basically, when the number of tenants reaches the number of available translation entries, every new request starts to miss in the DevTLB, and the translation request is forwarded to a chipset to complete the translation. Though its total processing time depends on a miss/hit to L[1-4]TLBs, for high bandwidth I/O devices even just PCIe latency will severely affect the total throughput.

D. HyperTRIO Evaluation

Scalability of the Partitioning Scheme

From the two previous studies, we observe that neither changing DevTLB replacement policies nor increasing its associativity allows us to scalably use the I/O link. In this case-study we evaluate how partitioning of the DevTLB and L[1-4]TLBs described in Section III affects total performance.

We set partition size to one 8-entry row per tenant, even though *mediastream* and *websearch* benchmarks have larger active translation sets per tenant. This decision is made in favor of having more isolated partitions for large numbers of tenants. Exploring the optimal number of partitions and the number of devices per partition is left outside of the scope of this work, giving an assumption that all the tenants fairly share an available I/O device.

Figure 12a presents results for the proposed scheme. Link utilization stays high until multiple devices start using the same partition. Partitioning an 8-way set-associative DevTLB provides significant improvements over the base case, but at the same time it limits the maximum possible cached translations available for every tenant. The benefit of the scheme comes from isolation and independent management of tenants, allowing translations to evict entries which belong only to the same partition. Overall, partitioning improves utilization better than simply increasing associativity or changing replacement policy of DevTLB, but it still does not solve the scalability challenge in hyper-tenant environments.

Pending Translation Buffer

The *Pending Translation Buffer (PTB)* allows HyperTRIO to hide translation latency in the case of a miss in the DevTLB. This case study assumes the DevTLB has 8 partitions as in the previous experiment, and it augments that configuration with a PTB, studying its effect on I/O link utilization.

Figure 12b shows simulation results for different PTB sizes. With eight entries it enables reaching full I/O bandwidth for a partitioned design with up to 16 tenants (compare with Figure 12a). Cumulative effect of both units gives an opportunity to hide translation latency when missing into DevTLB when the active translation size is larger than the number of DevTLB entries per partition. Translation requests which come after a missed one can be started before the previous one is finished, effectively enabling hit under miss. Further size increase of the PTB to 32 entries allows us to achieve an aggregated 136Gb/s for 1024 tenants for all benchmarks. Keeping track of more in-flight translations can improve utilization further, however it becomes expensive from a hardware point of view and does not scale for larger I/O bandwidth.

Translation Prefetching Scheme

Finally, we show the contribution of the Translation Prefetching scheme into total link utilization. As a base line we use the configuration from a previous case study with

partitioned DevTLB and L[1-4]TLBs, and with PTB of size 32, augmenting it with prefetching described in Section III.

To study the accuracy and timeliness of a prefetcher [45], we analyzed different Prefetch Buffer (PB) sizes, history length, and number of prefetch requests per Source ID. Since PB is a fully associative buffer, it should be kept small, and we found that eight entries are a good trade-off between precision and hardware resources used for the buffer. History length is a parameter which can be configured by a hypervisor after the addition/removal of a tenant. We found that for our simulated system a history depth of 48 requests is optimal across different number of tenants. The last parameter we studied was the number of prefetched translations for each tenant. In order to keep the translation for several tenants in a small PB, we prefetch the two most recently used translations per tenant.

The effect of a Prefetching Scheme on I/O link utilization is shown in Figure 12c. For hyper-tenant configurations it improves link utilization by up to 30% for the *websearch* benchmark. It also scales better than simply increasing the PTB size, since the prefetch buffer and history length can stay the same for larger number tenants. All page translation history of accesses is kept in the main memory which is an ample resource in today's servers.

VI. Related Work

Much work has evaluated the scalability of IOMMU design, management, protection, and their impact on I/O link utilization. Depending on message size, CPU utilization can become significant due to frequent page map and unmap operations [10]. However, it was shown only for a single client-server pair. To reduce the number of cycles required to process every packet, a hierarchical page-table was replaced by a flat one for every tenant's ring buffer [28]. However, it required modifications of device drivers and guest OSes, which is not always possible in hyper-tenant environments. Different protection schemes and their scalability as part of IOMMU design were also studied [29], [47]. Some schemes allow the use of up to 90% [29] of the total throughput of a device by reducing the cost of page unmapping. However, this work has focused on the software side of IOMMU management, without studying the interaction between multiple independent tenants in the system. Previous work also evaluated the performance of IOMMU page allocation for a 10Gb/s NIC, using 270 *netperf* instances to stress the system [39]. Compared to all the aforementioned work, HyperTRIO looks at the behavior of gIOVA to hPA translation subsystem from a hardware point of view in hyper-tenant environments with up to 1024 concurrently running tenants.

Prior work characterized IOVA streams showing that eviction policy for IOTLBs do not change miss/hit ratio and suggest using prefetching for adjacent pages [6]. They also proposed applying page coloring to offset address spaces of different I/O devices for their isolation. With HyperTRIO, we did partitioning in hardware. In our work, tenants use 2 MB pages, and we found that devices used in total 32 pages for data buffers instead of 10 as described before [6]. When studying replacement policies, in addition to LRU and *oracle* schemes, we evaluated LFU which was motivated by tenant's trace analysis. We also designed a Hyper-Tenant Performance Simulator for I/O which incorporates real latencies instead of just calculating the number of hits to IOTLB. Finally, our prefetching scheme uses a completely different approach by keeping track of translation history in the main memory and prefetching it in hardware. In contrast to previous work [6], HyperTRIO does not use software hints.

For GPUs it was shown that a highly threaded Page Table Walker significantly improves the performance [41], [42], [44], [46]. A SIMD instruction generates up to 64 translation requests [44], but multiple GPU lanes usually cooperate towards completion of a single task, providing opportunities for coalescing of multiple translation requests into one [44]. In contrast, every tenant sharing an I/O device is independent of others, and we cannot rely on any cooperation between them.

Partitioning and leveraging multiple data sources was previously done for the memory hierarchy [15], [17], [18]. Data cache bypassing and pinning was studied in the context of GPUs [25]. To the best of our knowledge, we make the first attempt to use partitioning for translation structures in a multi-level translation hierarchy for hyper-tenant environments.

Multiple efforts were done to optimize one- and two-dimensional page-table walks [12], [13], [16], [43], [44]. However, those studies focus on data-intensive applications and study MMU designs, while this work looks at IOMMU address translation. Shared I/O devices have multiple unique features including usage of ring-buffers, knowledge of tenant's bandwidth at the assignment time, and more strict requirements for total translation time due to limited buffering space.

VII. Conclusion

In this work, we presented *HyperTRIO*, an architecture for scalable guest I/O Virtual Address (gIOVA) to host Physical Address (hPA) translation in hyper-tenant environments. We showed that increasing the number of tenants leads to poor I/O link utilization and prevents the system from using all the available bandwidth of high-throughput devices. We proposed to use a *Pending Translation Buffer* to support multiple in-flight translations on a device, described a *Partitioned Device-TLB* for tenant isolation and uniform utilization of hardware resources, and proposed a scalable *Prefetching Scheme* which uses inter- and intra-tenant information to predict and translate gIOVAs to hPAs. In order to analyze and study hyper-tenant environments, we built the Hyper-Tenant Simulator of I/O - *HyperSIO*. Overall we find that the *HyperTRIO* architecture enables the system to utilize more than 90% of the 200Gb/s link in environments with up to 1024 independent tenants compared to only 6% for a design without the support of multiple in-flight translations, *Partitioned Device-TLB*, and *Prefetching Scheme*.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable and useful feedback. This material is based on research sponsored by the NSF under Grant No. CCF-1822949, Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7846, FA8650-18-2-7852, and FA8650-18-2-7862. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), the NSF, or the U.S. Government.

REFERENCES

[1] "2019 Ethernet roadmap," https://ethernetalliance.org/the-2019-ethernet-roadmap, accessed March, 2019.

[2] "How to achieve 20Gb and 30Gb bandwidth through network bonding," http://45drives.blogspot.com/2015/07/how-to-achieve-20gb-and-30gb-bandwidth.html, accessed November, 2019.

[3] "iperf - the ultimate speed test tool for tcp, udp and sctp," https://iperf.fr, accessed March, 2019.

[4] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[5] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster, "vIOMMU: efficient IOMMU emulation," *USENIX Annual Technical Conference (ATC)*, pp. 73–86, 2011.

[6] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "IOMMU: Strategies for mitigating the IOTLB bottleneck," in *Sixth Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2010.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[9] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

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

[11] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[12] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2. ACM, 2008, pp. 26–35.

[13] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2009, pp. 29–40.

[14] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, 2012.

[15] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 308–319.

[16] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 435–448, 2017.

[17] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 104–117.

[18] J. Gaur, M. Chaudhuri, P. Ramachandran, and S. Subramoney, "Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 13–24.

[19] Intel, "Intel ethernet controller x540 datasheet," https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-x540-datasheet.pdf, accessed March, 2019.

[20] ——, "Intel scalable I/O virtualization," https://software.intel.com/en-us/blogs/2018/06/25/introducing-intel-scalable-io-virtualization, accessed March, 2019.

[21] ——, "Intel virtualization technology for directed I/O," https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf, accessed March, 2019.

[22] ——, "Intel Xeon Scalable platform product brief," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-scalable-platform-brief.pdf, accessed March, 2019.

[23] ——, "4-level paging and 5-level EPT," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, 2017, accessed March, 2019.

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

[25] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 307–319, 2017.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 190–200.

[27] D. M. Davis, "Demystifying CPU ready (%RDY) as a performance metric," Tech. Rep., 2012.

[28] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 355–368, 2015.

[29] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir, "DAMN: Overhead-free IOMMU protection for networking," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018, pp. 301–315.

[30] Mellanox, "ConnectX-4 Lx EN adapter card for open compute project (OCP)," http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-4_Lx-EN_OCP.pdf, accessed March, 2019.

[31] ——, "ConnectX-5 EN card," http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf, accessed March, 2019.

[32] ——, "ConnectX-6 en card," http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, accessed March, 2019.

[33] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM*, 2018.

[34] NVIDIA, "NVIDIA Tesla M10," https://images.nvidia.com/content/tesla/pdf/188359-Tesla-M10-DS-NV-Aug19-A4-fnl-Web.pdf, accessed March, 2019.

[35] Open Compute Project, "OCP NIC 3.0 design specification," https://www.opencompute.org/documents/ocp-nic-30-specification, accessed March, 2019.

[36] ——, "OCP NIC 3.0 ethernet adapters portfolio product brief," https://docs.broadcom.com/docs/12395116, accessed March, 2019.

[37] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *Proceesings of 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 122–132.

[38] PCI SIG, "Single root I/O virtualization and sharing specification," https://pcisig.com/specifications/iov, 2010, accessed March, 2019.

[39] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU Scalably," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, ser. USENIX, 2015.

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

[41] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2014, pp. 743–758.

[42] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 568–578.

[43] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 469–480, 2017.

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

[45] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 63–74.

[46] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 161–171.

[47] P. Willmann, S. Rixner, and A. L. Cox, "Protection strategies for direct access to virtualized I/O devices," in *USENIX Annual Technical Conference*, 2008, pp. 15–28.