rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers

Moshe Malka Nadav Amit Muli Ben-Yehuda Dan Tsafrir

Technion – Israel Institute of Technology

{moshemal,namit,muli,dan}@cs.technion.ac.il

Abstract

The IOMMU allows the OS to encapsulate I/O devices in their own virtual memory spaces, thus restricting their DMAs to specific memory pages. The OS uses the IOMMU to protect itself against buggy drivers and malicious/errant devices. But the added protection comes at a cost, degrading the throughput of I/O-intensive workloads by up to an order of magnitude. This cost has motivated system designers to trade off some safety for performance, e.g., by leaving stale information in the IOTLB for a while so as to amortize costly invalidations.

We observe that high-bandwidth devices—like network and PCIe SSD controllers—interact with the OS via circular ring buffers that induce a sequential, predictable workload. We design a ring IOMMU (rIOMMU) that leverages this characteristic by replacing the virtual memory page table hierarchy with a circular, flat table. A flat table is adequately supported by exactly one IOTLB entry, making every new translation an implicit invalidation of the former and thus requiring explicit invalidations only at the end of I/O bursts. Using standard networking benchmarks, we show that rIOMMU provides up to 7.56x higher throughput relative to the baseline IOMMU, and that it is within 0.77–1.00x the throughput of a system without IOMMU protection.

Categories and Subject Descriptors B.3.2 [*memory structures*]: design styles—virtual memory; B.4.2 [*I/O and data communications*]: I/O devices—channels and controllers; D.4.2 [*operating systems*]: storage management—virtual memory, allocation/deallocation strategies

General Terms design, experimentation, performance

Keywords I/O memory management unit

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694355

1. Introduction

I/O device drivers initiate direct memory accesses (DMAs) to asynchronously move data from their devices into memory and vice versa. In the past, DMAs used physical memory addresses. But such unmediated access made systems vulnerable to (1) rogue devices that might perform errant or malicious DMAs [9, 12, 28, 32, 52], and to (2) buggy drivers that account for most operating system (OS) failures and might wrongfully trigger DMAs to arbitrary memory locations [8, 13, 23, 36, 47, 50]. Subsequently, all major chip vendors introduced I/O memory management units (IOMMUs) [3, 7, 25, 28], which allow DMAs to execute with I/O virtual addresses (IOVAs). The IOMMU translates the IOVAs into physical addresses according to I/O page tables that are setup by the OS. The OS thus protects itself by adding a translation just before the corresponding DMA, and by removing the translation right after [11, 16, 51]. We explain in detail how the IOMMU is implemented and used in §2.

DMA protection comes at a cost that can be substantial in terms of performance [4, 10, 51], for newer, high-throughput I/O devices like 10/40 Gbps network controllers (NICs), which can deliver millions of packets per second. Our measurements indicate that using DMA protection with such devices can reduce the throughput by up to 10x. This penalty has motivated OS developers to trade off some protection for performance. For example, when employing the "deferred" IOMMU mode, the Linux kernel defers IOTLB invalidations for a short while instead of performing them immediately, because invalidations are slow. Later, the kernel processes the accumulated invalidations en masse by flushing the entire IOTLB, thus amortizing the overhead at the risk of allowing devices to erroneously utilize stale IOTLB entries. While this tradeoff can double the performance relative to the stricter IOMMU mode, the throughput is still 5x lower than when the IOMMU is disabled. We analyze and model the overheads associated with using the IOMMU in §3.

We argue that the degraded performance is largely due to the IOMMU needlessly replicating the design of the regular MMU, which is based on hierarchical page tables. Our claim pertains high-bandwidth I/O devices, such as NICs and PCIe

SSD drives, which utilize circular "ring" buffers to interact with the OS. A ring is an array of descriptors that the OS driver sets when initiating DMAs. Descriptors encapsulate the DMA details, including the associated IOVAs. Importantly, ring semantics dictate that (1) the driver work through the ring in order, one descriptor after the other, and that (2) the I/O device process these descriptors in the same order. Thus, IOVAs are short-lived and the sequence in which they are used is linearly predictable: each IOVA is allocated, placed in the ring, used in turn, and deallocated.

We propose a ring IOMMU (rIOMMU) that supports this pervasive sequential model using flat (1D) page tables that directly correspond to the nature of rings. rIOMMU has three advantages over the baseline IOMMU that significantly reduce the overhead of DMA protection. First, building/destroying an IOVA translation in a flat table is quicker than in a hierarchical structure. Second, (de)allocation of IOVAs—the actual integers serving as virtual addresses—is faster, as IOVAs are indices of flat tables in our design. Finally, the frequency of IOTLB invalidations is substantially reduced, because the rIOMMU designates only one IOTLB entry per ring. One is enough because IOVAs are used sequentially, one after the other. Consequently, every translation inserted to the IOTLB removes the previous translation, eliminating the need to explicitly invalidate the latter. And since the OS handles high-throughput I/O in bursts, explicit invalidations become rare. We describe rIOMMU in §4.

We evaluate the performance of rIOMMU using networking benchmarks and find that it improves throughput by 1.00—7.56x, shortens latency by 0.80–0.99x, and reduces CPU consumption by 0.36–1.00x relative to the existing IOMMU. Our fastest rIOMMU variant is within 0.77–1.00x the throughput, 1.00–1.04x the latency, and 1.00–1.22x the CPU consumption of a system that disables the IOMMU entirely. We describe our experimental evaluation in §5.

2. Background

2.1 Operating System DMA Protection

The role the IOMMU plays for I/O devices is similar to the role the regular MMU plays for processes, as illustrated in Figure 1. Processes typically access the memory using virtual addresses, which are translated to physical addresses by the MMU. Analogously, I/O devices commonly access the memory via DMAs associated with IOVAs. The IOVAs are translated to physical addresses by the IOMMU.

The IOMMU provides *inter-* and *intra-OS protection* [4, 49, 51, 53]. Inter-OS protection is applicable in virtual setups. It allows for "direct I/O", where the host assigns a device directly to a guest virtual machine (VM) for its exclusive use, largely removing itself from the guest's I/O path and thus improving its performance [22, 36]. In this mode of operation, the VM directly programs device DMAs

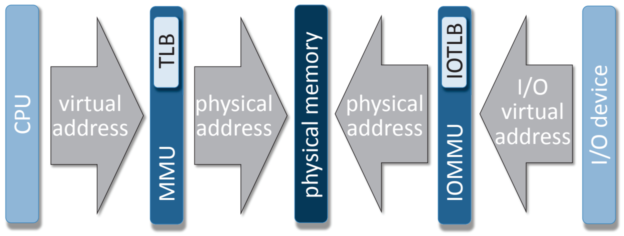

Figure 1. The IOMMU is for devices what the MMU is for processes.

using its notion of (guest) "physical" addresses. The host uses the IOMMU to redirect these accesses to where the VM memory truly resides, thus protecting its own memory and the memory of the other VMs. With inter-OS protection, IOVAs are mapped to physical memory locations infrequently, typically only upon such events as VM creation and migration. Such mappings are therefore denoted *static* or *persistent* [51]; they are not the focus of this paper.

Intra-OS protection allows the OS to defend against the DMAs of errant/malicious devices [9, 12, 17, 28, 32, 52] and of buggy drivers, which account for most OS failures [8, 13, 23, 36, 47, 50]. Drivers and their I/O devices can perform DMAs to arbitrary memory addresses, and IOMMUs allow OSes to protect themselves (and their processes) against such accesses, by restricting them to specific physical locations. In this mode of work, *map* operations (of IOVAs to physical addresses) and *unmap* operations (invalidations of previous maps) are frequent and occur within the I/O critical path, such that each DMA is preceded and followed by the mapping and unmapping of the corresponding IOVA [32, 40]. Due to their short lifespan, these mappings are denoted *dynamic* [11], *streaming* [16] or *single-use* [51]. This strategy of IOMMU-based intra-OS protection is the focus of this paper. It is recommended by hardware vendors [24, 28, 32] and employed by operating systems [6, 11, 16, 26, 38, 51].[1] It is applicable in non-virtual setups where the OS has direct control over the IOMMU. It is likewise applicable in virtual setups where IOMMU functionality is exposed to VMs via paravirtualization [10, 36, 45, 51], full emulation [4], and, more recently, hardware support for nested IOMMU translation [3, 28].

2.2 IOMMU Design and Implementation

Given a *target memory buffer* of a DMA, the OS associates the physical address (PA) of the buffer with an IOVA. The OS maps the IOVA to the PA by inserting the IOVA⇒PA

[1] For example, the DMA API of Linux notes that "DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer" [40]. In particular, "once a buffer has been mapped, it belongs to the device, not the processor. Until the buffer has been unmapped, the [OS] driver should not touch its contents in any way. Only after [the unmap of the buffer] has been called is it safe for the driver to access the contents of the buffer" [16].

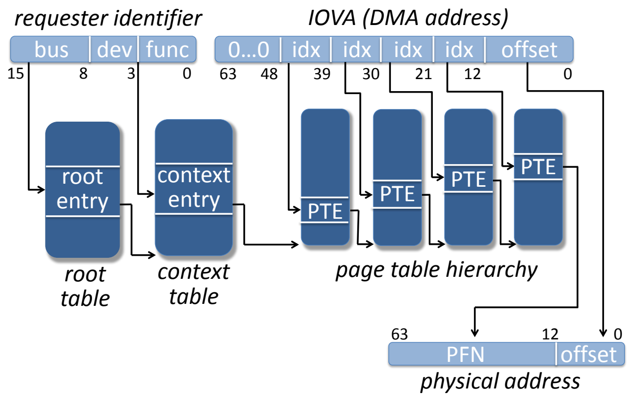

Figure 2. IOVA translation with the Intel IOMMU.

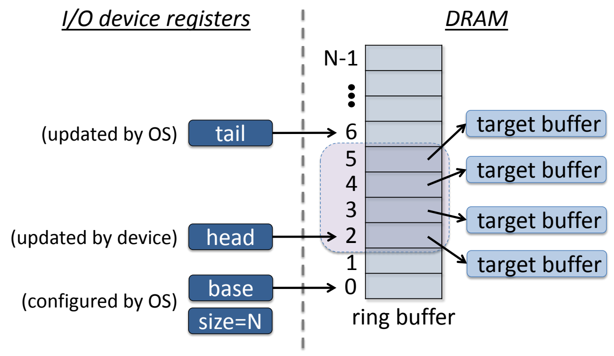

Figure 3. The driver drives its device via a ring. With an IOMMU, register/target pointers are IOVAs.

translation to the IOMMU data structures. Figure 2 depicts these structures as implemented by Intel x86-64 [28]. The PCI protocol dictates that each DMA operation is associated with a 16-bit *request identifier* comprised of a *bus-device-function* triplet that uniquely identifies the corresponding I/O device. The IOMMU uses the 8-bit bus number to index the *root table* in order to retrieve the physical address of the *context table*. It then indexes the context table using the 8-bit concatenation of the device and function numbers. The result is the physical location of the root of the page table hierarchy that houses all of the IOVA⇒PA translations of that I/O device.

The purpose of the IOMMU page table hierarchy is similar to that of the MMU hierarchy: recording the mapping from virtual to physical addresses by utilizing a 4-level radix tree. Each 48-bit (I/O) virtual address is divided into two: the 36 high-order bits, which constitute the *virtual page number*, and the 12 low-order bits, which are the *offset* within the page. The translation procedure applies to the virtual page number only, converting it into a *physical frame number* (PFN) that corresponds to the physical memory location being addressed. The offset is the same for both physical and virtual pages.

Let T_j denote a page table in the j-th radix tree level for $j = 1,2,3,4$, such that T_1 is the root of the tree. Each T_j is a 4KB page containing up to $2^9 = 512$ pointers to physical locations of next-level T_{j+1} tables. Last-level—T_4—tables contain PFNs of target buffer locations. Correspondingly, the 36-bit virtual page number is split into a sequence of four 9-bit indices i_1, i_2, i_3 and i_4, such that i_j is used to index T_j in order to find the physical address of the next T_{j+1} along the radix tree path. Logically, in C pointer notation, $T_1[i_1][i_2][i_3][i_4]$ is the PFN of the target memory location.

Similarly to the MMU translation lookaside buffer (TLB), the IOMMU caches translations using an IOTLB, which it fills on-the-fly as follows. Upon an IOTLB miss, the IOMMU hardware hierarchically *walks* the page table as described above, and it inserts the IOVA⇒PA translation to the IOTLB. IOTLB entries are invalidated explicitly by the OS as part of the corresponding unmap operation.

An IOMMU table walk fails if a matching translation was not previously established by the OS, a situation which is logically similar to encountering a null pointer value during the walk. A walk additionally fails if the DMA being processed conflicts with the read/write permission bits found within the page table entries along the traversed radix tree path. We note in passing that, at present, in contrast to MMU memory accesses, DMAs are typically not restartable. Namely, existing systems usually do not support "I/O page faults", and hence the OS cannot populate the IOMMU page table hierarchy on demand. Instead, IOVA translations of valid DMAs are expected to be successful, and the corresponding pages must be pinned to memory. (Albeit I/O page fault standardization does exist [42].)

2.3 I/O Devices Employing Ring Buffers

Many I/O devices—notably NICs and disk drives—deliver their I/O through one or more producer/consumer *ring* buffers. A ring is an array shared between the OS device driver and the associated device, as illustrated in Figure 3. The ring is circular in that the device and driver wrap around to the beginning of the array when they reach its end. The entries in the ring are called DMA *descriptors*. Their exact format and content vary between devices, but they specify at least the address and size of the corresponding target buffers. Additionally, the descriptors commonly contain status bits that help the driver and the device to synchronize.

Devices must also know the *direction* of each requested DMA, namely, whether the data should be transmitted from memory (into the device) or received (from the device) into memory. The direction can be specified in the descriptor, as is typical for disk controllers. Or the device can employ different rings for receive and transmit activity, in which case the direction is implied by the ring. The receive and transmit rings are denoted *Rx* and *Tx*, respectively. NICs employ at least one Rx and one Tx per port. They may employ multiple Rx/Tx rings per port to promote scalability, as different rings can be handled concurrently by different cores.

Upon initialization, the OS device driver allocates the rings and configures the I/O device with the ring sizes and base locations. For each ring, the device and driver utilize a *head* and a *tail* pointers to delimit the ring content that can be used by the device: $[head, tail)$. The device iteratively consumes (removes) descriptors from the head, and it increments the head to point to the next descriptor to be used next. Similarly, the driver adds descriptors to the tail, incrementing the tail to point to the entry it will use subsequently.

A device asynchronously informs its OS driver that data was transmitted or received by triggering an interrupt. The device coalesces interrupts when their rate is high. Upon receiving an interrupt, the driver of a high-throughput device handles the entire I/O burst. Namely, it sequentially iterates through and processes all the descriptors whose corresponding DMAs have completed,

3. Cost of Safety

This section enumerates the overhead components involved in using the IOMMU in the Linux/Intel kernel (§3.1). It experimentally quantifies the overhead of each component (§3.2). And it provides and validates a simple performance model that allows us to understand how the overhead affects performance and to assess the benefits of reducing it (§3.3).

3.1 Overhead Components

Suppose that a device driver that employs a ring wants to transmit or receive data from/to a target buffer. Figure 4 lists the actions it carries out. First, it allocates the target buffer, whose physical address is denoted p (1). (For simplicity, let us assume that p is page aligned.) It pins p to memory and then asks the IOMMU driver to map the buffer to some IOVA, such that the I/O device would be able to access p (2). The IOMMU driver invokes its IOVA allocator, which returns a new IOVA v—an integer that is not associated with any other page currently accessible to the I/O device (3). The IOMMU driver then inserts the $v \Rightarrow p$ translation to the page table hierarchy of the I/O device (4), and it returns v to the device driver (5). Finally, when updating the corresponding ring descriptor, the device driver uses v as the address for the target buffer of the associated DMA operation (6).

Assume that the latter is a receive DMA. Figure 5 details the activity taking place when the I/O device gets the data. The device reads the DMA descriptor through its head register. The address held by the head is an IOVA, so it is intercepted by the IOMMU (1). The IOMMU consults its IOTLB to find a translation for the head IOVA. If the translation is missing, the IOMMU walks the page table hierarchy of the device to resolve the miss (2). Equipped with the head's physical address, the IOMMU translates the head descriptor for of the device (3). The head descriptor specifies that v (IOVA defined above) is the address of the target buffer (4), so the device writes the received data to v (5). The IOMMU intercepts v,

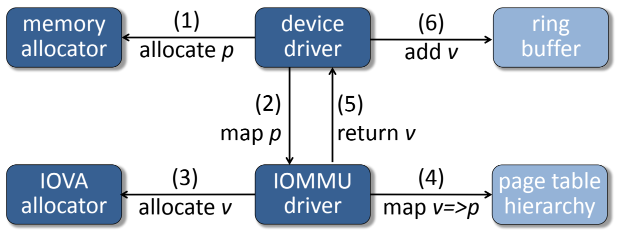

Figure 4. The I/O device driver maps an IOVA v to a physical target buffer p. It then assigns v to the DMA descriptor.

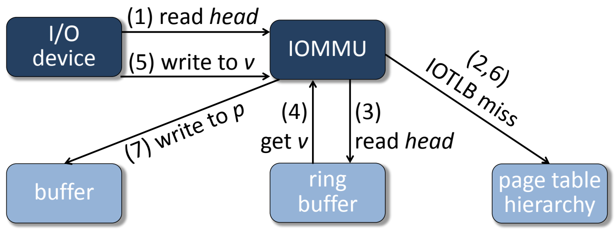

Figure 5. The I/O device writes the packet it receives to the target buffer through v, which the IOMMU translates to p.

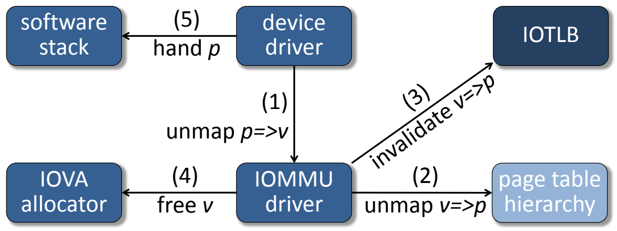

Figure 6. After the DMA completes, the I/O device driver unmaps v and passes p to a higher-level software layer.

walks the page table if the $v \Rightarrow p$ translation is missing (6), and redirects the received data to p (7).

Figure 6 shows the actions the device driver carries out after the DMA operation is completed. The device driver asks the IOMMU driver to unmap the IOVA v (1). In response, the IOMMU driver removes the $v \Rightarrow p$ mapping from the page table hierarchy (2), purges the mapping from the IOTLB (3), and deallocates v (4). (The order of these actions is important.) Once the I/O device can no longer access p, it is safe for the device driver to hand the buffer to higher levels in the software stack for further processing (5).

3.2 Protection Modes and Measured Overhead

We experimentally quantify the overhead components of the map and unmap functions—outlined in Figures 4 and 6—of the IOMMU driver. To this end, we execute the standard Netperf TCP stream benchmark, which attempts to maximize network throughput between two machines over a TCP connection. (The experimental setup is detailed in §5.)

function	*component*	*strict*	*strict+*	*defer*	*defer+*
map	iova alloc	3986	92	1674	108
	page table	588	590	533	577
	other	44	45	44	42
	sum	4618	727	2251	727
unmap	iova find	249	418	263	454
	iova free	159	62	189	57
	page table	438	427	471	504
	iotlb inv	2127	2135	9	9
	other	26	25	205	216
	sum	2999	3067	1137	1240

Table 1. Average cycles breakdown of the (un)map functions of the IOMMU driver for different protection modes.

Strict Protection We begin by profiling the Linux kernel in its safer IOMMU mode, denoted *strict*, which strictly follows the map/unmap procedures described in §3.1. Table 1 shows the average duration of the components of these procedures in cycles. The strict/map breakdown indicates that its most costly component is, surprisingly, IOVA allocation (Step 3 in Figure 4). Upon further investigation, we found that the reason for this high cost is a nontrivial pathology in the Linux IOVA allocator that regularly causes some allocations to be linear in the number of currently allocated IOVAs. We were able to come up with a more efficient IOVA allocator, which consistently allocates/frees in constant time [37]. We denote this optimized IOMMU mode—which is quicker than strict but equivalent to it in terms of safety—as *strict+*. Table 1 shows that strict+ indeed reduces the allocation time from nearly 4,000 cycles to less than 100.

The remaining dominant strict(+)/map overhead is the insertion of the IOVA to the IOMMU page table (Step 4 in Figure 4). The 500+ cycles of the insertion are due to explicit memory barriers and cacheline flushes that the driver performs when updating the hierarchy. Flushes are required, as the I/O page walk is incoherent with the CPU caches on our system. (This is common nowadays; Intel started shipping servers with coherent I/O page walks only recently.)

Focusing on the unmap components of strict/strict+, we see that finding the unmapped IOVA in the allocator's data structure is costlier in strict+ mode. The reason: like the baseline strict, strict+ utilizes a red-black tree to hold the IOVAs. But the strict+ tree is fuller, so the logarithmic search is longer. Conversely strict+/free (Step 4 in Figure 6) is done in constant time, rather than logarithmic, so it is quicker. The other unmap components are: removing the IOVA from the page tables (Step 2 in Figure 6) and the IOTLB (Step 3). The removal takes 400+ cycles, which is comparable to the duration of insertion. IOTLB invalidation is by far the slowest unmap component at around 2,000 cycles; this result is consistent with previous work [4, 53].

Deferred Protection In order to reduce the high cost of invalidating IOTLB entries, the Linux *deferred* protection mode relaxes strictness somewhat, trading off some safety for performance. Instead of invalidating entries right away, the IOMMU driver queues the invalidations until 250 freed IOVAs accumulate. It then processes all of them in bulk by invalidating the entire IOTLB. This approach affects the cost of (un)mapping in two ways, as shown in Table 1 in the defer and defer+ columns. (Defer+ is to defer what strict+ is to strict.) First, as intended, it eliminates the cost of invalidating individual IOTLB entries. And second, it reduces the cost of IOVA allocation in the baseline deferred mode as compared to strict (1,674 vs. 3,986), because deallocating IOVAs in bulk reduces somewhat the aforementioned linear pathology.

The drawback of deferred protection is that the I/O device might erroneously access target buffers through stale IOTLB entries after the buffers have already been handed back to higher software stack levels (Step 5 in Figure 6). Notably, at this point, the buffers could be (re)used for other purposes.

3.3 Performance Model

Let C denote the average number of CPU cycles required to process one packet. Figure 7 shows C for each of the aforementioned IOMMU modes in our experimental setup. The bottommost horizontal grid line shows C_{none}, which is C when the IOMMU is turned off. We can see, for example, that C_{strict} is nearly 10x higher than C_{none}.

Our experimental setup employs a NIC that uses two target buffers per packet: one for the header and one for the data. Each packet thus requires two map and two unmap invocations. So the processing of the packet includes: two IOVA (de)allocations; two page table insertions and deletions; and two invalidations of IOTLB entries. The corresponding aggregated cycles are respectively depicted as the three top stacked sub-bars in the figure. The bottom, "other" sub-bar embodies all the rest of the packet processing activity, notably TCP/IP and interrupt processing. As noted, the deferred modes eliminate the IOTLB invalidation overhead, and the "+" modes reduce the overhead of IOVA (de)allocation. But even C_{defer+} (the most performant mode, which introduces a vulnerability window) is still over 3.3x higher than C_{none}.

We find that the way the value of C affects the overall throughput of Netperf is simple and intuitive. Specifically, if S denotes the cycles-per-second clock speed of the core, then S/C is the number of packets the core can handle per second. And since every Ethernet packet carries 1,500 bytes, the throughput of the system in Gbps should be $Gbps(C) = 1500 \text{ byte} \times 8 \text{ bit} \times \frac{S}{C}$, assuming S is given in GHz. Figure 8 shows that this simple model (thick line) is accurate. It coincides with the throughput obtained when systematically lengthening C_{none} using a carefully controlled busy-wait loop (thin line). It also coincides with the throughput measured under the different IOMMU modes (cross points).

Consequences As our model is accurate, we conclude that the translation activity carried out by the IOMMU (as depicted in Figure 5) does not affect the performance of the system, even when servicing demanding benchmarks like

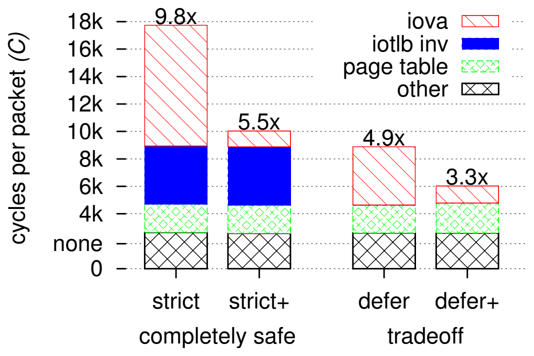

Figure 7. CPU cycles for processing one packet. The top bar labels are relative to C_{none}=1,816 (bottommost grid line).

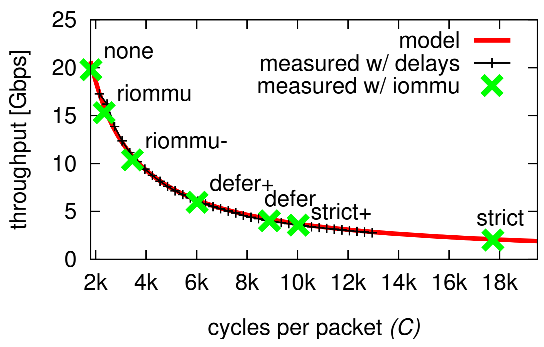

Figure 8. Throughput of Netperf as a function of the average number of cycles spent on processing one packet.

Netperf. Instead, the cost of IOMMU protection is entirely determined by the number of cycles the *core* spends creating and destroying IOVA mappings. Consequently, we can later simulate and accurately assess the expected performance of our proposed IOMMU by likewise spending cycles; there is no need to simulate the actual IOMMU hardware circuitry external to the core. A second conclusion rests on the understanding that throughput is proportional to $1/C$. If C is high (right-hand side of Figure 8), incrementally improving it would make little difference. The required change must be significant enough to bring C to the proximity of C_{none}.

4. Design

Our goal is to design rIOMMU—an efficient IOMMU for devices that employ rings. We aim to substantially reduce all IOVA-related overheads: (de)allocation, insertion/deletion to/from the page table hierarchy, and IOTLB invalidation (see Figure 7). We base our design on the observation that ring semantics (§2.3) dictate a well-defined memory mapping order, caused by the fact that the OS maps ring entries and shortly after unmaps them in the exact same order.

We contend that the x86 hierarchical structure of page tables is poorly suited for the ring model. For each DMA, the OS has to walk the hierarchical page table in order to map the associated IOVA. Then, the device faults on the IOVA and so the IOMMU must walk the table too. Shortly after, the OS has to walk the table yet again in order to unmap the IOVA. Contributing to this overhead are the aforementioned memory barriers and cacheline flushes required for propagating page table changes.

In a nutshell, we propose to replace the table hierarchy with a per-ring *flat* page table (1D array). IOVAs would constitute indices of the array, thus eliminating IOVA (de)allocation overheads. Not having to walk a hierarchical structure would additionally reduce the table walk cost. In accordance with Figure 9e, we propose an IOTLB that holds at most one entry per ring, such that each table walk removes the previous IOVA translation. Consequently, given a burst of unmaps, only the last IOVA in the sequence requires explicit invalidation. We further discuss this point later on.

We next describe the rIOMMU design in detail. There are several ways to realize the rIOMMU concept, and our description should be viewed as an example. Contrary to the baseline IOMMU, which provides protection in page granularity, our rIOMMU facilitates finer-grained protection of any specified size. This property is appealing because different target buffers can, and often do, reside on the same page. Therefore, the existing IOMMU allows the I/O device to access an already unmapped buffer if the page in which it resides additionally houses a still-mapped buffer. In contrast, rIOMMU eliminates this type of vulnerability.

Data Structures Figure 9 defines the rIOMMU data structures. The rDEVICE (Figure 9a) is to the rIOMMU what the root page table is to the baseline IOMMU. It is uniquely associated with a bus-device-function (bdf) triplet and is pointed to by the context table (Figure 2). As noted, each DMA carries with it a bdf, allowing the rIOMMU to find the corresponding rDEVICE when needed. The rDEVICE consists of a physical pointer to an array of rRING structures (Figure 9b) and a matching size. Each rRING entry represents a flat page table. It likewise contains the table's physical address and size. The OS associates with each rRING: (1) a tail pointing to the next entry to be allocated in the flat table, and (2) the current number of valid mappings in the table. The latter two are not architected and are unknown to the rIOMMU hardware. We include them in rRING to simplify the description.

Each ring buffer of the I/O device is associated with *two* rRINGs in the rDEVICE array. The first corresponds to IOVAs pointing to the device ring buffer (Step 1 of Figure 5 for translating the head register). The second corresponds to IOVAs that the device finds within its ring descriptors (Step 5 in Figure 5 for translating target buffers). The IOVAs that reside in the first flat table are mapped as part of the I/O device initialization. They will be unmapped only when the device is brought down, as the device rings are always accessible to the device. IOVAs residing in the second flat table are asso-

```
a) struct rDEVICE {
  u16   size;
  rRING rings[size];
};
```

```
b) struct rRING {
  u18 size;
  rPTE ring[size];
  u18 tail;    // SW only
  u18 nmapped; // SW only
};
```

```
c) struct rPTE {
  u64 phys_addr;
  u30 size;
  u02 dir;
  u01 valid;
  u31 unused;
}; // = 128bit
```

```
d) struct rIOVA {
  u30 offset;
  u18 rentry;
  u16 rid;
}; // = 64bit
```

```
e) struct rIOTLB_entry {
  u16  bdf;
  u16  rid;
  u18  rentry;
  rPTE rpte;
  rPTE next;
};
```

Figure 9. The rIOMMU data structures. rDEVICE is used only by hardware. Last two fields of rRING are used only by software.

ciated with DMA target buffers; they are mapped/unmapped repeatedly and are valid only while their DMA is in flight.

The flat table pointed to by rRING.ring is an array of rPTE structures (Figure 9c). An rPTE consists of the physical address and size associated with the corresponding IOVA; two bits that specify the DMA *direction*, which can be from the device, to it, or both; and a bit that indicates whether the rPTE (and thus the corresponding IOVA) are valid. The physical address need not be page aligned and the associated size can have any value, allowing for fine-grained protection.

The rIOVA structure (Figure 9d) defines the format of IOVAs. As noted, every DMA has a bdf that uniquely identifies its rDEVICE. The rIOVA.rid (ring ID) serves as an index to the corresponding rDEVICE.rings array, and thus it uniquely identifies the rRING of the rIOVA. Likewise, rIOVA.rentry serves as an index to the rRING.ring array, and thus it uniquely identifies the rPTE of the rIOVA. The target address of the rIOVA is computed by adding rIOVA.offset to rPTE.phys_addr.

The data structures discussed so far are used by both software and hardware. They are setup by the OS and utilized by the rIOMMU to translate rIOVAs. The last one (Figure 9e) is a hardware-only structure, representing one rIOTLB entry. The combination of its first two fields (bdf+rid) uniquely identifies a rRING flat page table, which we denote as T. The rIOTLB utilizes at most one rIOTLB_entry per T. The combination of the first three fields (bdf+rid+rentry) uniquely identifies T's "current" rPTE—the PTE associated with the most recently translated rIOVA that belongs to T. The current rPTE is cached by rIOTLB_entry.rpte (holds a copy). The rIOTLB_entry.next field may or may not contain a prefetched copy of T's subsequent rPTE. (Our design does not depend on the latter field and works just as well without it.)

Hardware The rtranslate routine (Figure 10 top/left) outlines how rIOMMU translates a rIOVA to a physical address. First, it searches for e, the rIOTLB_entry of the rRING that is associated with the rIOVA. (Recall that there is only one such entry per rRING.) If e is missing from the rIOTLB, rIOMMU walks the table using the data structures defined above, finds the rPTE, and inserts to the rIOTLB a matching entry. Doing the table walk ensures that e.rpte is the rPTE of the given rIOVA. However, if e was initially found in the rIOTLB, then e and the rIOVA might mismatch. RIOMMU therefore compares the rentry numbers of e and the IOVA, and it updates e if they are different. Now that e is up-to-date, rIOMMU checks that the direction of the DMA is permitted according to the rPTE. It also checks that the offset of the IOVA is in range, namely, smaller than the associated rPTE.size. Violating these conditions constitutes an error, causing rIOMMU to trigger an I/O page fault (IOPF). IOPFs are not expected to occur (drivers pin target buffers to memory), and OSes typically reinitialize the I/O device if they do. If no violation is detected, rIOMMU finally performs the translation by adding the offset of the IOVA to rPTE.phys_addr.

The rtable_walk routine (Figure 10 top/right) ensures that the rIOVA complies with the rIOMMU data structure limits as well as points to a valid rPTE. Noncompliance might be the result of, e.g., an errant DMA or a buggy driver. After validation, rtable_walk initializes the rIOTLB_entry in a straightforward manner based on the rIOVA and its rPTE. It additionally attempts to prefetch the subsequent rPTE by invoking rprefetch (Figure 10 bottom/right), which succeeds if the next rPTE is valid. Prefetching can be asynchronous.

The riotlb_entry_sync routine (Figure 10 bottom/left) is used by rtranslate to synchronize e (the rIOTLB_entry) with the current IOVA. The two become unsynchronized, e.g., whenever the device handles a new DMA descriptor. The required rPTE can be found in e.next if prefetching was previously successful, in which case the routine assigns e.next to e.rpte. Otherwise, it uses rtable_walk to fetch the needed rPTE. Finally, it attempts to prefetch the subsequent rPTE.

Software The (un)map functions comprising the rIOMMU OS driver are shown in Figure 11. Their prototypes are logically similar to the associated Linux functions from the baseline IOMMU OS driver (Figures 4 and 6), with minor adjustments. The map flow corresponds to Figure 4. It gets a device, a ring ID, a physical address to be mapped, and the associated size and direction of the DMA. The first part of the code allocates a ring entry rPTE at the ring's tail and then updates the tail/nmapped fields accordingly. This allocation—which consists of incrementing two integers—is analogous to the costly IOVA allocation of baseline Linux.

The second part of map initializes the newly allocated rPTE. When the rPTE is ready, the map function invokes sync_mem, which ensures that the rPTE memory updates are visible to the rIOMMU. This part of the code is analogous to walking and updating the page table hierarchy of the baseline IOMMU, but it is simpler since the page table is flat. The return statement of the map function packs the rentry index and its ring ID into an IOVA as dictated by the rIOVA data structure (Figure 9d). The offset is always set to be 0 by the rIOMMU driver. Callers of map can later manipulate the offset as they please,

```
u64 rtranslate(u16 bus_dev_func, rIOVA iova, u2 dir) {
    rIOTLB_entry e = riotlb_find( bus_dev_func, iova.rid );
    if( ! e  ) {
        e = rtable_walk( bus_dev_func, iova );
        riotlb_insert( e );
    }
    if( e.rentry != iova.rentry )
        riotlb_entry_sync( bus_dev_func, iova, e );
    if( iova.offset >= e.rpte.size || ! (e.rpte.dir & dir) )
        io_page_fault();
    return e.rpte.phys_addr + iova.offset;
}
void riotlb_entry_sync(u16 bus_dev_func, rIOVA iova,
                       rIOTLB_entry e)
{
    rDEVICE d = get_domain( bus_dev_func );
    u18 next = (e.rentry + 1) % d.rings[e.rid].size;

    if( e.next.valid && (iova.rentry===next) ) {
        e.rpte = e.next;  e.rentry = next;
        e.next.valid = 0;
    } else
        e = rtable_walk( bus_dev_func, iova );
    rprefetch( d, e );
}
```

```
rIOTLB_entry
rtable_walk(u16 bus_dev_func, rIOVA iova)
{
    rDEVICE d = get_domain( bus_dev_func );
    if( iova.rid    >= d.size ||
        iova.rentry >= d.rings[iova.rid].size ||
        ! d.rings[iova.rid].ring[iova.rentry].valid )
        io_page_fault();

    rIOTLB_entry e;
    rRING r  = d.rings[iova.rid];
    e.bdf    = bus_dev_func;
    e.rid    = iova.rid;
    e.rentry = iova.rentry;
    e.rpte   = r.ring[e.rentry]; // copy
    rprefetch( d, e );
    return e;
}
void rprefetch(rDEVICE d, rIOTLB_entry e) // async
{
    rRING r  = d.rings[e.rid];
    u18 next = (e.rentry + 1) % r.size;
    if( r.size > 1 && r.ring[next].valid )
        e.next = r.ring[next];  // copy
}
```

Figure 10. Outline of the rIOMMU hardware logic. All DMAs are carried out with IOVAs that are translated by rtranslate.

```
rIOVA map(rDEVICE d, u16 rid, u64 pa, u30 size, u2 direction)
{
    rRING r = d.rings[rid];
    locked { if( r.nmapped == r.size ) return OVERFLOW;
             u18 t  = r.tail;
             r.tail = (r.tail + 1) % r.size;
             r.nmapped++;  }

    r.ring[t].phys_addr = pa;
    r.ring[t].size      = size;
    r.ring[t].dir       = direction;
    r.ring[t].valid     = 1;
    sync_mem( & r.ring[t] );
    return pack_iova( 0/*offset*/, t/*rentry*/, rid );
}
```

```
void unmap(rDEVICE d, rIOVA iova, bool end_of_burst) {
    rRING r = d.rings[iova.rid];
    r.ring[iova.rentry].valid = 0;
    locked { r.nmapped--; }
    sync_mem( & r.ring[iova.rentry] );
    if( end_of_burst )
        riotlb_invalidate( bus_dev_func(d), iova.rid );
}
void sync_mem(void * line) {
    if( ! riommu_pt_is_coherent() ) {
        memory_barrier();
        cache_line_flush( line );
    }
    memory_barrier();
}
```

Figure 11. Outline of the rIOMMU OS driver map and unmap routines, which respectively correspond to Figures 4 and 6.

provided they conform to the size constraint encoded into the corresponding rPTE.

The flow of unmap (Figure 11/right) corresponds to Figure 6. Unmap gets an rIOVA, marks the associated rPTE as invalid (analogous to walking the table hierarchy), decrements the ring's nmapped counter (analogous to IOVA deallocation), and synchronizes the memory to make the rPTE update visible to the rIOMMU. Recall that when the device driver is notified that its device has finished some DMAs, it loops through the relevant descriptors and sequentially unmaps their IOVAs (§2.3). The driver sets the end_of_burst parameter of unmap to true at the end of this loop upon the last IOVA, signifying that an rIOTLB_entry invalidation is required. One such invalidation is sufficient for the entire burst because, by design, each rRING has at most one rIOTLB_entry allocated in the rIOTLB.

Our experimental measurements indicate that the average loop length of a throughput-sensitive workload such as Netperf is ~200 iterations. This is long enough to make the amortized cost of IOTLB invalidations negligible, as in the deferred mode, but without sacrificing safety. Amortization, however, does not apply to latency-sensitive workloads. Nonetheless, the invalidation cost is small in comparison to the overall latency as will shortly be demonstrated.

Finally, we consider the problem of synchronizing the memory between the IOMMU and its driver. In sync_mem (Figure 11 bottom/right), we see support for two hardware modes, corresponding to whether the IOMMU table walk is coherent with the CPU caches. The baseline Linux kernel queries the relevant IOMMU capability bit. If it finds that the two are not in the same coherency domain, it introduces an additional memory barrier followed by a cacheline flush. In the following section, we experimentally evaluate two simulated rIOMMU versions corresponding to these two modes.

Applicability and Limitations We note in passing that rIOMMU relies on the predictability of the order of IOVA (un)mappings—not the order by which the IOVAs are used by the device. That is, so long as IOVAs are valid (mapped), they can be used out of order. They are merely indices to a

1D-array. The only implication of an out of order access is that the rIOTLB prefetched 'next' entry would not satisfy that access, so the translation would have to be fetched from DRAM. (Namely, the riotlb_entry_sync hardware routine will be re-invoked; see left of Figure 10.)

Let R be a ring. Let D be the number of DMA descriptors comprising R. Let L be the maximal number of R's live IOVAs whose DMAs are currently in flight. And let N be the size of the associated rRING. N is set by the device driver upon startup. Optimally, $N \geq L$, or else the driver would experience overflow (2nd line of map in Figure 11). While suboptimal, overflow is legal as with other devices employing rings; it just means that the driver should slow down. D is typically hundreds or a few thousands. In some I/O devices, each descriptor can hold a constant number of IOVAs (K), so setting $N = D \times K$ would prevent overflow. Some devices support scatter-gather lists, whose K might be large or unbounded. Developers of device drivers must therefore make a judicious decision regarding N based on their domain-specific knowledge about L. (In our experiments, L was at most 8K for all rings.) Alternatively, developers can opt for using the baseline IOMMU as discussed below.

The IOMMU induces noticeable overhead only for high bandwidth I/O devices, notably NICs and PCIe SSDs. We have thus far focused on the former, but rIOMMU is also applicable to the latter. Specifically, PCIe SSDs adhere to the NVM Express (NVMe) specification for fast non-volatile memory devices [41], which dictates that OS-device interactions are conducted via circular rings that impose a strict (un)mapping order. (There can be up to 64K such rings, which are denoted "queues"; each queue may contain up to 64K descriptors, which are denoted "commands".)

Conversely, rIOMMU is inapplicable to slower SATA drives, which adhere to the Advanced Host Controller Interface Specification (AHCI) [29]. A SATA drive has a single queue with 32 slots (DMA descriptors) that can be processed by the drive in arbitrary order. It would be easy to extend rIOMMU to support this work mode as well. But such support seems unneeded, because SATA drives are too slow to benefit. Our measurements indicate indistinguishable performance results when running the Bonnie++ benchmark [15, 39] executing sequential I/O with (1) strict IOMMU protection and with (2) a disabled IOMMU. Regardless of whether we use a SATA HDD or a SATA SSD.

rIOMMU is likewise inapplicable in setups that exploit remote direct memory access (RDMA), where fine-grained protection is impractical. The reason: all the memory that could be accessed by the passive, receiving NIC must be persistently mapped by the IOMMU, as it is unknown which location would be accessed. We therefore do not propose to entirely replace the baseline IOMMU, only to supplement it. (PCIe allows for multiple IOMMUs and existing systems routinely support such configurations.)

5. Evaluation

5.1 Methodology

Simulating rIOMMU We experimentally evaluate the seven IOMMU modes defined in §3–4: (1) strict, which is the completely safe Linux baseline; (2) strict+, which enhances strict with our faster IOVA allocator; (3) defer, which is the Linux variant that trades off some protection for performance by batching IOTLB invalidations; (4) defer+, which is defer with our IOVA allocator; (5) riommu- (in lowercase), which is the newly proposed rIOMMU when assuming no I/O page table coherency; (6) riommu, which does assume coherent I/O page tables; and (7) none, which turns off the IOMMU.

The five non-rIOMMU modes are executed as is. They constitute full implementations of working systems and do not require a simulation component. To simulate the two rIOMMU modes, we start with the none mode as the baseline. We then supplement the baseline with calls to the (un)map functions, similarly to the way they are called in the non-simulated IOMMU-enabled modes. But instead of invoking the native functions of the Linux IOMMU driver (Figures 4 and 6), we invoke the (un)map functions that we implement in the simulated rIOMMU driver (Figure 11). All the code of the rIOMMU driver can be—and is—executed, with one exception. Since there is no real rIOTLB, we must simulate the invalidation of rIOTLB entries. We do so by busy waiting for 2,150 cycles upon each entry invalidation, in accordance to the measurements specified in Table 1.

Notice that our methodology does not account for differences between the existing and proposed IOMMU translation mechanism. We only account for actions shown in Figures 4 and 6 but not those in Figure 5. Notably, we ignore the fact that the IOMMU works harder than the rIOMMU due to IOTLB misses that rIOMMU avoids via prefetching. We likewise ignore the fact that rIOMMU works harder than the no-IOMMU mode, since it translates addresses whereas the no-IOMMU mode does not. We ignore these differences, as the model validated in §3.3 shows that throughput is entirely determined by the number of cycles it takes the core—not the device or the IOMMU—to process a DMA request, even for the most demanding I/O-intensive workloads. The system behaves this way probably because the device and IOMMU operate in parallel to the CPU and are apparently fast enough so as not to constitute a bottleneck.

We revalidate our methodology and show that it is also applicable for latency-sensitive workloads by using the standard Netperf UDP request-response (RR) benchmark, which repeatedly sends one byte to its peer and waits for an identical response. We run RR under two IOMMU modes: hardware pass-through (HWpt) and software pass-through (SWpt). With HWpt, the IOMMU is enabled, but it translates each IOVA to an identical physical address without consulting the IOTLB or any page table. SWpt provides an equivalent functionality by using a page table that maps the entire physical memory and associates each physical page address with an identical IOVA. Under SWpt, Netperf RR experiences an IOTLB miss on every packet it sends and receives. Nonetheless, we find that the performance of HWpt and SWpt is identical, because the network stack and interrupt processing introduce far greater latencies that hide the IOTLB miss penalty. Moreover, we find that the RR performance of HWpt/SWpt is identical to that of no-IOMMU.

Throughput performance of Netperf stream with HWpt and SWpt is smaller by ~10% relative to no-IOMMU. But here too the difference is entirely caused by the core: about 200 CPU cycles spent on unrelated kernel abstraction code that executes under HWpt/SWpt but not under no-IOMMU.

Experimental Setup In an effort to get more general results, we conduct the evaluation using two setups involving two different NICs, as follows.

The Mellanox setup (mlx for short) is comprised of two identical Dell PowerEdge R210 II Rack Server machines that communicate through Mellanox ConnectX3 40Gbps NICs connected back to back and configured to use Ethernet. We use one machine as the server and the other as a workload generator client. Each machine has a 8GB 1333MHz memory and a single-socket 4-core Intel Xeon E3-1220 CPU running at 3.10GHz. The chipset is Intel C202, which supports VT-d, Intel's Virtualization Technology that provides IOMMU functionality. We configure the server to utilize one core only and turn off all power optimizations—sleep states (C-states) and dynamic voltage and frequency scaling (DVFS)—to avoid reporting artifacts caused by nondeterministic events. The machines run Ubuntu 12.04 with the Linux 3.4.64 kernel. All experimental findings described thus far were obtained with the mlx setup.

The Broadcom setup (brcm for short) is similar, likewise utilizing two R210 machines. The differences are that the two machines communicate through Broadcom NetXtreme II BCM57810 10GbE NICs (connected by a CAT7 10GBASE-T cable for fast Ethernet); that they have 16GB memory; and that they run the Linux 3.11.0 kernel.

The mlx and brcm device drivers differ substantially. Notably, mlx utilizes more ring buffers and allocates more IOVAs (we observed a total of ~12K addresses for mlx and ~3K for brcm). The mlx driver uses two target buffers per packet (header and body) and thus two IOVAs, whereas the brcm driver allocates only one buffer/IOVA per packet.

Benchmarks To drive our experiments we utilize the following benchmarks. Netperf TCP stream [30] is a standard tool to measure networking throughput. It maximizes the amount of data sent over one TCP connection, simulating an I/O-intensive workload. Its default message size is 16KB. This is the application we used in §3.

Netperf UDP RR (request-response) models a latency sensitive workload by exchanging one byte messages in a ping-pong manner. The per message latency can be calculated as the inverse of the number of messages per second.

Apache [18, 19] is a HTTP web server. We drive it with ApacheBench [5] (distributed with Apache) that measures the number of requests per second that the server is capable of handling, issuing (32) concurrent requests of a static page of a given size. We run two instances of the benchmark, requesting a smaller (1KB) and a bigger (1MB) file.

Memcached [20] is an in-memory key-value storage server. We drive it with the Memslap benchmark [2] (part of the libmemcached client library), which measures the completion rate of the requests that it generates. By default, Memslap generates a workload comprised of 90% get and 10% set operations, with 64B and 1KB key and value sizes, respectively. It too is set to use 32 concurrent requests.

5.2 Results

We run each benchmark 100 times. Each individual run is configured to take ~10 seconds. We treat the first 10 runs as warmup and report the average of the remaining 90 runs. Figure 12 shows the resulting throughput and CPU consumption for the mlx (top) and brcm (bottom) setups. The corresponding normalized performance is shown in Table 2, specifying the relative improvement of the two rIOMMU variants over the other modes. The top/left plot in Figure 12 corresponds to the analysis and data shown in Figures 7–8.

Let us discuss the results in Figure 12, left to right. The greatest observed improvement by rIOMMU is attained with mlx / Netperf stream (Figure 12/top/left). This result is to be expected considering the model from §3.3 showing that every cycle shaved off the IOVA (un)mappings translates into increased throughput. CPU cycles constitute the bottleneck resource, as is evident from the mlx/stream/CPU curve, which is at 100% for all IOMMU modes. The notable difference between riommu- and riommu is due to ~1.1K cycles that the former adds to the latter, which is the cost of four additional memory barriers and four additional cacheline flushes, per packet. (Specifically, a barrier and a cacheline flush in both map and unmap for two IOVAs corresponding to the packet's header and data.) Riommu- and riommu provide 2.90–7.56x higher throughput relative to the completely safe IOMMU modes strict and strict+, and 1.74–3.79x higher throughput relative to the deferred modes. The latter, however, does not constitute an apple-to-apples comparison, since the deferred modes are vulnerable whereas the rIOMMU modes are safe. Riommu- and riommu deliver 0.52x and 0.77x lower throughout relative to the unprotected, no-IOMMU optimum.

The brcm/stream results (Figure 12/bottom/left) are quantitatively and qualitatively different. In particular, all IOMMU modes except strict have enough cycles to saturate the Broadcom NIC and achieve its line-rate, which is 10 Gbps. The brcm setup requires fewer cycles per packet because its device driver is more efficient, e.g., due to utilizing only one IOVA per packet instead of two. In setups of this type—where

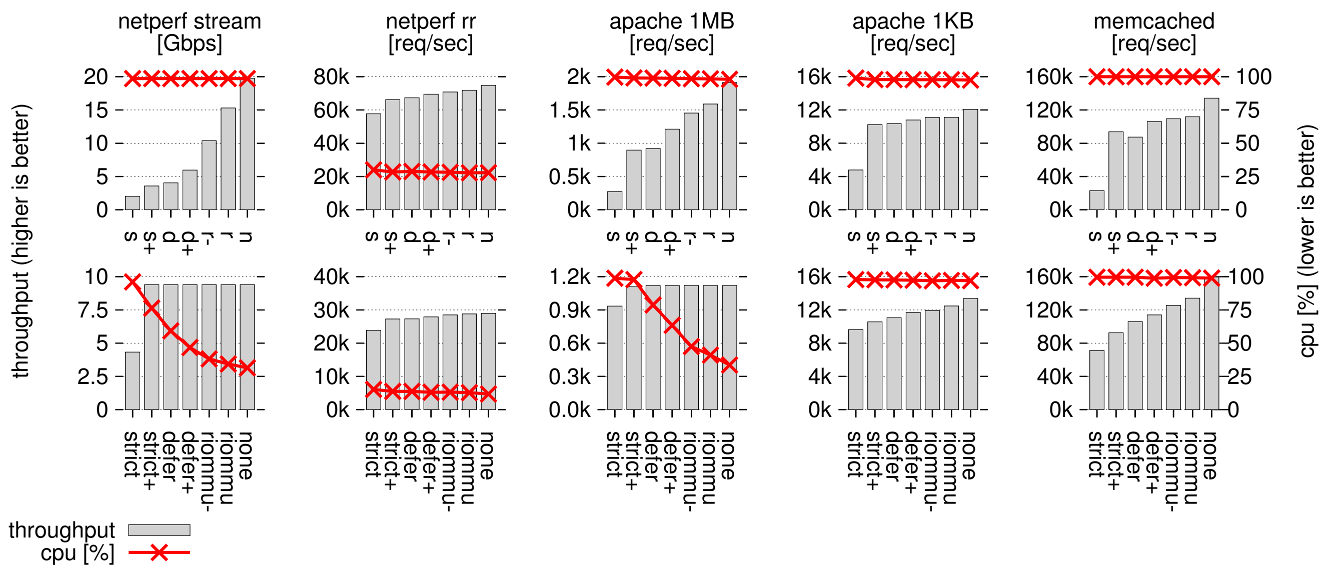

Figure 12. Performance of the IOMMU modes when using the Mellanox (top) and Broadcom (bottom) NICs.

NIC	*benchmark*	*throughput*										*cpu*									
		riommu- divided by					*riommu divided by*					*riommu- divided by*					*riommu divided by*				
		strict	*strict+*	*defer*	*defer+*	*none*	*strict*	*strict+*	*defer*	*defer+*	*none*	*strict*	*strict+*	*defer*	*defer+*	*none*	*strict*	*strict+*	*defer*	*defer+*	*none*
mlx	stream	5.12	2.90	2.57	1.74	0.52	7.56	4.28	3.79	2.57	0.77	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	rr	1.23	1.07	1.05	1.02	0.95	1.25	1.09	1.07	1.03	0.96	0.94	0.99	0.98	0.99	1.01	0.93	0.98	0.96	0.98	1.00
	apache 1M	5.30	1.62	1.58	1.20	0.76	5.80	1.77	1.73	1.31	0.83	0.99	0.99	1.00	1.00	1.00	0.99	0.99	0.99	1.00	1.00
	apache 1K	2.32	1.08	1.07	1.03	0.92	2.32	1.08	1.07	1.03	0.92	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00
	memcached	4.77	1.17	1.25	1.03	0.82	4.88	1.19	1.28	1.05	0.83	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
brcm	stream	2.17	1.00	1.00	1.00	1.00	2.17	1.00	1.00	1.00	1.00	0.40	0.50	0.64	0.81	1.21	0.36	0.45	0.58	0.73	1.09
	rr	1.19	1.05	1.04	1.02	0.99	1.21	1.06	1.05	1.03	1.00	0.86	0.96	0.96	1.00	1.11	0.84	0.93	0.93	0.98	1.08
	apache 1M	1.20	1.01	1.00	1.00	1.00	1.20	1.01	1.00	1.00	1.00	0.48	0.49	0.60	0.75	1.41	0.41	0.42	0.52	0.65	1.22
	apache 1K	1.24	1.13	1.08	1.02	0.89	1.29	1.18	1.13	1.07	0.93	0.99	0.99	0.99	1.00	1.00	0.99	1.00	1.00	1.00	1.00
	memcached	1.76	1.35	1.18	1.10	0.78	1.88	1.45	1.27	1.18	0.84	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2. Relative (normalized) performance numbers.

the network is saturated—the performance metric of interest becomes the CPU consumption. By Table 2, we can see that riommu and riommu- consume 0.36–0.50x fewer CPU cycles than the two strict modes; 0.58–0.81x fewer cycles than the deferred modes; and 1.09x and 1.21x more cycles than the no-IOMMU optimum, respectively.

The improvement by rIOMMU is less pronounced when running RR, in both mlx and brcm, with 1.02–1.25x higher throughput and 0.84–1.00x lower CPU consumption relative to the strict and deferred variants. It is less pronounced due to RR's ping-pong nature, which implies that CPU cycles are in low demand, as indicated by the CPU curves at 28–30% for mlx and at 12–15% for brcm. For this reason, in comparison to mlx/RR/none, rIOMMU has 4–5% lower throughput and nearly identical CPU consumption. In comparison to brcm/RR/none, rIOMMU has 8–11% higher CPU consumption and nearly identical throughput. Although the per-packet processing time at the core is smaller in brcm, overall, the mlx hardware transmits packets faster, as indicated by its higher RR throughput. The corresponding round-trip time of the different modes (which, as noted, is the inverse of the throughput in RR's case) is shown in Table 3.

NIC	*strict*	*strict+*	*defer*	*defer+*	*riommu-*	*riommu*	*none*
mlx	17.3	15.1	14.9	14.4	14.1	13.9	13.4
brcm	41.9	36.7	36.6	35.8	35.1	34.7	34.6

Table 3. Netperf RR round-trip time in microseconds.

The results of Apache 1MB are qualitatively identical to those of Netperf stream, because the benchmark transmits a lot of data per request and is thus throughput sensitive. Conversely, Apache 1KB is not throughput sensitive. Its smaller 1KB requests make the performance of mlx and brcm look remarkably similar despite their networking infrastructure difference. In both cases, the bottleneck is the CPU, while the volume of the transmitted data is only a small fraction of the NICs capacity. (Both deliver ~12K requests per second of 1KB files, yielding a transfer rate of ~0.1 Gbps.) The reason: Apache requires heavy processing for each http request. This overhead is amortized over hundreds of packets in the case of Apache 1MB but over only one packet in the case of 1KB. Thus, the computational processing dominates the throughput of Apache 1KB and the role of the networking infrastructure is marginalized. Even so, rIOMMU demonstrates a ~1.24x and ~2.32x throughput improvement over brcm/strict and mlx/strict, respectively. It is up to 1.18x higher relative to the other IOMMU-enabled modes. And ~0.9x lower relative to the unprotected optimum.[2]

The network activity of Apache 1KB is somewhat similar to that of the Memcached benchmark, because both are configured with 32 concurrent requests, both receive queries comprised of a few dozens of bytes (file name or key item), and both transmit 1KB responses (file content or data item). The difference is that the Memcached internal logic is simpler, as its purpose is merely to serve as an in-memory LRU cache. For this reason, it achieves an order of magnitude higher throughput relative to Apache 1KB.[3] The shorter per-request processing time makes the differences between the IOMMU configurations more pronounced, with rIOMMU throughput that is 1.17–4.88x higher than the completely safe modes, 1.03–1.28x higher than the deferred modes, and 0.78–0.84x lower than the optimum.

5.3 When IOTLB Miss Penalty Matters

Our experiments thus far indicated that using the IOMMU affects performance because it mandates the OS to spend CPU cycles on creating and destroying IOVA mappings. We were unable to measure the overhead caused by the actual IOMMU translation activity of walking the page tables upon an IOTLB miss (Figures 2 and 5). In §5.1, we attributed this inability to the substantially longer latencies induced by interrupt processing and the TCP/IP stack. In Table 3, we specified the round-trip latencies, whose magnitude (13–42 μs) seems to suggest that the occasional cost of 4 memory references per table walk is negligible in comparison.

There are, however, high performance environments that enjoy lower latencies in the order of a μs [14, 21, 43, 48], which is required, e.g., "where a fraction of a microsecond can make a difference in the value of a transaction" [1]. User-level I/O, for example, might permit applications to (1) utilize raw Ethernet packets to eliminate TCP/IP overheads, and to (2) poll the I/O device to eliminate interrupt delays.

With the help of the ibverbs library [27, 35], we established such a configuration on top of the mlx setup. We ran two experiments. The first iteratively and randomly selects a buffer from a large pool of previously mapped buffers and transmits it, thus ensuring that the probability for the corresponding IOVA to reside in the IOTLB is low. The second experiment does the same but with only one buffer, thus ensuring that the IOTLB always hits. The latency difference—which is the cost of an IOTLB miss—was ~0.5 μs (1532 cycles on average); we believe it is reasonable to assume that it approximates the benefit of using rIOMMU over the existing IOMMU in high performance environments of this type.

[2] We note in passing that our Apache 1KB throughput results coincide with that of Soares et al. [46], who reported a latency of 22ms for 256 concurrent requests, which translate to 1000/22 $\times$ 256 $\approx$ 12K requests/second.

[3] Our Memcached results are comparable to that of Gordon et al. [22].

5.4 Comparing to TLB Prefetchers

rIOMMU is not a prefetcher. Rather, it is a new IOMMU design that allows for efficient IOVA (un)mappings while minimizing costly IOTLB invalidations. (Unrelated to prefetching.) But rIOMMU does have a prefetching component, since it loads to the rIOTLB the next IOVA to be used ahead of time. While this component turned out to be useful only in specialized setups (§5.3), it is still interesting to compare this aspect of our work to previously proposed TLB prefetchers.

For lack of space, we only briefly describe the bottom line. We modified the IOMMU layer of KVM/QEMU to log the DMAs that its emulated I/O devices perform. We ran our benchmarks in a VM and generated DMA traces. We fed the traces to three simulated TLB prefetchers: Markov [31], Recency [44], and Distance [34], as surveyed by Kandiraju and Sivasubramaniam [33]. We found their baseline versions to be ineffective, as IOVAs are invalidated immediately after being used. We modified them and allowed them to store invalidated addresses, but mandated them to walk the page table and check that their predictions are mapped before making them. Distance was still ineffective. Recency and Markov, however, were able to predict most accesses, but only if the number of entries comprising their history data structure grew larger than the ring. In contrast, rIOTLB requires only two entries per ring and its "predictions" are always correct.

6. Conclusions

The IOMMU design is similar to that of the regular MMU, despite the inherent differences in their workloads. This design worked reasonably well when I/O devices were relatively slow as compared to the CPU. But it hampers the performance of contemporary devices like 10/40 Gbps NICs. We foresee that this problem will get worse due to the ever increasing speed of such devices. We thus contend that it makes sense to rearchitect the IOMMU such that it directly supports the unique characteristics of its workload. We propose rIOMMU as an example for such a redesign and show that the benefits of using it are substantial.

Acknowledgments We thank the anonymous reviewers for their feedback. We thank Ilya Lesokhin and Igor Smolyar for providing much appreciated technical help. This research was funded in part by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and by the Israeli Ministry of Science and Technology grant #3-9779. Mellanox graciously donated NICs.

References

[1] Dennis Abts and Bob Felderman. A guided tour through data-center networking. *ACM Queue*, 10(5):10:10–10:23, May 2012.

[2] Brian Aker. Memslap - load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memslap.html`. libmemcached 1.1.0 documentation.

[3] AMD Inc. AMD IOMMU architectural specification, rev 2.00. `http://support.amd.com/TechDocs/48882.pdf`, Mar 2011.

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[5] Apachebench. `http://en.wikipedia.org/wiki/ApacheBench`.

[6] Apple Inc. Thunderbolt device driver programming guide: Debugging VT-d I/O MMU virtualization. `https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/DebuggingThunderboltDrivers/DebuggingThunderboltDrivers.html`, 2013. (Accessed: May 2014).

[7] ARM Holdings. ARM system memory management unit architecture specification — SMMU architecture version 2.0. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062c/IHI0062C_system_mmu_architecture_specification.pdf`, 2013.

[8] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM Eurosys*, pages 73–85, 2006.

[9] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest applied security conference*, 2005.

[10] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007.

[11] James E.J. Bottomley. Dynamic DMA mapping using the generic device. `https://www.kernel.org/doc/Documentation/DMA-API.txt`. Linux kernel documentation.

[12] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, Feb 2014.

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[14] Cisco. Understanding switch latency. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white_paper_c11-661939.html`, Jun 2012. White paper. Accessed: Aug 2014.

[15] Russell Coker. The Bonnie++ benchmark. `http://www.coker.com.au/bonnie++/`. (Accessed: Jan 2015).

[16] Jonathan Corbet. *Linux Device Drivers*, chapter 15: Memory Mapping and DMA. O'Reilly, 3rd edition, 2005.

[17] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium*, pages 83–100, 2009.

[18] The Apache HTTP server project. `http://httpd.apache.org`.

[19] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, Jul 1997.

[20] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124), Aug 2004.

[21] Brice Goglin. Design and implementation of Open-MX: High-performance message passing over generic Ethernet hardware. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[22] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–422, 2012.

[23] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)*, pages 41–50, 2007.

[24] Brian Hill. Integrating an EDK custom peripheral with a LocalLink interface into Linux. Technical Report XAPP1129 (v1.0), XILINX, May 2009.

[25] IBM Corporation. PowerLinux servers — 64-bit DMA concepts. `http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm`. (Accessed: May 2014).

[26] IBM Corporation. AIX kernel extensions and device support programming concepts. `https://publib.boulder.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.kernelext/doc/kernextc/kernextc_pdf.pdf`, 2013. (Accssed: May 2014).

[27] ibverbs evaluation. `http://www.scalalife.eu/book/export/html/434`. Accessed: Aug 2014.

[28] Intel Corporation. Intel virtualization technology for directed I/O, architecture specification - architecture specification - rev. 2.2. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`, Sep 2013.

[29] Intel Corporation. Serial ATA advanced host controller interface (AHCI) 1.3.1. `http://www.intel.com/content/www/us/en/io/serial-ata/serial-ata-ahci-spec-rev1-3-1.html`, Mar 2014. (Accessed: Jan 2015).

[30] Rick A. Jones. A network performance benchmark (revision 2.0). Technical report, Hewlett Packard, 1995. `http:`

//www.netperf.org/netperf/training/Netperf.html.

[31] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 252–263, 1997.

[32] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–72, 2009.

[33] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 129–139, Jun 2002.

[34] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 195–206, 2002.

[35] Gregory Kerr. Dissecting a small InfiniBand application using the Verbs API. *Computing Research Repository (arxiv)*, abs/1105.1827, 2011. http://arxiv.org/abs/1105.1827.

[36] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2004.

[37] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient intra-operating system protection against harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)*, Feb 2015.

[38] Vinod Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007. (Accessed: May 2014).

[39] Ben Martin. Using Bonnie++ for filesystem performance benchmarking. Linux.com article. http://archive09.linux.com/feature/139742, 2008. (Accessed: Jan 2015).

[40] David S. Miller, Richard Henderson, and Jakub Jelinek. Dynamic DMA mapping guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. Linux kernel documentation.

[41] NVM Express Workgroup. NVM Express (NVMe) specification – revision 1.2. http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf, Nov 2014. (Accessed: Jan 2015).

[42] PCI-SIG. Address translation services revision 1.1. https://www.pcisig.com/specifications/iov/ats, Jan 2009.

[43] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. Technical Report UW-CSE-13-10-01, University of Washington, Jun 2014.

[44] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 117–127, 2000.

[45] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, 2007.

[46] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 33–46, 2010.

[47] Michael Swift, Brian Bershad, and Henry Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, Feb 2005.

[48] Transpacket Fusion Networks. Ultra low latency of 1.2 microseconds for 1G to 10G Ethernet aggregation. http://tinyurl.com/transpacket-low-latency, Dec 2012. Accessed: Aug 2014.

[49] Carl Waldspurger and Mendel Rosenblum. I/O virtualization. *Communications of the ACM (CACM)*, 55(1):66–73, Jan 2012.

[50] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 241–254, 2008.

[51] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008.

[52] Rafal Wojtczuk. Subverting the Xen hypervisor. In *Black Hat*, 2008. http://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf. (Accessed: May 2014).

[53] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 18:1–18:12, 2010.