Utilizing the IOMMU Scalably

Omer Peleg and Adam Morrison, *Technion—Israel Institute of Technology;*
Benjamin Serebrin, *Google;* Dan Tsafrir, *Technion—Israel Institute of Technology*

https://www.usenix.org/conference/atc15/technical-session/presentation/peleg

ISBN 978-1-931971-225

Utilizing the IOMMU Scalably

Omer Peleg Adam Morrison
Technion
{omer,mad}@cs.technion.ac.il

Benjamin Serebrin
Google
serebrin@google.com

Dan Tsafrir
Technion
dan@cs.technion.ac.il

Abstract

IOMMUs provided by modern hardware allow the OS to enforce memory protection controls on the DMA operations of its I/O devices. An IOMMU translation management design must *scalably* handle frequent concurrent updates of IOMMU translations made by multiple cores, which occur in high throughput I/O workloads such as multi-Gb/s networking. Today, however, OSes experience performance meltdowns when using the IOMMU in such workloads.

This paper explores *scalable IOMMU management designs* and addresses the two main bottlenecks we find in current OSes: (1) assignment of I/O virtual addresses (IOVAs), and (2) management of the IOMMU's TLB.

We propose three approaches for scalable IOVA assignment: (1) dynamic identity mappings, which eschew IOVA allocation altogether, (2) allocating IOVAs using the kernel's `kmalloc`, and (3) per-core caching of IOVAs allocated by a globally-locked IOVA allocator. We further describe a scalable IOMMU TLB management scheme that is compatible with all these approaches.

Evaluation of our designs under Linux shows that (1) they achieve 88.5%–100% of the performance obtained *without* an IOMMU, (2) they achieve similar latency to that obtained *without* an IOMMU, (3) scalable IOVA allocation and dynamic identity mappings perform comparably, and (4) `kmalloc` provides a simple solution with high performance, but can suffer from unbounded page table blowup.

1 Introduction

Modern hardware provides an I/O memory management unit (IOMMU) [2, 6, 24, 27] that mediates direct memory accesses (DMAs) by I/O devices in the same way that a processor's MMU mediates memory accesses by instructions. The IOMMU interprets the target address of a DMA as an I/O virtual address (IOVA) [32] and attempts to translate it to a physical address, blocking the DMA if no translation (installed by the OS) exists.

IOMMUs thus enable the OS to restrict a device's DMAs to specific physical memory locations, and thereby protect the system from errant devices [18, 29], malicious devices [8, 13, 46], and buggy drivers [7, 15, 22, 31, 41, 44], which may misconfigure a device to overwrite system memory. This *intra-OS* protection [45] is recommended by hardware vendors [23, 29] and implemented in existing OSes [5, 12, 25, 36]. OSes can employ intra-OS protection both in non-virtual setups, having direct access to the physical IOMMU, and in virtual setups, by exposing the IOMMU to the VM through hardware-supported nested IOMMU translation [2, 27], by paravirtualization [9, 31, 40, 45], or by full emulation of the IOMMU interface [3].

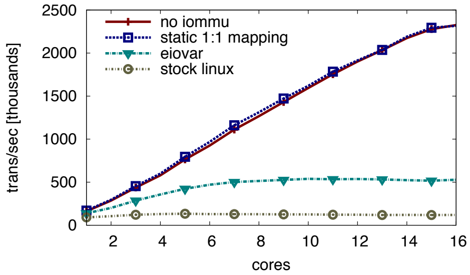

Figure 1. Parallel `netperf` throughput (Linux): Performance meltdown due to dynamic IOMMU mapping updates.

Intra-OS protection requires each DMA operation to be translated with a transient IOMMU mapping [12] dedicated to the DMA, which is destroyed once it completes so that the device cannot access the memory further [29, 37]. For example, a network interface card (NIC) driver maps the buffers it inserts into the NIC's receive (RX) rings to receive packets. Once a packet arrives (via DMA), the driver unmaps the packet's buffer.

These transient *dynamic* IOMMU mappings pose a performance challenge for driving high-throughput I/O workloads. Such workloads require dynamic mappings to be created and destroyed millions of times a second by *multiple cores concurrently*, since a single core often cannot sustain high enough throughput [30]. This paper specifically targets *multi-Gb/s networking*—NICs providing 10–40 Gb/s (and soon 100 Gb/s)—as a representative demanding case.

Current OSes melt down under load when the IOMMU is enabled in such a workload. Figure 1 demonstrates the problem on Linux. It shows the combined throughput of 270 `netperf` instances running a request-response workload (described in § 6) on a 16-core x86 server with a 10 Gb/s NIC, which we use as a running example throughout the paper. Mediating DMAs by the IOMMU imposes only negligible overhead by itself, as evidenced by the throughput obtained when the IOMMU is configured with *static* identity mappings that pass DMA requests through the IOMMU unchanged. In contrast, when IOMMU management is enabled, throughput drops significantly and ceases to increase with the number of cores. This occurs with both Linux's stock IOMMU subsystem or the recent EiovaR [35] optimization, which targets the *sequential* performance of Linux's IOMMU subsystem (see § 3.1). Other OSes suffer from similar scalability problems (§ 3).

This paper thus considers the **IOMMU management** problem of *designing a subsystem supporting high-throughput concurrent updates of IOMMU mappings*. We analyze the bottlenecks in current IOMMU management designs (§ 3) and explore the trade-offs in the design space of their solutions (§§ 4–5). Our designs address the two main bottlenecks we find in current OSes: IOVA assignment and IOTLB invalidation.

IOVA assignment Creating an IOMMU mapping for a physical memory location requires the OS to designate a range of IOVAs that will map to the physical location. The driver will later configure the device to DMA to/from the IOVAs. OSes presently use a dedicated, centralized (lock-protected) IOVA allocator that becomes a bottleneck when accessed concurrently.

We propose three designs for *scalable* IOVA assignment (§ 4), listed in decreasingly radical order: First, *dynamic identity mapping* eliminates IOVA allocation altogether by using a buffer's physical address for its IOVA. Consequently, however, maintaining the IOMMU page tables requires more synchronization than in the other designs. Second, *IOVA-kmalloc* eliminates only the specialized IOVA allocator by allocating IOVAs using the kernel's optimized `kmalloc` subsystem. This simple and efficient design is based on the observation that we can treat the addresses that `kmalloc` returns as IOVA page numbers. Finally, *per-core IOVA caching* keeps the IOVA allocator, but prevents it from becoming a bottleneck by using *magazines* [11] to implement per-core caches of free IOVAs, thereby satisfying allocations without accessing the IOVA allocator.

IOTLB invalidation Destroying an IOMMU mapping requires invalidating relevant entries in the IOTLB, a TLB that caches IOMMU mappings. The Linux IOMMU subsystem amortizes the invalidation cost by batching multiple invalidation requests and then performing a single global invalidation of the IOTLB instead. The batching data structure is lock-protected and quickly becomes a bottleneck. We design a compatible scalable batching data structure as a replacement (§ 5).

Design space exploration We evaluate the performance, page table memory consumption and implementation complexity of our designs (§ 6). We find that (1) our designs achieve 88.5%–100% of the throughput obtained *without* an IOMMU, (2) our designs achieve similar latency to that obtained *without* an IOMMU, (3) the savings dynamic identity mapping obtains from not allocating IOVAs are negated by its more expensive IOMMU page table management, making it perform comparably to scalable IOVA allocation, and (4) IOVA-kmalloc provides a simple solution with high performance, but it can suffer from unbounded page table blowup if empty page tables are not reclaimed (as in Linux).

Contributions This paper makes four contributions:

- Identifying IOVA allocation and IOTLB invalidation as the bottlenecks in the IOMMU management subsystems of current OSes.
- Three designs for scalable IOVA allocation: (1) dynamic identity mappings, (2) IOVA-kmalloc, and (3) per-core caching of IOVAs, as well as a scalable IOTLB invalidation scheme.
- Evaluation of the new and existing designs on several high throughput I/O workloads.
- Design space exploration: we compare the performance, page table memory consumption and implementation complexity of the proposed designs.

2 Background: IOMMUs

The IOMMU mediates accesses to main memory by I/O devices, much like the MMU mediates the memory accesses performed by instructions. IOMMUs impose a translation process on each device DMA. The IOMMU interprets the target address of the DMA as an *I/O virtual address* (IOVA) [32], and attempts to translate it to a physical address using per-device *address translations* (or *mappings*) previously installed by the OS. If a translation exists, the DMA is routed to the correct physical address; otherwise, it is blocked.

In the following, we provide a high-level description of Intel's x86 IOMMU operation [27]. Other architectures are conceptually similar [2, 6, 24].

IOMMU translations The OS maintains a *page table* hierarchy for each device, implemented as a 4-level radix tree (as with MMUs). Each radix tree node is a 4 KB page. An inner node (*page directory*) contains 512 pointers (*page directory entries*, or PDEs) to child radix-tree nodes. A leaf node (*page table*) contains 512 pointers (*page table entries*, or PTEs) to physical addresses. PTEs also encode the type of access rights provided through this translation, i.e., read, write or both.

The virtual I/O address space is 48-bit addressable.

The 36 most significant bits of an IOVA are its *page frame number* (PFN), which the IOMMU uses (when it receives a DMA request) to walk the radix tree (9 bits per level) and look up the physical address and access rights associated with the IOVA. If no translation exists, the IOMMU blocks the DMA and interrupts the processor. Unlike the analogous case in virtual memory, this is not a *page fault* that lets the OS install a new mapping and transparently resume operation of the faulting access. Instead, the DMA is simply dropped. The I/O device observes this and may not be able to recover. [1]

IOTLB translation caching The IOMMU maintains an IOTLB that caches IOVA translations. If the OS modifies a translation, it must *invalidate* (or *flush*) any TLB entries associated with the translation. The IOMMU supports individual invalidations as well as *global* ones, which flush all cached translations. The OS requests IOTLB invalidations using the IOMMU's *invalidation queue*, a cyclic buffer in memory into which the OS adds invalidation requests and the IOMMU processes them asynchronously. The OS can request to be notified when an invalidation has been processed.

2.1 IOMMU protection

IOMMUs can be used to provide *inter-* and *intra-OS* protection [3, 43, 45, 47]. IOMMUs are used for inter-OS protection in virtualized setups, when the host assigns a device for the exclusive use of some guest. The host creates a *static* IOMMU translation [45] that maps guest physical pages to the host physical pages backing them, allowing the guest VM to directly program device DMAs. This mode of operation does not stress the IOMMU management code and is not the focus of this work.

We focus on intra-OS protection, in which the OS uses the IOMMU to restrict a device's DMAs to specific physical memory locations. This protects the system from errant devices [18, 29], malicious devices [8, 13, 46], and buggy drivers [7, 15, 22, 31, 41, 44].

Intra-OS protection is implemented via the DMA API [12, 32, 37] that a device driver uses when programming the DMAs. To program a device DMA to a physical buffer, the driver must pass the buffer to the DMA API's *map* operation. The map operation responds with a *DMA address*, and it is the DMA address that the driver must program the device to access.

Internally, the map operation (1) allocates an IOVA range the same size as the buffer, (2) maps the IOVA range to the buffer in the IOMMU, and (3) returns the IOVA to the driver. Once the DMA completes, the driver must *unmap* the DMA address, at which point the mapping is destroyed and the IOVA range deallocated.

High throughput I/O workloads can create and destroy such *dynamic* IOMMU mappings [12] millions of times a second, on multiple cores concurrently, and thereby put severe pressure on the IOMMU management subsystem implementing the DMA API.

3 Performance Analysis of Present Designs

Here we analyze the performance of current IOMMU management designs under I/O workloads with high throughput and concurrency. We use Linux/Intel-x86 as a representative study vehicle; other OSes have similar designs (§ 3.4). Our test workload is a highly parallel RR benchmark, in which a `netperf` [28] server is handling 270 concurrent TCP RR requests arriving on a 10 Gb/s NIC. § 6 fully details the workload and test setup.

To analyze the overhead created by IOMMU management (shown in Figure 1), we break down the execution time of the parallel RR workload on 16 cores (maximum concurrency on our system) into the times spent on the subtasks required to create and destroy IOMMU mappings. Figure 2 shows this breakdown. For comparison, the last 3 bars show the breakdown of our scalable designs (§§ 4–5).

We note that this parallel workload provokes pathological behavior of the stock Linux IOVA allocator. This behavior, which does not exist in other OSes, causes IOVA allocation to hog $\approx 60\%$ of the execution time. The recent EiovaR optimization [35] addresses this issue, and we therefore use Linux with EiovaR as our baseline. We discuss this further in § 3.1.

In the following, we analyze each IOMMU management subtask and its associated overhead in the Linux design: IOVA allocation (§ 3.1), IOTLB invalidations (§ 3.2), and IOMMU page table management (§ 3.3).

3.1 Linux IOVA Allocation

In Linux, each device is associated with an IOVA allocator that is protected by a coarse-grained lock. Each IOVA allocation and deallocation for the device acquires its allocator's lock, which thus becomes a sequential bottleneck for frequent concurrent IOVA allocate/deallocate operations. Figure 2 shows that IOVA allocation lock acquisition time in the baseline accounts for 31.9% of the cycles. In fact, this is only because IOVA allocations are throttled by a *different* bottleneck, IOTLB invalidations (§ 3.2). Once we address the invalidations bottleneck, the IOVA allocation bottleneck becomes much more severe, accounting for nearly 70% of the cycles.

This kind of design—acquiring a global lock for each operation—would turn IOVA allocation into a sequential bottleneck no matter which allocation algorithm is used

[1] I/O page fault standardization exists, but since it requires support from the device, it is not widely implemented or compatible with legacy devices [2, 38, 39].

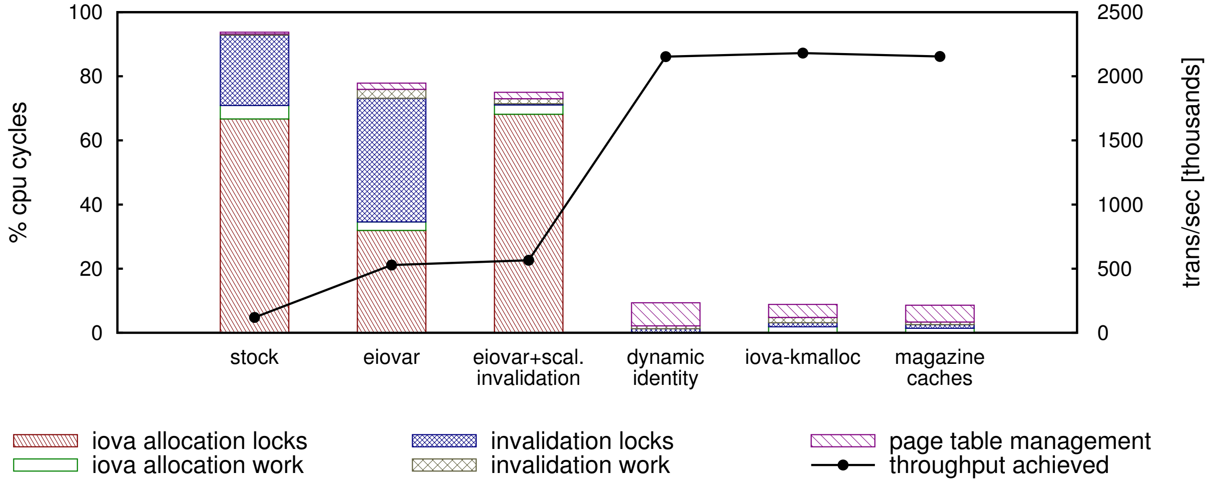

Figure 2. Throughput and cycle breakdown of time spent in IOMMU management on a 16-core parallel `netperf` RR workload. Stock Linux is shown for reference. Our baseline is Linux with **EiovaR** [35], which addresses a performance pathology in the stock Linux IOVA allocation algorithm (see § 3.1). In this baseline, the IOTLB invalidation bottleneck masks the IOVA allocation bottleneck, as evidenced by the third bar, which shows the breakdown after applying scalable IOTLB invalidations (§ 5) to the EiovaR baseline. Our designs are represented by the last three bars, nearly eliminating lock overhead.

once the lock is acquired. We nevertheless discuss the Linux IOVA allocation algorithm itself, since it has implications for IOMMU page table management.

The IOVA allocator packs allocated IOVAs as tightly as possible towards the end of the virtual I/O address space. This minimizes the number of page tables required to map allocated IOVAs—an important feature, because Linux rarely reclaims a physical page that gets used as an IOMMU page table (§ 3.3).

To achieve this, the IOVA allocator uses a red-black tree that holds pairwise-disjoint *ranges* of allocated virtual I/O page numbers. This allows a new IOVA range to be allocated by scanning the virtual I/O address space from highest range to lowest range (with a right-to-left traversal of the tree) until finding an unallocated gap that can hold the desired range. Linux attempts to minimize such costly linear traversals through a heuristic in which the scan starts from some previously cached tree node. This often finds a desired gap in constant time [35]. [2]

Unfortunately, the IOVA allocation patterns occurring with modern NICs can cause the heuristic to fail, resulting in frequent long linear traversals during IOVA allocations [35]. The EiovaR optimization avoids this problem by adding a cache of recently freed IOVA ranges that can satisfy most allocations without accessing the tree [35]. IOVA allocation time in Linux/EiovaR is thus comparable, if not superior, to other OSes. However, IOVA allocation remains a sequential bottleneck with EiovaR as well (Figure 2), since the EiovaR cache is accessed under the IOVA allocator lock.

3.2 Linux IOTLB Invalidation

Destroying an IOMMU mapping requires invalidating the IOTLB entry caching the mapping, both for correctness and for security. For correctness, if the unmapped IOVA gets subsequently remapped to a different physical address, the IOMMU will keep using the old translation and misdirect any DMAs to this IOVA. Security-wise, destroying a mapping indicates the device should no longer have access to the associated physical memory. If the translation remains present in the IOTLB, the device can still access the memory.

Unfortunately, waiting for the invalidation to complete prior to returning control from IOVA unmapping code is prohibitively expensive (§ 6). In addition to the added latency of waiting for the invalidation to complete, issuing the invalidation command requires writing to the IOMMU invalidation queue—an operation that must be serialized and thus quickly becomes a bottleneck.

As a result, Linux does not implement this *strict* invalidation policy by default. Instead, it implements a *deferred* invalidation policy that amortizes the cost of IOTLB invalidation across multiple unmappings. Here, an unmap operation buffers an invalidation request for its IOVA in a global *flush queue* data structure and returns without waiting for the invalidation to complete. Periodically (every 10 ms) or after batching 250 invalidation requests, Linux performs a single global IOTLB invalidation that empties the entire IOTLB (possibly flushing valid entries as well). Once the global invalidation com-

[2] We refer the reader to [35] for the exact details of the heuristic, which are irrelevant for our purpose.

pletes, the IOVA allocator is invoked to deallocate each IOVA range buffered in the flush queue.

Thus, deferred invalidation maintains correctness, but trades off some security—creating a window of time in which a device can access unmapped IOVAs—in exchange for performance. In practice, unmapped physical pages rarely get reused immediately upon returning from the unmap function. For example, a driver may unmap multiple pages—possibly triggering a global invalidation—before returning control to the system. Thus, deferred invalidation appears to be a pragmatic trade-off, and other OSes use similar mechanisms (§ 3.4).

While deferred invalidation amortizes the latency of waiting for invalidations to complete, the flush queue is protected by a single (per IOMMU) *invalidation lock*. As with the IOVA allocation lock, this is a non-scalable design that creates a bottleneck—21.7% of the cycles are spent waiting for the invalidation lock in our experiment.

Masking the IOVA allocator bottleneck Interestingly, the IOTLB invalidation bottleneck throttles the rate of IOVA allocation/deallocation operations, and thereby *masks* the severity of the IOVA allocator bottleneck. Deallocations are throttled because they occur while processing the flush queue—i.e., under the invalidation lock—and are therefore serialized. Allocations (mappings) are throttled because they are interleaved with unmappings. Since unmapping is slow because of the IOTLB bottleneck, the interval between mappings increases and their frequency decreases.

Once we eliminate the IOTLB invalidation bottleneck, however, pressure on the IOVA allocation lock increases and with it the severity of its performance impact. Indeed, as Figure 2 shows, adding scalable deferred IOTLB invalidation (§ 5) to Linux/EiovaR increases IOVA lock waiting time by 2.1×.

3.3 Linux Page Table Management

IOMMU page table management involves two tasks: updating the page tables when creating/destroying mappings, and reclaiming physical pages that are used as page tables when they become empty.

3.3.1 Updating Page Tables

We distinguish between updates to *last-level* page tables (leafs in the page table tree) and *page directories* (inner nodes).

For last-level page tables, the Linux IOVA allocator enables synchronization-free updates. Because each mapping is associated with a unique IOVA page range, updates of distinct mappings involve distinct page table entries. Further, the OS does not allow concurrent mapping/unmapping of the same IOVA range. Consequently, it is safe to update entries in last-level page tables without locking or atomic operations.

Updating page directories is more complex, since each page directory entry (PDE) may map multiple IOVA ranges (any range potentially mapped by a child page table), and multiple cores may be concurrently mapping/unmapping these ranges. A new child page table is allocated by updating an empty PDE to point to a new page table. To synchronize this action in a parallel scenario, we allocate a physical page P and then attempt to point the PDE to P using an atomic operation. This attempt may fail if another core has pointed the PDE to its own page table in the mean time, but in this case we simply free P and use the page table installed by the other core. Deallocation of page tables is more complex, and is discussed in the following section.

The bottom line, however, is that updating page tables is relatively cheap. Page directories are updated infrequently and these updates rarely experience conflicts. Similarly, last-level page table updates are cheap and conflict free. As a result, page table updates account for about 2.8% of the cycles in the system (Figure 2).

3.3.2 Page Table Reclamation

To reclaim a page table, we must first be able to remove any reference (PDE) to it. This requires some kind of synchronization to atomically (1) determine that the page table is empty, (2) remove the pointer from the parent PDE to it, (3) prevent other cores from creating new entries in the page table in the mean time. In addition, because the IOTLB could cache PDEs [27], we can only reclaim the physical page that served as the now-empty page table after invalidating the IOTLB. Before that, it is not safe to reclaim this memory or to map any other IOVA in the range controlled by it.

Due to this complexity, the Linux design sidesteps this issue and does not reclaim page table memory, unless the entire region covered by a PDE is freed in one unmap action. Thus, once a PDE is set to point to some page P, it is unlikely to ever change, which in turn reduces the number of updates that need to be performed for PDEs. This simple implementation choice is largely enabled by the IOVA allocation policy of packing IOVA ranges close to the top of the address space. This policy results in requiring a minimal number of page tables to map the allocated IOVA ranges, which makes memory consumption by IOMMU page tables tolerable.

3.4 IOMMU Management in Other OSes

This section compares the Linux/Intel-x86 IOMMU management design to the designs used in the FreeBSD, Solaris[3], and Mac OS X systems. Table 1 summarizes our findings, which are detailed below. In a nutshell, we find that (1) all OSes have scalability bottlenecks sim-

[3] Our source code references are to illumos, a fork of OpenSolaris. However, the code in question dates back to OpenSolaris.

	IOVA allocation		**IOTLB invalidation**		**PT management**	
	Allocator	Scales?	Strict?	Scales?	Scales?	Free mem?
Linux/Intel-x86 (§§ 3.1–3.3)	Red-black tree: linear time (made constant by EiovaR).	✗	✗	✗	✓	Rarely
FreeBSD [20]	Red-black tree: logarithmic time	✗	✓	✗	✗	Yes
Solaris [21]	Vmem [11]: constant time	✗	✗	✗	✗	No
Mac OS X [4]	Buddy allocator/red-black tree (size dependent): logarithmic time	✗	✗	✗	✗	No

Table 1. Comparison of IOMMU management designs in current OSes

ilar to—or worse than—Linux, (2) none of the OSes other than FreeBSD reclaim IOMMU page tables, (3) FreeBSD is the only OS to implement strict IOTLB invalidation. The other OSes loosen their intra-OS protection guarantees for increased performance. The last observation supports our choice of optimizing the Linux deferred invalidation design in this work. Our scalable IOMMU management designs (or simple variants thereof) are thus applicable to these OSes as well.

IOVA allocation All systems use a central globally-locked allocator, which is invoked by each IOVA allocation/deallocation operation, and is thus a bottleneck. The underlying allocator in FreeBSD is a red-black tree of allocated ranges, similarly to Linux. However, FreeBSD uses a different traversal policy, which usually finds a range in logarithmic time [35]. Solaris uses the Vmem resource allocator [11], which allocates in constant time. Mac OS X uses two allocators, both logarithmic—a buddy allocator for small ($\leq$ 512 MB) ranges and a red/black tree allocator for larger ranges.

IOTLB invalidation FreeBSD is the only OS that implements strict IOTLB invalidations, i.e., waits until the IOTLB is invalidated before completing an IOVA unmap operation. The other OSes defer invalidations, although differently than Linux: Solaris does not invalidate the IOTLB when unmapping. Instead, it invalidates the IOTLB when mapping an IOVA range, to flush any previous stale mapping. This creates an unbounded window of time in which a device can still access unmapped memory. An unmap on Mac OS X buffers an IOTLB invalidation request in the cyclic IOMMU invalidation queue and returns without waiting for the invalidation to complete. All these designs acquire the lock protecting the IOMMU invalidation queue for each operation, and thus do not scale.

Page table management Linux has the most scalable IOMMU page table management scheme—exploiting IOVA range disjointness to update last-level PTEs without locks and inner PDEs with atomic operations. In contrast, FreeBSD performs page table manipulations under a global lock. Solaris uses a more fine-grained technique, protecting each page table with a read/write lock. However, the root page table lock is acquired by every operation and thus becomes a bottleneck, since even acquisitions of a read/write lock in read mode create contention on the lock's shared cache line [33]. Finally, Mac OS X updates page tables under a global lock when using the buddy allocator, but outside of the lock—similarly to Linux—when allocating from the red-black tree.

Page table reclamation Similarly to Linux, Solaris does not detect when a page table becomes empty and thus does not attempt to reclaim physical pages that serve as page tables. Mac OS X bounds the number of IOMMU page tables (and therefore the size of I/O virtual memory supported) and allocates them on first use while holding the IOVA allocator lock. Mac OS X does not reclaim page tables as well. Only FreeBSD actively manages IOMMU page table memory; it maintains a count of used PTEs in each page table, and frees the page table when it becomes empty.

4 Scalable IOVA Allocation

Here we describe three designs for scalable IOVA assignment, exploring several points in this design space: (1) dynamic identity mapping (§ 4.1), which does away with IOVA allocation altogether, (2) IOVA-kmalloc (§ 4.2) which implements IOVA allocation but does away with its dedicated allocator, and (3) scalable IOVA allocation (§ 4.3), which uses *magazines* [11] to alleviate contention on the IOVA allocator using per-core caching of freed IOVA ranges.

4.1 Dynamic Identity Mapping

The fastest code is code that never runs. Our *dynamic identity mapping* design applies this principle to IOVA allocation. We observe that ordinarily, the buffers that a driver wishes to map are already physically contiguous. We thus propose to use such a physically contiguous buffer's physical address as its IOVA when mapping it, resulting in an identity (1-to-1) mapping in the IOMMU. Due to intra-OS protection reasons, when the driver unmaps a buffer we must destroy its identity mapping. We therefore refer to this scheme as *dynamic* identity mappings—while the same buffer will always use the same mapping, this mapping is dynamically created and destroyed to enforce protection of the buffer's memory.

Dynamic identity mapping eliminates the work and locking involved in managing a distinct space of IOVAs, replacing it with the work of translating a buffer's kernel virtual address to a physical address. In most OSes, this is an efficient and scalable operation—e.g., adding a constant to the kernel virtual address. However, while dynamic identity mapping completely eliminates the IOVA allocation bottleneck, it turns out to have broader implications for other parts of the IOMMU management subsystem, which we now discuss.

Page table entry reuse Standard IOVA allocation associates each mapping with a distinct IOVA. As a result, multiple mappings of the same page (e.g., for different buffers on the same page) involve different page table entries (§ 3.3). Dynamic identity mapping breaks this property: a driver—or concurrent cores—mapping a previously mapped page will need to update exactly the same page table entries (PTEs).

While repeated mapping operations will always write the same value (i.e., the physical address of the mapped page), unmapping operations pose a challenge. We need to detect when the last unmapping occurs, so we can clear the PTE. This requires maintaining a reference count for each PTE. We use 10 of the OS-reserved bits in the last-level PTE structure [27] to maintain this reference count. When the reference count hits zero, we clear the PTE and request an IOTLB invalidation.

Because multiple cores may concurrently update the same PTE, all PTE updates—including reference count maintenance—require atomic operations. In other words, we need to (1) read the PTE, (2) compute the new PTE value, (3) update it with an atomic operation that verifies the PTE has not changed in the mean time, and (4) repeat this process if the atomic operation fails.

Conflicting access permissions A second problem posed by not having unique PTEs for each mapping is how to handle mappings of the same physical page with conflicting access permission (read, write, or both). For example, two buffers may get allocated by the network stack on the same page—one for RX use, which the NIC should write to, and one for TX, which the NIC should only read. To maintain intra-OS protection, we must separately track mappings with different permissions, e.g., so that once all writable mappings of a page are destroyed, no PTE with write permissions remains. Furthermore, even when a mapping with write permissions exists, we want to avoid promoting PTEs that should only be used to read (and vice-versa), as this kind of access should not happen during the normal course operation for a properly functioning device and should be detected and blocked.

To solve this problem, we exploit the fact that current x86 processors support 48-bit I/O virtual memory addresses, but only 46-bits of physical memory addresses. The two most significant bits in each IOVA are thus "spare" bits, which we can use to differentiate mappings with conflicting access permissions. Effectively, we partition the identity mapping space into regions: three regions, for read, write and read/write mappings, and a fourth fallback region that contains addresses that cannot be mapped with identity mappings—as discussed next.

Non-contiguous buffers Some mapping operations are for physically non-contiguous buffers. For example, Linux's scatter/gather mapping functions map nonconsecutive physical memory into contiguous virtual I/O memory. Since identity mappings do not address this use case, we fall back to using the IOVA allocator in such cases. To avoid conflicts with the identity mappings, IOVAs returned by the IOVA allocator are used in the fourth fallback region.

PTE reference count overflow We fall back to standard IOVA allocation for mappings in which the 10-bit reference count overflows. That is, if we encounter a PTE whose reference count cannot be incremented while creating a dynamic identity mapping, we abort (decrementing the references of any PTEs previously incremented in the mapping process) and create the mapping using an IOVA obtained from the IOVA allocator.

Page table memory Because physical addresses mapped by drivers are basically arbitrary, we lose the property that IOVAs are densely packed. Consequently, more memory may be required to hold page tables. For example, if the first 512 map operations are each for a single page, IOVA allocation will map them all through the same last-level page table. With physical addresses, we may need at least a page table per map operation. Unfortunately, the physical pages used to hold these page tables will not get reclaimed (§ 3.3). In the worst case, page table memory consumption may keep increasing as the system remains active.

4.2 IOVA-kmalloc

Our IOVA-kmalloc design explores the implications of treating IOVA allocation as a general allocation problem. Essentially, to allocate an IOVA range of size R, we can allocate a block of R bytes in physical memory using the system's `kmalloc` allocator, and use the block's address as an IOVA (our actual design is much less wasteful, as discussed below). The addresses `kmalloc` returns are unique per allocation, and thus IOVA-kmalloc maintains the efficient conflict-free updates of IOMMU page tables enabled by the original IOVA allocator.

One crucial difference between kmalloc and IOVA allocation is that IOVAs are abstract—essentially, opaque integers—whereas kmalloc allocates physically contiguous memory. It might therefore seem that the IOVA-kmalloc approach unacceptably wastes memory, since allocating R bytes to obtain an R-byte IOVA range doubles the system's memory consumption. Fortunately, we observe that the IOVA allocator need only allocate virtual I/O *page frame numbers* (PFNs), and not arbitrary ranges. That is, given a physical buffer to map, we need to find a range of *pages* that contains this buffer. This makes the problem tractable: we can treat the physical addresses that kmalloc returns as PFNs. That is, to map a range of R bytes, we kmalloc $\lceil R/4096 \rceil$ bytes and interpret the address of the returned block B as a range of PFNs (i.e., covering the IOVA range of $[4096B, 4096(B + \lceil R/4096 \rceil))$.

While this still wastes physical memory, the overhead is only 1 byte per virtual I/O page, rounded up to 8 bytes (the smallest unit kmalloc allocates internally). By comparison, the last-level PTE required to map a page is itself 8 bytes, so the memory blowup caused by IOVA-kmalloc allocating actual physical memory is never greater than the memory overhead incurred by stock Linux for page table management.

In fact, being a full-blown memory allocator that manages actual memory (not abstract integers) might actually turn out to be advantageous for kmalloc, as this property enables it to use the many techniques and optimizations in the memory allocation literature. In particular, kmalloc implements a version of *slab allocation* [10], a fast and space-efficient allocation scheme. One aspect of this technique is that kmalloc stores the slab that contains metadata associated with an allocated address in the page structure of the address. This allows kmalloc to look up metadata in constant time when an address gets freed. In contrast, the Linux IOVA allocator has to maintain metadata *externally*, in the red-black tree, because there is no physical memory backing an IOVA. It must thus access the globally-locked tree on each deallocation.

Page table memory blowup The main downside of IOVA-kmalloc is that we have no control over the allocated addresses. Since kmalloc makes no effort to pack allocations densely, the number of page tables required to hold all the mappings will be larger than with the Linux IOVA allocator. Moreover, if the same physical address is mapped, unmapped, and then mapped again, IOVA-kmalloc may use a different IOVA when remapping. Because Linux does not reclaim page table memory, the amount of memory dedicated to page tables can grow without bound. In contrast, dynamic identity mapping always allocates the same IOVA to a given physical buffer. However, in an OS that manages page table memory, unbounded blowup cannot occur.

To summarize, IOVA-kmalloc represents a classic *time/space* trade-off—we relinquish memory in exchange for the efficiency and scalability of kmalloc, which is highly optimized due to its pervasive use throughout the kernel. Notably, these advantages come *for free*, in terms of implementation complexity, debugging and maintenance, since kmalloc is already there, performs well, and is trivial to use.

Handling IOVA collisions IOVAs are presently 48 bits wide [27]. x86 hardware presently supports 46-bits of physical address space [26]. Thus, because we treat kmalloc addresses as virtual I/O PFNs, IOVA-kmalloc may allocate two addresses that collide when interpreted as PFNs in the 48-bit I/O virtual address space. That is, we have 2^{36} possible IOVA PFNs since pages are 4 KB, but kmalloc allocates from a pool of up to 2^{46} bytes of physical memory.

Such collisions are mathematically impossible on systems with at most 64 GB of physical memory (whose physical addresses are 36-bit). To avoid these collisions on larger systems, IOVA allocations can invoke kmalloc with the GFP_DMA flag. This flag instructs kmalloc to satisfy the allocation from a low memory zone whose size we can configure to be at most 64 GB.[4]

Why allocate addresses? We could simply use a mapped buffer's kernel virtual address as its IOVA PFN. However, we then lose the guarantee that different mappings obtain different IOVA PFNs (e.g., as the same buffer can be mapped twice). This is exactly the problem dynamic identity mapping tackles, only here we do not have "spare" bits to distinguish mappings with different access rights as the virtual address space is 48 bits.

4.3 Scalable IOVA Allocation with Magazines

Our final design addresses the IOVA allocator bottleneck head-on, turning it into a scalable allocator. The basic idea is to add a *per-core cache* of previously deallocated IOVA ranges. If most allocations can be satisfied from the per-core cache, the actual allocator—with its lock—will be invoked rarely.

Per-core caching poses several requirements. First, the per-core caches should be *bounded*. Second, they should efficiently handle producer/consumer scenarios observed in practice, in which one core continuously allocates IOVAs, which are later freed by another core. In a trivial design, only the core freeing IOVAs will build up a cache, while the allocating core will always invoke the under-

[4] In theory, the GFP_DMA memory zone must be accessible to legacy devices for DMA and thus addressable with 24 bits. But as we have an IOMMU, we only need to ensure that the IOVA ranges mapped to legacy devices fall into the 24-bit zone.

lying non-scalable allocator. We require that *both* cores avoid interacting with the IOVA allocator.

Magazines [11] provide a suitable solution. A *magazine* is an M-element per-core cache of objects—IOVA ranges, in our case—maintained as a stack of objects. Conceptually, a core trying to allocate from an empty magazine (or deallocate into a full magazine) can return its magazine to a globally-locked *depot* in exchange for a full (or empty) magazine. Magazines actually employ a more sophisticated *replenishment policy* which guarantees that a core can satisfy at least M allocations and at least M deallocations from its per-core cache before it must access the depot. Thus, a core's miss rate is bounded by $1/M$.

We implement magazines on top of the original Linux IOVA allocator, maintaining separate magazines and depots for different allocation sizes. Thus, this design still controls page table blowup, as the underlying allocator densely packs allocated IOVAs.

5 Scalable IOTLB Invalidation

This section describes a scalable implementation of deferred IOTLB invalidations. Recall that Linux maintains a *flush queue* containing a globally ordered list of all IOVA ranges pending invalidation. We observe, however, that a global flush queue—with its associated lock contention—is overkill. There is no real dependency between the invalidation process of distinct IOVA ranges. Our only requirements are that until an entire IOVA range mapping is invalidated in the IOTLB:

1. The IOVA range will not be released back to the IOVA allocator, to avoid having it allocated again before the old mapping is invalidated.

2. The page tables that were mapping the IOVA range will not be reclaimed. Since page directory entries are also cached in the IOTLB, such a reclaimed page table might get reused and data that appears as a valid page table entry be written to it, and this data might then be used by the IOMMU.

Our approach We satisfy these requirements in a much more scalable manner by batching invalidation requests *locally* on each core instead of globally. Implementing this approach simply requires replicating the current flush queue algorithm on a per-core basis. With this design, the lock on a core's flush queue is almost always acquired by the owning core. The only exception is when the queue's global invalidation timeout expires—the resulting callback, which acquires the lock, may be scheduled on a different core. However, not only does such an event occur rarely (at most once every 10 ms), but it suggests that the IOMMU management subsystem is not heavily used in the first place.

The remaining source of contention in this design is the serialization of writes to the cyclic IOMMU invalidation queue—which is protected by a lock—in order to buffer global IOTLB invalidation requests. Fortunately, accesses to the invalidation queue are infrequent, with at most one invalidation every 250 invalidations or 10 ms, and short, as an invalidation request is added to the buffer and the lock is released; the core waits for the IOMMU to process its queued invalidation without holding the lock.

Security-wise, while we now might batch 250 invalidation requests per core, a destroyed IOVA range mapping will usually be invalidated in the IOTLB just as quickly as before. The reason is that *some* core performs a global IOTLB invalidation, on average, every 250 global invalidation requests. Thus, we do not substantially increase the window of time in which a device can access an unmapped physical page. Unmapped IOVA ranges may, however, remain in a core's flush queue and will not be returned to the IOVA allocator for a longer time than the baseline design—they will be freed only when the core itself processes its local flush queue. We did not observe this to be a problem in practice, especially when the system experiences high IOMMU management load.

6 Evaluation

Here we evaluate our designs (§§ 4–5) and explore the trade-offs they involve. We study two metrics: the performance obtained (§ 6.1), and the complexity of implementing the designs (§ 6.2).

6.1 Performance

Experimental setup We implement the designs in Linux 3.17.2. Our test setup consists of a client and a server, both Dell PowerEdge R430 rack machines. Each machine has dual 2.4 GHz Intel Xeon E5-2630 v3 8-core processors, for a total of 16 cores per machine (hyperthreading is disabled). Both machines are equipped with 32 GB 2133 MHz memory. The server is equipped with a Broadcom NetXtreme II BCM57810 10 Gb/s NIC. The client is equipped with an Intel 82599 10 Gb/s NIC and runs an unmodified Ubuntu 3.13.0-45 Linux kernel. The client's IOMMU is disabled for all evaluations. The client and the server NICs are cross-connected to avoid network interference.

Methodology To obtain the best scalability, we (1) configure the NIC on the server to use 15 rings, which is the maximum number supported by the Broadcom NIC, allowing cores to interact with the NIC with minimal lock contention (no ring contention with < 16 cores), and (2) distribute delivery of interrupts associated with different rings across different cores.[5] Benchmarks are

[5] Default delivery is to core #0, which overloads this core.

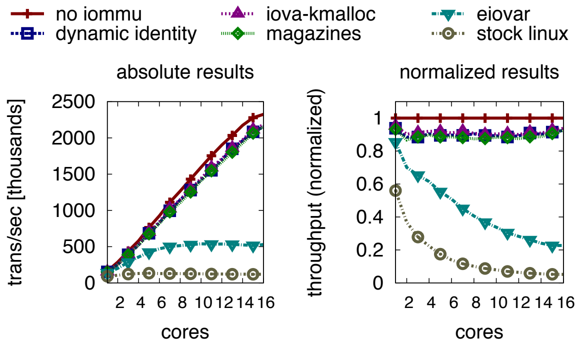

Figure 3. Highly parallel RR throughput

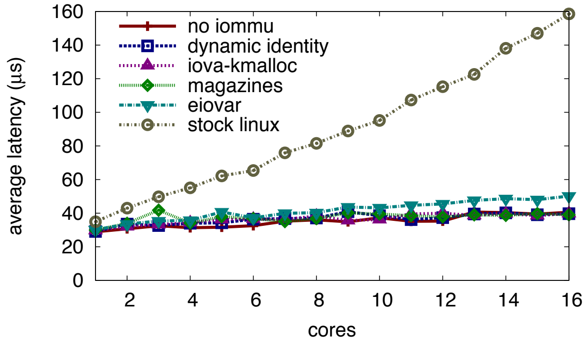

Figure 4. 1 netperf TCP RR instance per core latency test

executed in a round-robin fashion, with each benchmark running once for 10 seconds for each possible number of cores, followed by all benchmarks repeating. The cycle is run five times and the reported results are the medians of the five runs.

Benchmarks In our benchmarks, we aim to use concurrent workloads that stress the IOMMU management subsystem.

Our first few benchmarks are based on `netperf` [28], a prominent network analysis tool. In our highly parallel RR (request-response) benchmark, we run 270 instances of the `netperf` TCP RR test on the client, and limit the server side of `netperf` to the number of cores we wish to test on. Each TCP RR instance sends a TCP packet with 1 byte of payload to the server and waits for a 1 byte response before sending the next one. In all, our benchmark has 270 ongoing requests to the `netperf` server at any given time, which bring the server close to 100% CPU utilization even with IOMMU disabled. We report the total number of such request-response transactions the server responds to during the testing period.

The second benchmark we use `netperf` for is a parallel latency test. To achieve that, we run the `netperf` TCP RR test with as many instances as we allow server cores. This allows each `netperf` request-response connection to run on its own dedicated core on both the client and the server.

Our third benchmark is `memcached` [19], a key-value store service used by web applications to speed up read operations on slow resources such as databases and remote API calls. To avoid lock contention within `memcached`, we run multiple instances, each pinned to run on a specific core. We measure `memcached` throughput using `memslap` [1], which distributes requests among the `memcached` instances. We configure `memslap` to run on the client with 16 threads and 256 concurrent requests (16 per thread). We use `memslap`'s default configuration of a 64-byte key, 1024-byte value and a ratio of 10%/90% `SET`/`GET` operations.

We note that network bandwidth benchmarks would be less relevant here, as a single core can saturate the network on our machines. As an example, `netperf` TCP STREAM, which makes use of the NIC's offloading features, is able to saturate the network using 22% CPU time on a single core even running under EiovaR-Linux.

Results Figure 3 depicts the throughput achieved by our highly parallel RR benchmark. Without an IOMMU, Linux scales nicely and obtains 14.4× TPS with 16 cores than with a single core. Because of the IOMMU management bottlenecks, EiovaR-Linux does not scale as well, obtaining only a 3.8× speedup. In particular, while EiovaR obtains 86% of the No-IOMMU throughput with 1 core (due to the single-threaded overheads of IOMMU management), it only achieves 23% of the No-IOMMU throughput at 16 cores.

Our designs all perform at 93%–95% of No-IOMMU on a single core and scale far better than EiovaR, with 16-core throughput being 93% (magazines and dynamic identity mapping) and 94% (IOVA-kmalloc) of the No-IOMMU results. Because of the overhead dynamic identity mapping adds to page table updates, it does not outperform the IOVA allocating designs—despite not performing IOVA allocation at all.

To summarize, in our designs IOMMU overhead is essentially *constant* and does not get worse with more concurrency. Most of this overhead consists of page tables updates, which are essential when managing the IOMMU in a transient manner.

In our parallel latency test, shown in Figure 4, we see that Linux's latency increases with multiple cores, even without IOMMU, from 29μs with one instance to 41μs with 16 instances, each with its own core. While EiovaR starts within 1.2μs of Linux's latency on one core, the contention caused by its locks causes a 10μs gap at 16 cores. For the most part, our designs are on par with Linux's performance, actually achieving slightly shorter latency than No-IOMMU Linux on 16 cores.

The evaluation of `memcached` in Figure 5 paints a similar picture to Figure 3 with up to 12 cores. The IOMMU subsystem is indifferent to the different packet size in this workload (1052 bytes here, 1 byte in TCP RR) as the

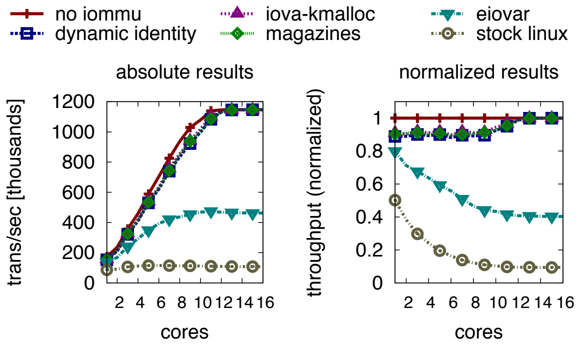

Figure 5. `memcached` throughput

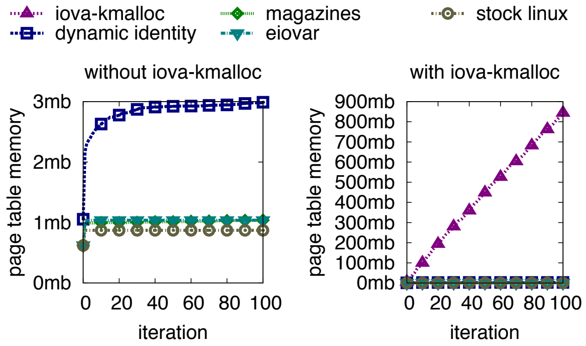

Figure 6. Page table memory over time

mappings are done per page and no data copying takes place. Starting at 12 cores, all of our designs achieve line rate and therefore close the gap with No-IOMMU performance. Here, too, IOVA-kmalloc demonstrates the best performance out of our designs by a small margin (in all but 2 cores and > 12 cores, where they all achieve line rate), as it is the most highly optimized of the three. With a single core, all of our designs are between 89% (dynamic identity) and 91% (IOVA-kmalloc) of No-IOMMU performance. At 9 cores, after which No-IOMMU Linux throughput begins to near line rate, our designs' relative throughput is between 89.5% (dynamic identity) and 91.5% (IOVA-kmalloc). EiovaR also stops scaling well before 16 cores, but as opposed to our designs, it does not do that due to achieving line rate, plateauing at 40% of it, starting with 9 cores.

Page table memory consumption Linux rarely reclaims a page used as an IOMMU page table (§ 3.3). Figure 6 illustrates the memory consumption dynamics this policy creates. We run 100 iterations of our parallel RR benchmark and plot the number of IOMMU page tables (in all levels) that exist in the system before the tests start (but after system and NIC initialization) and after each iteration.

Both EiovaR and our magazine-based design, which are based on Linux's current IOVA allocator, minimize page table memory consumption by packing allocated IOVAs at the top of the address space. As Figure 6 shows,

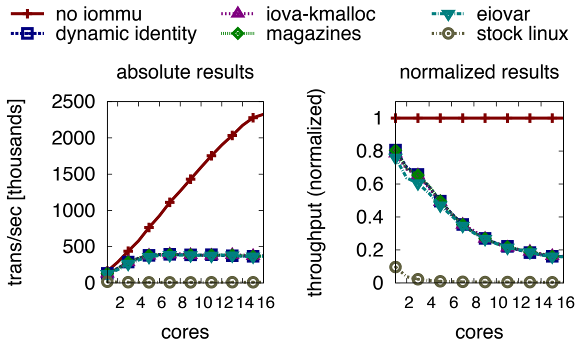

Figure 7. Highly parallel `netperf` txns/sec with strict IOTLB invalidation

both of them, as well as stock Linux, allocate most page tables at NIC initialization, when all RX rings are allocated and mapped, as well as all TX/RX descriptors and other metadata.

Our magazine-based design consumes 1.2× more page table memory than stock Linux, because the cores cache freed IOVAs in their per-core caches instead of returning them to the global allocator, requiring other cores to allocate new IOVAs from the allocator and thus increasing page table use.

While dynamic identity mapping does not guarantee an upper bound on the number of IOVAs it utilizes over time, it turns out that the APIs used by the driver to allocate buffers use caching themselves. Consequently, mapped addresses repeat across the iterations and the number of page tables does not explode. Still, 3.4× more page tables than stock Linux are allocated, since the addresses are not packed.

In contrast to the other schemes, IOVA-kmalloc exhibits a blowup of page table memory use. Since the addresses IOVA-kmalloc receives from `kmalloc` are influenced by system-wide allocation activity, IOVA-kmalloc uses a much wider and constantly changing set of IOVAs. This causes its page table memory consumption to grow in an almost linear fashion. We conclude that while IOVA-kmalloc is the best performing design, by a slight margin, its use must be accompanied by memory reclamation of unused page tables. This underscores the need for managing the IOMMU page table memory in a scalable manner—§ 3.3 describes the challenges involved.

Strict invalidation Figure 7 shows the parallel RR benchmark throughput with *strict* IOTLB invalidation, i.e., full intra-OS protection. As invalidations are not deferred, our scalable IOTLB invalidation optimization is irrelevant for this case. While we observe very minor throughput difference between the IOVA allocation designs when using a small number of cores, even this difference is no longer evident when more than 6 cores are used, with all designs obtaining about $1/6$ the TPS

Design	Lines		Files	Impl. time
	add	del		
Dynamic identity mapping (§ 4.1)	+397	-92	1	Weeks
IOVA-kmalloc (§ 4.2)	+56	-44	1	Hours
IOVA allocation with magazines (§ 4.3)	+362	-79	3	Days
Scalable IOTLB invalidation (§ 5)	+97	-37	1	Hours

Table 2. Implementation complexity of our designs

of No-IOMMU Linux by 16 cores. In all designs, the contention over the invalidation queue lock becomes the dominating factor (§ 3.2). Thus, strict IOTLB invalidation appears incompatible with a scalable implementation.

6.2 Implementation Complexity

Table 2 reports the source code changes required to implement our designs in Linux 3.17.2. The table also estimates the time it would take us to re-implement each approach from scratch. IOVA-kmalloc is the simplest design to implement, essentially replacing calls to the IOVA allocator with a `kmalloc()` call. Most of the line changes reported for IOVA-kmalloc are due to technical changes, replacing the `struct` representing an IOVA with an integer type. Implementation of the magazines design is a bit more complex, requiring an implementation of a magazine-based caching layer on top of the existing IOVA allocator, as well as optimizing its locking strategy. Dynamic identity mapping is the most complicated and intrusive of our IOVA assignment designs, as it calls for a surgical modification of page table management itself, maintaining mapping book-keeping within the table itself and synchronizing updates to the same mapping from multiple cores in parallel.

7 Related Work

Most work addressing the poor performance associated with using IOMMUs tackles only sequential performance [3, 9, 14, 34, 42, 45, 47, 35], for example by reducing the number of mapping operations [45], deferring or offloading IOTLB invalidation work [3], and improving the IOVA allocator algorithm [14, 35, 42]. Cascardo [14] does consider multicore workloads, but his proposal improves only the sequential part of the IOVA allocator. In contrast, our work identifies and attacks the scalability bottlenecks in current IOMMU management designs. In addition, our scalable IOVA allocation is orthogonal to sequential improvements in the IOVA allocator, since it treats it as a black box.

Clements et al. propose designs for scalable management of process virtual address spaces [16, 17]. I/O virtual memory has simpler semantics than standard virtual memory, which allows us to explore simpler designs. In particular, (1) IOVA ranges cannot be partially unmapped, unlike standard `mmap()`ed ranges, and (2) IOVA mappings can exploit the preexisting address of the mapped buffers (as in dynamic identity mapping), while creation of a virtual memory region can occur before the allocation of the physical memory backing it.

Bonwick introduced magazines to improve scalability in the Vmem resource allocator [11]. Since Vmem is a general allocator, however, it does not minimize page table memory consumption of allocated IOVAs, in contrast to the specialized Linux allocator. Our IOVA-kmalloc design additionally shows that a dedicated resource allocator may not be required in the first place. Finally, part of our contribution is the re-evaluation of magazines on modern machines and workloads.

8 Conclusion

Today, IOMMU-based intra-OS protection faces a performance challenge in high throughput and highly concurrent I/O workloads. In current OSes, the IOMMU management subsystem is not scalable and creates a prohibitive bottleneck.

Towards addressing this problem, we have explored the design space of scalable IOMMU management approaches. We proposed three designs for scalable IOVA assignment—dynamic identity mapping, IOVA-kmalloc, and per-core IOVA caching—as well as a scalable IOTLB invalidation scheme. Our designs achieve 88.5%–100% of the performance obtained *without* an IOMMU.

Our evaluation demonstrates the trade-offs of the different designs. Namely, (1) the savings dynamic identity mapping obtains from not allocating IOVAs are negated by its more expensive IOMMU page table management, making it perform comparably to scalable IOVA allocation, and (2) IOVA-kmalloc provides a simple solution with high performance, but suffers from unbounded page table blowup. This emphasizes the importance of managing the IOMMU *page table memory* in a scalable manner as well, which is interesting future work.

References

[1] AKER, B. Memslap - load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memslap.html`. libmemcached 1.1.0 documentation.

[2] AMD INC. AMD IOMMU architectural specification, rev 2.00. `http://developer.amd.com/wordpress/media/2012/10/488821.pdf`, Mar 2011.

[3] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)* (2011), pp. 73–86.

[4] APPLE COMPUTER. IOPCIFamily/IOPCIFamily-196.3/vtd.c, Source Code File of Mac OS X. `http:`

//www.opensource.apple.com/source/IOPCIFamily/IOPCIFamily-196.3/vtd.c. (Accessed: Jan 2015).

[5] 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).

[6] 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.

[7] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *ACM EuroSys* (2006), pp. 73–85.

[8] Becher, M., Dornseif, M., and Klein, C. N. FireWire: all your memory are belong to us. In *CanSecWest Applied Security Conference* (2005).

[9] Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., and van Doorn, L. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)* (2007), pp. 9–20.

[10] Bonwick, J. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Annual Technical Conference* (1994), pp. 87–98.

[11] Bonwick, J., and Adams, J. Magazines and Vmem: Extending the Slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference (ATC)* (2001), pp. 15–44.

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

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

[14] Cascardo, T. DMA API performance and contention on IOMMU enabled environments. http://events.linuxfoundation.org/images/stories/slides/lfcs2013_cascardo.pdf, 2013.

[15] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 73–88.

[16] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable Address Spaces Using RCU Balanced Trees. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 199–210.

[17] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *ACM EuroSys* (2013), pp. 211–224.

[18] Criswell, J., Geoffray, N., and Adve, V. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium* (2009), pp. 83–100.

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

[20] FreeBSD Foundation. x86/iommu/intel_gas.c, source code file of FreeBSD 10.1.0. https://github.com/freebsd/freebsd/blob/release/10.1.0/sys/x86/iommu/intel_gas.c. (Accessed: Jan 2015).

[21] Garrett D'Amore. i86pc/io/immu_dvma.c, Source Code File of illumos. https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/i86pc/io/immu_dvma.c. (Accessed: Jan 2015).

[22] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)* (2007), pp. 41–50.

[23] Hill, B. Integrating an EDK custom peripheral with a LocalLink interface into Linux. Tech. Rep. XAPP1129 (v1.0), XILINX, May 2009.

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

[25] IBM Corporation. AIX kernel extensions and device support programming concepts. http://public.dhe.ibm.com/systems/power/docs/aix/71/kernextc_pdf.pdf, 2013. (Accssed: May 2015).

[26] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide. http://download.intel.com/products/processor/manual/325384.pdf, 2013.

[27] Intel Corporation. Intel Virtualization Technology for Directed I/O, Architecture Specification - Architecture Specification - Rev. 2.3. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Oct 2014.

[28] Jones, R. A. A network performance benchmark (revision 2.0). Tech. rep., Hewlett Packard, 1995. http://www.netperf.org/netperf/training/Netperf.html.

[29] Kadav, A., Renzelmann, M. J., and Swift, M. M. Tolerating hardware device failures in software. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 59–72.

[30] Leitao, B. H. Tuning 10Gb network cards on Linux. In *Ottawa Linux Symposium (OLS)* (2009), pp. 169–189.

[31] LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 17–30.

[32] Documentation/intel-iommu.txt, Linux 3.18 documentation file. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt. (Accessed: Jan 2015).

[33] Liu, R., Zhang, H., and Chen, H. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *USENIX Annual Technical Conference (ATC)* (2014), pp. 219–230.

[34] Malka, M., Amit, N., Ben-Yehuda, M., and Tsafrir, D. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015), pp. 355–368.

[35] Malka, M., Amit, N., and Tsafrir, D. Efficient Intra-Operating System Protection Against Harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 29–44.

[36] Mamtani, V. 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).

[37] Miller, D. S., Henderson, R., and Jelinek, J. Dynamic DMA mapping guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. Linux kernel documentation.

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

[39] ROEDEL, J. IOMMU Page Faulting and MM Integration. http://www.linuxplumbersconf.org/2014/ocw/system/presentations/2253/original/iommuv2.pdf. Linux Plumbers Conference 2014.

[40] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 335–350.

[41] SWIFT, M., BERSHAD, B., AND LEVY, H. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS) 23*, 1 (Feb 2005), 77–110.

[42] TOMONORI, F. DMA representations sg_table vs. sg_ring IOMMUs and LLD's restrictions. In *USENIX Linux Storage and Filesystem Workshop (LSF)* (2008). https://www.usenix.org/legacy/events/lsf08/tech/IO_tomonori.pdf.

[43] WALDSPURGER, C., AND ROSENBLUM, M. I/O virtualization. *Communications of the ACM (CACM) 55*, 1 (Jan 2012), 66–73.

[44] WILLIAMS, D., REYNOLDS, P., WALSH, K., SIRER, E. G., AND SCHNEIDER, F. B. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (2008), pp. 241–254.

[45] WILLMANN, P., RIXNER, S., AND COX, A. L. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)* (2008), pp. 15–28.

[46] WOJTCZUK, R. 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).

[47] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)* (2010), pp. 18:1–18:12.