Efficient Intra-Operating System Protection Against Harmful DMAs

Moshe Malka Nadav Amit Dan Tsafrir
Technion – Israel Institute of Technology

Abstract

Operating systems can defend themselves against misbehaving I/O devices and drivers by employing intra-OS protection. With "strict" intra-OS protection, the OS uses the IOMMU to map each DMA buffer immediately before the DMA occurs and to unmap it immediately after. Strict protection is costly due to IOMMU-related hardware overheads, motivating "deferred" intra-OS protection, which trades off some safety for performance.

We investigate the Linux intra-OS protection mapping layer and discover that hardware overheads are not exclusively to blame for its high cost. Rather, the cost is amplified by the I/O virtual address (IOVA) allocator, which regularly induces linear complexity. We find that the nature of IOVA allocation requests is inherently simple and constrained due to the manner by which I/O devices are used, allowing us to deliver constant time complexity with a compact, easy-to-implement optimization. Our optimization improves the throughput of standard benchmarks by up to 5.5x. It delivers strict protection with performance comparable to that of the baseline deferred protection.

To generalize our case that OSes drive the IOMMU with suboptimal software, we additionally investigate the FreeBSD mapping layer and obtain similar findings.

1 Introduction

The role that the I/O memory management unit (IOMMU) plays for I/O devices is similar to the role that the regular memory management unit (MMU) plays for processes. Processes typically access the memory using virtual addresses translated to physical addresses by the MMU. Likewise, I/O devices commonly access the memory via direct memory access operations (DMAs) associated with I/O virtual addresses (IOVAs), which are translated to physical addresses by the IOMMU. Both hardware units are implemented similarly with a page table hierarchy that the operating system (OS) maintains and the hardware walks upon an (IO)TLB miss.

The IOMMU can provide *inter*- and *intra-OS protection* [4, 44, 54, 57, 59]. Inter 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 [27, 42]. In this mode, the VM directly programs device DMAs 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 protection, IOVAs are mapped to physical memory locations infrequently, only upon such events as VM creation and migration, and host management operations such as memory swapping, deduplication, and NUMA migration. Such mappings are therefore denoted *persistent* or *static* [57].

Intra-OS protection allows the OS to defend against errant/malicious devices and buggy drivers, which account for most OS failures [19, 49]. Drivers/devices can initiate/perform DMAs to arbitrary memory locations, and IOMMUs allow OSes to protect themselves by restricting these DMAs to specific physical memory locations. Intra-OS protection is applicable in: (1) non-virtual setups where the OS has direct control over the IOMMU, and in (2) virtual setups where IOMMU functionality is exposed to VMs via paravirtualization [12, 42, 48, 57], full emulation [4], or, recently, hardware support for nested IOMMU translation [2, 36]. In this mode, IOVA (un)mappings are frequent and occur within the I/O critical path. The OS programs DMAs using IOVAs rather than physical addresses, such that each DMA is preceded and followed by the mapping and unmapping of the associated IOVA to the physical address it represents [38, 46]. For this reason, such mappings are denoted *single-use* or *dynamic* [16]. The context of this paper is intra-OS protection (§2).

To do its job, the intra-OS protection mapping layer must allocate IOVA values: integer ranges that serve as page identifiers. IOVA allocation is similar to regular memory allocation. But it is different enough to merit its own allocator (§3). One key difference is that regular allocators dedicate much effort to preserving locality and to combating fragmentation, whereas the IOVA allocator disallows locality and enjoys a naturally "unfragmented" workload. This difference makes the IOVA allocator 1–2 orders of magnitude smaller in terms of lines of code.

Another difference is that, by default, the IOVA subsystem trades off some safety for performance. It delays the completion of IOVA deallocations while letting the OS believe that the deallocations have been processed. Specifically, freeing an IOVA implies purging it from the IOTLB such that the associated physical buffer is no longer accessible to the I/O device. But invalidating IOTLB entries is a costly, slow operation. So the IOVA subsystem opts for batching the invalidations until enough accumulate and then invalidating all the IOTLB en masse, thus reducing the amortized price. This default mode is called *deferred protection*. Users can turn it off at boot time by instructing the kernel to use *strict protection*.

The activity that stresses the IOVA mapping layer is associated with I/O devices that employ *ring buffers* in order to communicate with their OS drivers in a producer-consumer manner. A ring is a cyclic array whose entries correspond to DMA requests that the driver initiates and the device fulfills. Ring entries contain IOVAs that the mapping layer allocates/frees before/after the associated DMAs are processed by the device. We carefully analyze the performance of the IOVA mapping layer and find that its allocation scheme is efficient despite its simplicity, but only if the device is associated with a single ring (§4).

Devices, however, often employ more rings, in which case our analysis indicates that the IOVA allocator seriously degrades the performance (§5). We study this deficiency and find that its root cause is a pathology we call *long-lasting ring interference*. The pathology occurs when I/O asynchrony prompts an event that causes the allocator to migrate an IOVA from one ring to another, henceforth repetitively destroying the contiguity of the ring's I/O space upon which the allocator relies for efficiency. We conjecture that this harmful effect remained hidden thus far because of the well-known slowness associated with manipulating the IOMMU. The hardware took most of the blame for the high price of intra-OS protection even though software is equally guilty, as it turns out, in both OSes that we checked (Linux and FreeBSD).

We address the problem by adding the Efficient IOVA allocatoR (EIOVAR) optimization to the kernel's mapping subsystem (§6). EIOVAR exploits the fact that its workload is (1) exclusively comprised of power-of-two allocations, and is (2) ring-induced, so the difference D between the cumulative number of allocation and deallocation requests at any given time is proportional to the ring size, which is relatively small. EIOVAR is accordingly a simple, thin layer on top of the baseline IOVA allocator that proxies all (de)allocations. It caches all freed ranges and reuses them to quickly satisfy subsequent allocations. It is successful because the requests are similar. It is frugal with memory because D is small. And it is compact (implementation-wise) because it consists of an array of freelists with a bit of minimal logic. EIOVAR entirely eliminates the baseline allocator's aforementioned reliance on I/O space contiguity, ensuring all (de)allocations are efficient.

We evaluate the performance of EIOVAR using different I/O devices (§7). On average, EIOVAR satisfies (de)allocations in about 100 cycles. It improves the throughput of Netperf, Apache, and Memcached benchmarks by up to 5.50x and 1.71x for strict and deferred protection, respectively, and it reduces the CPU consumption by up to 0.53x. Interestingly, EIOVAR delivers strict protection with performance that is similar to that of the baseline system when employing deferred protection.

Accelerating allocation (of IOVAs in our case) using freelists is a well-known technique commonly utilized by memory allocators [13, 14, 15, 29, 40, 53, 55] (§8). Our additional contributions are: identifying that the performance of the IOMMU mapping layer can be dramatically improved by employing this technique across the OSes we tested and thus refuting the common wisdom that the poor performance is largely due to the hardware slowness; carefully studying the IOMMU mapping layer workload; finding that it is very "well behaved"; which ensures that even our simplistic EIOVAR freelist provides fast, constant-time IOVA allocation while remaining compact in size (§9).

2 Intra-OS Protection

DMA refers to the ability of I/O devices to read from or write to the main memory without CPU involvement. It is a heavily used mechanism, as it frees the CPU to continue to do work between the time it programs the DMA until the time the associated data is sent or received. As noted, drivers of devices that stress the IOVA mapping layer initiate DMA operations via a *ring buffer*, which is a circular array in main memory that constitutes a shared data structure between the driver and its device. Each entry in the ring contains a DMA *descriptor*, specifying the address(es) and size(s) of the corresponding *target buffer(s)*; the I/O device will write/read the data to/from the latter, at which point it will trigger an interrupt to let the OS know that the DMA has completed. (Interrupts are coalesced if their rate is high.) I/O device are commonly associated with more than one ring, e.g., a receive ring denoted *Rx* for DMA read operations, and a transmit ring denoted *Tx* for DMA write operations.

In the past, I/O devices used physical addresses in order to access main memory, namely, each DMA descriptor contained a physical address of its target buffer. Such unmediated DMA activity directed at the memory makes the system vulnerable to rogue devices performing errant or malicious DMAs [9, 17, 38, 58], or to buggy drivers that might program their devices to overwrite any part of the system memory [8, 30, 42, 49, 56]. Subsequently, all major chip vendors introduced IOMMUs [2, 7, 34, 36], alleviating the problem as follows.

The OS associates each DMA target buffer with some IOVA, used instead of the physical address when filling out the associated ring descriptor. The I/O device is oblivious to the change, processing the DMA as if the IOVA was a physical memory address. The IOMMU then translates the IOVA, routing the operation to the appropriate

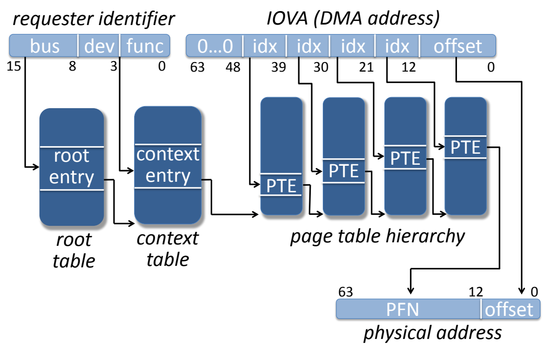

Figure 1: *IOVA translation using the Intel IOMMU.*

memory location. Figure 1 depicts the translation process of the Intel IOMMU used in this paper. The PCI protocol dictates that each DMA operation is associated with a 16 bit *request identifier* comprised of a *bus-device-function* triplet unique to the corresponding I/O device. The IOMMU uses the 8 bit bus number to index the *root table*, retrieving the physical address of the *context table*. It then indexes the latter using the device-function 8 bit concatenation, yielding the physical location of the root of the page table hierarchy that houses the device's IOVA translations. Similarly to the MMU, the IOMMU accelerates translations using an IOTLB.

The functionality of the IOMMU is equivalent to that of the regular MMU. It permits IOVA memory accesses to go through only if the OS previously inserted matching translations. The OS can thus protect itself by allowing a device to access a target buffer just before the corresponding DMA occurs (add mapping), and by revoking access just after (remove mapping), exerting fine-grained control over what portions of memory may be used in I/O transactions at any given time. This state-of-the-art strategy of IOMMU-based protection was termed *intra-OS protection* by Willmann et al. [57].It is recommended by hardware vendors [31, 38], and it is used by operating systems [6, 16, 35, 45]. For example, the DMA API of Linux—which we use in this study—notes that "DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer" [46].

3 IOVA vs. Memory Allocation

The task of generating IOVAs—namely, the actual integer numbers that the OS assigns to descriptors and the devices then use—is similar to regular memory allocation. But it is sufficiently different to merit its own allocator, because it optimizes for different objectives, and because it is required to make different tradeoffs, as follows.

Locality Memory allocators spend much effort in trying to (re)allocate memory chunks in a way that maximizes reuse of TLB entries and cached content. The IOVA mapping layer of the OS does the opposite. The numbers it allocates correspond to whole pages, and they are not allowed to stay warm in hardware caches in between allocations. Rather, they must be purged from the IOTLB and from the page table hierarchy immediately after the DMA completes. Moreover, while purging an IOVA, the mapping layer must flush each cache line that it modifies in the hierarchy, as the IOMMU and CPU do not reside in the same coherence domain.[1]

Fragmentation Memory allocators invest much effort in combating fragmentation, attempting to eliminate unused memory "holes" and utilize the memory they have before requesting the system for more. As we further discuss in §5–§6, it is trivial for the IOVA mapping layer to avoid fragmentation due to the simple workload that it services, which exclusively consists of requests whose size is a power of two number of pages. The IOMMU driver rounds up all IOVA range requests to 2^j for two reasons. First, because IOTLB invalidation of 2^j ranges is faster [36, 39]. And second, because the allocated IOVA range does not correspond to 2^j pages of real memory. Rather it merely corresponds to to a pair of integers marking the beginning and end of the range. Namely, the IOMMU driver maps only the physical pages it was given, but it reserves a bigger IOVA range so as to make the subsequent associated IOTLB invalidation speedier. It can thus afford to be "wasteful". (In our experiments, the value of j was overwhelmingly 0. Namely, the allocated IOVA ranges almost always consist of one page only.)

Complexity Simplicity and compactness matter and are valued within the kernel. Not having to worry about locality and fragmentation while enjoying a simple workload, the mapping layer allocation scheme is significantly simpler than regular memory allocators. In Linux, it is comprised of only a few hundred lines of codes instead of thousands [40, 41] or tens of thousands [13, 32].

Safety & Performance Assume a thread T_0 frees a memory chunk M, and then another thread T_1 allocates memory. A memory allocator may give M to T_1, but only after it processes the free of T_0. Namely, it would never allow T_0 and T_1 to use M together. Conversely, the IOVA mapping layer purposely allows T_0 (the device) and T_1 (the OS) to access M simultaneously for a short period of time. The reason: invalidation of IOTLB entries is costly [4, 57]. Therefore, by default, the mapping layer trades off safety for performance by (1) accumulating up to W unprocessed 'free' operations and only then (2) freeing those W IOVAs and (3) invalidating the entire IOTLB en masse. Consequently, target buffers are actively being used by the OS while the device might still access them through

[1]Intel IOMMU specification documents a capability bit that indicates whether the IOMMU and CPU coherence could be turned on [36], but we do not own such hardware and believe it is not yet common.

stale IOTLB entries. This weakened safety mode is called *deferred protection*. Users can instead employ *strict protection*—which processes invalidations immediately—by setting a kernel command line parameter.

Metadata Memory allocators typically use the memory that their clients (de)allocate to store their metadata. For example, by inlining the size of an allocated area just before the pointer that is returned to the client. Or by using linked lists of free objects whose "next" pointers are kept within the areas the comprise the lists. The IOVA mapping layer cannot do that, because the IOVAs that it invents are pointers to memory that is used by some other entity (the device or the OS). An IOVA is just an *additional* identifier for a page, which the mapping layer does not own.

Pointer Values Memory allocators running on 64-bit machines typically use native 64-bit pointers. The IOVA mapping layer prefers to use 32-bit IOVAs, as utilizing 64-bit addresses for DMA would force a slower, dual address cycle on the PCI bus [16].

4 Supposed O(1) Complexity of Baseline

In accordance to §3, the allocation scheme employed by the Linux/x86 IOVA mapping layer is different than, and independent of, the regular kernel memory allocation subsystem. The underlying data structure of the IOVA allocator is the generic Linux kernel red-black tree. The elements of the tree are *ranges*. A range is a pair of integer numbers $[L,H]$ that represent a sequence of currently allocated I/O virtual page numbers $L, L+1, ..., H-1, H$, such that $L \leq H$ stand for "low" and "high", respectively. Ranges are pairwise disjoint, namely, given two ranges $[L_1,H_1] \neq [L_2,H_2]$, then either $H_1 < L_2$ or $H_2 < L_1$.

Newly requested IOVA integers are allocated by scanning the tree right-to-left from the highest possible value downwards towards zero in search for a gap that can accommodate the requested range size. The allocation scheme attempts and—as we will later see—ordinarily succeeds to allocate the new range from within the highest gap available in the tree.

The allocator begins to scan the tree from a *cache node* that it maintains, denoted C. The allocator iterates from C through the ranges in a descending manner until a suitable gap is found. C is maintained such that it usually points to a range that is higher than (to the right of) the highest free gap, as follows. When (1) a range R is freed and C currently points to a range lower than R, then C is updated to point to R's successor. And (2) when a new range Q is allocated, then C is updated to point to Q; if Q was the highest free gap prior to its allocation, then C still points higher than the highest free gap after this allocation.

Figure 2 lists the pseudo code of the IOVA allocation

```
struct range_t {int lo, hi;};

range_t alloc_iova(rbtree_t t, int rngsiz) {

  range_t  new_range;
  rbnode_t right = t.cache;
  rbnode_t left  = rb_prev( right );

  while(right.range.lo - left.range.hi <= rngsiz)
      right = left;
      left  = rb_prev( left );

  new_range.hi = right.lo - 1;
  new_range.lo = right.lo - rngsiz;
  t.cache      = rb_insert( t, new_range );

  return new_range;
}

void free_iova(rbtree_t t, rbnode_t d) {
  if( d.range.lo >= t.cache.range.lo )
      t.cache = rb_next( d );
  rb_erase( t, d );
}
```

Figure 2: *Pseudo code of the baseline IOVA allocation scheme. The functions* rb_next *and* rb_prev *return the successor and predecessor of the node they receive, respectively.*

scheme as was just described. Clearly, the algorithm's worst case complexity is linear due to the 'while' loop that scans previously allocated ranges beginning at the cache node C. But when factoring in the actual workload that this algorithm services, the situation is not so bleak: the complexity turns out to actually be constant rather than linear (at least conceptually).

Recall that the workload is commonly induced by a circular ring buffer, whereby IOVAs of DMA target buffers are allocated and freed in a repeated, cyclic manner. Consider, for example, an Ethernet NIC with a Rx ring of size n, ready to receive packets. Assume the NIC initially allocates n target buffers, each big enough to hold one packet (1500 bytes). The NIC then maps the buffers to n newly allocated, consecutive IOVAs with which it populates the ring descriptors. Assume that the IOVAs are $n, n-1, ..., 2, 1$. (The series is descending as IOVAs are allocated from highest to lowest.) The first mapped IOVA is n, so the NIC stores the first received packet in the memory pointed to by n, and it triggers an interrupt to let the OS know that it needs to handle the packet.

Upon handling the interrupt, the OS first unmaps the corresponding IOVA, purging it from the IOTLB and IOMMU page table to prevent the device from accessing the associated target buffer (assuming strict protection). The unmap frees IOVA=n, thus updating C to point to n's successor in the red-black tree (free_iova in Figure 2).

The OS then immediately re-arms the ring descriptor for future packets, allocating a new target buffer and associating it with a newly allocated IOVA. The latter will be n, and it will be allocated in constant time, as C points to n's immediate successor (alloc_iova in Figure 2). The same scenario will cyclically repeat itself for $n-1, n-2, \ldots, 1$ and then again $n, \ldots, 1$ and so on as long as the NIC is operational.

Our soon to be described experiments across multiple devices and workloads indicate that the above description is fairly accurate. IOVA allocations requests are overwhelmingly for one page ranges ($H = L$), and the freed IOVAs are indeed re-allocated shortly after being freed, enabling, in principle, the allocator in Figure 2 to operate in constant time as described. But the algorithm succeeds to operate in this ideal manner only for some bounded time. We find that, inevitably, an event occurs and ruins this ideality thereafter.

5 Long-Lasting Ring Interference

The above $O(1)$ algorithm description assumes there exists only one ring in the I/O virtual address space. In reality, however, there are often two or more, for example, the Rx and Tx receive and transmit rings. Nonetheless, even when servicing multiple rings, the IOVA allocator provides constant time allocation in many cases, so long as each ring's free_iova is immediately followed by a matching alloc_iova for the same ring (the common case). Allocating for one ring and then another indeed causes linear IOVA searches due to how the cache node C is maintained. But large bursts of I/O activity flowing in one direction still enjoy constant allocation time.

The aforementioned event that forever eliminates the allocator's ability to accommodate large I/O bursts with constant time occurs when a free-allocate pair of one ring is interleaved with that of another. Then, an IOVA from one ring is mapped to another, ruining the contiguity of the ring's I/O virtual address. Henceforth, every cycle of n allocations would involve one linear search prompted whenever the noncontiguous IOVA is freed and reallocated. We call this pathology *long-lasting ring interference* and note that its harmful effect increases as additional inter-ring free-allocate interleavings occur.

Table 1 illustrates the pathology. Assume that a server mostly receives data and occasionally transmits. Suppose that Rx activity triggers a Rx.free_iova(L) of address L **(1)**. Typically, this action would be followed by Rx.alloc_iova, which would then return L **(2)**. But sometimes a Tx operation sneaks in between. If this Tx operation is Tx.free_iova(H) such that $H > L$ **(3)**, then the allocator would update the cache node C to point to H's successor **(4)**. The next Rx.alloc_iova would be satisfied by H **(5)**, but then the subsequent Rx.alloc_iova would have to iterate through the tree from H **(6)** to L **(7)**, inducing a linear overhead. Notably, once H is mapped for Rx, the pathology is repeated every time H is (de)allocated. This repetitiveness is experimentally demonstrated in Figure 3, showing the per-allocation number of rb_prev invocations. The calls are invoked in the loop in alloc_iova while searching for a free IOVA.

operation	without Tx			with Tx		
	return value	C before	C after	return value	C before	C after
Rx.free(L=151) **(1)**		152	152		152	152
Tx.free(H=300) **(3)**					152	**(4)** 301
Rx.alloc	**(2)** 151	152	151	**(5)** 300	301	300
Rx.free(150)		151	151		300	**(6)** 300
Rx.alloc	150	151	150	**(7)** 151	300	151

Table 1: *Illustrating why Rx-Tx interferences cause linearity, following the baseline allocation algorithm detailed in Figure 2. (Assume that all addresses are initially allocated.)*

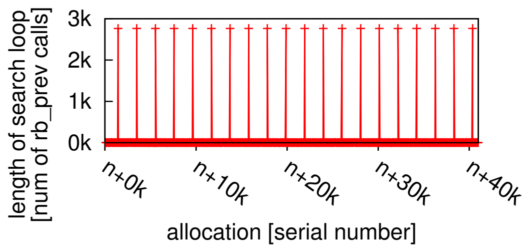

Figure 3: *The length of each alloc_iova search loop in a 40K (sub)sequence of alloc_iova calls performed by one Netperf run. One Rx-Tx interference leads to regular linearity.*

We show below that the implications of long-lasting ring interference can be dreadful in terms of performance. How, then, is it possible that such a deficiency is overlooked? We contend that the reason is twofold. The first is that commodity I/O devices were slow enough in the past such that IOVA allocation linearity did not matter. The second reason is the fact that using the IOMMU hardware is slow and incurs a high price, motivating the deferred protection safety/performance tradeoff. Being that slow, the hardware served as a scapegoat, wrongfully held accountable for most of the overhead penalty and masking the fact that software is equally to blame.

6 The EIOVAR Optimization

Suffering from frequent linear allocations, the baseline IOVA allocator is ill-suited for high-throughput I/O devices that are capable of performing millions of I/O transactions per second. It is too slow. One could proclaim that this is just another case of a special-purpose allocator proved inferior to a general-purpose allocator and argue that the latter should be favored over the former despite the notable differences between the two as listed in §4. We contend, however, that the simple, repetitive, and inherently ring-induced nature of the workload can be adequately served by the existing simplistic allocator—with only a small, minor change—such that the modified version is able to consistently support fast (de)allocations.

We propose the EIOVAR optimization (Efficient IOVA allocatoR), which rests of the following observation. I/O devices that stress the intra-OS protection mapping layer are not like processes, in that the size of their virtual address spaces is relatively small, inherently bounded by the size of their rings. A typical ring size n is a few hundreds or a few thousands of entries. The number of per-device virtual page addresses that the IOVA allocator must simultaneously support is proportional to the ring size, which means it is likewise bounded and relatively small. Moreover, unlike "regular" memory allocators, the IOVA mapping layer does not allocate real memory pages. Rather, it allocates integer identifiers for those pages. Thus, it is reasonable to keep O(n) of these identifiers alive under the hood for quick (de)allocation, without really (de)allocating them (in the traditional, malloc sense of (de)allocation).

In numerous experiments with multiple devices and workloads, the maximal number of per-device different IOVAs we have observed is 12K. More relevant is that, across all experiments, the maximal number of previously-allocated-but-now-free IOVAs has never exceeded 668 (and was 155 on average). Additionally, as noted earlier, the allocated IOVA ranges have a power of two size $H-L+1=2^j$, where j is overwhelmingly 0. EIOVAR leverages these workload characteristic to efficiently cache freed IOVAs so as to satisfy future allocations quickly, similarly to what regular memory allocators do when allocating real memory [13, 14, 15, 29, 40, 53, 55].

EIOVAR is a thin layer that masks the red-black tree, resorting to using it only when EIOVAR cannot fulfill IOVA allocation on its own using previously freed elements. When configured to have enough capacity, all tree allocations that EIOVAR is unable to mask are assured to be fast and occur in constant time.

EIOVAR's main data structure is a one-dimensional array called "the freelist", or f for short. The array consists of M linked lists of IOVA ranges. Lists are empty upon initialization. When an IOVA range $[L,H]$ whose size is $H-L+1=2^j$ is freed, instead of actually freeing it, EIOVAR adds it to the head of the linked list of the corresponding exponent, namely, to $f[j]$. Because most ranges are comprised of one page ($H=L$), most ranges end up in the $f[0]$ list after they are freed. The upper bound on the size of the ranges supported by EIOVAR is 2^{M+12} bytes (assuming 2^{12} = 4KB pages), as EIOVAR allocates page numbers. Thus, $M=28$ is enough, allowing for up to a terabyte range.

EIOVAR allocation performs the reverse operation of freeing. When a range whose exponent is j is being allocated, EIOVAR removes the head of the $f[j]$ linked list in order to satisfy the allocation request. EIOVAR resorts to utilizing the baseline red-black tree only if a suitable range is not found in the freelist.

When no limit is imposed on the freelist, after a very short while, all EIOVAR (de)allocation operations are satisfied by f due to the inherently limited size of the ring-induced workload. All freelist (de)allocations are performed in constant time, taking 50-150 cycles per operation. Initial allocations that EIOVAR satisfies by resorting to the baseline tree are likewise done in constant time, because the freelist is limitless and so the tree never observes deallocations, which means its cache node C always points to its smallest, leftmost node (Figure 2).

We would like to make sure that the freelist is compact and is not effectively leaking memory. To bound the size of the freelist, EIOVAR has a parameter k that serves as f's maximal capacity of freed IOVAs. We use the EIOVAR$_k$ notation to express this limit, with $k=\infty$ indicating no upper bound. We demonstrate that setting k to be a relatively small number is equivalent to setting it to ∞, because the number of previously-allocated-but-now-free IOVAs is constrained by the size of the corresponding ring. Consequently, we can be certain that the freelist of EIOVAR$_\infty$ is a compact. At the same time, $k=\infty$ guarantees that (de)allocations are always satisfied in constant time.

6.1 EIOVAR with Strict Protection

To understand the behavior and effect of EIOVAR, we begin by analyzing five EIOVAR$_k$ variants as compared to the baseline under strict protection, where IOVAs are (de)allocated immediately before and after the associated DMAs. We use the standard Netperf stream benchmark that maximizes throughput on one TCP connection. We initially restart the NIC interface for each allocation variant (thus clearing IOVA structures), and then we execute the benchmark iteratively. The exact experimental setup is described in §7. The results are shown in Figure 4.

Figure 4a shows that the throughput of all EIOVAR variants is similar and is 20%–60% better than the baseline. The baseline gradually decreases except for the last iteration. Figure 4b highlights why even EIOVAR$_1$ is sufficient to provide the observed benefit. It plots the rate of IOVA allocations that are satisfied by the freelist, showing that $k=1$ is enough to satisfy nearly all allocations. This result indicates that each call to free_iova is followed by alloc_iova, such that the IOVA freed by the former is returned by the latter, coinciding with the ideal scenario outlined in §4. Figure 4c supports this observation by depicting the average size of the freelist. The average of EIOVAR$_1$ is inevitably 0.5, as every allocation and deallocation contributes to the average 1 and 0 respectively. Larger k values are similar, with an average of 2.5 be-

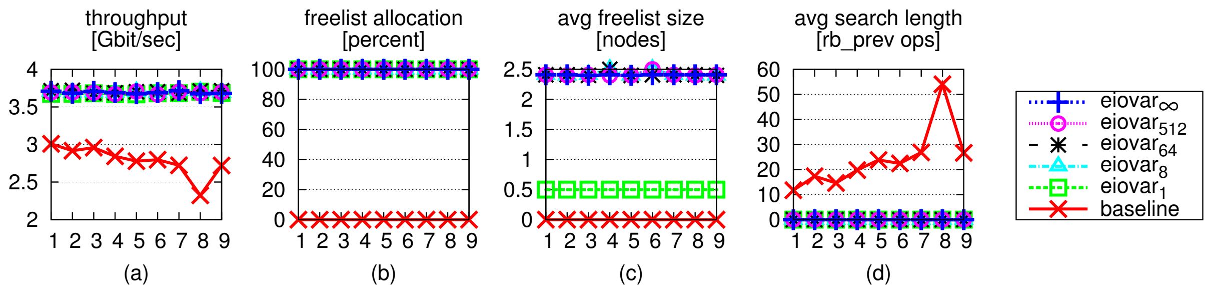

Figure 4: *Netperf TCP stream iteratively executed under strict protection. The x axis shows the iteration number.*

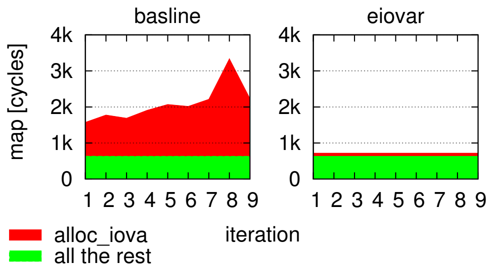

Figure 5: *Cycles breakdown of map with Netperf/strict.*

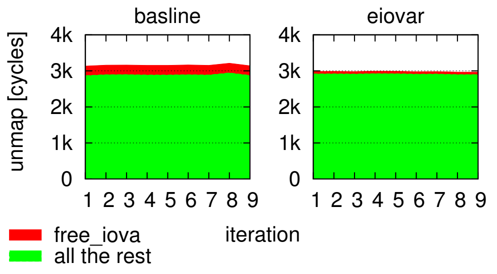

Figure 6: *Cycles breakdown of unmap with Netperf/strict.*

cause of two additional (de)allocation that are performed when Netperf starts running and that remain in the freelist thereafter. Figure 4d shows the average length of the 'while' loop from Figure 2, which searches for the next free IOVA. It depicts a rough mirror image of Figure 4a, indicating throughput is tightly negatively correlated with the traversal length.

Figure 5 (left) shows the time it takes the baseline to map an IOVA, separating allocation from the other activities. Whereas the latter remains constant, the former exhibits a trend identical to Figure 4d. Conversely, the alloc_iova time of EIOVAR (Figure 5, right) is negligible across the board. EIOVAR is immune to long-lasting ring interface, as interfering transactions are absorbed by the freelist and reused in constant time.

6.2 EIOVAR with Deferred Protection

Figure 6 is similar to Figure 5, but it pertains to the unmap operation rather than to map. It shows that the duration of free_iova remains stable across iterations with both EIOVAR and the baseline. EIOVAR deallocation is still faster as it is performed in constant time whereas the baseline is logarithmic. But most of the overhead is not due to free_iova. Rather, it is due to the costly invalidation that purges the IOVA from the IOTLB to protect the corresponding target buffer. This is the aforementioned hardware overhead that motivated deferred protection, which amortizes the cost by delaying invalidations until enough IOVAs are accumulated and then processing all of them together. As noted, deferring the invalidations trades off safety for performance, because the relevant memory is accessible by the device even though it is already used by the kernel for other purposes.

Figure 7 compares between the baseline and the EIOVAR variants under deferred protection. Interestingly, the resulting throughput divides the variants into two, with EIOVAR_{512} and EIOVAR_{∞} above 6Gbps and all the rest at around 4Gbps (Figure 7a). We again observe a strong negative correlation between the throughput and the length of the search to find the next free IOVA (Figure 7a vs. 7d).

In contrast to the strict setup (Figure 4), here we see that EIOVAR variants with smaller k values roughly perform as bad as the baseline. This finding is somewhat surprising, because, e.g., 25% of the allocations of EIOVAR_{64} are satisfied by the freelist (Figure 7b), which should presumably improve its performance over the baseline. A finding that helps explain this result is noticing that the average size of the EIOVAR_{64} freelist is 32 (Figure 7c), even though it is allowed to hold up to $k = 64$ elements. Notice that EIOVAR_{∞} holds around 128 elements on average, so we know there are enough deallocations to fully populate the EIOVAR_{64} freelist. One might therefore expect that the latter would be fully utilized, but it is not.

The average size of the EIOVAR_{64} freelist is 50% of its capacity due to the following reason. Deferred invalidations are aggregated until a high-water mark W (kernel parameter) is reached, and then all the W addresses are deallocated in bulk.[2] When $k < W$, the freelist fills up to

[2]They cannot be freed before they are purged from the IOTLB, or else they could be re-allocated, which would be a bug since their stale mappings might reside in the IOTLB and point to somewhere else.

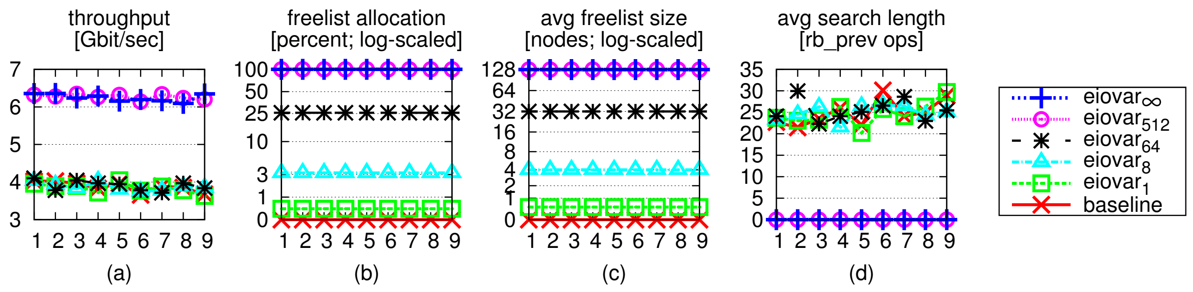

Figure 7: *Netperf TCP stream iteratively executed under deferred protection. The x axis shows the iteration number.*

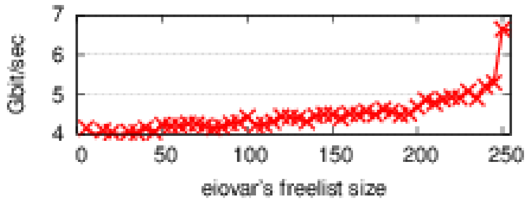

Figure 8: *Under deferred protection,* EIOVAR$_k$ *eliminates costly linear searches when k exceeds the high-water mark W.*

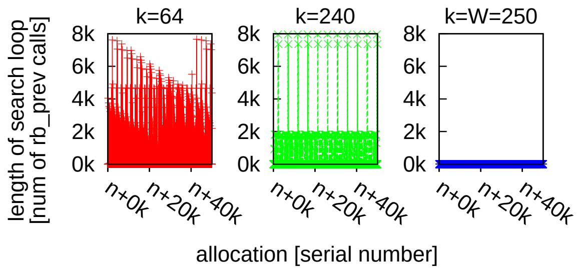

Figure 9: *Length of the* alloc_iova *search loop under the* EIOVAR$_k$ *deferred protection regime for three k values when running Netperf TCP Stream. Bigger capacity implies that the searches become shorter on average. Big enough capacity ($k \geq W = 250$) eliminates the searches altogether.*

hold k elements, which become $k-1$ after the subsequent allocation, and then $k-2$ after the next allocation, and so on until zero is reached, yielding an average size of $\frac{1}{k+1}\Sigma_{j=0}^{k} j \approx k/2$ as our measurements show.

Importantly, when $k < W$, EIOVAR$_k$ is unable to absorb all the W consecutive deallocations. The remaining $W-k$ deallocations are thus freed by the baseline free_iova. Thus, only k of the W subsequent allocation are satisfied by the freelist, and the remaining $W-k$ are serviced by the baseline alloc_iova. The baseline free_iova and alloc_iova are therefore regularly invoked in an uncoordinated way despite the freelist. As described in §5, the interplay between these two routines eventually causes long-lasting ring interference that induces repeated linear searches. In contrast, when k is big enough ($\geq W$), the freelist has sufficient capacity to absorb all W deallocations, which are then used to satisfy the subsequent W allocations and thus secure the conditions for preventing the harmful effect.

Figure 8 demonstrates this threshold behavior, depicting the throughput as a function of the maximal freelist size k. Increasingly bigger k slowly improves performance, as more—but not yet all—allocations are served by the freelist. When k reaches $W = 250$, the freelist is finally big enough, and the throughput suddenly increases by 26%. Figure 9 provides further insight into this result. It shows the per-allocation length of the loop within alloc_iova that iterates through the red-black tree in search for the next free IOVA (similarly to Figure 3). The subgraphs correspond to 3 points from Figure 8 with k values 64, 240, and 250. We see that the smaller k (left) yields longer searches relative to the bigger k (middle), and that the length of the search becomes zero when $k = W$ (right).

7 Evaluation

Experimental Setup We implement EIOVAR in the Linux kernel, and we experimentally compare its performance against the baseline IOVA allocation. In an effort to attain more general results, we conducted the evaluation using two setups involving two different NICs with two corresponding different device drivers that generate different workloads for the IOVA allocation layer.

The *Mellanox setup* consists of two identical Dell PowerEdge R210 II Rack Servers that communicate through Mellanox ConnectX3 40Gbps NICs. The NICs are connected back to back configured to use Ethernet. One machine is the server and the other is a workload generator client. Each machine has 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 we 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 two machines run Ubuntu 12.04 and utilize the Linux 3.4.64 kernel.

The *Broadcom setup* is similar, with the difference that: the two R210 machines communicate through Broadcom NetXtreme II BCM57810 10GbE NICs (connected via a

CAT7 10GBASE-T cable for fast Ethernet); have 16GB memory; and run the Linux 3.11.0 kernel.

The drivers of the Mellanox and Broadcom NICs differ in many respects. Notably, the Mellanox driver uses more ring buffers and allocates more IOVAs (we observed around 12K addresses for Mellanox and 3K for Broadcom). In particular, the Mellanox driver uses two buffers per packet and hence two IOVAs, whereas the Broadcom driver allocates only one buffer and thus only one IOVA.

Benchmarks We use the following benchmarks to drive our experiments. Netperf TCP stream [37] is a standard tool to measure networking throughput. It attempts to maximize the amount of data sent over one TCP connection, simulating an I/O-intensive workload. This is the benchmark used when studying long-lasting ring interference (§5) and the impact of *k* on EIOVAR$_k$ (§6). We use the default 16KB message size unless otherwise stated.

Netperf UDP RR (request-response) is the second canonical configuration of Netperf. It models a latency sensitive workload by repeatedly sending a single byte and waiting for a matching single byte response. The latency is then calculated as the inverse of the observed number of transactions per second.

Apache [23, 24] is a HTTP web server. We drive it with ApacheBench [5] (a.k.a. "ab"), a workload generator distributed with Apache. ApacheBench assess the number of concurrent requests per second that the server is capable of handling by requesting a static page of a given size from within several concurrent threads. We run it on the client machine configured to generate 100 concurrent requests. We use two instances of the benchmark to request a smaller (1KB) and a bigger (1MB) file. Logging is disabled to avoid disk write overheads.

Memcached [25] is an in-memory key-value storage server. It is used, e.g., by websites for caching results of slow database queries, thus improving the sites' overall performance. We run Memslap [1] (part of the libmemcached client library) on the client machine, generating requests and measuring the completion rate. By default, Memslap generates a random workload comprised of 90% get and 10% set operations. Unless otherwise stated, Memslap is set to use 16 concurrent requests.

Methodology Before running each benchmark, we shut down and bring up the interface of the NIC using the ifconfig utility, such that the IOVA allocation is redone from scratch using a clean tree, clearing the impact of previous harmful long-lasting ring interferences. We then iteratively run the benchmark 150 times, such that individual runs are configured to take about 20 seconds. We present the corresponding results, on average.

Results Figure 10 shows the resulting average performance for the Mellanox (top) and Broadcom (bottom) setups. Higher numbers indicate better throughput in all cases but for Netperf RR, which depicts latency (inverse of throughput). The corresponding normalized values—specifying relative improvement—are shown in the first part of Table 2. Here, for consistency, the normalized throughput is shown for all benchmarks including RR.

Mellanox Setup We first examine the results of the Mellanox setup (left of Table 2). In the topmost part, we see that EIOVAR yields throughput 1.07–4.58x better than the baseline, and that improvements are more pronounced under strict protection. The second part of the table shows that the improved performance of EIOVAR is due to reducing the average IOVA allocation time by 1–2 orders of magnitude, from up to 50K cycles to around 100–200. EIOVAR further reduces the average IOVA deallocation time by about 75%–85%, from around 250–550 cycles to around 65–85 (4th part of the table).

As expected, the duration of the IOVA allocation routine is tightly correlated to the length of the search loop within this routine, such that a longer loop implies a longer duration (3rd part of Table 2). Notice, however, that there is not necessarily such a direct correspondence between EIOVAR's throughput improvement (1st part of table) and the associated IOVA allocation overhead (2nd part). The reason: latency sensitive applications are less affected by the allocation overhead, because other components in their I/O paths have higher relative weights. For example, under strict protection, the latency sensitive Netperf RR has higher allocation overhead as compared to the throughput sensitive Netperf Stream (10,269 cycles vs. 7,656, respectively), yet the throughput improvement of RR is smaller (1.27x vs. 2.37x). Similarly, the IOVA allocation overhead of Apache/1KB is higher than that of Apache/1MB (49,981 cycles vs. 17,776), yet its throughput improvement is lower (2.35x vs. 3.65x).

While there is not necessarily a direct connection between throughput and allocation overheads when examining strict safety only, the connection becomes apparent when comparing strict to deferred protection. Clearly, the benefit of EIOVAR in terms of throughput is greater under strict protection because the associated baseline allocation overheads are higher than that of deferred protection (7K–50K cycles for strict vs. 2K–3K for deferred).

Broadcom Setup Let us now examine the results of the Broadcom setup (right of Table 2). Strict EIOVAR yields throughput that is 1.07–2.35x better than the baseline. Deferred EIOVAR, on the other hand, only improves the throughput by up to 10%, and, in the case of Netperf Stream and Apache/1MB, it offers no improvement. Thus, while still significant, throughput improvements in this setup are less pronounced. The reason for this difference is twofold. First, as noted above, the driver of the Mellanox NIC utilizes more rings and more IOVAs, increasing the load on the IOVA allocation layer relative to the Broad-

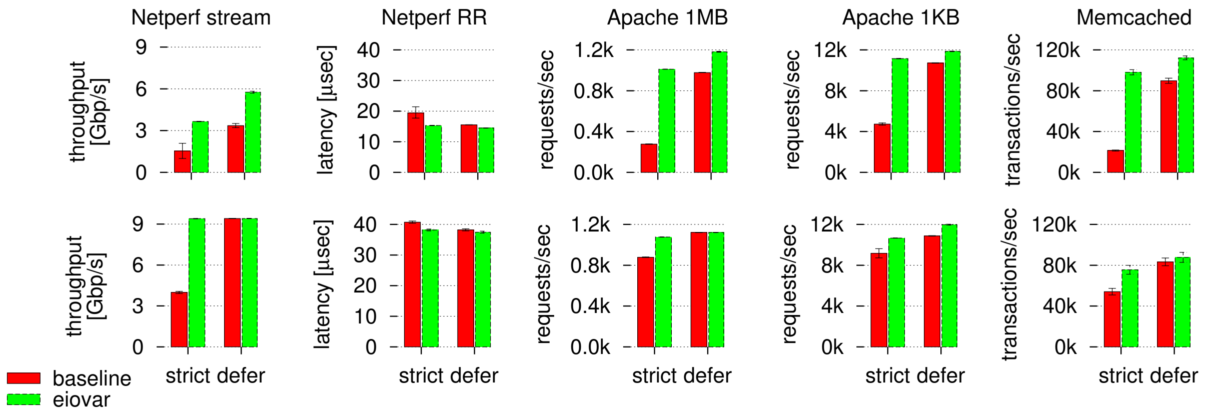

Figure 10: *The performance of baseline vs.* EIOVAR *allocation, under strict and deferred protection regimes for the Mellanox (top) and Broadcom (bottom) setups. Except for in the case of Netperf RR, higher values indicated better performance. Error bars depict the standard deviation (sometimes too small to be seen).*

com driver and generating more opportunities for ring interference. This difference is evident when comparing the duration of alloc_iova in the two setups, which is significantly lower in the Broadcom case. In particular, the average allocation time in the Mellanox setup across all benchmarks and protection regimes is about 15K cycles, whereas it is only about 3K cycles in the Broadcom setup.

The second reason for the less pronounced improvements in the Broadcom setup is that the Broadcom NIC imposes a 10 Gbps upper bound on the bandwidth, which is reached in some of the benchmarks. Specifically, the aforementioned Netperf Stream and Apache/1MB—which exhibit no throughput improvement under deferred EIOVAR—hit this limit. These benchmarks are already capable of obtaining line rate (maximal throughput) in the baseline/deferred configuration, so the lack of throughput improvement in their case should come as no surprise. Importantly, when evaluating I/O performance in a setting whereby the I/O channel is saturated, the interesting evaluation metric ceases to be throughput and becomes CPU usage. Namely, the question becomes which system is capable of achieving line rate using fewer CPU cycles. The bottom/right part of Table 2 shows that EIOVAR is indeed the more performant alternative, using 21% less CPU cycles in the case of the said Netperf Stream and Apache/1MB under deferred protection. (In the Mellanox setup, it is the CPU which is saturated in all cases but the latency sensitive Netperf RR.)

Deferred Baseline vs. Strict EIOVAR We explained above that deferred protection trades off safety to get better performance. We now note that, by Figure 10, the performance attained by EIOVAR when strict protection is employed is similar to the performance of the baseline configuration that uses deferred protection (the default in Linux). Specifically, in the Mellanox setup, on average, strict EIOVAR achieves 5% higher throughput than the deferred baseline, and in the Broadcom setup EIOVAR achieves 3% lower throughput. Namely, if strict EIOVAR is made the default, it will simultaneously deliver similar performance *and* better protection as compared to the current default configuration.

Different Message Sizes The default configuration of Netperf Stream utilizes a 16KB message size, which is big enough to optimize throughput. Our next experiment systematically explores the performance tradeoffs when utilizing smaller message sizes. Such messages can overwhelm the CPU and thus reduce the throughput. Another issue that might negatively affect the throughput of small packets is the maximal *number* of packets per second (PPS), which NICs commonly impose in conjunction with an upper bound on the throughput. (For example, the specification of our Broadcom NIC lists a maximal rate of 5.7 million PPS [33], and a rigorous experimental evaluation of this NIC reports that a single port in it is capable of delivering less than half that much [21].)

Figure 11 shows the throughput (top) and consumed CPU (bottom) as a function of message size for strict (left) and deferred safety (right) using the Netperf Stream benchmark in the Broadcom setup. With a 64B message size, the PPS limit dominates the throughput in all four configurations. Strict/baseline saturates the CPU with a message size as small as 256B; from that point on it achieves the same throughput (4Gbps), because the CPU remains its bottleneck. The other three configurations enjoy a gradually increasing throughput until line rate is reached. However, to achieve the same level of throughput, strict/EIOVAR requires more CPU than deferred/baseline, which in turn requires more CPU than deferred/EIOVAR.

Concurrency We next experiment concurrent I/O streams, as concurrency amplifies the harmful long-lasting ring interference. Figure 12 depicts the results of running

Mellanox	*protect*	*benchmark*	*baseline*	*EiovaR*	*diff*
throughput	strict	Netperf stream	1.00	2.37	+137%
(normalized)		Netperf RR	1.00	1.27	+27%
		Apache 1MB	1.00	3.65	+265%
		Apache 1KB	1.00	2.35	+135%
		Memcached	1.00	4.58	+358%
	defer	Netperf stream	1.00	1.71	+71%
		Netperf RR	1.00	1.07	+7%
		Apache 1MB	1.00	1.21	+21%
		Apache 1KB	1.00	1.11	+11%
		Memcached	1.00	1.25	+25%
alloc	strict	Netperf stream	7656	88	-99%
(cycles)		Netperf RR	10269	175	-98%
		Apache 1MB	17776	128	-99%
		Apache 1KB	49981	204	-100%
		Memcached	50606	151	-100%
	defer	Netperf stream	2202	103	-95%
		Netperf RR	2360	183	-92%
		Apache 1MB	2085	130	-94%
		Apache 1KB	2642	206	-92%
		Memcached	3040	171	-94%
search	strict	Netperf stream	153	0	-100%
(length)		Netperf RR	206	0	-100%
		Apache 1MB	381	0	-100%
		Apache 1KB	1078	0	-100%
		Memcached	893	0	-100%
	defer	Netperf stream	32	0	-100%
		Netperf RR	32	0	-100%
		Apache 1MB	30	0	-100%
		Apache 1KB	33	0	-100%
		Memcached	33	0	-100%
dealloc / free	strict	Netperf stream	289	66	-77%
(cycles)		Netperf RR	446	87	-81%
		Apache 1MB	360	70	-81%
		Apache 1KB	565	85	-85%
		Memcached	525	73	-86%
	defer	Netperf stream	273	65	-76%
		Netperf RR	242	66	-73%
		Apache 1MB	278	65	-76%
		Apache 1KB	300	66	-78%
		Memcached	334	65	-80%
cpu	strict	Netperf stream	100	100	+0%
(%)		Netperf RR	32	29	-8%
		Apache 1MB	100	99	-0%
		Apache 1KB	99	98	-1%
		Memcached	100	100	+0%
	defer	Netperf stream	100	100	+0%
		Netperf RR	30	29	-5%
		Apache 1MB	99	99	-0%
		Apache 1KB	98	98	-0%
		Memcached	100	100	+0%

Broadcom	*protect*	*benchmark*	*baseline*	*EiovaR*	*diff*
throughput	strict	Netperf stream	1.00	2.35	+135%
(normalized)		Netperf RR	1.00	1.07	+7%
		Apache 1MB	1.00	1.22	+22%
		Apache 1KB	1.00	1.16	+16%
		Memcached	1.00	1.40	+40%
	defer	Netperf stream	1.00	1.00	+0%
		Netperf RR	1.00	1.02	+2%
		Apache 1MB	1.00	1.00	+0%
		Apache 1KB	1.00	1.10	+10%
		Memcached	1.00	1.05	+5%
alloc	strict	Netperf stream	14878	70	-100%
(cycles)		Netperf RR	3359	100	-97%
		Apache 1MB	1469	74	-95%
		Apache 1KB	2527	116	-95%
		Memcached	5797	110	-98%
	defer	Netperf stream	1108	96	-91%
		Netperf RR	1029	118	-89%
		Apache 1MB	833	88	-89%
		Apache 1KB	1104	133	-88%
		Memcached	1021	130	-87%
search	strict	Netperf stream	345	0	-100%
(length)		Netperf RR	68	0	-100%
		Apache 1MB	27	0	-100%
		Apache 1KB	39	0	-100%
		Memcached	128	0	-100%
	defer	Netperf stream	13	0	-100%
		Netperf RR	9	0	-100%
		Apache 1MB	9	0	-100%
		Apache 1KB	9	0	-100%
		Memcached	9	0	-100%
dealloc / free	strict	Netperf stream	294	47	-84%
(cycles)		Netperf RR	282	48	-83%
		Apache 1MB	250	50	-80%
		Apache 1KB	425	52	-88%
		Memcached	342	47	-86%
	defer	Netperf stream	268	47	-82%
		Netperf RR	273	47	-83%
		Apache 1MB	234	47	-80%
		Apache 1KB	279	47	-83%
		Memcached	276	47	-83%
cpu	strict	Netperf stream	100	53	-49%
(%)		Netperf RR	13	12	-12%
		Apache 1MB	99	99	-0%
		Apache 1KB	98	98	-0%
		Memcached	99	95	-4%
	defer	Netperf stream	55	44	-21%
		Netperf RR	12	11	-7%
		Apache 1MB	91	72	-21%
		Apache 1KB	98	98	-0%
		Memcached	93	92	-2%

Table 2: *Summary of the results obtained with the Mellanox setup (left) and the Broadcom setup (right).*

Memcached in the Mellanox setup with an increasing number of clients. The left sub-graph reveals that the baseline allocation hampers scalability, whereas EiovaR allows the benchmark to scale such that it is up to 5.5x more performant than the baseline (with 32 clients). The right sub-graphs highlights why, showing that the baseline IOVA allocation becomes costlier proportionally to the number of clients, whereas EiovaR allocation remains negligible across the board.

FreeBSD We hypothesize that, like Linux, other OSes drive the IOMMU with suboptimal software, likely due to the perception that the IOMMU hardware is slow, possibly combined with the fact that I/O devices that are fast enough to significantly suffer from the consequences have become prevalent fairly recently. We test this hypothesis by studying the IOMMU mapping layer of FreeBSD. Our hypothesis coincides with the announcement of IOMMU support being added to FreeBSD, which says that "it

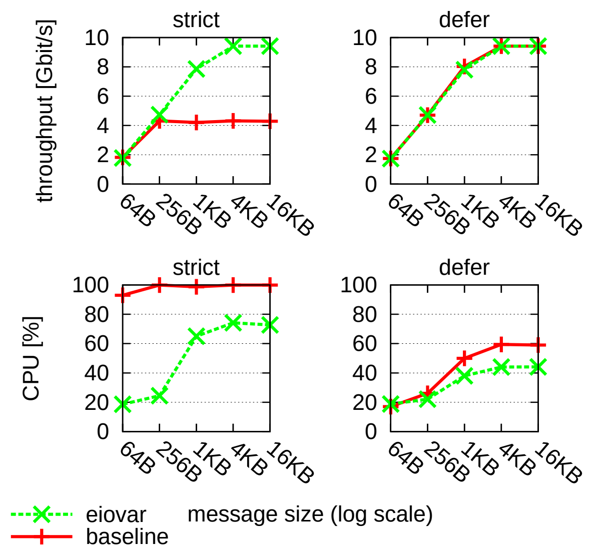

Figure 11: *Netperf Stream throughput (top) and used CPU (bottom) for different message sizes in the Broadcom setup.*

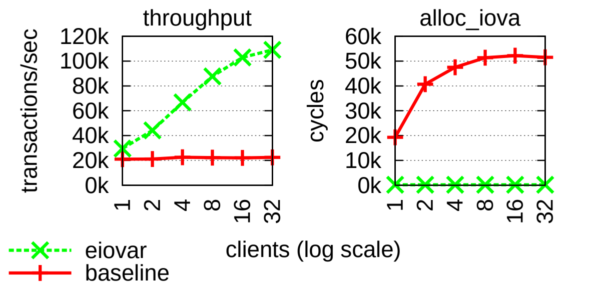

Figure 12: *Impact of increased concurrency on Memcached in the Mellanox setup.* EIOVAR *allows the performance to scale.*

is known that IOMMU adds overhead due to the mapping and unmapping for each I/O [and therefore it is] not plan[ned] to enable [the] IOMMU by default" [10].

We find that, similarly to Linux, FreeBSD uses a red-black tree for IOVA space management. Although it does not employ the problematic cached node optimization, the relevant source code can call fall back to a linear iteration through the tree nodes upon allocation. The comment preceding the linear iteration acknowledges that "this falls back to linear iteration over the free space in the high region"; however, the comment further notes that the said "high regions are almost unused" [26].

Using DTrace, the dynamic tracing tool [28], we profiled the IOVA mapping layer of FreeBSD while running the Netperf TCP stream benchmark. We measured each function along the call stack in a separate run, because multiple probe points affected the perceived results. Table 3 show the outcome, indicating that the FreeBSD IOMMU mapping layer overheads are larger than those of baseline Linux (compare with left of Figures 5–6). Specifically, whereas FreeBSD IOVA allocation is comparable

	map	*unmap*
iova	1,103	2,178
all the rest	8,557	13,825
total	9,660	16,003

Table 3: *FreeBSD IOMMU mapping layer overheads in cycles. (Compare with Linux's overheads in Figures 5–6.)*

to that of Linux, IOVA freeing takes an order of magnitude longer, and the (un)mapping is 4–5x slower altogether.

Our profiling reveals some of the root causes for these overheads. The aforementioned linear iteration remained inactive, as promised. But IOVA allocation turned out to nevertheless require the traversal of 11 red-black tree nodes on average. And the tree was rebalanced in almost every deallocation, introducing an overhead that is considerably higher than that of baseline Linux.

In addition to its inefficient IOVA (de)allocation, FreeBSD makes several suboptimal implementation choices that significantly slow down its mapping layer as compared to Linux. For instance, when a page within the IOMMU page table hierarchy is no longer in use, Linux usually does not reclaim it, rightfully assuming that it is likely to get reused soon. Conversely, FreeBSD does reclaim such pages, thereby reducing the memory footprint somewhat at the cost of increased CPU overheads.

The most wasteful unoptimized FreeBSD code we observed relates to synchronizing the I/O page table hierarchy between the IOMMU and the CPU. Upon every unmapping (ctx_unmap_buf_locked), FreeBSD flushes all the cachelines of the corresponding page table, although merely flushing the affected page table entries (PTEs) would have been enough. We applied the latter optimization to the FreeBSD unmap code and thus shortened it by $\sim$10K cycles on average, which improved the throughput of Netperf from 530 to 935 Mbps (1.76x higher).[3]

Consequently, in according to our hypothesis, we find that the FreeBSD mapping layer consists of suboptimal code that allows for easy optimizations that dramatically boost performance, possibly due to the perception that IOMMU hardware overheads are inherently high.

8 Related Work

Several studies recognized the poor performance associated with using the IOMMU [4, 12, 18, 44, 50, 57, 59]. Willmann et al. suggested to alleviate IOMMU overheads somewhat via "shared mappings", creating only one mapping for buffers that happen to reside on the same page instead of associating each of them with a different IOVA [57]. Amit et al. proposed to use "optimistic teardown",

[3]We confirmed this optimization with the relevant FreeBSD maintainer [11] and committed a patch that will be included in the next FreeBSD release [3].

whereby unmappings are delayed for a few milliseconds in the hope they will get immediately reused, creating a riskier policy than deferred protection that is more performant [4]. They also proposed to transparently offload the (un)mapping activity to computational cores different than the ones that perform the I/O. These approaches leave the original, unoptimized code intact and therefore EIOVAR is complementary to them.

Tomonori suggested to manage the IOVA space using bitmaps instead of trees, reporting an improvement in performance of 9% [50, 51]. Cascardo showed that performance is greatly improved if the driver of the I/O device can be modified to perform much fewer (un)mappings [18]. In a followup work, we proposed to redesign the IOMMU hardware to directly support the ring-induced workload and thus provide strict safety within 0.77–1.00x the throughput, 1.00–1.04x the latency, and 1.00–1.22x the CPU consumption of a system without an IOMMU [44].

Using freelists to speed up object allocation—as in EIOVAR—is a standard technique among memory allocators [13, 14, 15, 29, 40, 53, 55]. We discuss the contributes of this paper relative to such allocators in the next section.

9 Discussion, Conclusions, Future Work

Clements et al. made the case that implementations of OS kernels can be made scalable if they are designed beforehand such that their system calls commute, contending that "this rule aids developers in building more scalable software" [20]. Conversely, Linus Torvalds proclaimed that "Linux is evolution, not intelligent design" [22], likely more accurately reflecting the manner by which OSes are built, typically using the simplest implementation until experience proves that this is the wrong way to go.

When implementing new kernel functionality, a linear algorithm is often favored as being the simplest. For example, such was the case with the original linear Linux scheduler, which survived a decade [47]. And such is still the case with vmalloc, which is the internal Linux kernel function that is responsible for allocating virtually contiguous memory [52] (as opposed to kmalloc, which allocates *physically* contiguous memory). The pro of favoring linearity is simplicity. The con is that it might hinder performance when assumptions change.

The Intel/Linux IOVA allocation algorithm admittedly models the vmalloc algorithm [43]. From examining the source code, we see that both use a red-black tree for storing address ranges; both cache the location of the last freed range; and both use the cache as the starting point for subsequent allocations, traversing the tree elements in search for a large enough hole. We are not aware of workloads that utilize vmalloc whose performance noticeably degrades as a consequence. We demonstrate, however, that I/O intensive workloads suffer greatly form the linearity of IOVA allocation, which is induced by the "long-term ring interference" pathology that we characterize.

We conjecture that this deficiency exists because the IOMMU has been falsely perceived as the main responsible party for the significant overheads of intra-OS protection, and possibly because I/O devices fast enough to be noticeably affected have become widespread only in the last few years. We support our conjecture with experimental data from both Linux and FreeBSD.

We employ the compact EIOVAR optimization that proxies IOVA (de)allocations, resorting to the underlying red-black tree only if EIOVAR is unable to satisfy requests with its freelist. EIOVAR makes the baseline allocator orders of magnitude faster, improving the performance of common benchmarks by up to 5.5x.

Using freed object caches for fast allocation similarly to EIOVAR's freelist is not new. It is a standard technique employed by memory allocators [13, 14, 15, 29, 40, 53, 55]. Our contribution lies not in inventing the technique but rather: in (1) noticing it is applicable to, and substantially improves the performance of, the IOMMU mapping layer, which goes against the common wisdom that the slowness of this layer is due to the slowness of the hardware; in (2) carefully characterizing the workload experienced by the mapping layer; and in (3) finding that the workload characteristics allow for even the most basic/minimal freelist mechanism to deliver high performance, since (3.1) allocation requests exclusively consist of rounded up power-of-two areas that accelerate IOTLB invalidations without wasting real memory, and since (3.2) the freelist population size is inherently constrained by the relatively small size of the associated ring, so it can be used without worrying that the population of the previously-allocated-but-now-free IOVAs would explode.

EIOVAR eliminates one serious bottleneck of the IOMMU mapping layer. But we suspect that other bottlenecks exist, notably in relation to its locking regime, which affects subsystems different than the IOVA allocator and might hinder scalability. In the future, we therefore intend to study how the mapping layer scales as core-count increases. Another interesting question we intend to study is whether it is possible, and how hard is it, to exploit the window of vulnerability inherent to deferred protection as compared to strict protection.

Acknowledgments We thank the reviewers, our shepherd Carl Waldspurger, Muli Ben-Yehuda, and Dror Feitelson for providing valuable feedback. 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] Brian Aker. 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://support.amd.com/TechDocs/48882.pdf, Mar 2011.

[3] Nadav Amit and Konstantin Belousov. svn commit: r277023 – head/sys/x86/iommu, FreeBSD kerenl commit. https://lists.freebsd.org/pipermail/svn-src-head/2015-January/066777.html, Jan 2015. (Accessed: Jan 2015).

[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] Konstantin Belousov. FreeBSD x86 IOMMU support (DMAR). http://lists.freebsd.org/pipermail/freebsd-arch/2013-May/014368.html, May 2013. (Accessed: Jan 2015).

[11] Konstantin Belousov. FreeBSD VT-d IOMMU implementation. Private email communication, Jan 2015.

[12] 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.

[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, 2000.

[14] Jeff Bonwick. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Annual Technical Conf*, pages 87–98, 1994.

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

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

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

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

[19] 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.

[20] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, 2013.

[21] Demartek, LLC. QLogic FCoE/iSCSI and IP networking adapter evaluation (previously: Broadcom BCM957810 10Gb). http://www.demartek.com/Reports_Free/Demartek_QLogic_57810S_FCoE_iSCSI_Adapter_Evaluation_2014-05.pdf, May 2014. (Accessed: May 2014).

[22] Dror G. Feitelson. Perpetual development: A model of the Linux kernel life cycle. *Journal of Systems and Software*, 85(4):859–875, 2012.

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

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

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

[26] 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#L447. (Accessed: Jan 2015).

[27] 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.

[28] Brendan Gregg and Jim Mauro. *DTrace Dynamic Tracing in Oracle, Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[29] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software Practice and Experience (SPE)*, 23(8):851–869, 1993. Aug.

[30] 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.

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

[32] The hoard memory allocator. `http://www.hoard.org/`. (Accessed: May 2014).

[33] HP Development Company. Family data sheet: Broadcom NetXtreme network adapters for HP ProLiant Gen8 servers. `http://www.broadcom.com/docs/features/netxtreme_ethernet_hp_datasheet.pdf`, Aug 2013. Rev. 2. (Accessed: May 2014).

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

[35] 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).

[36] 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.

[37] Rick A. Jones. A network performance benchmark (revision 2.0). Technical report, Hewlett Packard, 1995. `http://www.netperf.org/netperf/training/Netperf.html`.

[38] 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.

[39] Anil S. Keshavamurthy. [patch -mm][Intel-IOMMU] optimize sg map/unmap calls. Linux Kernel Mailing List `https://lkml.org/lkml/2007/8/1/402`, Aug 2007.

[40] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, 2000.

[41] Doug Lea. malloc.c. `ftp://g.oswego.edu/pub/misc/malloc.c`, Aug 2012. (Accessed: May 2014).

[42] 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.

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

[44] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 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. To appear.

[45] 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).

[46] 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.

[47] Ingo Monar. Goals, design and implementation of the new ultra-scalable O(1) scheduler. `http://lxr.free-electrons.com/source/Documentation/scheduler/sched-design.txt?v=2.6.25`, Apr 2002.

[48] 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.

[49] 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.

[50] Fujita Tomonori. 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`.

[51] Fujita Tomonori. Intel IOMMU (and IOMMU for virtualization) performances. `https://lkml.org/lkml/2008/6/4/250`, Jun 2008. (Accessed: Jan 2015).

[52] Linus Torvalds and others. mm/vmalloc.c, source code file of Linux 3.17. `http://lxr.free-electrons.com/source/mm/vmalloc.c?v=3.17`. (Accessed: Jan 2015).

[53] Jim Van Sciver. Zone garbage collection. In *USENIX Mach Workshop*, pages 1–16, 1990.

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

[55] C. B. Weinstock and W. A. Wulf. Quick Fit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–148, Oct 1988.

[56] 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.

[57] 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.

[58] 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).

[59] 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.