# vIOMMU: Efficient IOMMU Emulation

Nadav Amit
*Technion & IBM Research*

Muli Ben-Yehuda
*Technion & IBM Research*

Dan Tsafrir
*Technion*

Assaf Schuster
*Technion*

## Abstract

Direct device assignment, where a guest virtual machine directly interacts with an I/O device without host intervention, is appealing, because it allows an unmodified (non-hypervisor-aware) guest to achieve near-native performance. But device assignment for unmodified guests suffers from two serious deficiencies: (1) it requires pinning all of the guest's pages, thereby disallowing memory overcommitment, and (2) it exposes the guest's memory to buggy device drivers.

We solve these problems by designing, implementing, and exposing an emulated IOMMU (vIOMMU) to the unmodified guest. We employ two novel optimizations to make vIOMMU perform well: (1) waiting a few milliseconds before tearing down an IOMMU mapping in the hope it will be immediately reused ("optimistic teardown"), and (2) running the vIOMMU on a sidecore, and thereby enabling for the first time the use of a sidecore by unmodified guests. Both optimizations are highly effective in isolation. The former allows bare-metal to achieve 100% of a 10Gbps line rate. The combination of the two allows an unmodified guest to do the same.

## 1 Introduction

I/O activity is a dominant factor in the performance of virtualized environments [29, 37], motivating *direct device assignment* whereby a guest virtual machine (VM) sees a real device and interacts with it directly. As direct access does away with the software intermediary that other I/O virtualization approaches require, it can provide much better performance than the alternative I/O virtualization approaches. This increased performance comes at a cost of complicating virtualization use-cases where the hypervisor interposes on guest I/O, such as live migration [20, 50]. Nonetheless, the importance of increased I/O performance cannot be overstated, as it makes virtualization applicable to common I/O-intensive workloads that would otherwise experience unacceptable performance degradation [26, 28, 33, 45, 48].

### 1.1 Motivation

Despite its advantages, direct device assignment suffers from at least three serious deficiencies that limit its applicability. First, it requires the entire memory of the unmodified guest to be pinned to the host physical memory. This is so because I/O devices typically access the memory by triggering DMA (direct memory access) transactions, and those can potentially target any location of the physical memory; importantly, unlike regular memory accesses, computer systems are technically unable to gracefully tolerate DMA page misses, reacting to them by either ignoring the problem, by restarting the offending domain, or by panicking. The hypervisor cannot tell which pages are designated by the unmodified guest for DMA transactions, and so, to avoid such unwarranted behavior, it must pin all the guest's pages to physical memory. This necessity negates a primary reason for using virtualization—server consolidation—because it hinders the ability of the hypervisor to perform memory overcommitment, whereas memory is *the* main limiting factor for server consolidation [16, 41, 47].

The second deficiency of direct device assignment is that the unmodified guest is unable to utilize the IOMMU (I/O memory management unit) so as to protect itself against bugs in the corresponding drivers. It is well-known that device drivers constitute the dominant source of OS (operating system) bugs [5, 17, 25, 38, 43]. Notably, the devices' ability to perform DMA to arbitrary physical memory locations is a main reason why such bugs are detrimental. IOMMUs were introduced by all major chip manufacturers to solve exactly this problem. They allow the OS to restrict DMA transactions to specific memory locations by having devices work with IOVAs (I/O virtual addresses) instead of physical addresses, such that every IOVA is validated by the IOMMU hardware circuitry upon each DMA transaction and is then redirected to a physical address according to the IOMMU mappings. The hypervisor cannot allow guests to program the IOMMU directly (otherwise every guest would be able to access the entire physical memory), and so all the related work that provided ways for guests to enjoy the IOMMU functionality [12, 13, 25, 35, 44, 49] involved paravirtualization. Namely, the guest's OS was modified to explicitly inform the hypervisor regarding the DMA mappings it requires. Clearly, such an approach is inapplicable to unmodified (fully virtualized) guests.

A third deficiency of direct device assignment is that, in general, it prevents the unmodified guest from taking advantage of the IOMMU remapping capabilities, which

are useful in contexts other than just defending against faulty device drivers. One such context is legacy devices that do not support memory addresses wider than 32bit, an issue that can be easily resolved by programming the IOMMU to map the relevant 32bit-addresses to higher memory locations [18]. Another such context is "nested virtualization", which allows one hypervisor to run other hypervisors as guests [11] and, hence, mandates granting a nested hypervisor the ability to program the IOMMU to protect its guests from one another (when those utilize directly-assigned devices).

## 1.2 Contributions and Preview of Results

**IOMMU Emulation** The root cause of all of the above limitations is the fact that current hypervisors do not provide unmodified guests with an emulated IOMMU. Our initial contribution is therefore to implement and evaluate such an emulation, for the first time. We do so within KVM on Intel x86, following the proposal made by Intel [1]. We denote the emulation layer "vIOMMU". And we note in passing that we are aware of a similar effort that is currently being done for AMD processors [31].

By emulating the IOMMU, our patched hypervisor intercepts, monitors, and acts upon DMA remapping operations. Knowing which of the unmodified guest's memory pages serve as DMA targets allows it to: (1) pin/unpin the corresponding host physical pages, and only these pages, thereby enabling memory overcommitment; (2) program the physical IOMMU to enable device access to the said physical pages, and only to these pages, thereby enabling the guest to protect its memory image against faulty drivers; and (3) redirect DMA transactions through the physical IOMMU according the unmodified guest's wishes, thereby retrieving the indirection level needed to support legacy 32bit devices, certain user-mode DMA usage models, and nested virtualization. (See Section 2 for details.)

Utilizing the IOMMU without relaxing the protection it offers is costly, even for a bare metal (unvirtualized) OS. Our experiments using Netperf [19] show that bare metal Linux 2.6.35 achieves only 43% of the line-rate of a 10Gbps NIC when the IOMMU is used with strict protection; the corresponding unmodified guest achieves less than one fourth of that with the vIOMMU.

**Optimistic Teardown** The default mode of Linux, however, relaxes IOMMU protection. It does so by batching the invalidation of stale IOTLB entries and by collectively purging them from the IOTLB every 10ms (IOTLB is the I/O translation look-aside buffer within the IOMMU). The protection is relaxed, because, during this short interval, a faulty device might successfully perform a DMA transaction through a stale entry. Nonetheless, for bare metal, the resulting improvement is dramatic, transforming the aforesaid 43% throughput to 91% and arguably justifying the risk. Alas, the corresponding unmodified guest does not experience such an improvement, as its throughput remains more or less the same when the protection is relaxed.

To improve the performance of the vIOMMU, our second contribution is investigating a set of optimizations that exercise the protection/performance tradeoff in various ways (see Section 3 for details). We find that the "optimistic teardown" optimization is the most effective.

While the default mode of Linux removes stale IOTLB entries en masse at 10ms intervals, it nevertheless tears down individual invalidated IOVA translations with no delay, immediately removing them from the IOMMU page table. The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the OS mapping data structure, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [3, 44], one can be hopeful that the newly proposed optimization would work. Importantly, for each reused translation, optimistic teardown would avoid the overhead of (1) tearing the translation down from the IOMMU page table, (2) invalidating it in the IOTLB, (3) immediately reconstructing it, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Optimistic teardown is remarkably successful, pushing the throughput of bare metal from 91% to 100% (and reducing its CPU consumption from 100% to 60%). The improvement is more pronounced for an unmodified guest with vIOMMU: from 11% throughput to 82%.

**Sidecore Emulation** To further improve the performance of the unmodified guest, we implement the vIOMMU functionality on an auxiliary sidecore. Traditional "samecore" emulation of hardware devices (where hypervisor invocations occur on the guest's core) has been extensively studied in the literature [6, 10, 21, 37]. Likewise, offloading of computation to a sidecore for speeding up I/O in a paravirtualized system has been explored as well [15, 23, 27]. But in this paper, for the first time, we present "sidecore emulation", which combines the best of both approaches. Specifically, sidecore emulation maintains the exact same hardware interface between the guest and the sidecore as exists in a non-virtualized setting between a bare metal OS and the real hardware device. Consequently, sidecore emulation is able to offload the computation while requiring no guest modifications. (See details in Section 4.)

By running the vIOMMU on a sidecore, we triple the

| *setting* | *strict* | *relaxed (default)* | *optimistic teardown* |
|---|---|---|---|
| samecore | 10% | 11% | 82% |
| sidecore | 30% | 49% | 100% |
| bare metal | 43% | 91% | 100% |

Table 1: *Summary of preview of results (percent of line-rate throughput on 10GbE).*

throughput of the strict unmodified guest, quintuple its throughput if its protection is relaxed, and achieve 100% of the line-rate if employing optimistic teardown. The results mentioned so far are summarized in Table 1.

**Roadmap** We describe: our "samecore" vIOMMU design (§2); the set of optimizations we explore and the associated performance/protection tradeoffs (§3); our "sidecore" vIOMMU design (§4); how to reason about risk and protection (§5); evaluation of the performance of our proposals using micro and macro benchmarks (§6); the related work (§7); and our conclusions (§8).

## 2 Samecore IOMMU Emulation

I/O device emulation for virtualized guests is usually implemented by trapping guest accesses to device registers and emulating the appropriate behavior [2, 10, 37]. Correspondingly, in this section, we present the rudiments of emulating an IOMMU. We choose to emulate Intel's VT-d IOMMU [18], as it is commonly available and as most x86 OSes/hypervisors have drivers for it. Conveniently, Intel's VT-d specification [18] proposes how to emulate an IOMMU. We largely follow their suggestions.

The emulated guest BIOS uses its ACPI (Advanced Configuration and Power Interface) tables to report to the guest that the (virtual) hardware includes Intel's IOMMU. Recognizing that the hardware supports an IOMMU, the guest will ensure that any DMA buffer in use will first be mapped in the IOMMU for DMA [12]. The emulated IOMMU registers reside in memory pages that the hypervisor marks as "not present", causing any guest access to them to trap to the hypervisor. The hypervisor monitors the emulated registers and configures the platform's physical IOMMU accordingly. The hypervisor further monitors changes in related data structures such as the IOMMU page tables in guest memory.

Figure 1 illustrates the flow of a single DMA transaction in an emulated environment: a guest I/O device calls the IOMMU mapping layer when it wishes to map an I/O buffer (1); the layer accordingly allocates an IOVA region and, within the emulated IOMMU, maps the corresponding page table entries (PTEs) to point to the GPA (guest physical address) given by the I/O device driver (2); the layer performs an explicit mapping invalidation of these PTEs (3), thereby triggering a write access to a certain IOMMU register, which traps to the hypervisor; the hypervisor then updates the status of the emulated IOMMU registers (4), reads the IOVA-to-GPA mapping from the updated emulated IOMMU PTEs (5), pins the relevant page to the host physical memory (not shown), and generates physical IOMMU PTEs to perform IOVA-to-HPA (host physical address) mapping (6); when the physical hardware requires it, the hypervisor also performs physical IOTLB invalidation (7); the guest is then resumed, and the I/O device driver initiates the DMA transaction, delivering the IOVA as the destination address to the device (8); the device performs memory access to the IOVA (9), which is appropriately redirected by the physical IOMMU (10-11); the guest OS can then unmap the IOVA, triggering a flow similar to the mapping flow except that the hypervisor unmaps the I/O buffer and unpins its page-frames.

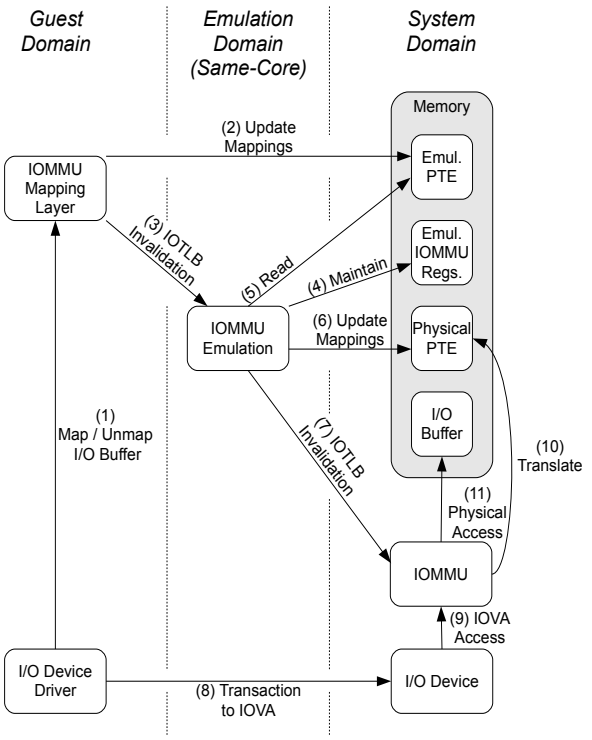


Figure 1: *IOMMU emulation architecture (samecore).*

## 3 Optimizing IOMMU Mapping Strategies

Operating systems can employ multiple mapping strategies when establishing and tearing down IOMMU mappings. Different mapping strategies tradeoff performance vs. memory consumption vs. protection [13, 44, 49].

Taking Linux as an example, the default mapping strategy of the Intel VT-d IOMMU driver is to *defer* and batch IOTLB invalidations, thereby improving performance at the expense of reduced protection from errant DMAs. Batching IOTLB invalidations helps performance because IOTLB invalidations are expensive. Unlike an MMU TLB, which resides on a CPU core, an IOMMU and its IOTLB usually reside away from the CPU on the PCIe bus.

An alternative mapping strategy is the *strict* mapping strategy. In the strict strategy Linux's IOMMU mapping layer executes IOTLB invalidations as soon as device drivers unmap their I/O buffers and waits for the invalidations to complete before continuing.

In this section we investigate the different tradeoffs possible on bare metal and in a virtualized system employing an emulated IOMMU, where both the guest and the host may employ different mapping strategies. We discuss different IOMMU mapping performance optimizations and their effect on system safety, starting with the least dangerous strategy and ending with the best performing—but also most dangerous—strategy.

## 3.1 Approximate Shared Mappings

Establishing a new mapping in the IOMMU translation table and later tearing it down are inherently costly operations. Shared mappings can alleviate some of the costs [44]. We can reuse a mapping when another valid mapping which points to the same physical page frame already exists. Using the same mapping for two mapping requests saves the time required for the setup and eventual teardown of a new mapping.

Willmann, Rixner and Cox propose a precise lookup method for an existing mapping. Their approach relies on an inverted data structure translating from physical address to IOVA for all mapped pages [44]. This approach is problematic with modern IOMMUs that can map all of physical memory and employ a separate I/O virtual address space for each protection context (usually for each I/O device). Maintaining a direct-map data structure to enable precise lookups is impractical for such IOMMUs as it would require too much memory. We expect that using a smaller but more complex data structure, such as a red-black tree, will incur prohibitively high overhead [32].

To avoid the overhead associated with complex datastructures, we propose *approximate shared mappings*. Instead of maintaining a precise inverted data structure, we perform reverse lookups using heuristics which may fail to find a translation from physical address to IOVA, even though there exists a mapping of that physical address. Our implementation of approximate shared mappings used a software LRU cache, which requires temporal locality in I/O buffers allocation in order to perform well, in addition to spatial locality of the I/O buffers. Many applications experience such temporal locality [44].

## 3.2 Asynchronous Invalidations

IOTLB invalidation is a lengthy process that on bare metal takes over 40% of the overall unmapping process. *Asynchronous invalidation* is an invalidation scheme targeted at alleviating the cost of the lengthy IOTLB invalidation process by a minor relaxation of protection. The default IOTLB invalidation scheme is synchronous: the OS writes an invalidation request to the IOMMU's invalidation register or (when the hardware supports it) to an invalidation queue [18] and blocks the execution thread until the IOMMU completes the invalidation. In asynchronous invalidation, the OS does *not* wait for the invalidation to complete before continuing. Doing so on bare metal can save the few hundred cycles it takes the IOMMU to write the invalidation completion message back to memory after the invalidation is done.

Asynchronous invalidation enables multiple in-flight invalidations when the hardware supports an invalidation queue. However, to maintain correctness, asynchronous invalidation must not permit an IOVA range which is being invalidated to be mapped again to a different physical address until the invalidation process is completed. Unfortunately there is no practical way to ensure with Linux that the page allocator will not reuse the physical memory backing those IOVAs while the invalidation is outstanding [49].

On bare metal asynchronous invalidation relaxes protection only slightly, since the IOMMU hardware performs the invalidation process in silicon, taking only hundreds of cycles to complete. In our experiments with asynchronous invalidation, the invalidation queue never held more than two pending invalidations at the same time.

## 3.3 Deferred Invalidation

Deferring IOTLB invalidations, as currently implemented by Linux, makes it possible to aggregate IOTLB invalidations together and possibly coalesce multiple invalidation requests so that they will be invalidated in a single request, if the hardware supports it. Instead of the OS invalidating each translation entry as it is torn down, the OS collects multiple invalidations in a queue, which it then flushes periodically. The current Linux implementation coalesces up to 250 invalidations for periods of no longer than 10ms.

Holding back the invalidations makes the deferred method less secure than the asynchronous method, where

the "window of vulnerability" for an errant DMA is only hundreds of cycles. But deferred invalidation reduces the number of software/hardware interactions, since a whole batch of invalidations is executed at once. This savings is more pronounced when the hardware is emulated by software, in which case deferred invalidation can save multiple, expensive guest/host interactions.

### 3.4 Optimistic Teardown

Reusing IOVA translations is key to IOMMU performance [3, 13, 44, 49]. Reusing a translation avoids the overhead of (1) tearing a translation down from the IOMMU page table, (2) invalidating it from the IOTLB, (3) immediately reconstructing it in the page table, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Even when approximate shared mapping is used, the opportunities to reuse IOVA translations are limited. The default Linux deferred invalidation scheme removes stale IOTLB entries en masse at 10ms intervals, but nevertheless tears down individual unmapped IOVA translations with no delay, immediately removing them from the IOMMU page tables.

The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the IOMMU page table, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [3, 44, 49], one can be hopeful that the newly proposed optimization would work.

We thus developed an optimistic teardown mapping strategy, which keeps mappings around even after an unmapping request for them has been received. Unmapping operations of I/O buffers are deferred and executed at a later, configurable time. If an additional mapping request of the same physical memory page arrives to the IOMMU mapping layer while a mapping already exists for that page, the old mapping is reused. If an old mapping is not used within the pre-defined time limit, it is unmapped completely and the corresponding IOMMU PTEs are invalidated, limiting the overall window of vulnerability for an errant DMA to the pre-defined time limit. We determined experimentally that on our system a modest limit of ten milliseconds is enough to achieve a 92% hit rate.

We keep track of all cached mappings in the same software LRU cache, regardless of how many times each mapping is shared. Mappings which are not currently in use are also kept in a deferred-unmappings first-in first-out (FIFO) queue with a fixed size limit. The queue size and the residency constraints are checked whenever the queue is accessed, and also periodically. Invalidations are performed when mappings are removed from the queue.

## 4 Sidecore IOMMU Emulation

Samecore emulation uses the classical approach of trapping device register access and switching to the hypervisor for handling. We now present an alternative, novel approach for device emulation which uses a second core to handle device register accesses, thus avoiding expensive VM-exits. We call this sidecore emulation. While the discussion below focuses on Intel's VT-d, our approach is generic and can be applied to most other IOMMUs and I/O devices.

Samecore hardware emulation suffers from an inherent limitation. Each read or write access to the hardware registers requires a VM-transition to the hypervisor, which then emulates the hardware behavior. VM-transitions are known to be expensive, partly due to cache pollution [2, 11].

Offloading computation to a sidecore for speeding up I/O for modified (paravirtualized) guests has been explored by Kumar et al. [23], Gavrilovska et al. [15], and Liu and Abali [27]. Sidecore emulation offloads device emulation to a sidecore. In contrast with previous paravirtualized sidecore approaches, which require guest modifications, sidecore emulation maintains the same hardware interface between the guest and the sidecore as between a bare-metal OS and the real hardware device, and thus requires no guest modifications. As we show in Section 6, sidecore emulation on its own can achieve 69% of bare metal performance—for unmodified guests and without any protection relaxation.

In general, hardware emulation by a sidecore follows the same principles as samecore emulation. The guest programs the device, the hypervisor detects that the guest has accessed the device, decodes the semantics of the access, and emulates the hardware behavior. But sidecore emulation differs from samecore emulation in two fundamental aspects. First, there are no expensive traps from the guest to the hypervisor when the guest accesses device registers. Instead, the device register memory areas are shared between the guest and the hypervisor, and the hypervisor polls the emulated control registers for updates. Second, the guest code and the hypervisor code run on different cores, leading to reduced cache pollution and improved utilization of each core's exclusive caches.

Efficient hardware emulation by a sidecore is dependent on the interface between the I/O device and the guest OS, since the sidecore polls memory regions instead of

receiving notifications on discrete register access events. In general, efficient sidecore emulation requires that the physical hardware have the following (commonly found) properties.

**Synchronous Register Write Protocol** Sidecore emulation relies on a synchronous protocol between the device driver and the device for a single register's updates, in the sense that the device driver expects some indication from the hardware before writing to a register a second time. Such a protocol ensures that the sidecore has time to process the first write before a second write to the same register overwrites the first write's contents.

**A Single Register Holds Only Read-Only or Write Fields** Registers which hold both read-only and write fields are challenging for a sidecore to handle, since the sidecore has no efficient way of ensuring the guest device driver would not change read-only fields.

**Loose Response Time Requirements** Sidecore emulation is likely to be slower than physical hardware. If the device has strict specifications of the "wall time" device operations take (e.g., "this operation completes within 3ns") or the device driver makes other strong timing assumptions which hold for real hardware but not for emulated hardware, then the device driver might assume that the hardware is malfunctioning when operations take longer than expected. This property must hold for device emulation in general.

**Explicit Update of Memory-Resident Data Structures** Since the sidecore cannot poll large memory regions efficiently, update to its memory-resident data structures should be explicit, by requiring the device-driver to perform a write-access to the device control registers indicating exactly which data structure it updated.

An additional, optional property that can boost sidecore emulation performance is a limited number of control registers. Since the sidecore needs to sample the control registers of the emulated hardware, a large number of registers would result in long latency between the time the guest sets the control register and the time the sidecore detects the change. In addition, polling a large number of registers may result in cache thrashing.

Intel's IOMMU has all of the properties required for efficient sidecore emulation. This is in contrast to AMD's IOMMU, which cannot require the OS's mapping layer to explicitly update the IOMMU registers upon every change to the memory-resident page tables. We note, however, that the emulated IOMMU and the platform's physical IOMMU are orthogonal, and Intel's IOMMU can be emulated when only AMD's IOMMU is physically present or even when no physical IOMMU is present and bounce buffers are used instead [12].

# 5 Reasoning About Risk and Protection

## 5.1 Risk and Protection Types

The IOMMU was designed to protect those pages which do not hold I/O buffers from errant DMA transactions. To achieve complete protection, the IOMMU mapping layer must ensure a page is accessible for DMA transactions only if it holds an I/O buffer that may be used for DMA transaction and only while a valid DMA transaction may target this page [49].

However, IOMMU mapping layer optimizations may relax protection by completing the synchronous unmap function call by the I/O device driver (*logical unmapping*) before tearing down the mapping in the physical IOMMU page-tables and completing the physical IOTLB invalidation (*physical unmapping*).

Deferring physical unmapping this way, as done by the deferred invalidation scheme, the asynchronous invalidation scheme, and the optimistic teardown scheme, could potentially compromise protection for any page which has been logically unmapped but not yet physically unmapped. We differentiate, however, between *inter-guest protection*, protection between different guest OS instances, and *intra-guest protection*, protection within a particular guest OS [44].

vIOMMU maintains full inter-guest protection—full isolation between VMs—in all configurations. It maintains inter-guest protection by keeping pages pinned in physical memory until they have been physically unmapped. vIOMMU *pins* a page in physical memory before mapping it in the IOMMU page table, and only *unpins* it once the IOMMU mapping of that page is torn down and the IOTLB invalidation is complete. Consequently, any page that is used for a DMA transaction by a guest OS will not be re-allocated to any other guest OS as long as it may be the target of a valid DMA transaction by the first guest OS.

Full intra-guest protection—protecting a guest OS from itself—is arguably less important than inter-guest protection in a virtualized setting. Intra-guest protection may be relaxed by both the host's and the guest's mapping layer optimizations. Maintaining complete intra-guest protection with optimal performance in an operating system such as Linux without modifying all drivers remains an open challenge [49], since Linux drivers assume that any page that has been logically unmapped is also physically unmapped. Consequently, such pages are often re-used by the driver or the I/O stack for other purposes as soon as they have been logically unmapped.

## 5.2 Quantifying Risk

We do not assess the risk posed by arbitrary malicious adversaries, since such adversaries might sometimes be able exploit even very short vulnerability windows [40]. In our discussion of protection and risk we focus instead on the "window of vulnerability", when an errant DMA may sneak in and read or write an exposed I/O buffer through a *stale mapping*. A stale mapping is a mapping which exists after a page has been logically unmapped but before it has been physically unmapped. A stale mapping occurs when the device driver asks to unmap an IOVA translation and receives an affirmative response, despite the actual teardown of the physical IOMMU PTE or physical IOTLB invalidation having been deferred.

We quantify risk along two axes: the *duration of vulnerability* during which an I/O buffer is open for reading or writing through a stale mapping, and the *stale mapping bound*, which indicates the maximum number of stale mappings at any given point in time.

We classify the mapping strategies mentioned above into four classes according to their duration: no risk, nanosecond risk, microsecond risk, and millisecond risk.

**No Risk** The only times when there is no risk are when an OS on bare metal uses the strict mapping strategy, or when both guest and host use the strict mapping strategy. Since buffers are unmapped and their mappings invalidated without any delay, there can be no stale mappings regardless of whether we run on bare metal, use samecore emulation, or sidecore emulation. The use of approximate shared mappings does not affect the risk.

**Nanosecond Risk** The time that elapses between the moment when the host posts an invalidation request to the invalidation queue and the invalid translation is actually flushed from the physical IOTLB can be measured in nanoseconds. Since the flush happens in silicon, this duration is a physical property of the platform IOMMU, and the risk only applies to bare metal with asynchronous invalidation. With samecore or sidecore emulation, guest/host communication costs overshadow this duration. We determined experimentally that on our system the stale mapping bound for nanosecond risk is at most two mappings, and the duration of vulnerability is 128 cycles per entry on average.

**Microsecond Risk** Microsecond risk only applies to sidecore emulation and comes into play when the guest does not wait for the host to process an invalidation (i.e., when the guest uses asynchronous invalidation). Here, inter-core communication costs determine the window of vulnerability, since the host must realize that the guest posted an invalidation before it can handle it. In general, the stale mapping bound for microsecond risk is the number of outstanding invalidation requests in the emulated invalidation queue. In our experimental setup the queue was sized to hold at most 128 outstanding entries.

**Millisecond Risk** Millisecond risk applies when either the guest or the host uses the deferred invalidation or optimistic teardown strategies. Regardless of whether the guest or the host defers invalidations or keeps around cached mappings, the window of vulnerability is likely to be in the order of milliseconds. Software configures the stale mappings bounds by setting a quota on the number of cached mappings and a residency time limit on each mapping.

**Overall Risk** When a guest OS uses an emulated IOMMU, the combination of the guest's and host's mapping strategies determines the overall protection level. The hypervisor cannot override the guest mapping strategy to provide greater protection, since the hypervisor is unaware of any cached mappings or deferred invalidations in the guest until the guest unmaps them and executes the invalidations. Therefore, the hypervisor can either keep the guest's level of protection by using a strict invalidation scheme, or relax it for better performance.

# 6 Performance Evaluation

## 6.1 Methodology

**Experimental Setup** We implement the samecore and sidecore emulation of Intel IOMMU, as well as the mapping layer optimizations presented above. We use the KVM hypervisor [21] and Ubuntu 9.10 running Linux 2.6.35 for both host and guest. Our experimental setup is comprised of an IBM System x3550 M2, which is a dual-socket, four-cores per socket server equipped with Intel Xeon X5570 CPUs running at 2.93GHz. The Chipset is Intel 5520, which supports VT-d. The system includes 16GB of memory and an Emulex OneConnect 10Gbps NIC. We use another identical remote server (connected directly by 10Gbps optical fiber) as a workload generator and a target for I/O transactions. In order to obtain consistent results and to avoid reporting artifacts caused by nondeterministic events, all power optimizations are turned off, namely, sleep states (C-states) and DVFS (dynamic voltage and frequency scaling).

To have comparable setups, guest-mode configurations execute with a single VCPU (virtual CPU), and native-mode configurations likewise execute with a single core enabled. In guest-mode setups, the VCPU and the sidecore are pinned to two different cores on the same die, and 2GB of memory is allocated to the guest.

**Microbenchmarks** We use two well-known Netperf [19] instances in order to assess the overheads induced by vIOMMU in terms of throughput, CPU cycles,

| config. | *guest/native invalidation* | *guest/native reuse* | *guest/native Linux* | *host invalidation* | *native max stale#* | *guest max stale#* | *duration magnitude* |
|---|---|---|---|---|---|---|---|
| strict | strict | none | unpatched | strict | none | none | 0 |
| shared | strict | shared | patched | strict | none | none | 0 |
| async | async | shared | patched | async | 32 | 128+32 | $\mu$sec |
| deferred | deferred | none | unpatched | deferred | 32 | 250+32 | ms |
| opt256 | async | shared+tear | patched | deferred | 256+32 | 256+128+32 | ms |
| opt4096 | async | shared+tear | patched | deferred | 4096+32 | 4096+128+32 | ms |
| off | n/a | n/a | unpatched | deferred | all | all | $\infty$ |

Table 2: *Evaluated configurations. The host column is meaningless when running the native configuration. The maximal number of stale mappings for async and deferred host is the size of the IOTLB, namely, 32.*

and latency. The first instance—Netperf TCP stream—attempts to maximize the amount of data sent over a single TCP connection, simulating an I/O-intensive workload. The second instance–Netperf UDP RR (request-response)—models a latency-sensitive workload by repeatedly sending a single byte and waiting for a matching single byte response. Latency is calculated as the inverse of the number of transactions per second.

**Macrobenchmarks** We use two macrobenchmarks to asses the performance of vIOMMU on real applications. The first is MySQL SysBench OLTP (version 0.4.12; executed with MySQL database version 5.1.37), which was created for benchmarking the MySQL database server by generating OLTP inspired workloads. To simulate high-performance storage, the database is placed on a remote machine's RAM-drive, which is accessed through NFS and mounted in synchronous mode. The database contains two million records, which collectively require about 1GB. We disable data caching on the server by using the InnoDB engine and the `O_DIRECT` flush method.

The second macrobenchmark we use is Apache Bench, evaluating the performance of the Apache web server. Apache Bench is a workload generator that is distributed with Apache to help assess the number of concurrent requests per second that the server is capable of handling. The benchmark is executed with 25 concurrent requests. The logging is disabled to avoid the overhead of writing to disk.

**Configurations** There are many possible combinations of emulation approaches, which are comprised of the guest and host mapping layers and their reuse and invalidation strategies. Each such combination is associated with different protection and performance levels. We cannot evaluate all combinations. We instead choose to present several meaningful ones in the hope that they provide reasonable coverage. The configurations are listed in Table 2. Each line in the table pertains to two scenarios: a virtualized setting, with a guest serviced by a host, and a "native" setting with only the bare metal OS running. The latter scenario provides a baseline. It is addressed because our optimizations apply to virtualized settings and bare metal settings alike. We next describe the configurations one by one, from safest to riskiest.

The **strict** configuration involves no optimizations in guest, host, or native modes, and hence it involves no risk; it is the least performant configuration. While strict is not the default mode of Linux, it requires no OS modification, but rather setting an already-existing configurable parameter. Hence it is marked as "unpatched".

The **shared** configuration is nearly identical to strict except that it adds the approximate shared mapping optimization (Section 3.1); it is still risk-free, merely attempting to avoid allocating more than one IOVA for a given physical location and preferring instead to reuse. Notice that for the virtualized setting this optimization is meaningless for the host, as the hypervisor cannot override the IOVA chosen by the guest. The OS is patched because Linux does not natively support shared mappings.

The **async** configuration is similar to the shared configuration, yet in addition it utilizes the asynchronous IOTLB invalidation optimization (Section 3.2). The latter immediately invalidates unmapped translations, but does not wait for the IOTLB invalidation to complete, reducing invalidation cost by the time it takes the IOMMU to write its invalidation completion message back to memory. Realistically, the risk exists only for the sidecore setting, which is dominated by inter-core communication cost that is approximated by not more than a handful of $\mu$secs. The theoretical maximal number of stale entries is the size of the IOTLB (32) in the host and native settings; in this guest's case, this is supplemented by the default size of the invalidation queue (128).

The **deferred** configuration is the default configuration of Linux, whereby IOTLB invalidations are aggregated and processed together every 10ms (Section 3.3). In the guest's case, stale entries might reside in the IOTLB (32) or in the deferred entries queue (up to 250 by

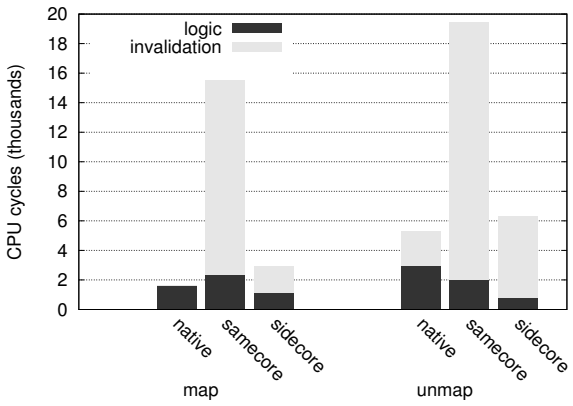


Figure 2: *Average breakdown of (un)mapping a single page using the strict invalidation scheme.*

default). While the entries are in the guest's queue, the host does not know about them and hence cannot invalidate them. As both guest and host use a 10ms interval, the per-entry maximal vulnerability window is 20ms for the guest and half that much for the host and bare metal.

The **opt** configuration (short for "optimistic") deploys all optimizations save deferred invalidation, which is substituted by optimistic teardown (Section 3.4). The maximal number of stale entries we keep alive (for up to 10ms) is 256, similarly to the 250 of deferred; in a more aggressive configuration we increase that number to 4096.

Finally, the **off** configuration does not employ an IOMMU in the native setting, and does not employ a vIOMMU in the virtualized setting. (In the latter case the physical IOMMU is nevertheless utilized by the host, because the device is still assigned to the guest.) In this configuration, neither the guest nor the native bare metal enjoy any form of protection, which is why we marked "all" the mappings as unsafe for their entire lifetime ("$\infty$").

## 6.2 Overhead of (Un)mapping

The IOMMU layer provides exactly two primitives: map and unmap. Before we delve into the benchmark results, we first profile the overhead induced by the vIOMMU with respect to these two operations. Figure 2 presents the cycles breakdown of each operation to IOTLB "invalidation", which is the direct interactions of the OS with the IOMMU, and to "logic", which encapsulates the rest of the code that builds and destroys the mappings within the I/O page tables.

Notice that guest invalidation overhead is induced even when performing the map operation; this happens because the hypervisor turns on the "caching mode bit", which, by the IOMMU specification, means that the OS is mandated to first invalidate every new mapping it creates (which allows the hypervisor to track this activity). Most evident in the figure is the fact that the sidecore dramatically cuts down the price of invalidation when compared to samecore, which is a direct result of eliminating the associated VM exits and associated world switches. The other interesting observation is that the rest of the (un)map logic can be accomplished faster by the vIOMMU. This better-than-native performance is a product of the vIOMMU registers being cacheable, as opposed to those of the physical IOMMU.

## 6.3 Benchmark Results

Figure 3(a) depicts the throughput of Netperf/TCP for each configuration, from safest to riskiest, along the X axis. The values displayed are normalized by the maximal throughput achieved by bare metal and off, which in this case is 100% of the attainable bandwidth of the 10Gbps NIC. Figure 3(b) presents the very same data, but the normalization is done against native on a per-configuration basis; accordingly, the native curve coincides with the "1" grid line. Figure 3(c) presents the CPU consumed by Netperf/TCP while doing the corresponding work; observe that the sidecore is associated with two curves in this figure, the lower one corresponds to the useful work done by the sidecore (aside from polling) and the upper one pertains to the main core.

The safe (shared) or nearly safe (async) configurations provide no benefit for the samecore setting, but they can slightly improve the performance of sidecore and native by 2–5 percentage points each. Deferred delivers a much more pronounced improvement, especially in the native case, which manages to attain 91% of the line-rate. By consulting Figure 3(c), we can see that native/deferred is not attaining 100%, because the CPU is a bottleneck. Utilizing opt solves this problem, not only for the native setting, but also for the sidecore; opt allows both to fully exploit the NIC. The sidecore/CPU curve (bottom of Figure 3(c)) implies that the work required form the IOMMU software layer is little when optimal teardown is employed, allowing the sidecore to catch up with native performance and the samecore to reduce to gap to 0.82x the optimum.

Similarly to the above, Figure 4 depicts the latency as measured with Netperf/UDP-RR and the associated CPU consumption. The results largely agree with what we have seen for Netperf/TCP. Deferring the IOTLB invalidation allows the native setting to achieve optimal latency, but only slightly improves the virtualized settings. However, when optimistic teardown is employed, the latency of both sidecore and samecore drops significantly (by about 60 percentage point in the latter case), and they manage attain the optimum. Importantly, the

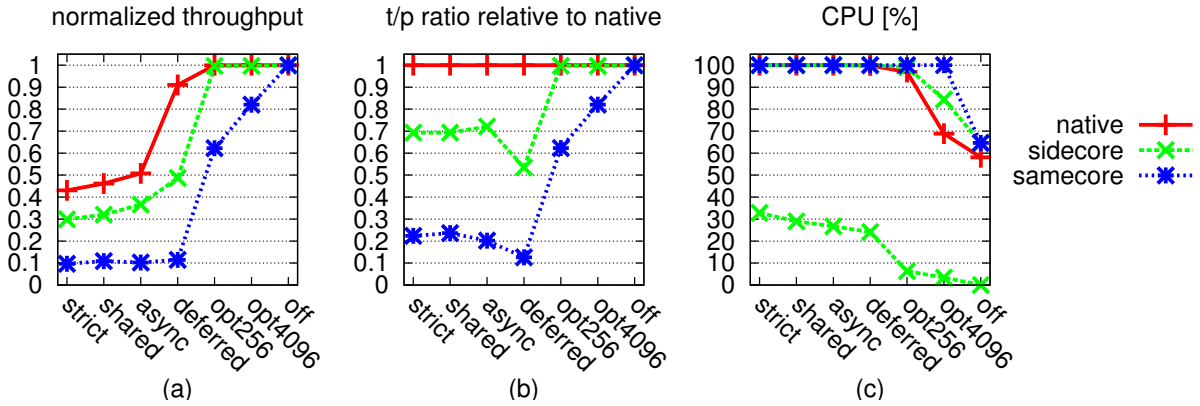


Figure 3: *Measuring throughput with Netperf TCP; the baseline for normalization is the optimal throughput attainable by our 10Gbps NIC.*

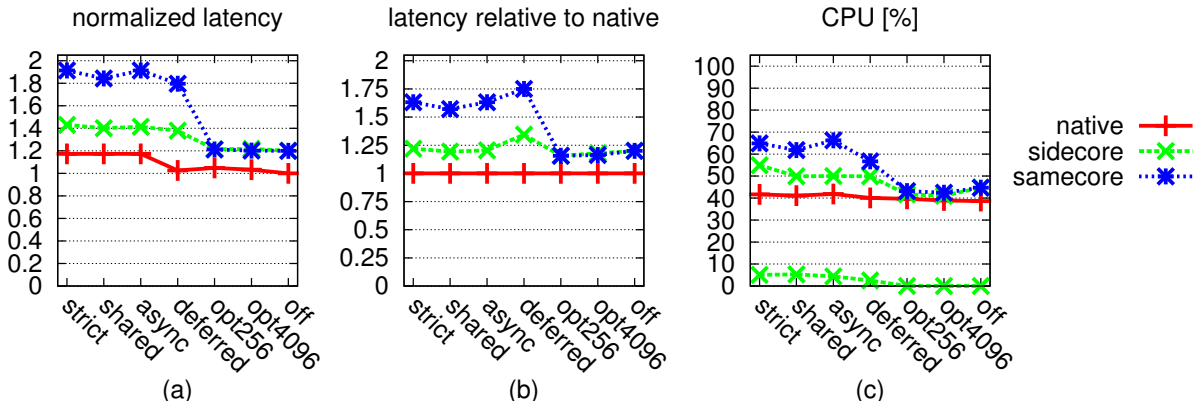


Figure 4: *Measuring latency with the Netperf UDP request-response benchmark; the baseline for normalization (latency of bare metal with no IOMMU) is 41 $\mu$secs.*

optimum for the samecore and sidecore settings is not the "1" that is shown in Figure 4(a); rather, it is the value that is associated with the off configuration of the virtualized settings (guest with no IOMMU protection), which is roughly 1.2 in this case.

Examining Figure 4(c), we unsurprisingly see that the CPU is not a bottleneck for this benchmark. We further see that optimistic teardown is the most significant optimization for this metric, allowing the virtualized settings to nearly reach the bare metal optimum.

Figures 5–6 present the results of the macrobenchmarks, showing trends that are rather similar. Optimistic teardown is most meaningful to the samecore setting, boosting its throughput by about 1.5x. For sidecore, however, the optimization has a lesser effect. Specifically, opt4096 improves upon deferred by 1.07x in the case of MySQL, and by 1.04x in the case of Apache. Before the optimistic teardown is applied, the sidecore setting delivers 1.52x and 1.63x better throughput than samecore for MySQL and Apache, respectively. But once it is applied, then these figures respectively drop to 1.12x and 1.10x. In other words, for the real applications that we have chosen, sidecore is better than samecore by 50%–60% for safe configurations (as well as for deferred), but when optimistic teardown is applied, this gap is reduced to around 10%. This should come as no surprise as we have already established above that optimistic teardown dramatically reduces the IOMMU overhead.

It is important to note, once again, that the optimum for sidecore and samecore is the off configuration in the virtualized setting, namely 0.86 and 0.68 for MySQL and Apache in Figures 5(a) and 6(a), respectively. Thus, it is not that the optimistic teardown all of a sudden became less effective for the macrobenchmarks; rather, it is that in comparison to the microbenchmarks the applications attain much higher throughput to begin with, and so the optimization has less room to shine.

The bottom line is that combining sidecore and op-

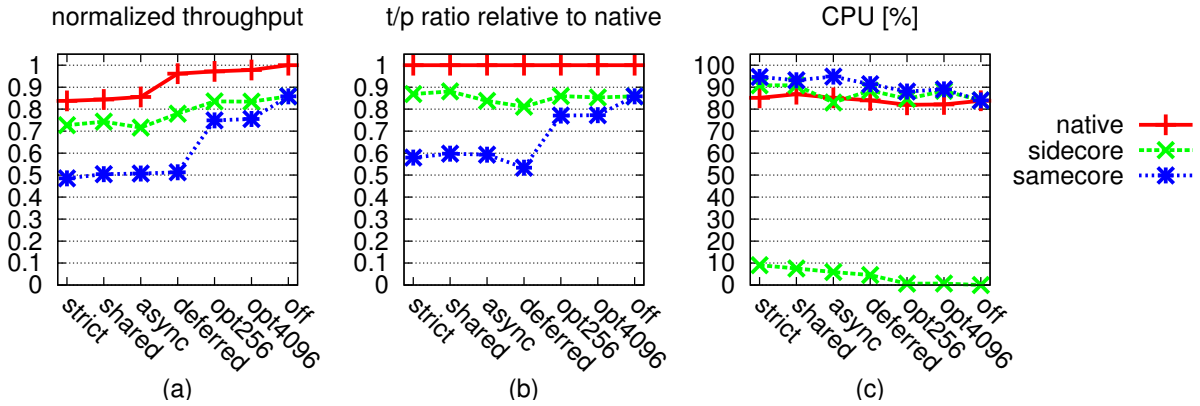


Figure 5: *Measuring MySQL throughput; the baseline for normalization is 243 transactions per second.*

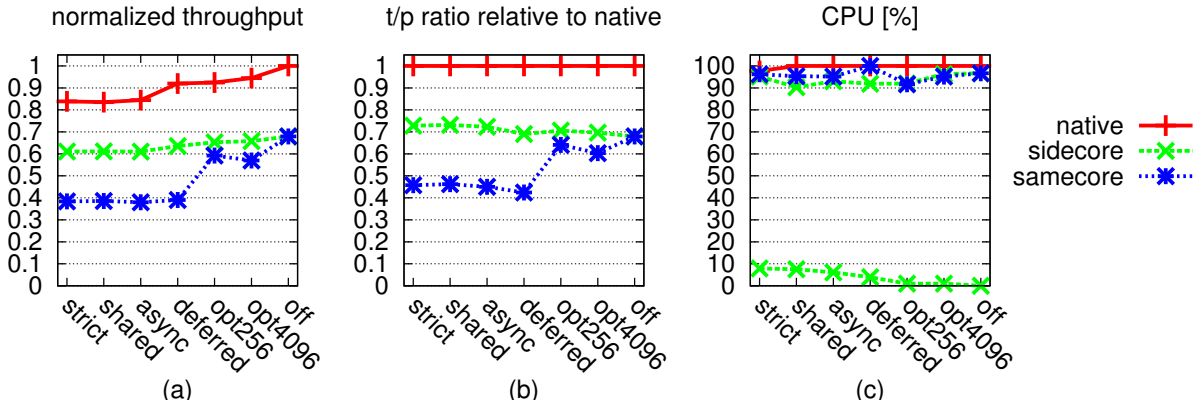


Figure 6: *Measuring Apache throughput; the baseline for normalization is 6828 requests per second.*

| | *Throughput(Mbps)* | *VCPUs load* | *Sidecore load* |
|---|---|---|---|
| samecore | 1345 (+49%) | 76% | |
| sidecore | 4312 (+54%) | 83% | 49% (+50%) |

Table 3: *Measuring the Netperf TCP throughput of 2-VCPUs with strict configuration compared to a single VCPU.*

timistic teardown brings both MySQL and Apache throughputs to be only 3% less than their respective optima.

## 6.4 Sidecore Scalability and Power-Efficiency

Performance gain from the sidecore approach requires the emulating sidecore to be co-scheduled with the VCPUs to achieve low-latency IOMMU emulation. Therefore, it is highly important that the sidecore performs its tasks efficiently with high utilization.

One method for better utilizing the sidecore is to set one emulating sidecore to serve multiple VCPUs or multiple guest CPUs. Table 3 presents the performance of a 2 VCPUs setup, using the strict configuration, relative to a single VCPU setup. As shown, sidecore emulation scales up similarly to samecore emulation, and the performance of both improves by approximately 50% in 2 VCPUs setup.

This method, however, may encounter additional latency in a system that consists multiple sockets (dies), as the affinity of the sidecore thread has special importance in such systems. If both the virtual guest and the sidecore are located on the same die, fast cache-to-cache micro-architectural mechanisms can be used to propagate modifications of the IOMMU data structures, and the interconnect imposes no additional latency. In contrast, when the sidecore is located on a different die, the latency of accessing the emulated IOMMU data structures is increased by interconnect imposed latency. The Intel QuickPath Interconnect (QPI) protocol used on our system requires write-backs of modified cache lines to main memory, which results in latency that can exceed 100ns—over four times the latency of accessing a modi-

fied cache line on the same die [30].

Another method for better utilizing the sidecore is to use its spare cycles productively. Even though the nature of the sidecore is that it is constantly working, a sidecore can have spare cycles—those cycles in which it polled memory and realized it has no pending emulation tasks. One way of improving the system's overall efficiency is to use such cycles for polling paravirtual devices in addition to emulated devices. Another way is to allow the sidecore to enter a low-power sleep state when it is otherwise idle. We can make sidecore IOMMU emulation more power-efficient by using the CPU's monitor/mwait capability, which enables the core to enter a low-power state until a monitored cache range is modified [4].

However, current x86 architecture only enables monitoring of a single cache line, and the Linux scheduler already uses the monitoring hardware for its internal purposes. Moreover, the sidecore must monitor and respond to writes to multiple emulated registers which do not reside in the same cache line.

We overcame these challenges by using the mapping hardware to monitor the *invalidation queue tail* (IQT) register of the IOMMU invalidation queue while we periodically monitored the remaining emulated IOMMU registers. (This is possible because the IOMMU mapping layer performs most of its writes to a certain IQT register.) We also relocated the memory range monitored by the scheduler (the `need_resched` variable) to a memory area which is reserved according to the IOMMU specifications and resides in the same cache line as the IQT register.

Nonetheless, entering a low-power sleep state is suitable only in an extended quiescence period, in which no accesses to the IOMMU take place. This is because entering and exiting low power state takes considerable time [39]. Thus, sidecore emulation is ideally suited for an asymmetric system [22]. Such systems, which include both high power high performance cores and low power low performance cores, can schedule the hardware emulation code to a core which will provide the desired performance/power consumption tradeoff.

The impact of these two scaling related methods, using sidecore to serve a guest whose VCPU is located on another package, and entering low power state instead of polling, appear in Figure 7. According to our experiments, when the sidecore was set on another package, the mapping and unmapping cost increased by 23%, resulting in 25% less TCP throughput than when the sidecore was located on the same package. Entering low-power state increased the cycle cost of mapping and unmapping by 13%, and optimally would decrease performance very little using good heuristics for detecting idle periods. Regardless, in both cases, the cost of sidecore emulation is still considerably lower than that of samecore emulation.

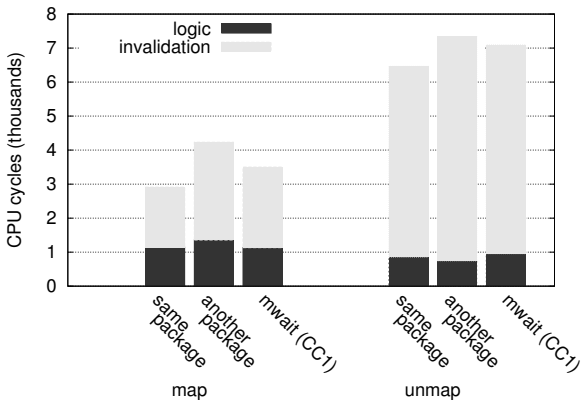


Figure 7: *The effect of power-saving and CPU affinity on the mapping/unmapping cost of a single page.*

## 7 Related Work

We survey related work along the following dimensions: I/O device emulation for virtual machines, IOMMU mapping strategies for paravirtualized and unmodified guests, and offloading computation to a sidecore.

All common hypervisors in use today on x86 systems emulate I/O devices. Sugerman, Venkitachalam, and Lim discuss device emulation in the context of VMware's hypervisor [37], Barham et al. discuss it in the context of the Xen hypervisor [6], Kivity et al. discuss it in the context of the KVM hypervisor [21], and Bellard discusses it in the context of QEMU [10]. In all cases, device emulation suffered from prohibitive performance [11], which led to the development of paravirtualized I/O [6, 34] and direct device assignment I/O [25, 26]. To our knowledge, we are the first to demonstrate the feasibility of high-speed I/O device emulation with performance approaching that of bare metal.

Maximizing OS protection from errant DMAs by minimizing the DMA vulnerability duration is important, because devices might be buggy or exploited [9, 14, 25, 46]. Several IOMMU mapping strategies have been suggested for trading off protection and performance [44, 49]. For unmodified guests, the only usable mapping strategy prior to this work was the direct mapping strategy [44], which provides no protection to the guest OS. Once we expose an emulated IOMMU to the guest OS, the guest OS may choose to use any mapping strategy it wishes to protect itself from buggy or malicious devices.

Additional mapping strategies were possible for paravirtualized guests. The single-use mapping and the shared mapping strategies provide full protection at sizable cost to performance [44]. The persistent mappings strategy provides better performance at the expense of reduced protection. In the persistent mapping strategy

mappings persist forever. The on-demand mapping strategy [49] refines persistent mapping by tearing down mappings once a set quota on the number of mappings was reached. On-demand mapping, however, does not limit the duration of vulnerability. Optimistic teardown provides performance that is equivalent to that of persistent and on-demand mapping, but does so while limiting the duration of vulnerability to mere milliseconds.

Offloading computation to a dedicated core is a well-known approach for speeding up computation [7, 8, 36]. Offloading computation to a sidecore in order to speed up I/O for paravirtualized guests was explored by Kumar et al. [23], Gavrilovska et al. [15], and in the virtualization polling engine (VPE) [27]. In order to achieve near native performance for 10GbE, VPE required modifications of the guest OS and a set of paravirtualized drivers for each emulated device. In contrast, our sidecore emulation approach requires no changes to the guest OS.

Building in part upon vIOMMU, the SplitX project [24] takes the sidecore approach one step further. SplitX aims to run each unmodified guest and the hypervisor on a disjoint set of cores, dedicating a set of cores to each guest and offloading all hypervisor functionality to a disjoint set of sidecores.

## 8 Conclusions

We presented vIOMMU, the first x86 IOMMU emulation for unmodified guests. By exposing an IOMMU to the guest we enable the guest to protect itself from buggy device drivers, while simultaneously making it possible for the hypervisor to overcommit memory. vIOMMU employs two novel optimizations to perform well. The first, "optimistic teardown", entails simply waiting a few milliseconds before tearing down an IOMMU mapping and demonstrates that a minuscule relaxation of protection can lead to large performance benefits. The second, running IOMMU emulation on a sidecore, demonstrates that given the right software/hardware interface and device emulation, unmodified guests can perform just as well as paravirtualized guests.

The benefits of IOMMU emulation rely on the guest using the IOMMU. Introducing software and hardware support for I/O page faults could relax this requirement and enable seamless memory overcommitment even for non-cooperative guests. Likewise, introducing software and hardware support for multiple levels of IOMMU page tables [11] could in theory provide perfect protection without any decrease in performance. In practice, multiple MMU levels cause more page-faults and higher TLB miss-rates, resulting in lower performance for many workloads [42]. Similarly, a single level of IOMMU emulation may perform better than multiple levels of IOMMU page tables, depending on workload.

## Acknowledgments

We thank Ben-Ami Yassour, Abel Gordon, Nadav Har'El, and Alex Landau for their insightful comments and joyful discussions. We also thank the anonymous reviewers and Carl Waldspurger (our shepherd) for their much appreciated feedback. The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement numbers 248615 (IOLanes) and 248647 (ENCORE).

## References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, Aug 2006.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Archit. Support for Prog. Lang. & Operating Syst. (ASPLOS)*, pages 2–13, 2006.

[3] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Workshop on Interaction between Opearting Syst. & Comput. Archit. (WIOSCA)*, 2010.

[4] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pages 1–8, 2008.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS European Conf. on Comput. Syst. (EuroSys)*, pages 75–85, 2006.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 164–177, 2003.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 29–44, 2009.

[8] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *USENIX Workshop on Hot Topics in Operating Syst. (HOTOS)*, page 12, 2009.

[9] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

[10] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Ann. Technical Conf. (ATC)*, pages 41–46, 2005.

[11] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 423–436, 2010.

[12] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symp. (OLS)*, pages 71–86, 2006.

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

[14] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[15] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, 2007.

[16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Comm. of the ACM (CACM)*, 53(10):85–93, Oct 2010.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *IEEE Int'l Conf. on Dependable Syst. & Networks (DSN)*, pages 41–50, 2007.

[18] Intel virtualization technology for directed I/O, architecture specification. `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`, Feb 2011. Revision 1.3. Intel Corporation. (Accessed Apr 2011).

[19] R. Jones. The netperf benchmark. `http://www.netperf.org`. (Accessed Apr, 2011).

[20] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *USENIX Workshop on I/O Virtualization (WIOV)*, page 2, 2008.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symp. (OLS)*, pages 225–230, 2007. `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`. (Accessed Apr, 2011).

[22] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *ACM/IEEE Int'l Symp. on Comput. Archit. (ISCA)*, pages 81–92, 2004.

[23] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Syst. & Comput. Archit. (WIOSCA)*, 2007.

[24] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 17–30, 2004.

[26] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pages 1–12, 2010.

[27] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM Int'l Conf. on Supercomputing (ICS)*, pages 225–234, 2009.

[28] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Ann. Technical Conf. (ATC)*, pages 29–42, 2006.

[29] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 13–23, 2005.

[30] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *ACM/IEEE Int'l Conf. on Parallel Archit. & Compilation Techniques (PACT)*, pages 261–270, 2009.

[31] E. G. Munteanu. AMD IOMMU emulation patchset. KVM mailing list, `http://www.spinics.net/lists/kvm/msg38514.html`, Jul 2010. (Accessed Apr, 2011).

[32] B. Pfaff. Performance analysis of BSTs in system software. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 32(1):410–411, Jun 2004.

[33] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Int'l Symp. on High Performance Distributed Comput. (HPDC)*, pages 179–188, 2007.

[34] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Syst. Review (OSR)*, 42(5):95–103, Jul 2008.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 335–350, 2007.

[36] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Ann. Technical Conf. (ATC)*, page 5, 2010.

[37] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Ann. Technical Conf. (ATC)*, pages 1–14, 2001.

[38] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. (TOCS)*, 23(1):77–110, Feb 2005.

[39] D. Tsafrir, Y. Etsion, and D. G. Feitelson. General purpose timing: the failure of periodic timers. Technical Report 2005-6, School of Computer Science & Engineering, the Hebrew University, Feb 2005.

[40] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX Conf. on File & Storage Technologies (FAST)*, pages 189–206, 2008.

[41] C. A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[42] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 217–226, 2011.

[43] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 241–254, 2008.

[44] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Ann. Technical Conf. (ATC)*, pages 15–28, 2008.

[45] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *IEEE Int'l Symp. on High Performance Comput. Archit. (HPCA)*, pages 306–317, 2007.

[46] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat*, 2008. `http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf`. (Accessed Apr, 2011).

[47] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 31–40, 2009.

[48] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008.

[49] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Haifa Experimental Syst. Conf. (SYSTOR)*, 2010.

[50] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symp. (OLS)*, pages 261–268, 2008.