Protection Strategies for Direct Access to Virtualized I/O Devices

Paul Willmann, Scott Rixner, and Alan L. Cox
Rice University
{willmann, rixner, alc}@rice.edu

Abstract

Commodity virtual machine monitors forbid direct access to I/O devices by untrusted guest operating systems in order to provide protection and sharing. However, both I/O memory management units (IOMMUs) and recently proposed software-based methods can be used to reduce the overhead of I/O virtualization by providing untrusted guest operating systems with safe, direct access to I/O devices. This paper explores the performance and safety tradeoffs of strategies for using these mechanisms.

The protection strategies presented in this paper provide equivalent inter-guest protection among operating system instances. However, they provide varying levels of intra-guest protection from driver software and incur varying levels of overhead. A simple direct-map strategy incurs the least overhead, providing native-level performance but offering no enhanced protection from misbehaving device drivers within the guest operating system. Additional protection against guest drivers can be achieved by limiting IOMMU page-table mappings to memory buffers that are actually used in I/O transfers. Furthermore, the cost incurred by this limitation can be minimized by aggressively reusing these mappings. Surprisingly, a software-only strategy that does not use an IOMMU at all performs competitively, and sometimes better than, hardware-based strategies while maintaining strict inter-guest isolation.

1 Introduction

In many organizations, the economics of supporting a growing number of Internet-based application services has created a demand for server consolidation. Consequently, there has been a resurgence of interest in machine virtualization [1, 2, 5, 8, 9, 10, 14, 19, 22]. However, virtualization can impose performance penalties up to a factor of 5 on I/O-intensive workloads [16, 19]. These penalties stem from the overhead of providing shared and protected access to I/O devices by untrusted guest operating systems. Commodity virtualization architectures provide protection in software by forbidding direct access to I/O hardware by untrusted guests—instead, all I/O accesses, both commands and data, are routed through a single software entity that provides both protection and sharing.

Preferably, guest operating systems would be able to directly access I/O devices without the need for the data to traverse an intermediate software layer within the virtual machine monitor [17, 23]. However, if a guest can directly access an I/O device, then it can potentially direct the device to access memory that it is not entitled to via direct memory access (DMA). Therefore, the virtual machine monitor must be able to ensure that guest operating systems do not access each other's memory indirectly through the shared I/O devices in the system. Both I/O memory management units (IOMMUs) [7] and recently proposed software-based methods [23] can provide DMA memory protection for the virtual machine monitor. They do so by preventing guest operating systems from directing I/O devices to access memory that it is not entitled to access, while still allowing the guest to directly access the device.

These DMA protection mechanisms can also be used by a guest operating system to enhance safety and isolation among its own drivers and processes. The state-of-the-art single-use IOMMU-based protection strategy employed by many existing non-virtualized operating systems provides just such a level of enhanced safety. This strategy creates a mapping for each I/O transaction and then destroys that mapping as soon as the transaction completes. In conjunction with IOMMU hardware, the operating system's protection strategy can exert fine-grained control over what portions of memory may be

This work was supported in part by the National Science Foundation under Grant Nos. CCF-0546140 and CNS-0720878 and by gifts from Advanced Micro Devices and Hewlett-Packard. Paul Willmann was supported in part by SFE Technology, Inc.

used in an I/O transaction at any given time.

This paper explores and experimentally compares five different strategies for ensuring memory isolation of untrusted, virtualized operating systems that each have direct access to I/O hardware. These strategies all ensure isolation among OS instances and the virtual machine monitor, but they vary in the level of protection within a particular guest they can support and the overheads they incur. Though upcoming commodity platforms will feature IOMMUs for efficient I/O virtualization, there exists no comprehensive study about how to best leverage IOMMUs, what the tradeoffs are among efficiency and protection for different possible strategies, and what the comparative costs associated with the various protection strategies are.

The first IOMMU-based strategy is based on state-of-the-art strategies and uses single-use I/O memory mappings that are created before each I/O operation and immediately destroyed after each I/O operation. The second IOMMU-based strategy is introduced in this paper and uses shared I/O memory mappings that can be reused by multiple, concurrent I/O operations. The third IOMMU-based strategy is also introduced in this paper and uses persistent I/O memory mappings that can be reused. The fourth IOMMU-based strategy uses a static direct map of the guest's physical memory to isolate that guest's I/O transactions. Finally, the software-based strategy is based on previous work [23] and uses VMM-managed DMA descriptors that can only be used for one I/O operation.

The comparison of these five strategies yields several insights. First, all five strategies provide equivalent inter-guest protection among OS instances. However, the strategies support differing levels of protection within a particular guest (*intra-guest* protection). For example, the direct-map strategy incurs almost no performance overhead but supports no intra-guest protection. Conversely, the single-use strategy provides the maximum possible intra-guest protection, but it imposes the largest performance penalty. Second, there is significant opportunity to reuse IOMMU mappings, which can reduce protection overheads. Multiple concurrent I/O operations are able to share the same mappings often enough that there is a noticeable decrease in the overhead of providing protection. Sharing mappings only among concurrent I/O operations provides the same level of intra-guest protection as the single-use strategy but with less overhead. Relaxing this intra-guest protection guarantee by allowing mappings to persist so that they can be reused in future I/O operations can significantly decrease this overhead, allowing the guest to achieve performance levels very close to that of the direct-map strategy while still maintaining some amount of intra-guest protection. Finally, the software-based protection strategy performs competitively with several of the better-performing IOMMU-based strategies while maintaining strong inter-guest protection guarantees and enabling intra-guest protection capabilities.

The next section provides background on how I/O devices access main memory and the possible memory protection violations that can occur when doing so. Sections 3 and 4 discuss the four IOMMU-based protection strategies and the one software-based protection strategy. Section 5 then describes the protection properties afforded by the five strategies. Section 6 discusses IOMMU hardware architectures. Section 7 describes the experimental methodology and Section 8 evaluates the protection strategies. Section 9 then describes related work and Section 10 concludes the paper.

2 Background

Modern server I/O devices, including disk and network controllers, utilize direct memory access (DMA) to move data between the host's main memory and the device's on-board buffers. The device uses DMA to access memory independently of the host CPU, so such accesses must be controlled and protected. To initiate a DMA operation, the device driver within the operating system creates *DMA descriptors* that refer to regions of memory. Each DMA descriptor typically includes an address, a length, and a few device-specific flags. In commodity x86 systems, devices lack support for virtual-to-physical address translation, so DMA descriptors always contain physical addresses for main memory. Once created, the device driver passes the descriptors to the device, which will later use the descriptors to transfer data to or from the indicated memory regions autonomously. When the requested I/O operations have been completed, the device raises an interrupt to notify the device driver.

For example, to transmit a network packet, the network interface's device driver might create two DMA descriptors. The first descriptor might point to the packet headers and the second descriptor might point to the packet payload. Once created, the device driver would then notify the network interface that there are new DMA descriptors available. The precise mechanism of that notification depends on the particular network interface, but typically involves a programmed I/O operation to the device telling it the location of the new descriptors. The network interface would then retrieve the descriptors from main memory using DMA—if they were not written to the device directly by programmed I/O. The network interface would then retrieve the two memory regions that compose the network packet and transmit them over the network. Finally, the network interface would interrupt the host to indicate that the packet has been transmitted. In practice, notifications from the device

driver and interrupts from the network interface would likely be aggregated to cover multiple packets for efficiency.

Three potential memory access violations can occur on every I/O transfer initiated using this DMA architecture:

1. The device driver could create a DMA descriptor with an incorrect address (a "bad-address" fault).

2. The operating system could repurpose the memory referenced by a DMA descriptor, or the device driver could later reuse a valid DMA descriptor without permission (an "invalid-use" fault).

3. The device itself could initiate a DMA transfer to a memory address not referenced by the DMA descriptor (a "bad-device" fault).

These violations could occur either because of failures or because of malicious intent. However, as devices are typically not user-programmable, the last type of violation is only likely to occur as a result of a device failure.

In a non-virtualized environment running on commodity x86 hardware, the operating system is solely responsible for preventing "bad-address" and "invalid-use" violations. This requires the operating system to trust the device driver to create correct DMA descriptors using only physical memory addresses of buffers that have been pinned by the OS. In practical terms, however, trusting the driver can be disastrous in terms of system stability. For example, nearly 85% of all system crashes of the Windows XP operating system are caused by drivers [20]. As will be discussed, operating systems running on platforms that feature IOMMUs can leverage those hardware capabilities to isolate device drivers and explicitly authorize the I/O transactions requested by the driver, thus reducing the trust requirement for the driver. Regardless, a failure of the operating system to prevent these memory access violations could potentially result in system failure.

In a virtualized environment, however, the virtual machine monitor cannot trust the guest operating systems to prevent these memory access violations, as a memory access violation incurred by one guest operating system can potentially harm other guest operating systems or even bring down the whole system. Therefore, a virtual machine monitor requires mechanisms to prevent one guest operating system from intentionally or accidentally directing an I/O device to access the memory of another guest operating system. The only way that would be possible is via either a "bad-address" or "invalid-use" violation. Depending on the reliability of the I/O devices, it may also be desirable to try to prevent "bad-device" violations as well (although it is frequently not possible to protect against a misbehaving device, as will be discussed in Section 5). The following sections describe mechanisms and strategies for preventing these memory access violations.

3 IOMMU-based Protection

A VMM can utilize an IOMMU to help provide DMA memory protection when allowing direct access to I/O devices. Whereas a virtual memory management unit enforces access control and provides address translation services for a processor as it accesses memory, an IOMMU enforces access control and provides address translation services for an I/O device as it accesses memory.

In general, the structures defined by IOMMUs for expressing access control and address translation are fairly similar to those defined by virtual memory management units. Typically, the VMM or OS maintains one or more page table structures for use by the IOMMU. There are, however, some differences. The VMM or OS must also maintain another structure for the IOMMU that maps each I/O device to one of the page tables. Although every device must have an entry in this mapping, not every device must have its own page table. If two or more devices are entitled to access the same memory, then they can share a single page table. On every memory access by an I/O device the IOMMU consults the I/O device's designated page table to determine if the access should be allowed or blocked.

Regardless of the page-table organization, all IOMMU-based systems require that a valid IOMMU mapping exists for each host memory buffer to be used in an upcoming DMA descriptor. Otherwise, the DMA descriptor will refer to a region unmapped by the IOMMU, and the I/O transaction will fail. The following subsections present four strategies for using an IOMMU to provide DMA memory protection in a VMM. The strategies primarily differ in the extent to which IOMMU mappings are allowed to be reused. The underlying IOMMU hardware architectures that may be used to implement these strategies are discussed in more detail in Section 6

3.1 Single-use Mappings

A common strategy for managing an IOMMU is to create a single-use mapping for each I/O transaction. The Linux `DMA-Mapping` interface, for example, implements a single-use mapping strategy. Ben-Yehuda, *et al.* also explored a single-use mapping strategy in the context of virtual machine monitors [7]. In such a single-use strategy, the driver must ensure that a new IOMMU mapping

is created for each DMA descriptor. Even, when separate DMA descriptors refer to the same physical page, the single-use strategy always creates distinct IOMMU mappings for each descriptor. Each IOMMU mapping is destroyed once the corresponding I/O transaction has completed. In a virtualized system, the trusted virtual machine monitor is responsible for creating and destroying IOMMU mappings at the driver's request. If the VMM does not create the mapping, either because the driver did not request it or because the request referred to memory not owned by the guest, then the device will be unable to perform the corresponding DMA operation.

To carry out an I/O transaction using a single-use mapping strategy, the virtual machine monitor (VMM), untrusted guest operating system (GOS), and the device (DEV) carry out the following steps:

1. GOS: The guest OS requests an IOMMU mapping for the memory buffer involved in the I/O transaction.

2. VMM: The VMM validates that the requesting guest OS has appropriate read or write permission for each memory page in the buffer to be mapped.

3. VMM: The VMM marks the memory buffer as "in I/O use", which prevents the buffer from being reallocated to another guest OS during an I/O transaction.

4. VMM: The VMM creates one or more IOMMU mappings for the buffer. As with virtual memory management units, one mapping is usually required for each memory page in the buffer.

5. GOS: The guest OS creates a DMA descriptor with the IOMMU-mapped address that was returned by the VMM.

6. DEV: The device carries out its I/O transaction as directed by the DMA descriptor and it notifies the driver upon completion.

7. GOS: The driver requests destruction of the corresponding IOMMU mapping(s).

8. VMM: The VMM validates that the mappings belong to the guest OS making the request.

9. VMM: The VMM destroys the IOMMU mappings.

10. VMM: The VMM clears the "in I/O use" marker associated with each memory page referred to by the recently-destroyed mapping(s).

3.2 Shared Mappings

Rather than creating a new IOMMU mapping for each new DMA descriptor, it is possible to share a mapping among DMA descriptors so long as the mapping points to the same underlying memory page and remains valid. Unlike the single-use strategy, the shared-mapping strategy detects when a valid IOMMU mapping to a memory page already exists and reuses that mapping rather than generating a new one. Sharing IOMMU mappings is advantageous because it avoids the overhead of creating and destroying a new mapping for each I/O request. In practical terms, this sharing can happen when an application repeats the same I/O message or when an application sends or receives small I/O messages that reside in the same memory page.

To implement sharing, the guest operating system must keep track of which IOMMU mappings are currently valid, and it must keep track of how many pending I/O requests are currently using the mapping. To protect a guest's memory from errant device accesses, an IOMMU mapping should be destroyed once all outstanding I/O requests that use the mapping have been completed. Though the untrusted guest operating system has responsibilities for carrying out a shared-mapping strategy, it need not function correctly to ensure isolation among operating systems, as is discussed further in Section 5.

To carry out a shared-mapping strategy, the guest OS and the VMM perform many of the same steps that are required by the single-use strategy. The shared-mapping strategy differs at the initiation and termination of an I/O transaction. Before step 1 would occur in a single-use strategy, the guest operating system first queries a table of known, valid IOMMU mappings to see if a mapping for the I/O memory buffer already exists. If so, the driver uses the previously established IOMMU-mapped address for a DMA descriptor, and then passes the descriptor to the device, in effect skipping steps 1–4. If not, the guest and VMM follow steps 1–4 to create a new mapping. Whether a new mapping is created or not, before step 5, the guest operating system increments its own reference count for the mapping. This reference count is separate from the reference count maintained by the VMM.

Steps 5 and 6 then proceed as in the single-use strategy. After these steps have completed, the driver calls the guest operating system to decrement its reference count. If the reference count is zero, no other I/O transactions are in progress that are using this mapping, so the guest calls the VMM to destroy the mapping, as in steps 7–10 of the single-use strategy. Otherwise, the IOMMU mapping is still being used by another I/O transaction within the guest OS, so steps 7–10 are skipped.

3.3 Persistent Mappings

IOMMU mappings can further be reused by allowing them to persist even after all I/O transactions using the mapping have completed. Compared to a shared mapping strategy, such a persistent mapping strategy attempts to further reduce the overhead associated with creating and destroying IOMMU mappings inside the VMM. Whereas sharing exploits reuse among mappings only when a mapping is being actively used by at least one I/O transaction, persistence exploits temporal reuse across periods of inactivity.

The infrastructure and mechanisms for implementing a persistent mapping strategy are similar to those required by a shared mapping strategy. The primary difference is that the guest operating system does not request that mappings be destroyed after the I/O transactions using them complete. Therefore, in contrast to the shared mapping strategy, when the guest's reference count is decremented after step 6, the I/O transaction is complete and steps 7–10 are always skipped. This should dramatically reduce the number of potentially costly invocations of the VMM.

Eventually, it is possible that all of a guest's memory would become mapped using this strategy. Compared to a shared mapping strategy, this increases the guest's exposure to intra-guest protection violations, which will be discussed in Section 5.2. To limit this exposure, the guest operating system can implement a reclamation policy that eventually removes mappings that are not currently in use by an I/O operation. For example, the simple reclamation policy that is used in the experiments of this paper limits the total number of mappings. Once this total is reached, a mapping that is not currently in use would have to be destroyed before a new mapping can be created.

3.4 Direct Mappings

To allow maximum reuse of IOMMU mappings and to further reduce runtime overhead, it is possible to permanently map the entire physical address space of the guest operating system. Such a strategy is sometimes referred to as a *direct map*, because this arrangement creates a one-to-one mapping between IOMMU entries and physical pages for each physical page owned by the guest operating system.

4 Software-based Protection

IOMMU-based protection strategies enforce safety even when untrusted software provides unverified DMA descriptors directly to hardware, because the DMA operations generated by any device are always subject to later validation. However, an IOMMU is not necessary to ensure full isolation among untrusted guest operating systems, even when they use DMA-capable hardware that directly reads and writes host memory. Rather than relying on hardware to perform late validation during I/O transactions, a lightweight software-based system performs early validation of DMA descriptors before they are used by hardware. The software-based strategy also must protect validated descriptors from subsequent unauthorized modification by untrusted software, thus ensuring that all I/O transactions operate only on buffers that have been approved by the VMM. This software-based strategy was previously introduced as a means for ensuring DMA memory protection by untrusted guest operating systems that have concurrent direct access to a prototype network interface [23].

The runtime operation of a software-based protection strategy works much like a single-use IOMMU-based strategy, since both validate permissions for each I/O transaction. Whereas the single-use IOMMU-based strategy uses the VMM to create IOMMU mappings for each transaction, software-based I/O protection creates the actual DMA descriptor. The descriptor is valid only for the single I/O transaction. Unlike an IOMMU-based system, an untrusted guest OS's driver must first register itself with the VMM during initialization. At that time, the VMM takes ownership of the driver's DMA descriptor region and the driver's status region, revoking write permissions from the guest. This prevents the guest from independently creating or modifying DMA descriptors, or modifying the status region. Finally, the VMM must prevent the guest from changing the descriptor and status regions. This can be accomplished by only mapping the device's configuration registers into the VMM's address space, and not into the guests' address spaces.

After initialization, the operation of the software-based strategy is similar to the single-use IOMMU-based strategy outlined in Section 3.1. Steps 1–3 of a software-based strategy are nearly identical, with the exception that the Guest OS is requesting a DMA descriptor, not an IOMMU mapping, in Step 1. In step 4, the VMM creates a DMA descriptor in the write-protected DMA descriptor region, obviating the OS's role in step 5. The device carries out the requested operation using the validated descriptor, as in step 6, and because the descriptor is write-protected, the untrusted guest cannot modify the descriptor and thus cannot induce a transaction that has not been explicitly authorized by the VMM. When the device signals completion of the transaction, the VMM inspects the device's state (which is usually written via DMA back to the host) to see which DMA descriptors have been used. The VMM then processes those completed descriptors, as in step 10, permitting the associated guest memory buffers to be reallocated.

	Inter-Guest			Intra-Guest		
	Bad Address	**Invalid Use**	**Bad Device**	**Bad Address**	**Invalid Use**	**Bad Device**
Direct-map	x	x	x			
Single-use	x	x	x	x		x
Shared	x	x	x	x		x
Persistent	x	x	x	x		
Software	x	x		x	x	

Table 1: Types of protection supported by the different DMA protection strategies.

In contrast to the IOMMU-based strategies, the software-based strategy requires that the VMM actually insert DMA descriptors to the I/O device. This requires that the VMM know the method of insertion (*i.e.*, programmed I/O or DMA) and the structure of the DMA descriptor for a particular device. Furthermore, the VMM must be able to determine the descriptor state of the device. Descriptor state specifies whether or not the device can accept more descriptors and indicates when previously posted descriptors have been processed. Though these device-specific requirements inherently require some device-specific methods in the VMM, the general method of software protection described here apply to DMA-capable devices in general. As described in greater detail in [23], the implementation for software-based DMA protection described here applies to devices that organize their DMA descriptors in contiguous rings, which includes many high-performance devices.

5 Protection Properties

The protection strategies presented in Sections 3 and 4 can be used to prevent the memory access violations presented in Section 2. Those violations can happen by creating a DMA descriptor using a bad address, by repurposing and reusing a DMA descriptor after its initial use, or by suffering a fault inflicted by a malfunctioning I/O device. These violations can occur both across multiple guests (inter-guest) and within a single guest (intra-guest). A virtual machine monitor must, at minimum, provide inter-guest protection in order to operate reliably. A guest operating system may additionally benefit if the system hardware or the virtual machine monitor can be used to help provide intra-guest protection. This section describes the protection properties of the five previously presented protection strategies. Table 1 summarizes these faults and shows which strategy prevents what faults in both the inter-guest and intra-guest cases.

5.1 Inter-Guest Protection

Perhaps surprisingly, all five strategies provide equivalent inter-guest protection against "bad-address" and "invalid-use" faults. In all of the IOMMU-based strategies, if the device driver creates a DMA descriptor that refers to memory that is not owned by that guest operating system, the device will be unable to perform that DMA, as no IOMMU mapping will exist. The only requirement to maintain this protection is that the VMM must never create an IOMMU mapping for a guest that does not refer to that guest's memory. Similarly, only the VMM can repurpose memory to another guest, so as long as it does not do so while there is an existing IOMMU mapping to that memory, inter-guest "invalid-use" faults can never occur. The software-based approach provides exactly the same guarantees by only allowing the VMM to create DMA descriptors. Therefore, these strategies allow the VMM to provide protection.

"Bad-device" faults are more difficult to prevent. If the device is shared among multiple guest operating systems, then no strategy can prevent this type of fault. For example, if a network interface is allowed to receive packets for two guest operating systems, the VMM cannot prevent the device from sending the traffic destined for one guest to the other. This is one simple example of many the many problems that a shared device can cause.

However, if a device is privately assigned to a single guest operating system, the IOMMU-based strategies can be used to provide protection against faulty device behavior. In this case, the VMM simply has to ensure that there are only IOMMU mappings to the guest that is assigned the device. In this manner, all four IOMMU-based strategies can protect against such a fault. However, the software-based strategy cannot provide this level of protection. Though DMA descriptors are validated by the VMM to ensure that they only point to memory for which the associated guest has appropriate permissions, there is no software-only mechanism capable of stopping the device from simply ignoring the DMA descriptor and accessing any physical memory.

5.2 Intra-Guest Protection

As is shown in Table 1, the five protection strategies discussed in this paper vary significantly according to how they may be used by the guest OS to prevent intra-guest

faults of the types listed in Section 2. In order to prevent a driver from creating a DMA descriptor with the wrong address or using memory that has been repurposed by the guest OS (i.e., errors of the first two types), the OS must implement its own isolation strategy for inspecting and verifying each I/O transaction. Typically, operating systems designed for commodity platforms simply trust that the driver will request a valid, current address for an I/O transaction.

To enable the OS to protect itself from drivers that may construct DMA descriptors using bad intra-guest addresses, the OS must be able to act as the gate-keeper to I/O translations and thus DMA addresses. Unlike the other four strategies, the direct-map strategy ensures that all of a guest's memory is mapped in the IOMMU at any given time, and thus the driver does not have to request specific permission for I/O memory from the OS. In this direct-map case, it is possible for the driver to create a valid DMA descriptor to an arbitrary region of the guest's memory. This I/O transaction would succeed with the blessing of the IOMMU so long as the memory location is somewhere within the guest's memory, even though the OS may have never approved use of this location for I/O. In all the other strategies, at least one specific request for memory by the driver is required before the OS will approve construction of the necessary IOMMU mapping, and hence, all the other strategies can protect against intra-guest "bad-address" faults.

Once the first request to create the IOMMU mapping has happened, however, none of the IOMMU-based strategies can prevent a driver from invalidly reusing that same mapping for a subsequent I/O transaction. In these strategies, the driver is responsible for informing the OS when it is done with an IOMMU mapping. Even if the OS was modified to automatically revoke an IOMMU mapping when it detected the completion of a corresponding I/O event (as in the completion of a `sendfile()` operation or the `free()` of an `skbuff`), the driver could still invalidly reuse a mapping after the original I/O event finished, but before the OS could intervene to terminate the IOMMU mapping. In the software strategy, however, the VMM automatically detects when the device has completed a specific I/O transaction and ensures that individual DMA descriptors can never be reused. Thus, the software-based DMA protection strategy can be used by an operating system to detect and then prevent "invalid-use" faults.

Preventing "bad-device" faults from corrupting an individual guest's memory requires that the device can only access that guest's memory while a valid I/O transaction is in-flight and has been authorized by the OS. The direct-map strategy is incapable of preventing these faults because it permanently maps all of a guest's memory for I/O, and thus it may be used for I/O at any given time. The persistent strategy allows IOMMU mappings to exist in a valid state even after I/O transactions have completed, and thus in this time period, it is possible for a device fault to access one of those mapped locations and corrupt memory. As in the inter-guest case, the software-based mechanism has no hardware enforcement mechanism that could prevent the device from initiating an invalid transfer. Conversely, both the single-use and shared-mapping strategies are specifically designed to only permit valid IOMMU mappings to exist during active I/O transactions, and hence they both guard against device faults.

6 IOMMU Architectures

As discussed in Section 3, an IOMMU performs memory translation and protection for I/O devices. For each direct memory access (DMA) performed by an I/O device, the IOMMU validates that the device is allowed to access that memory and translates the memory address appropriately. If the device is not allowed to access that memory or there is no valid translation, then the IOMMU terminates the DMA transaction.

A graphics address relocation table (GART) provides similar memory translation functionality for I/O devices. A GART translates memory addresses within a contiguous range in the physical address space, called the GART *aperture*. Unlike an IOMMU, only addresses that fall within this aperture are translated by the GART. This translation functionality is typically used by graphics software libraries to simplify memory accesses performed by the grapics card. Operating systems also use GART hardware to provide translation of addresses for devices that only support 32-bit addressing on 64-bit platforms. Though the GART translates memory addresses in a similar manner to an IOMMU, the GART does not protect memory from the device. Devices can still access all physical memory directly using addresses outside of the GART aperture.

Table 2 shows the performance differences between the AMD Opteron with an integrated GART and two modern IOMMU platforms, the IBM Calgary platform and the Intel VT-d architecture. The table shows the average cost, in processor cycles, to update an I/O page table (PT) entry, to flush the platform's I/O translation lookaside buffer (IOTLB) which caches translations, and to both update an I/O page table entry and then immediately flush the IOTLB. The Calgary, VT-d and GART platforms feature processors operating at 2.5, 2.66, and 2.4 GHz, respectively. The performance differences among the platforms arise directly from the differing page table organizations and IOTLB overheads.

As the table shows, the Calgary platform has the highest overhead to install or modify a translation for a single

Platform	I/O PT Update	IOTLB Flush	Update & Flush
IBM Calgary IOMMU	673	10207	10887
Intel VT-d IOMMU	991	1217	2213
AMD GART	27	486	579

Table 2: Microbenchmarks examining costs associated with modern translation hardware, in processor cycles.

memory page. This is primarily caused by its unusually high cost for flushing the system's IOTLB, which is discussed at length by Ben-Yehuda *et al*. [6]. The Intel VT-d architecture is more efficient, but its multilevel page table and its IOTLB are still more expensive to update than the flat page table organization of the GART and its simpler IOTLB interface.

The Opteron's GART was used to evaluate the protection strategies presented in this paper for two reasons. First, at the time this research began, there were no IOMMUs available for x86-based systems—the AMD Opteron with a GART was the only suitable device. Second, as Table 2 shows, the currently available IOMMUs for x86-based systems, Calgary and VT-d, have higher overheads than the GART. The only implication of this choice is that the strategies that perform more frequent mappings, single-use and shared, perform better than they would with a higher-overhead IOMMU. In fact, Ben-Yehuda, *et al*. found that the single-use strategy had higher overhead than found in this paper [7].

7 Experimental Setup

The protection strategies described in Sections 3 and 4 were evaluated on a system with an AMD Opteron 250 processor. The Opteron's GART is used to model the behavior and functionality of IOMMU hardware, as described in the previous section. The IOMMU- and software-based protection strategies are implemented in the open source Xen 3 virtual machine monitor [5]. Xen differs from many virtualization systems in that it exposes host physical addresses to the guest OS. In particular, the guest OS, and not the VMM, is responsible for translating between pseudo-physical addresses that are used at most levels of the guest OS and host physical addresses that are used at the device level. This does not, however, fundamentally change the implementation of the various protection strategies.

We evaluate these strategies on a variety of network-intensive workloads, including a TCP stream microbenchmark, a voice-over-IP (VoIP) server benchmark, and a static-content web server benchmark. The stream microbenchmark either transmits or receives bulk data over a TCP connection to a remote host. The VoIP benchmark uses the OpenSER server. In this benchmark, OpenSER acts as a SIP proxy and 50 clients simultaneously initiate calls as quickly as possible. The web server benchmark uses the lighttpd web server to host static HTTP content. In this benchmark, 32 clients simultaneously replay requests from various web traces as quickly as possible. Three web traces are used in this study: "CS", "IBM", and "WC". The CS trace is from the Rice University computer science departmental web server and has a working set of 1.2 GB of data. The IBM trace is from an IBM web server and has a working set of 1.1 GB of data. The WC trace is from the 1998 World Cup soccer web server and has a working set of 100 MB of data. For all benchmarks, the client machine is directly connected to the server without the use of a switch, and the client is monitored to ensure that it is never saturated. Hence, it is assured that the server machine is always the bottleneck. Each benchmark (TCP stream, VoIP, and each web benchmark) is tested using the specified strategy a minimum of 5 times each. The performance reported is average performance, because there is no significant variance across runs.

The server under test has two Gigabit Ethernet network interface cards and features DDR 400 DRAM. The network interfaces are publicly available prototypes that are user-programmable, support shared direct access by virtual machines, and support line-rate Gigabit Ethernet speeds [18]. For each configuration except the direct-map strategy, a single unprivileged guest operating system has 1.4 GB of memory, and the IOMMU-based strategies employ 512 MB of physical GART address space for remapping (which corresponds to 131,072 unique mappings). For the Direct-map strategy, we use a guest operating system with 512 MB of memory. We simplify the implementation of the Direct-map strategy to its minimum possible overhead by pre-mapping the entire guest's physical memory space permanently at boot-time. Hence, this model represents the minimum possible I/O overhead on this platform, since no mappings are created, destroyed, or modified during the experiments. Its limited memory footprint, however, prevents a fair evaluation of web-based workloads that have a working set larger than 512 MB, including the IBM and CS traces, and hence those benchmarks are not evaluated using the Direct-map strategy.

In each benchmark, direct access for the guest to the hardware is granted only for the network interface cards. Because the guest's memory allocation is large enough to hold each benchmark and its corresponding data set, other I/O is insignificant. For the web-based workloads, the guest's buffer cache is warmed prior to the tests.

Protection Strategy	CPU %		Reuse (%)		HC/ DMA
	Total	Prot.	TX	RX	
Stream Transmit					
None	41	0	N/A	N/A	0
Direct-map	41	0	N/A	N/A	0
Single-use	64	23	N/A	N/A	.88
Shared	58	17	39	0	.55
Persistent	43	2	100	100	0
Software	56	15	N/A	N/A	.90
Stream Receive					
None	53	0	N/A	N/A	0
Direct-map	54	0	N/A	N/A	0
Single-use	79	26	N/A	N/A	.37
Shared	73	20	39	0	.10
Persistent	59	5	100	100	0
Software	64	11	N/A	N/A	.39

Table 3: TCP Stream Profile.

8 Evaluation

Network server applications can stress network I/O in different ways, depending on the characteristics of the application and its workload. Applications may generate large or small network packets, and may or may not utilize zero-copy I/O. For an application running on a virtualized guest operating system, these network characteristics interact with the I/O protection strategy implemented by the VMM. Consequently, the efficiency of the I/O protection strategy can affect application performance in different ways. Furthermore, the application's behavior for a given workload can directly affect the amount of mapping reuse that is exploitable by a given strategy. We first provide an overview of performance and efficiency under several different network workloads, and then we discuss the sources of mapping reuse for the different workloads.

For all applications, we evaluate the five protection strategies presented earlier, and we compare each to the performance of a system lacking any I/O protection at all ("None"). "Single-use", "Shared", "Persistent", and "Direct-map" all use an IOMMU to enforce protection, using either single-use, shared-mapping, persistent-mapping, or direct-mapping strategies, respectively, as described in Section 3. "Software" uses software-based I/O protection, as described in Section 4.

8.1 TCP Stream

A TCP stream microbenchmark either transmits or receives bulk TCP data and thus isolates network I/O performance. This benchmark does not use zero-copy I/O. Table 3 shows the CPU efficiency and overhead associated with each protection mechanism when streaming data over two network interfaces. The table shows the total percentage of CPU consumed while executing the benchmark and the percentage of CPU spent implementing the given protection strategy. The table also shows the percentage of times a buffer to be used in an I/O transaction (either transmit or receive) already has a valid IOMMU mapping that can be reused. Finally, the table shows the number of VMM invocations, or hypercalls (HC), required per DMA descriptor used by the network interface driver.

When either transmitting or receiving, all of the strategies achieve the same TCP throughput (1865 Mb/s transmitting, 1850 Mb/s receiving), but they differ according to how costly they are in terms of CPU consumption. The single-use protection strategy is the most costly, with its repeated construction and destruction of IOMMU mappings consuming 23% of total CPU resources for transmit and 26% for receive. The shared strategy reclaims some of this overhead through its sharing of in-use mappings, though this reuse only exists for transmitted packets (data in the transmit-stream case, TCP ACK packets in the receive case). The lack of reuse for received packets is caused by the paravirtualized (PV) Linux buffer allocator, which dedicates an entire 4 KB page for each receive buffer, regardless of the buffer's actual size. This over-allocation is an artifact of the PV-Linux I/O architecture, which was designed to remap received packets to transfer them between guest operating systems.

Regardless, the persistent strategy achieves 100% reuse of mappings, as the small number of persistent mappings that cover network buffers essentially become permanent. This further reduces overhead relative to single-use and shared. Notably, the number of hypercalls per DMA operation rounds to zero. However, detailed L2 cache profile statistics (not shown in the table) reveals that management of the persistent mappings—mapping lookup and reclamation, as described in Section 3.3—incurs additional overhead in the processor's memory system. This consumes 2% of the processor resources for the transmit-based workload and 5% for the receive-based workload. The direct-map strategy does not require any protection management at runtime and has the same measured CPU utilization as the "None" case for the transmit case. However, the direct-map strategy incurs a small overhead in the receive case. This represents the measured overhead on the system of simply using the GART for I/O transactions rather than using non-translated addresses. However, through extensive reuse of existing IOMMU mappings, the persistent-mapping strategy achieves nearly the same efficiency as the direct-map case.

Surprisingly, the overhead incurred by the software-based technique is noticeably less than the IOMMU-based shared-mapping and single-use strategies. The software-based technique certainly requires far more hy-

Protection Strategy	Calls/ Sec.	CPU % Prot.	Reuse (%)		HC/ DMA
			TX	RX	
None	3005	0	N/A	N/A	0
Direct-map	2997	0	N/A	N/A	0
Single-use	2790	6.1	N/A	N/A	.68
Shared	2835	6.0	4	0	.65
Persistent	2997	0.1	100	100	0
Software	2895	3.5	N/A	N/A	.67

Table 4: OpenSER Profile.

percalls per DMA than the IOMMU-based strategies. The cost of those VMM invocations and the associated page-verification operations is similar to the cost of inspecting mapping requests for shared- and single-use strategies. However, the software strategy does not incur the additional overhead of flushing the IOMMU's IOTLB via a programmed-I/O write, as is required with the other strategies whenever a group of changes to an I/O page table must be committed.

8.2 VoIP Server

Table 4 shows the performance and overhead profile for the OpenSER VoIP application benchmark for the various protection strategies. The OpenSER benchmark is largely CPU-intensive and therefore only uses one of the two network interface cards. Though the strategies rank similarly in efficiency for the OpenSER benchmark as in the TCP Stream benchmark, Table 4 shows one significant difference with respect to reuse of IOMMU mappings. Whereas the shared strategy was able to reuse mappings 39% of the time for transmit packets under the TCP Stream benchmark, OpenSER sees only 4% reuse. Unlike typical high-bandwidth streaming applications, OpenSER only sends and receives very small TCP messages in order to initiate and terminate VoIP phone calls. Consequently, the shared strategy provides only a minimal efficiency and performance improvement over the high-overhead single-use strategy for the OpenSER benchmark, indicating that sharing alone does not provide an efficiency gain for applications that are heavily reliant on small messages.

8.3 Web Server

Table 5 shows the performance, overhead, and sharing profiles of the various protection strategies when running a webserver under each of three different trace workloads, "CS", "IBM", and "WC". As in the TCP Stream and OpenSER benchmarks, the different strategies rank identically among each other in terms of performance and overhead. Note that the direct-map strategy is evaluated only for the "WC" trace, since it is the only web trace whose workload will fit entirely within the direct-map configuration's smaller guest operating system memory allocation. Each of the different traces generates messages of different sizes and requires different amounts of web-server compute overhead. For the `write()`-based implementation of the web server, however, the server is always completely saturated for each workload shown. "CS" is primarily network-limited, generating relatively large response messages with an average HTTP message size of 34 KB. "IBM" is largely compute-limited, generating relatively small HTTP responses with an average size of 2.8 KB. "WC" lies in between, with an average response size of 6.7 KB. As the table shows, the amount of reuse exploited by the shared strategy is dependent on the average HTTP response being generated. Larger average messages lead to larger amounts of reuse for transmitted buffers under the shared strategy. Though larger amounts of reuse slightly reduce the CPU overhead for the shared strategy relative to the single-use strategy, the reuse is not significant enough under these workloads to yield significant performance benefits.

Protection Strategy	HTTP Mbps	CPU % Prot.	Reuse (%)		HC/ DMA
			TX	RX	
CS Trace					
None	1336	0	N/A	N/A	0
Single-use	1142	18.2	N/A	N/A	.66
Shared	1162	16.3	40	0	.42
Persistent	1292	3.3	100	100	0
Software	1212	9.1	N/A	N/A	.67
IBM Trace					
None	359	0	N/A	N/A	0
Single-use	322	8.5	N/A	N/A	.70
Shared	322	8.3	22	0	.58
Persistent	350	1.3	100	100	0
Software	326	4.5	N/A	N/A	.71
WC Trace					
None	714	0	N/A	N/A	0
Direct-map	697	0	N/A	N/A	0
Single-use	617	11.8	N/A	N/A	.68
Shared	619	11.1	30	0	.50
Persistent	681	1.8	100	100	0
Software	632	5.9	N/A	N/A	.69

Table 5: Web Server Profile Using `write()`.

As in the other benchmarks, receive buffers are not subject to reuse with the shared-mapping strategy. Regardless of the workload, the persistent strategy is 100% effective at reusing existing mappings as the mappings again become effectively permanent. As in the other benchmarks, the software-based strategy achieves application performance consistently between the shared and persistent IOMMU-based strategies.

Protection Strategy	HTTP Mbps	CPU %		Reuse (%)			HC/ DMA
		Idle	Prot.	TX Hdr.	TX File	RX	
CS Trace							
None	1378	35.0	0	N/A	N/A	N/A	0
Single-use	1291	7.0	27.6	N/A	N/A	N/A	.37
Shared	1330	17.0	17.7	82	72	0	.17
Persistent	1363	28.0	6.7	100	96	100	.02
Software	1351	21.0	13.7	N/A	N/A	N/A	.37
IBM Trace							
None	475	0	0	N/A	N/A	N/A	0
Single-use	403	0	14.0	N/A	N/A	N/A	.43
Shared	413	0	12.3	34	50	0	.35
Persistent	455	0	2.4	100	99	100	0
Software	422	0	6.2	N/A	N/A	N/A	.43
WC Trace							
None	961	0	0	N/A	N/A	N/A	0
Direct-map	953	0	0	N/A	N/A	N/A	0
Single-use	760	0	19.9	N/A	N/A	N/A	.39
Shared	796	0	16.0	53	62	0	.27
Persistent	914	0	2.7	100	100	100	0
Software	833	0	8.7	N/A	N/A	N/A	.40

Table 6: Web Server Profile Using Zero-Copy `sendfile()`.

For all of the previous workloads, the network application utilized the `write()` system call to send any data. Consequently, all buffers that are transmitted to the network interface have been allocated by the guest operating system's network-buffer allocator. Using the zero-copy `sendfile()` interface, however, the guest OS generates network buffers for the packet headers, but then appends the application's file buffers rather than copying the payload. This interface has the potential to change the amount of reuse exploitable by a protection strategy, because data reused by the application can translate to reused IOMMU mappings. Using `sendfile()`, the packet-payload footprint for IOMMU mappings is no longer limited to the number of internal network buffers allocated by the OS, but instead is limited only by the size of physical memory allocated to the guest.

Table 6 shows the performance, efficiency, and sharing profiles for the different protection strategies for web-based workloads when the server uses `sendfile()` to transmit HTTP responses. Note that for the "CS" trace, the host CPU is not completely saturated, and so the CPU's idle time percentage is nonzero. This idle time is useful as a means to compare efficiency. For the other traces, the CPU is completely saturated. The table separates reuse statistics for transmitted buffers according to whether or not the buffer was a packet header or packet payload. As compared to Table 5, Table 6 shows that the shared strategy is more effective overall at exploiting reuse using `sendfile()` than with `write()`. Consequently, the shared strategy gives a larger performance and efficiency benefit relative to the single-use strategy when using `sendfile()`. Table 6 also shows that the persistent strategy is highly effective at capturing file reuse, even though the total working-set size of the "CS" and "IBM" traces are each more than twice as large as the 512 MB mapping space afforded by the GART. As in the other benchmarks, the persistent strategy achieves performance that closely approaches the minimal-overhead direct-map strategy for the "WC" trace. Finally, the table shows that though the shared-mapping strategy benefits from better reuse characteristics and achieves better performance with the `sendfile()`-based workload, the software-based strategy still performs better than both the shared or single-use IOMMU strategies for all workloads.

8.4 Sources of Reuse

The benchmarks explored in this study show varying levels of reuse, as enumerated in the "Reuse" column of Tables 3, 4, 5, and 6. For these network-based workloads, there are two primary sources of reuse: reuse within the network-buffer (`skbuff`) allocator of the operating system, and reuse among the payload buffers provided from user-space for `sendfile()` operations. For each of these types of reuse, there is both spatial and temporal reuse.

Network buffers (called `skbuffs` in Linux) are data buffers managed and allocated by the operating system to either hold the data copied in as payload from a trans-

mitting application, to hold the packet header that is to be prepended onto a zero-copy data payload packet, or to hold data that is received from the network interface. Spatial reuse of an IOMMU mapping happens when more than one usable `skbuff` can be allocated out of a single memory page, since pages are the granularity of IOMMU mappings. For transmitted packet data, this spatial reuse happens when either the packet size (which may be dictated by the maximum transmission unit (MTU) size) is less than that of a physical page, or when the network stack and network card are not utilizing TCP segmentation offloading (TSO). For packet headers prepended onto zero-copy `sendfile()` packets, spatial reuse can be fairly common because many of the small TCP/IP headers can be allocated from a single physical page.

Spatial reuse of `skbuffs` is entirely dependent on the behavior of the `skbuff` allocator, however. As is illustrated by the lack of reuse in the "Shared" IOMMU mappings for received buffers, when the PV-Linux `skbuff` allocator dedicates an entire physical page to a single network buffer, spatial reuse is completely eliminated. Likewise, temporal reuse of `skbuffs` is dependent on the behavior of the `skbuff` allocator. Clearly the default `skbuff` allocator behavior is enabling some temporal reuse, because received packets benefit from IOMMU-mapping reuse in the "Persistent" case (and as previously established, spatial reuse for this receive case is not possible given the allocator's design).

Similarly, for payload data transmitted via zero-copy `sendfile`, spatial reuse is dependent on the packet size, the page size, and the presence (or lack) of TSO capability. Temporal reuse, however, is dependent on the application reuse patterns. Kim *et al.* have explored NIC-based data caching using the same web workloads that are presented in Table 6 and found significant opportunity for reuse [13], so temporal reuse of IOMMU mappings for these payload buffers is expected.

The experimental prototype hardware used in this prototype does not support MTU sizes larger than 1500 bytes and does not support TSO, and so some additional spatial reuse is present in these experiments that might not be present on different hardware. The "Shared" and "Persistent" strategies effectively recapture part of the IOMMU-mapping efficiency that might be gained simply by using TSO- or large-MTU-capable hardware with large messages. In both cases (IOMMU-mapping reuse or large-packet aggregation) a single IOMMU mapping is used for all the data within a physical page. However, the "Shared" and "Persistent" strategies leverage the abundant spatial reuse for workloads that also have many small packets, as in the VoIP benchmark and several of the web workloads.

9 Related Work

Contemporary commodity virtualization solutions forbid direct I/O access and instead use software to implement both protection and sharing of I/O resources among untrusted guest operating systems. Confining direct I/O accesses only within the trusted VMM ensures that all DMA descriptors used by hardware have been constructed by trusted software. Though commodity VMMs confine direct I/O within privileged software, they provide shared access to their unprivileged VMs using different software interfaces. For example, the Denali isolation kernel provides a high-level interface that operates on packets [22]. The Xen VMM provides an interface that mimics that of a real network interface card but abstracts away many of the register-level management details [9]. VMware can support either an emulated register-level interface that implements the precise semantics of a hardware NIC, or it can support a higher-level interface similar to Xen's [19, 21].

IBM's high-availability virtualization platforms feature IOMMUs and can support direct I/O by untrusted guest operating systems. The POWER4 platform supports logical partitioning of hardware resources among guest operating systems but does not permit concurrent sharing of resources [12]. The POWER5 platform adds support for concurrent sharing using software, effectively sacrificing direct I/O access to gain sharing [4]. This sharing mechanism works similarly to commodity solutions, effectively confining direct I/O access within what IBM refers to as a "Virtual I/O Server". Unlike commodity VMMs, however, this software-based interface is used solely to gain flexibility, not safety. When a device is privately assigned to a single untrusted guest OS, the POWER5 platforms can still use its IOMMU to support safe, direct I/O access.

Previous studies have found that software-based approaches for I/O sharing and protection are especially costly for network I/O. Sugerman *et al.* reported a factor-of-6 network overhead penalty compared to native OS execution in a 2001 study of VMware's network virtualization [19]. Menon *et al.* later reached similar results using the Xen virtual machine monitor, reporting a factor-of-5 penalty versus native execution in 2005 [16]. Menon *et al.* subsequently developed software-based mechanisms to reduce this overhead for transmit-oriented workloads but did not find any such mechanism for receive-based workloads [15].

The high overhead of software-based network virtualization motivated recent research toward hardware-based techniques that support simultaneous, direct-access network I/O by untrusted guest operating systems. Liu *et al.* developed an Infiniband-based prototype that supports direct access by applications running within un-

trusted virtualized guest operating systems [14]. This work adopted the Infiniband model of registration-based direct I/O memory protection, in which trusted software (the VMM) must validate and register the application's memory buffers before those buffers can be used for network I/O. Registration is similar to programming an IOMMU but has different overhead characteristics, because registrations require interaction with the device rather than modification of IOMMU page table entries. Furthermore, unlike an IOMMU, registration alone cannot provide any protection against a malfunctioning device, since the protection mechanism is partially enforced within the I/O device.

Willmann *et al.* previously developed an Ethernet-based prototype that also supports concurrent, direct network access by untrusted guest operating systems [23]. Rather than relying on hardware-based buffer registration for I/O protection, that work introduced the software-based mechanism for ensuring DMA transaction safety that is described in Section 4 of this paper. This method is based on validating DMA descriptors to be enqueued to a device and on guaranteeing the integrity of those descriptors throughout the duration of an I/O transaction. Like registration-based virtualized hardware, this software-based strategy offers no protection against faulty device behavior.

Raj and Schwan also developed an Ethernet-based prototype device that supports shared, direct I/O access by untrusted guests [17]. Because of hardware-implementation constraints, their prototype has limited addressability of main memory and thus requires all network data to be copied through VMM-managed bounce-buffers. This strategy permits the VMM to validate each buffer but does not provide any protection against faulty accesses by the device within its addressable memory range.

AMD and Intel have recently proposed the addition of IOMMUs to their upcoming architectures [3, 11]. However, IOMMUs are an established component in high-availability server architectures [6]. Ben-Yehuda *et al.* recently explored the TCP-stream network performance of IBM's state-of-the-art IOMMU-based architectures using both non-virtualized, "bare-metal" Linux and paravirtualized Linux running under Xen [7]. As is found in this paper, they reported that the state-of-the-art single-use IOMMU-management strategy can incur significant overhead. They also identified platform-specific architectural limitations that reduce performance, such as the inability to individually replace IOMMU mappings without globally flushing the CPU cache. They hypothesized that modifications to the single-use IOMMU-management strategy could avoid such penalties. Though the GART-based IOMMU implementation used in this paper does not incur the cache-flush penalties associated with the IBM platform, single-use mappings are costly nonetheless. Furthermore, this paper proposes two specific strategies for IOMMU management that reduce IOMMU-related overhead, and examines their safety characteristics and effectiveness at reducing overhead to increase performance across a variety of real-world workloads.

10 Conclusions

This paper has evaluated a variety of DMA memory protection strategies for direct access to I/O devices within virtual machine monitors by untrusted guest operating systems. All of these strategies prevent the guest operating systems from directing the device to access memory that does not belong to that guest. The strategies do, however, differ in their performance overhead, the level of intra-guest protection, and their ability to deal with misbehaving devices.

The traditional single-use strategy provides inter-guest protection at the greatest cost, consuming from 6–26% of the CPU. However, there is significant opportunity to reuse IOMMU mappings, which in turn can reduce the cost of providing protection. This reuse and its efficiency advantages are demonstrated by the new shared- and persistent-mapping strategies introduced in this paper. Multiple concurrent network transmit operations are typically able to share the same mappings 20–40% of the time, yielding small performance improvements. However, due to Xen's I/O architecture, network receive operations are usually unable to share mappings. In contrast, using persistent mappings with a limit of 131,072 mappings enables nearly 100% reuse in almost all cases, reducing the overhead of protection to only 2–13% of the CPU.

The protection strategy supported by the VMM and the OS can also greatly affect the degree to which each guest OS can potentially protect itself by isolating the behavior of the hardware or isolating its own device drivers. Though the direct-map strategy has the least overhead, it is the only strategy that provides no mechanism for the guest OS to protect itself from its own device drivers. The persistent-mapping strategy, however, offers nearly the same performance as the direct-map strategy while still allowing some protection against misbehaving device drivers.

Finally, the software-based protection strategy evaluated in this paper performs better than two of the IOMMU-based strategies (single-use and shared), consuming only 3–15% of the CPU for protection. However, the software-based mechanism still maintains strict inter-guest memory protection. And though it cannot guard against errors that originate in the hardware, the software-based strategy supports the implementation of

enhanced intra-guest driver isolation. Therefore, an IOMMU-based protection strategy does not necessarily deliver superior performance or protection relative to software-only strategies.

Acknowledgments

We wish to thank Muli Ben-Yehuda and Ben-Ami Yassour for contributing IOMMU benchmark data for the IBM Calgary and Intel VT-d IOMMU platforms. We also wish to thank this paper's conference shepherd, Michael Swift, and our anonymous reviewers for their insightful comments and suggestions that improved this paper.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Oct. 2006).

[2] ADVANCED MICRO DEVICES. *Secure Virtual Machine Architecture Reference Manual*, May 2005. Revision 3.01.

[3] ADVANCED MICRO DEVICES. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb. 2007. Publication 34434, Revision 1.20.

[4] ARMSTRONG, W. J., ARNDT, R. L., BOUTCHER, D. C., KOVACS, R. G., LARSON, D., LUCKE, K. A., NAYAR, N., AND SWANBERG, R. C. Advanced virtualization capabilities of POWER5 systems. *IBM Journal of Research and Development 49*, 4/5 (2005), 523–532.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (Oct. 2003).

[6] BEN-YEHUDA, M., MASON, J., KRIEGER, O., XENIDIS, J., DOORN, L. V., MALLICK, A., NAKAJIMA, J., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the Linux Symposium* (July 2006).

[7] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND DOORN, L. V. The price of safety: Evaluating IOMMU performance. In *Proceedings of the 2007 Linux Symposium* (July 2007).

[8] DEVINE, S., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent #6,397,242* (Oct. 1998).

[9] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On Demand IT InfraStructure (OASIS)* (Oct. 2004).

[10] INTEL. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*, Apr. 2005. Revision 2.0.

[11] INTEL CORPORATION. *Intel Virtualization Technology for Directed I/O*, May 2007. Order Number D51397-002, Revision 1.0.

[12] JANN, J., BROWNING, L. M., AND BURUGULA, R. S. Dynamic reconfiguration: Basic building blocks for autonomic computing on ibm pseries servers. *IBM Systems Journal 42*, 1 (2003), 29–37.

[13] KIM, H., PAI, V. S., AND RIXNER, S. Improving web server throughput with network interface data caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2002), pp. 239–250.

[14] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).

[15] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing network virtualization in Xen. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).

[16] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the ACM/USENIX Conference on Virtual Execution Environments* (June 2005).

[17] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing* (June 2007).

[18] SHAFER, J., AND RIXNER, S. RiceNIC: A reconfigurable network interface for experimental research and education. In *Proceedings of the Workshop on Experimental Computer Science* (June 2007).

[19] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).

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

[21] VMWARE INC. VMware ESX server: Platform for virtualizing servers, storage and networking. `http://www.vmware.com/pdf/esx_datasheet.pdf`, 2006.

[22] WHITAKER, A., SHAW, M., AND GRIBBLE, S. Scale and performance in the Denali isolation kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2002).

[23] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture* (Feb. 2007).