On the DMA Mapping Problem in Direct Device Assignment

Ben-Ami Yassour
IBM Research–Haifa
Haifa, Israel
benami@il.ibm.com

Muli Ben-Yehuda
IBM Research–Haifa
Haifa, Israel
muli@il.ibm.com

Orit Wasserman
IBM Research–Haifa
Haifa, Israel
oritw@il.ibm.com

ABSTRACT

I/O intensive workloads running in virtual machines can suffer massive performance degradation. Direct assignment of I/O devices to virtual machines is the best performing I/O virtualization mechanism, but its performance still remains far from the bare-metal (non-virtualized) case. The primary gap between direct assignment I/O performance and bare-metal I/O performance is the overhead of mapping the VM's memory pages for DMA in IOMMU translation tables. One could avoid this overhead by mapping all of the VM's pages for the lifetime of the VM, but this leads to memory consumption which is unacceptable in many scenarios.

The DMA mapping problem can be stated briefly as "when should a memory page be mapped or unmapped for DMA?" We begin by presenting a theoretical framework for reasoning about the DMA mapping problem. Then, using a quota-based approach, we propose the *on-demand DMA mapping strategy*, which provides the best DMA mapping performance for a given amount of memory consumed. In particular, on-demand mapping can achieve the same performance as state-of-the-art mapping strategies while consuming much less memory (exact amount depends on the workload's requirements). We present the design and implementation of on-demand mapping in the Linux-based KVM hypervisor and an experimental evaluation of its application to various workloads.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.4 [**Operating Systems**]: Communications Management—*Input/output*; D.4.8 [**Operating Systems**]: Performance—*Operational analysis*

Keywords

IOMMU, device assignment, direct access, I/O virtualization, SR-IOV, DMA mapping, on-demand mapping, IOMMU protection strategies.

SYSTOR 2010 May 24-26, Haifa, Israel

1. INTRODUCTION

A constantly increasing number of computer systems in today's data-centers are running multiple operating systems simultaneously, using *virtualization* technology. Most new CPUs manufactured for servers, desktops, laptops, and even some embedded systems, has virtualization capabilities built into the hardware. Virtualization is clearly here to stay.

Virtualizing a computer system's CPU and memory is a challenging but fairly well understood problem. However, a computer system has three equally important components: CPU, memory, and I/O. Virtualizing I/O is far more challenging and not nearly as well understood as virtualizing the CPU and memory.

There are three prevailing approaches to I/O virtualization: emulating a real (hardware) I/O device [29], using para-virtualized I/O drivers [5, 17, 25], and giving a virtual machine direct access to an I/O device [21, 23, 32, 34]. Other approaches, such as virtualizing the entire I/O stack or dedicating a core to I/O processing, are also possible [20, 27, 28].

Emulation means that the host emulates a device that the guest already has a driver for [29]. The host traps all device accesses and converts them to operations on a real, possibly different, device, as depicted in Figure 1.

With para-virtualized I/O devices, special hypervisor-aware I/O drivers are installed in the guest. All modern hypervisors implement such para-virtualized drivers [5, 17, 25], but their performance is still far from native [24, 26] and they require special drivers.

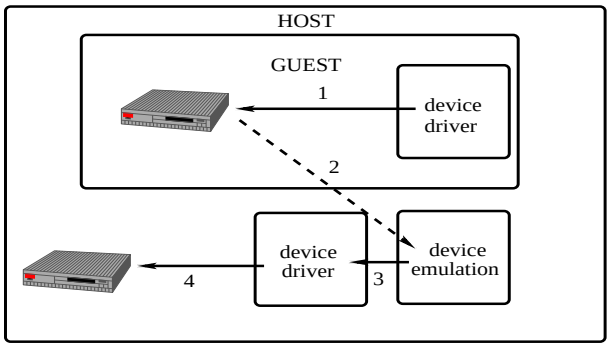

Figure 1: Emulation flow

Direct device access (interchangeably referred to as "direct device assignment", "direct access" or "pass-through access") means that the guest sees a real device and interacts with it directly, *without a software intermediary* (see Figure 2).

Direct access does away with the software intermediary which other I/O virtualization approaches require. Direct access can provide much better performance than the alternative I/O virtualization approaches [19, 21, 23, 32, 34]. This is its primary benefit and its importance cannot be overstated: the difference in performance for I/O intensive workloads is such that direct access makes it possible to virtualize workloads that otherwise would bring the virtualization system to its knees. Another benefit of direct access is that the guest can use any device it has a driver for.

To fully benefit from direct access, some hardware support is necessary. An I/O memory management unit (IOMMU) is needed to protect and translate device memory accesses [3, 7], and a self-virtualizing adapter is needed in order to share the adapter between different virtual machines [21, 23, 32]. IOMMUs such as Intel's VT-d [2, 3], IBM's Calgary [7], and AMD's IOMMU [1], and PCI standard SR-IOV devices [13, 19] are now becoming available.

Although direct access can in theory provide bare-metal performance (i.e., the same performance as running the same workload in a non-virtualized environment), in practice sizable performance gaps remain. This paper deals with one such performance gap—the so-called "DMA mapping problem."

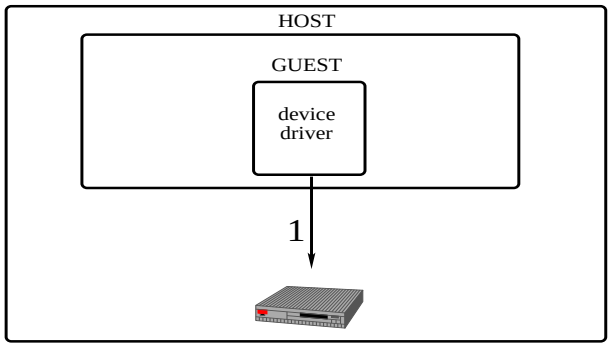

Figure 2: Direct access flow

1.1 DMA and IOMMUs

In virtualized environments, guests have their own view of physical memory, which in the Linux-based KVM hypervisor is referred to as "guest physical", and which is distinct from the host's "host physical" view of memory [17]. Although there are ways of giving fully-virtualized guests access to portions of host memory without hardware support [18], such approaches can only work for *trusted* guests. Giving *untrusted* guests and devices access to system memory requires hardware support in the form of an IOMMU [7, 15].

An I/O Memory Management Unit (IOMMU) validates and translates all device accesses to host memory. To understand why an IOMMU is needed, consider what would happen if we didn't have one: first, the guest operating system would need to know that it is running on a virtualized system, and be able to translate guest physical addresses to host physical addresses on its own before handing them over to the device. This conflicts with one of the key value propositions of x86 virtualization, running unmodified guest operating systems. Second, we would need to *trust* the OS to program the device with the right addresses, since otherwise the OS could program the device to DMA anywhere in system memory, including on top of the hypervisor or other virtual machines. This would defeat another key value proposition of virtualization, running untrusted operating systems in isolation.

IOMMUs work by interposing on DMAs made by devices and *validating* and *translating* the addresses in those DMAs before letting them read or write main memory. This work used Intel's VT-d IOMMU [3], but the details are substantially similar for other IOMMUs. In order to validate and translate DMAs generated by different devices VT-d maintains a separate translation table in memory for each PCI device (as denoted by its PCI Bus/Device/Function ID). When the device initiates a DMA operation, the IOMMU walks through the translation table and checks whether the access is valid. If the translation entry is not valid (for example if the device is trying to write to memory that it only has read permissions for), the DMA is aborted. If the translation entry is valid, the address is translated and the DMA operation reads or writes memory at the translated address.

In order to speed up translation entry lookups, most IOMMUs, including VT-d, cache translations in IOTLBs. An IOTLB serves the same purpose a TLB serves for the MMU. IOTLBs improve IOMMU performance, but software must keep them coherent when it modifies a translation table. VT-d defines three modes of IOTLB invalidations: global, page level, and directory level. When software invalidates the IOTLB it can either poll for invalidation completion or request an interrupt. There are two mechanisms for invalidating the entries: register based and queued invalidations. With the register based approach software needs to wait for the completion of each address invalidation. When several entries need to be modified, queued invalidation is more efficient. This feature is important, since as we show, an important optimization is to batch modifications. With the queuing method the IOTLB invalidation time can be reduced. Unfortunately, the VT-d revision on our systems did not support queued invalidations.

When new mappings are added to the translation table which cause an entry to go from non-present to present then no IOTLB invalidation is needed, since VT-d does not cache non-present entries. On the other hand, VT-d's internal write-buffer[1] does need to be flushed, and software needs to wait for completion. We note that when multiple addresses are to be added, only a single write-buffer flush is needed. Therefore, batching multiple mappings can significantly reduce the mapping cost.

1.2 The DMA Mapping Problem

The key goal of direct access is to allow a guest operating system to access a device *directly*, which requires mapping the guest OS's memory in an IOMMU so that the device could DMA to it directly. An obvious question which follows is "when should a page of memory be mapped or unmapped in the IOMMU?". This, in a nutshell, is the DMA mapping problem.

The DMA mapping problem arises because mapping (and unmapping) a page of memory in an IOMMU translation table is expensive, as shown in previous works by Ben-Yehuda et al. [8] and by Willman, Rixner, and Cox [31], and con-

[1]Internal write-buffers may hold updates to memory-resident data-structure. Flushing them may be required to make those updates visible to the hardware. For further details, consult the VT-d specification [2].

firmed in our experiments. A breakdown of the costs is provided in Section 6.

The early direct access implementations used one of two extreme approaches for DMA mapping: they either mapped all of a guest operating system's memory up-front (thus incurring minimal run-time overhead), or they only mapped memory once immediately before it was DMA'd to or from, and unmapped it immediately when the DMA operation was done [7]. Willman, Rixner, and Cox named these strategies *direct mapping* and *single-use mapping*, respectively. In addition, they presented two other strategies: *shared mapping* and *persistent mapping* [31].

Single-use mapping has a non-negligible performance overhead [8] but protects the guest's memory from malicious devices and buggy drivers. Thus it sacrifices performance for reduced memory consumption and increased protection. *Direct mapping*, on the other hand, is transparent to the guest and requires minimal CPU overhead—but requires pinning all of the guest's memory, and provides no protection inside a guest (intra-guest protection), only between different guests (inter-guest protection). Thus it sacrifices memory and protection for increased performance.

Shared mapping and *persistent mapping* provide different tradeoffs between performance, memory consumption, and protection. *Shared mapping* reuses a single mapping if more than one device is trying to DMA to the same memory location at the same time, and *persistent mapping* keeps mappings around once they have been created in case they will be reused in the future. A key question is whether there is an optimal mapping strategy, and if yes, what is it. We attempt to address this question in the remainder of this paper.

The DMA mapping problem is not specific to virtual machine direct access. Variants of it also appear in many other areas, such as high-speed networks (Infiniband, Quadrics, Myrinet) with OS-bypass and/or hypervisor-bypass [21, 33], userspace drivers and micro-kernels. In general, whenever a trusted entity wishes to grant access to memory to a device controlled by an untrusted entity, the privileged entity needs to solve a variant of the DMA mapping problem.

1.3 Our Contributions

We make the following contributions in this paper:

1. A theoretical framework for reasoning about the DMA mapping problem is presented in Section 2. Using the framework, we build a quota-based model in Section 2.1.

2. The *on-demand mapping* strategy, which provides the best performance for a given amount of memory consumed, is presented in Section 2.2.

3. A prefetching algorithm which reduces the DMA mapping overhead with small quotas is presented in Section 2.5, and several batching mechanisms are presented in Section 3.2.

4. The design and implementation of on-demand mapping in the KVM is presented in Section 3, with an evaluation of its performance and memory consumption with various workloads and quotas in Section 4. Intra-guest protection is discussed in Section 5, and a cycle breakdown of the cost of creating and destroying a DMA mapping is presented in Section 6.

2. THEORETICAL FRAMEWORK

We begin by presenting several requirements that any DMA mapping scheme must satisfy. The first requirement is for the assigned device to operate correctly, that is, to operate as it would in a non-virtualized environment. Our working assumption is that in the majority of cases when a device initiates a DMA request and the DMA request fails it will require device reset, since no current hardware supports I/O page faults [27]. It is therefore a requirement that the IOMMU must have a valid translation for every address that the device can validly DMA to. Naturally, if the device tries to DMA to an invalid address, then the request will be blocked.

This requirement leads to the following proposition:

PROPOSITION 2.1. *Every guest physical address (gpa) that the guest programmed the device to DMA to must be backed up by a valid guest-physical-to-host-physical mapping in the IOMMU translation table for that device.*

For correct operation, every guest-physical-to-host-physical mapping in the IOMMU translation table for a given device must also be backed up by an equivalent mapping in the guest-physical to host-physical translation table used for MMU address translations for the guest VM driving that device (i.e., the software or hardware (EPT/NPT) shadow page table) [4, 9]. Note that if a host physical address (hpa) that has a valid translation leading to it in the IOMMU translation table does not have a valid translation in the shadow page table, then it might be in use by another guest or the host. Since the guest-physical-to-host-physical mapping exists in the IOMMU translation table, the device could read or write to it, thereby reading or writing a host frame that is owned by another guest or by the host and violating the protection (isolation) guarantees of the IOMMU.

PROPOSITION 2.2. *Every guest-physical-to-host-physical mapping in the IOMMU translation table of a given device must have a corresponding mapping in the guest-physical-to-host-physical translation table for CPU MMU translations for the guest VM driving that device.*

From Propositions 2.1 and 2.2 we have:

PROPOSITION 2.3. *Every valid gpa that the guest programmed the device to DMA to or from must have a mapping to hpa, which must be pinned (cannot be changed) as long as the device may legitimately use that gpa.*

The host can apply various policies for programming the IOMMU translation table. Since changing the IOMMU mapping requires a world switch and an IOMMU IOTLB flush, both of which are expensive, the main optimization target is to minimize the number of IOMMU remappings. Mapping the entire guest physical address space requires a single IOMMU mapping operation that is not changed throughout the period that the device is assigned to that guest. On the down side, based on Proposition 2.3, mapping the entire guest physical address space means that that the entire guest address space would then be pinned!

Mapping and pinning the entire guest physical memory might be a valid solution in some scenarios, but is not acceptable in the general case, since memory is a precious resource and significant effort is expended in trying to conserve, share, and otherwise make full use of it in virtualized

environments [16, 22, 30]. Indeed, memory is considered the most important resource and the main barrier to scalability in virtualized environments. All commonly used hypervisors over-commit memory and try to balance the memory needs of VMs dynamically by continually adjusting the amount of memory assigned to each VM. We need a better solution for DMA mapping, one that does not require pinning a large amount of host physical memory, for an unbounded length of time under the control of an untrusted guest.

The simplest solution to avoid mapping the entire guest memory is for the guest to notify the host of the guest pages which are the targets of each DMA request. This way the host can update the IOMMU translation tables only for the addresses that the guest is using for DMA. However, our experiments with KVM in Section 6 confirm earlier experimental results: remapping IOMMU translation entries is expensive[8, 31]. If the guest initiates a hypercall for each DMA transaction, performance is degraded significantly. Therefore it is crucial for the system to minimize the number of IOMMU remappings. In an attempt to reduce the number of IOMMU remappings, the *persistent mapping* strategy keeps guest physical addresses mapped in the IOMMU translation tables once they have been mapped. Then, if and when the guest tries to reuse that address, the overhead of remapping is avoided since the address is already mapped. Clearly, with such a strategy, after a while all of the guest's memory pages will be pinned as in the case of the *direct mapping* strategy, leading to unacceptable memory consumption.

We propose a model in which one first defines a *quota* of guest memory pages that the guest can pin at any given time for DMA. Without such restrictions a selfish or malicious guest with direct device access can pretend to DMA to the entire guest memory, thus making sure that all of its physical memory is pinned.

Note that with such an approach, the correct minimal quota needs to be determined. If the quota is insufficient for correct operation of the guest, it is equivalent to running a guest with insufficient host physical memory.

The quota can either be defined manually in a static fashion just like the amount of memory that is assigned to the guest, or be changed dynamically by the host. A combination of the two where a range of quotas is provide statically, and the exact quota within that range is determined dynamically is also possible. We now ask two questions:

- **Guest perspective** Given a quota of DMA mappings, what is the optimal eviction strategy? In other words, how should the guest use its given quota of DMA mappings so that the total number of remappings (evictions) is minimized?
- **Host perspective** What is the optimal allocation of memory for DMA purposes (i.e., quotas) between one or more guests, so that the shared resource (memory) is shared fairly between the guests and each guest's I/O performance is maximized?

We addressed both questions when designing the on-demand mapping implementation, as detailed in Section 3. We also note that while keeping pages mapped in the IOMMU translation tables improves performance and preserves inter-guest protection, it does not guarantee intra-guest protection. We discuss intra-guest protection in Section 5.

2.1 A Model for DMA Mapping

Next we define a formal model for the DMA mapping problem. Given a quota Q and a series of requests by a guest driver to map and unmap guest pages $< g_i >$, where the guest can make a hypercall to the host and ask the host to create or destroy one or more mappings in the IOMMU translation table mapping those pages, what is the optimal guest mapping strategy such that at every point in time all of the following properties hold:

1. The guest has a set S of guest pages which are mapped in the IOMMU translation table. The set S, which we term the *map cache*, is composed of the set of guest pages which are candidates for eviction (E) and the set of guest pages which are pinned (P). $S = E \cup P$.
2. Each guest page g_i is either a candidate for eviction because the driver has already asked the map cache to unmap it: $g_i \in E$, or is pinned because the driver is still using it: $g_i \in P$.
3. The quota is never exceeded, i.e., the total size of the map cache is smaller than or equal to the quota: $|S| \leq Q$.
4. When a driver tries to map a new guest page g_{new}, if $g_{new} \notin S$ than g_{new} is added to S: $S = S \cup \{g_{new}\}$. If adding g_{new} to S would cause the size of S to exceed the quota, than a guest page g_e which is a candidate for eviction ($g_e \in E$) is evicted from S: $E = E/\{g_e\}$. If there are no candidates for eviction, the mapping request is denied.
5. Adding or removing one or more guest pages from the map cache S requires a remapping hypercall.
6. The number of remapping hypercalls is minimized.

Note that our target is to minimize the number of hypercalls rather than minimizing the number of map or unmap operations, since a single hypercall can map or unmap several guest pages, and we assume that the cost of remapping a single page or several pages is similar. This is a reasonable assumption since the cost of the remapping (other than the hypercall) is the cost of the IOMMU write buffer and/or IOTLB flush, and a single IOMMU write buffer and/or IOTLB flush can be used for changing several mappings at the same time.

The DMA mapping problem, as formalized above, has two variants: offline—requests are known in advance, and online—-requests are *not* known in advance.

Once formalized this way, the DMA mapping problem is remarkably similar to the well known page replacement problem [10]. However, there is a key difference. In the page replacement problem a single page is accessed at a given time, and *any other page* can be evicted to make place for it. In the DMA mapping problem, there is a set of pages which cannot be evicted (the pages which a driver mapped but not yet unmapped, i.e., the pages which a device may be actively using at the moment). These pages cannot be evicted from the set of existing mappings; only pages which were previously unmapped are candidates for eviction. This difference stems from Proposition 2.1, which itself stems from the fact that there is no mechanism for I/O page faults in current IOMMUs, I/O devices, and protocols [27].

It therefore follows that in the DMA mapping problem there are some access patterns which *cannot* be satisfied. If the guest driver tries to map more pages than the quota allows and there are no evictable pages than the mapping operation will fail.

2.2 The On-Demand Mapping Strategy

Using the quota model presented in the previous section, we devise a mapping strategy which maximizes performance (i.e., minimizes remapping hypercalls) for a given amount of memory consumed (i.e., a given quota). We restrict the discussion here to strategies which provide inter-guest protection only; intra-guest protection is discussed in Section 5.

In the *direct mapping* strategy, we pin and map the entire guest memory in advance. This maximizes performance since it does not require any remapping hypercalls, but it also requires pinning all of the guest's memory (i.e., has the worst possible memory consumption). Thus direct mapping is optimal when the quota is equal to the guest's entire physical memory, but cannot be used when the quota is smaller than the guest's physical memory.

The *single-use mapping* strategy is to map and unmap a guest page in the IOMMU translation table whenever the driver maps or unmaps a page for DMA. This strategy incurs the highest number of hypercalls and IOMMU remappings, but it can be used with very small quotas since it also minimizes memory consumption.

The *persistent mapping* strategy maps guest pages in the IOMMU translation table when the drivers map them for the first time. It is not specified when the pages are unmapped. We introduce a refinement of persistent mapping which we term *on-demand mapping*, based on the quota-based model. In the on-demand mapping strategy, mappings are created in the IOMMU translation table when the guest driver maps them for the first time, assuming the number of existing mappings is less than the quota. When the guest tries to create a new mapping which would cause the quota to be exceeded, one or more old mappings are evicted from the map cache and the new mapping created in their place.

On-demand mapping avoids the need to map the entire guest memory up front, and maps it only as needed, until the quota is reached. If the quota is set to some minimal value, on-demand behaves like single mapping. If the quota is set to the size of guest physical memory, on-demand behaves like direct and persistent mapping. If the quota is set to some value in-between then the behavior of on-demand depends on the workload. See Section 4.1 for a discussion of the "right" quota to set.

We have observed that a quota which is significantly smaller than guest physical memory is often sufficient for a workload's needs (see Section 4.1). In this case on-demand has the same performance as persistent or direct mapping, while consuming much less memory, and unlike persistent, it will not grow to consume all of memory. Other times, when the host is under memory pressure, on-demand enables the host to decide how much memory to allow a given guest to pin, while also enabling the guest to achieve the best performance given the amount of memory allotted to it.

2.3 Optimal Solution to the Offline DMA Mapping Problem

Let us ignore batching of requests for a moment, and assume that we can only map or unmap a single page per hypercall. Let us further assume that the set of pinned pages P is always strictly smaller than the quota ($|P| < Q$). Under these assumptions, the offline version of the DMA mapping problem is for all intents and purposes equivalent to the page replacement problem, for which the optimal offline solution is Belady's theoretical memory caching algorithm [6]. Briefly stated, this algorithm always evicts the page that is going to be accessed later then any other page in the cache.

The optimal offline batching algorithm is to simply map the next N different pages in the access pattern in a single batch. The upper bound on the miss rate is $\frac{1}{N+1}$, which is the theoretical bound, and is much lower than the miss rate without batching, which is 1 when there is no reuse.

2.4 Online Algorithms

Since the DMA mapping problem is related to the page replacement problem, what can we learn from the known online algorithms for the page replacement problem?

The most familiar online algorithm is the Least Recently Used (LRU) algorithm that is used as a base in many systems with additional optimizations. However, LRU also has well-known deficiencies with certain access patterns.

The best known algorithms for the page replacement problem make use of the access graph model [10, 11]. In this model, there is a graph which models the input sequence. In the graph there is an edge between two vertexes (v, u) if an access to v is followed by an access to u (or vice versa, since the graph is usually not directed). Obviously a single graph can model many different input sequences, provided they share the same access patterns. The best known algorithms for deciding which page to evict next in the access graph model ares *FAR* [12] and *FARL* [14]. Intuitively, both perform an online approximation of what the optimal offline algorithm does, evicting the graph node whose next access is the farthest in the future.

FAR and FARL show us that as long as an access pattern has *some* repetition (some reuse) and is not completely random, then inspecting the history and deciding which pages to evict and which to keep, based on history, is likely to be useful. We present a prefetching algorithm for on-demand mapping which exploits this insight in the next section.

2.5 Prefetching Algorithm

The frequency-based prefetching algorithm takes advantage of the low cost of batching. Many I/O workloads exhibit recurring patterns in their access sequences, often because pages are used as buffers which are reused over and over again. We exploit the fact that if a guest driver maps a page, there is a high probability that it will next try to map additional pages which were mapped in the past following this page.

The prefetching algorithm is as follows: for each page g_i keep track of which pages were mapped most frequently in the past right after g_i. We say that a page g_i has a follower if there is a page g_j that was mapped right after g_i at least twice. If there is more than one such page g_j, the follower is the page which was mapped most often after g_i.

When a page g_i is mapped by the driver which is not in the map cache, we make a hypercall to map it, and if it has a follower page, we also add the follower page to the batch. If the follower page has a follower page we add that page to the batch as well, repeating the process until there is no follower page or a maximal batch size is reached.

Assuming that the quota has been reached and that the remapping hypercall is trying to map n new pages, n old pages need to be evicted to make room for them. The pages to be evicted are chosen by a standard LRU algorithm.

3. ON-DEMAND MAPPING

We designed and implemented on-demand mapping in the Linux-based KVM hypervisor [17]. This implementation built upon our earlier implementation of direct access for KVM [34]. The original implementation of direct access for KVM used the direct mapping strategy, enabling it to run unmodified guests and avoiding guest changes. The on-demand mapping strategy, however, requires a para-virtualized guest DMA mapping interface, which we implemented using hypercalls for guest-to-host communication.

3.1 Map Cache

The key guest-side component of on-demand mapping is the *map cache*, which caches DMA mappings inside the guest in order to avoid the overhead of going to the hypervisor to create or destroy a new mapping in the IOMMU translation table. We implemented the map cache as a Linux DMA-API implementation, which all DMA-using drivers call into [7]. The map cache is limited in size: it has a set quota of mappings it caches. The quota can be changed at run-time, and is dictated by the hypervisor, as discussed in Section 2.1.

Pages (mappings) in the map cache are either pinned or candidates for eviction. When a driver asks the map cache to create a mapping for some guest page, the map cache checks if a mapping for the page already exists in the map cache. If it is, a reference count is incremented. If the page isn't mapped, the map cache makes a hypercall and asks the hypervisor to map the page in the IOMMU translation table. If the call succeeds, the map cache increments the reference count on the page. Unmap requests by the driver are handled similarly: the reference count on mappings of that page is decremented, and if it drops to zero, the page is moved to the candidates for eviction set. Note however that the page remains mapped in the IOMMU and used for DMA from the host's perspective.

When the map cache makes a map or unmap hypercall, the hypervisor makes any necessary checks (e.g., that the guest is not trying to unmap a page that it hasn't mapped, or that the guest is not about to go over quota) and then makes the necessary changes to the IOMMU translation table.

The main data structure used by the map cache is a red-black tree, using the standard Linux red-black tree implementation. The tree holds all of the pages that are mapped and their reference counts. All pages with a reference count of zero are stored in a free list and are candidates for unmapping from the IOMMU translation table and eviction from the map cache.

While the map cache greatly improves performance by caching mappings—thereby reducing the number of DMA remappings—further optimizations are possible. The next two sections describe two classes of optimizations which reduce the amount of interaction between the guest and the hypervisor further.

3.2 Batching Driver Mapping Requests

Many commonly used drivers such as the e1000e NIC driver use the Linux kernel functions *dma_map_single* and *dma_unmap_single* to map or unmap memory for DMA. When a large buffer with multiple pages is mapped for DMA, the driver calls *dma_map_single* several times to map the entire buffer. Each such call into the map cache could cause a map cache miss and a subsequent hypercall to map the page. Since the driver will not use the mappings until it has completed the entire sequence of mapping requests, we can minimize the number of hypercalls by batching these requests into a single hypercall at the end of the sequence. Batching the sequence reduces the number of hypercalls and enables us to use a single IOMMU write buffer flush for multiple changes.

We identified and implemented the following batching opportunities:

- **Batch map requests:** When a large buffer is mapped by the NIC driver in the guest we map it using a single hypercall. We note that this optimization does not violate the intra-guest protection property.
- **Batch unmap requests:** The equivalent of "batch map requests" for unmap requests.
- **Unmap piggyback:** Perform a single hypercall including both map and unmap requests. In general when the on-demand strategy is used, then there is no reason to unmap a page before the quota is exceeded. Since we only unmap as a result of a new map request, the most efficient way to execute the unmap request is to piggyback the unmap request on the map request replacing it.

Adding the batching code involves minimal changes to the driver. For exapmle in the case of the e1000 driver we added only six new lines of code.

3.3 Prefetching Mappings

In addition to the batching optimizations mentioned above, we also implemented the prefetching algorithm as described in Section 2.5. The key practical difference between batching and prefetching is that batching requires driver changes, and prefetching doesn't. Put differently, batching would need to be implemented in every Linux driver, while prefetching can be implemented once in the DMA-API layer.

In order to minimize the algorithm's memory consumption, we keep for each page no more than 3 pages which followed it, and the number of occurrences for each of these 3 pages. For each mapped page we prefetch the follower with the highest number of occurrences—if the number of occurrences is higher than 2.

3.4 Quota Control Policy

From the point of view of the guest, the host communicates a quota and the guest needs to keep its map cache within the confines of the quota. From the point of view of the host, what quota should it set to the guest?

3.4.1 Cooperative Guests

For cooperative guests, we note that with all modern hypervisors, virtual machines already have a certain amount of memory assigned to them (regardless of direct access) and that amount can be changed dynamically by the hypervisor, either by paging virtual machine pages out to disk, or by using a para-virtualized balloon approach [30].

The balloon driver is a hypervisor-controlled guest driver which the hypervisor uses to "steal" or "withdraw" memory

pages from the guest, by asking the balloon driver to allocate them inside the guest. Once the balloon driver has allocated them, the hypervisor can safely remove the guest-physical-to-host-physical mappings of these pages in both CPU MMU page tables and IOMMU translation tables (if the page was previously mapped for DMA) and give the host frames to some other virtual machine.

Therefore, one possible approach for the dynamic quota policy for cooperative guests is to define the quota as whatever amount of memory is allocated to the guest at the moment anyhow. We already gave the guest all of that memory, so we might as well allow it to use it for DMA too, relying on its cooperation (via the balloon driver) when we want to reclaim some of those pages.

Let us assume that the host would like to share memory between multiple virtual machines, and that the load on each virtual machine might change. The host decides how much memory to assign to each guest at each point in time according to a predefined policy. We denote the guest total memory size M_{max}, and the amount of memory that is allocated to the guest at any given time as $M(t), M(t) \leq M_{max}$. We denote the current DMA mapping quota as $Q(t), Q(t) \leq M(t)$. The host will use the balloon driver in the guest to "withdraw" memory from the guest as need for other purposes: we denote the amount of memory withdrawn (i.e,. the balloon's current size) as $B(t)$.

It follows that when the balloon is used in the guest, the host does not need to explicitly define a DMA mapping quota, since the quota is implicitly defined by the current size of the balloon: $Q(t) \leq M(t) = M_{max} - B(t)$. Assuming the host has given the guest enough memory for its regular operation, the guest can also use as much of that memory as it wishes for DMA.

In other words, the quota will be indirectly applied since the pages that are used by the balloon will never be added to the map cache, and pages which were previously in the map cache and the balloon allocated will be removed from the map cache. If the hypervisor wishes to give the guest more pages (whether for DMA purposes or for other purposes) it will shrink the balloon; if the hypervisor wishes to withdraw some pages from the guest, it will inflate the balloon and cause the the guest to release some of its pages. Since the map cache is only limited by the amount of currently available memory $M(t)$ it will eventually fill and in the steady state there are going to be no hypercalls for DMA remappings, maximizing performance.

3.4.2 Selfish Guests

With an implicit DMA mapping quota, the host does not limit the amount of pages the guest can pin. A selfish guest could map its entire physical memory in the map cache, and ignore or disable the balloon. In such cases, setting a quota $Q(t)$ that is strictly smaller than available memory $M(t)$ is critical for fair sharing of memory between multiple guests. The specific quota which should be set could depend on multiple factors (e.g., workload, quality of service the host wishes to provide to different guests, or I/O adapters in use) and determining it is left as future work.

It is interesting to note that there are circumstances where the host will want to limit the amount of DMAs done by the guest even when the guest is cooperative, for example in order to conserve memory bandwidth or because there is only a limited number of IOMMU mappings available. In such cases the host can also set $Q(t)$ to be strictly smaller than $M(t)$.

4. EXPERIMENTAL EVALUATION

As noted previously, if the host sets the quota to be larger than the workload's requirement, then on-demand mapping achieves a steady state where the workload's pages are mapped and no DMA remapping is needed, i.e., we achieve maximal performance from the point of view of DMA mapping. However, if the workload size is equal to the entire guest memory size, then we might as well use persistent mapping. We therefore evaluated the quota requirements of two common networking protocols, and show that the needed quota is related to the workload size rather than the guest's memory size, thereby demonstrating the benefits of on-demand mapping over persistent mapping.

Our setup consisted of two Lenovo M57p machines with the Intel Q35 chipset which includes VT-d. Each machine had a 2.66GHz dual-core Intel Core 2 Duo CPU with 4GB of memory. The machines were connected directly with a 1GbE cable. One Lenovo machine ran native Linux (Ubuntu 7.10 for x86_64) and the other ran Linux (Ubuntu 7.10 for x86_64) with KVM, with a single virtual machine running Fedora Core 8 (64 bit), with 1GB of memory. All runs, native and virtualized, used the on-board e1000e PCI-e NIC.

4.1 Quota Requirements of Common Workloads

We began by looking at applications which use the standard TCP/IP stack via the socket API. Due to its semantics, the `send` socket API must copy the data from the application buffer to a kernel buffer associated with that socket. The kernel buffer is then mapped for DMA and accessed by the NIC. Since the Linux memory allocator recycles pages, there is high likelihood that the same guest pages will be reused for socket buffers. Each socket has an upper bound on the socket buffer size, so the total number of mappings needed to cover all of the send and receive sockets at a given point is given by:

$$\begin{aligned}\text{Total} =& \text{SendBufferSize} \times \text{NumSendSockets} + \\ & \text{ReceiveBufferSize} \times \text{NumReceiveSockets}\end{aligned}$$

If the quota is larger than this total and there is a high probability of reuse then after the initial mapping, no further remappings will be required. On the other hand if the quota is smaller than this total and there is little reuse then the IOMMU translation table will constantly need to change. This means that the "right" quota is a function of the number of sockets in the system and the level of reuse. Both are likely to be fairly steady for a workload which is in a steady state.

To avoid copying of the data from userspace to kernel space most systems and specifically Linux offer a zero-copy API, namely `sendpage` and `sendfile`. In this case the data is passed to the device without being copied from userspace to kernel space.

`sendpage` can be used to map any number of pages, and the usage is completely dependent on the application. However in the case of `sendfile` the number of pages is bounded by the file size. We tested the Apache webserver and found that the optimal quota is equal to the sum of the sizes of

the files being served by Apache at a given point in time. Again, for a workload that is in the steady state, the needed quota is likely to be fairly steady.

4.2 Eviction Strategies

Except when the quota is equal to the entire guest's memory—which is wasteful—there will always be cases where the quota is not sufficient for a workload's needs. We therefore evaluated the eviction strategies described in Section 2.

In order to evaluate each strategy's performance separately from a specific implementation, we began by considering the map cache hit rate that is achieved by each strategy on different workloads. We recorded I/O access patterns of real workloads and applied each of the strategies to them. As described in Section 2.1, we assume that batching has no additional cost. This assumption is not completely accurate in the real world, since it is dependent on the IOMMU implementation details.

We compare the following eviction strategies for selecting the page to evict from the cache:

- **FIFO** Evict pages in a first in first out order.
- **LRU** Evict the least recently used page.
- **OPT** Evict the page that is going to be used later then any other page in the cache. This is the optimal *offline* algorithm *without batching*, i.e., only a single page is replaced at a time.
- **Optimal batching** The optimal *offline* algorithm, but with batching as defined in Section 2.3. Multiple pages can be replaced at the same time.
- **Prefetching** The prefetching algorithm as described in Sections 2.5 and 3.3.

We recorded access patterns of two workloads, `netperf` send with a 65KB socket buffer, and an Apache webserver serving an `httperf` client that is requesting static wiki pages. For each workload we calculated the total working set size which is the total number of pages that are accessed during the run. We executed each of the eviction strategies for different quota values varying from 0 to 100% of the working set size. For each execution we calculated the cache hit rate. We note that with both workloads the total workload size depended on the workload rather than the guest memory size.

Figure 3 shows the hit rate for a `netperf` send access pattern for each of the eviction strategies and for different quota values. The basic strategies, non-batching LRU and FIFO, might be useful for large quota due to their simplicity, but are highly inefficient for low quotas. The optimal batching strategy performs significantly better and achieves close to 100% hit rate even for quota that is 10% of the working set! This indicates the great potential of batching to reduce the DMA mapping CPU utilization overhead. The prefetching strategy takes advantage of extra knowledge to batch multiple page replacements together rather than replacing pages one by one. With this extra knowledge it achieves better results than the non-batching strategies including the optimal non-batching strategy.

Figure 4 shows the hit rate for an Apache access pattern for each of the eviction strategies. Again prefetching does very well, getting a 90% hit rate even with a quota that is only 10% of the working set.

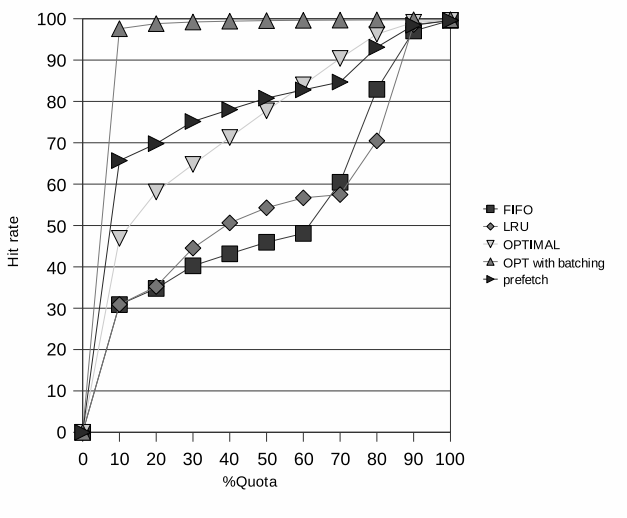

Figure 3: `Netperf` send: Hit rate vs. %Quota

4.3 CPU utilization

Next we evaluated the effect of the different batching options and prefetching on the performance of two workloads, as measured by the CPU utilization (all tests saturated the 1GbE link used). We used the same `netperf` workload with a 65KB socket size, and an Apache workload where the client is downloading various large files. Again we looked at the different quota values ranging from minimal quota (i.e., no caching allowed) to 100%, where the the quota is equal to the total total amount of pages that are required by the workload.

We evaluated the following optimizations:

- **LRU** Default LRU algorithm where no batching or caching is used.
- **Piggyback** Piggybacking unmaps on top of maps.
- **Prefetching** As previously described.
- **Batching** LRU with the map and unmap batching optimizations.

The first thing to note in Figures 5 and 6 is the high CPU utilization regardless of optimization when no caching is used (minimal quota), highlighting again the importance of improving the DMA handling for direct access. As expected the CPU utilization for IOMMU remapping is reduced as the quota is enlarged.

The piggyback optimization reduces CPU utilization by approximately 10% when compared with LRU and its impact is reduced as the quota increases. Prefetching reduces CPU utilization further, but its impact is reduced fast as the quota increases.

Piggybacking and prefetching require no driver changes. Observing the batching optimization we see that making changes to drivers can reduce the CPU utilization significantly. In the case of minimal quota the batching optimization halves the CPU utilization! Moreover, batching is the only optimization available when we wish to provide intra-guest protection. The benefit of batching is clear and leads us to the conclusion that significant improvement can be gained by changing drivers to batch mappings.

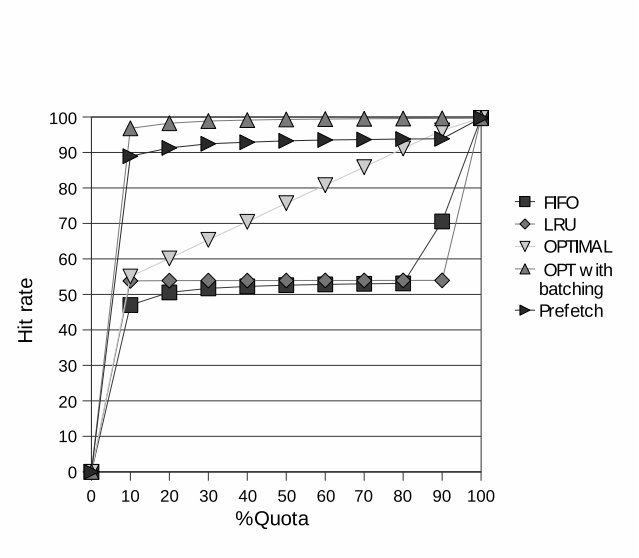

Figure 4: Apache: Hit rate vs. %Quota

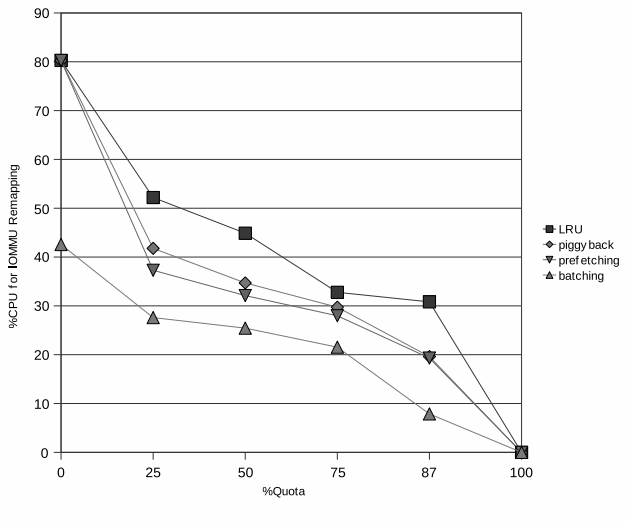

Figure 5: `Netperf` send: CPU utilization vs. quota

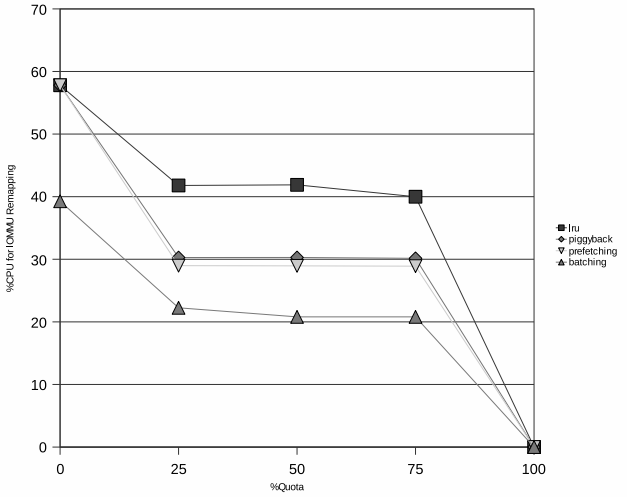

Figure 6: Apache: CPU utilization vs. quota

5. INTRA-GUEST PROTECTION

Every mapping strategy discussed in this paper, and in particular the on-demand mapping strategy, provides inter-guest protection, i.e., protection of one virtual machine from another. In some circumstances it is also desirable to provide intra-guest protection: protection inside a single virtual machine from malicious or buggy devices.

The only mapping strategies which provide intra-guest protection are single-use mapping and shared mapping. Unsurprisingly, they also have the lowest performance of the other mapping strategies. The on-demand mapping strategy, as formulated in Section 2.2, keeps around unmapped mappings, which opens a hole through which a buggy or malicious device could DMA to a page that is no longer being used for DMA. Therefore the on-demand mapping does not support intra-guest protection.

In order to adapt on-demand to intra-guest protection in Linux and close that hole, we need to add two pieces of information to every guest page which do not currently exist: what is the page used for, and who owns that page. If every page in the system was marked as either "allocated for DMA" or not, we could cache only pages which have been marked for DMA, thereby providing a limited amount of intra-guest protection. If in addition we knew which component owns a given page, we could keep a page mapped only as long as it was owned by that component.

The prevalent mode of DMA mapping for Linux device drivers is to map arbitrary pages for DMA and unmap them when done. Adding an "allocated for DMA" marker to every page can be done, by changing device drivers to allocate "DMA" pages through special interfaces (which already exist) rather than map them on the fly. However, changing every Linux device driver is a tall order. Even more problematic is the fact that a driver may be handed a page from some other entity (e.g., the TCP/IP stack) and asked to DMA to or from that page. Changing the way the different components in Linux interact is an even taller order and is left for future work.

Linux is a monolithic operating system, with no easily defined boundaries between different components (e.g., the page cache, TCP/IP stack, and device drivers). The different components freely pass around pages, which makes tracking page ownership difficult. Implementing the second form of intra-guest protection, where a page is only mapped as long as it is owned by the mapping component, requires both clear boundaries between components and tracking ownership. We believe that either would require extensive changes to Linux, but could be done fairly easily in micro-kernel based operating systems where different components run with different MMU address spaces.

We note that there is another potential relaxation of intra-guest protection, where the map cache is only used for caching read-only mappings of pages. This provides protection against a device writing to a page of memory it shouldn't (the most common form of device misbehavior that we would like to prevent), while providing a nice boost to workloads which mainly use DMA to read from memory rather than write to

it.

6. ANALYZING THE COST OF A SINGLE MAPPING

As noted in Section 1.2, in order to reduce the total cost of DMA mapping operations, one could either reduce the frequency of operations, or one could reduce the cost of a single mapping request. Just how expensive is it to create or destroy a single mapping?

Using the experimental setup detailed in Section 4, we measured the time needed to create and destroy DMA mappings, breaking the cycle cost down into different steps. In order to create a mapping, the following steps need to take place:

1. The guest needs to communicate to the host a request to create a mapping. This is most often done using a hypercall, although other techniques are also possible. A single empty hypercall took approximately 6000 cycles on our systems, just for the world switch.

2. The host needs to perform some internal implementation details such as retrieving the arguments to the hypercall from the guest's memory space, and translating the guest physical address to host physical address, which might also require faulting in a page in case it is non-present. This step took approximately 4400 cycles in our setup.

3. The host needs to update the IOMMU translation table (IOMMU translation table walk and creation of the new I/O PTE): this step took approximately 1400 cycles.

4. The host needs to flush the IOMMU write buffer: an additional 1200 cycles, on average. We note that with the Intel VT-d IOMMU in our systems, there is no need to flush the IOTLB when an I/O PTE goes from not present to present. In the general case an IOTLB flush may be required here as well.

In order to destroy a mapping, the following steps need to take place:

1. Again, the guest needs to communicate to the host a request to destroy a mapping: 6000 cycles for the hypercall.

2. The host needs to translate the guest page into a host physical address and perform other related bookkeeping. This step took approximately 2600 cycles in our setup.

3. The host needs to update the IOMMU translation table (page table walk and clearing of the I/O PTE): this step took approximately 1100 cycles.

4. When destroying a mapping, the host must flush the entry out of the IOMMU's IOTLB, which took approximately 2000 cycles on average.

Clearly optimizations can be made in both hardware and software which will reduce the single boundary crossing cost, such as optimizing the transfer of hypercall arguments and making world switches more efficient. It is also possible to switch from a hypercall based communication mechanism to a polling mechanism wherein guest and host use shared memory areas to communicate map and unmap requests. But as long as the cost of a request is not negligible, we should also strive to make less requests.

7. CONCLUSIONS AND FUTURE WORK

Efficient DMA mapping is a challenge for virtual machine direct access to I/O devices. Using a theoretical framework for the DMA mapping problem and a quota-based model, we developed the on-demand DMA mapping strategy. On-demand DMA mapping provides the best DMA mapping performance for a given amount of memory pinned for DMA.

There are two complementary aspects to on-demand mapping. From the host's perspective, the question is what is the right quota for a given guest. From the guest's perspective, the question is how to achieve the best performance with a given quota. When given a quota that is sufficient for the workload, on-demand provides maximal performance. When the quota is smaller than the workload's needs, a heuristic prefetching algorithm that takes advantage of repeating access patterns in I/O workloads can improve performance, without requiring driver modifications. If driver modifications are feasible, batching of map and unmap requests can provide even better performance. In both cases more extensive changes or more computationally-intensive algorithms might reduce the number of re-mappings even further.

With this work we minimized the DMA mapping overhead and made direct access performance closer to bare-metal, but closing the gap completely requires dealing with other sources of overhead such as interrupts. In addition, providing intra-guest protection, with minimal performance hit, and without requiring extensive changes to the guest operating system, still remains an open challenge.

Last, but certainly not least, we believe this work demonstrates that fundamentally, an I/O device should be considered just another core that reads and writes system memory. IOMMUs will end up resembling MMUs even more than they do today, and DMA memory management algorithms will keep inching closer to CPU memory management algorithms. The key missing ingredient for such unification, which we are also pursuing, is an efficient I/O page fault mechanism [27].

References

[1] *AMD I/O Virtualization Technology (IOMMU) Specification.* http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf.

[2] *Intel Virtualization Technology for Directed I/O Architecture Specification.* http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.

[3] 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(03):179–192, August 2006.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5):2–13, December 2006.

[5] 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 *SOSP '03: 19th ACM Symposium on Operating Systems Principles*.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems*, (5):78–101, 1966.

[7] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 71–86, July 2006.

[8] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The price of safety: Evaluating IOMMU performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.

[9] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, New York, NY, USA, 2008. ACM.

[10] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[11] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *J. Comput. Syst. Sci.*, 50(2):244–258, 1995.

[12] A. Borodin, P. Raghavan, S. Irani, and B. Schieber. Competitive paging with locality of reference. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 249–259, New York, NY, USA, 1991. ACM.

[13] Y. Dong, Z. Yu, and G. Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *WIOV '08: The First Workshop on I/O Virtualization*.

[14] A. Fiat and Z. Rosen. Experimental studies of access graph based heuristics: beating the lru standard? In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 63–72, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[15] K. Fraser, H. Steven, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[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. In *OSDI '08: 8th USENIX Symposium on Operating System Design and Implementation*.

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[18] J. Levasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI '04: 6th Symposium on Opearting Systems Design & Implementation*, page 2, Berkeley, CA, USA.

[19] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IPDPS '10: IEEE International Parallel and Distributed Processing Symposium*, 2010.

[20] J. Liu and B. Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2009. ACM.

[21] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX '06 Annual Technical Conference*, page 3.

[22] D. Magenheimer, C. Mason, D. Mccracken, and K. Hackel. Paravirtualized paging. In *WIOV '08: The First Workshop on I/O Virtualization*.

[23] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07*, pages 179–188, 2007.

[24] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10Gbps using safe and transparent network interface virtualization. In *VEE '09: 2009 Conference on Virtual Execution Environments*.

[25] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[26] J. R. Santos, Y. Turner, j. G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference*, pages 29–42, June 2008.

[27] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky. Scalable I/O—a well-architected way to do scalable, secure and virtualized I/O. In *WIOV '08: The First Workshop on I/O Virtualization*, 2008.

[28] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack—highly efficient network processing on dedicated cores. In *USENIX ATC '10: USENIX Annual Technical Conference*, 2010.

[29] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *2002 USENIX Annual Technical Conference*, pages 1–14.

[30] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI '02: 5th Symposium on Operating System Design and Implementation*.

[31] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference, 2008*, pages 15–28.

[32] 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 *High Performance Computer Architecture*, pages 306–317, 2007.

[33] P. Wyckoff and J. Wu. Memory registration caching correctness. In *CCGRID '05*, pages 1008–1015.

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