coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O

Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong, Intel Corporation

https://www.usenix.org/conference/atc20/presentation/tian

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O

Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, Yaozu Dong
Intel Corporation

Abstract

Direct assignment of I/O devices (Direct I/O) is the best performant I/O virtualization method. However, it requires the hypervisor to statically pin the entire guest memory, thereby hindering the efficiency of memory management. This problem can be fixed by presenting a virtual IOMMU (vIOMMU). Emulation of its DMA remapping capability carries sufficient information about guest DMA buffers, allowing the hypervisor to do fine-grained pinning of guest memory. However, established vIOMMUs are not widely used by commodity guests due to the emulation cost, thus cannot reliably eliminate static pinning in direct I/O.

We propose and implement *coIOMMU*, a new vIOMMU architecture for efficient memory management with a cooperative DMA buffer tracking mechanism. The new mechanism provides a dedicated interface for hypervisor and guest to exchange DMA buffer information over a shared DMA tracking table (DTT), orthogonal to the costly DMA remapping interface. We also explore two techniques: smart pinning and lazy unpinning, to minimize the impact on the performance of direct I/O. Our evaluation results show that coIOMMU dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost. Moreover, the desired semantics of DMA remapping can be sustained when cooperative tracking is enabled alongside. Overall, we believe that coIOMMU can serve as a reliable solution for efficient memory management in direct I/O.

1. Introduction

Direct I/O [1, 21, 29, 31, 37, 39, 48, 49, 50] is the best performant I/O virtualization method and a cornerstone capability in data centers and clouds. It allows the guest to directly interact with I/O devices without the intervention from software intermediary. An I/O memory management unit (IOMMU) [3, 14, 16] helps prevent Direct Memory Access (DMA) attacks in direct I/O by providing the capability of *DMA remapping*. Each assigned device is associated with an IOMMU page table (IOPT), configured by the hypervisor in a way that only the memory of the guest that owns the device is mapped. The IOMMU walks the IOPTs to validate and translate DMA requests, achieving inter-guest protection among directly assigned devices.

Most devices do not tolerate DMA faults, implying that guest buffers must be pinned in host memory and mapped in the IOPT before they are accessed by DMAs. However, the hypervisor does not know which pages are mapped by the guest when it is eliminated from the direct I/O path. Consequently, it has to pin the entire guest memory upfront, a.k.a *static pinning* [7, 44]. This heavily hinders the efficiency of memory management and worsens memory utilization, as pinned pages cannot be reclaimed for other purposes.

Presenting a virtual IOMMU (vIOMMU) [8, 23, 29, 52, 60] to the guest allows *fine-grained pinning* of guest memory for efficient memory management, although its primary purpose is to help the guest protect itself against buggy drivers. The hypervisor emulates the DMA remapping interface by: 1) walking the virtual IOPT (vIOPT) to identify the affected buffers; 2) pinning and unpinning the buffers in the host memory; and 3) mapping and unmapping them in the physical IOMMU to enforce protection as desired by the guest. Naturally, the emulation leads to a fine-grained pinning scheme, if the guest always uses the vIOMMU to remap its DMA buffers.

Unfortunately, established vIOMMUs are not applicable as a reliable solution for *fine-grained pinning*. Their virtual DMA remapping capabilities are disabled by most guests [8, 24, 30, 38, 51] in typical usages such as public cloud, because significant emulation cost may be incurred due to frequent mapping operations in the guest. Such cost could be alleviated through side-core emulation [8] or para-virtualized extension [23, 52]. However, the side-core emulation requires an additional CPU core to perform the emulation; and can only achieve optimal performance with deferred IOTLB invalidation, leading to compromised security. Para-virtualized extension reduces the virtualization overhead with optimized interfaces, but it still involves large number of VM-exits at the time of guest DMA mappings/unmappings, hence limiting the performance. Therefore, they did not change the fact that established vIOMMUs are used only in limited circumstances, e.g. when intra-guest protection is valued over the overhead of DMA remapping.

We argue that mixing the requirements of protection and pinning, through the same costly DMA remapping interface, is needlessly constraining. Protection is a guest requirement, while pinning is for host memory management. The two do not always match, thus favoring one may easily break the other. Instead, we aim to provide a reliable solution for fine-grained pinning by decoupling it from protection.

We propose and implement a new vIOMMU architecture called coIOMMU, which helps the hypervisor achieve efficient memory management in direct I/O. It introduces a dedicated mechanism for cooperative DMA buffer tracking, orthogonal to the costly DMA remapping interface. coIOMMU allows the hypervisor and guest to communicate over a DMA tracking table (DTT) located in a shared memory region. The guest records the mapping status of its DMA buffers in the DTT and the hypervisor walks the DTT to identify the corresponding pinning requirement. coIOMMU further minimizes the number of notifications from the guest, with two optimizations: (1) *smart pinning*, which heuristically pins frequently used pages and timely shares its pinning status with the guest, to enable precise notification in guest-mapping operations; and (2) *lazy unpinning*, which asynchronously unpins guest pages to eliminate notifications in guest-unmapping operations. On the other hand, the new mechanism does not affect the desired semantics of DMA remapping. It can be enabled with or without DMA remapping, as a reliable and standard interface to achieve fine-grained pinning in direct I/O.

We implement coIOMMU by extending KVM/QEMU vIOMMU and Linux guest. The concept and implementation can be easily ported to other hypervisors, vIOMMUs and guest OSes. Overall, the main contributions of this paper are:

- Observing that established vIOMMUs cannot reliably fix the problem of static pinning in direct I/O, due to the costly DMA remapping interface.
- Proposing and implementing coIOMMU, the first vIOMMU that introduces a dedicated DMA buffer tracking mechanism for fine-grained pinning.
- Introducing smart pinning and lazy unpinning to dramatically reduce the tracking overhead in fine-grained pinning.
- Conducting comprehensive evaluations under different Linux protection policies, with benchmarks in direct networking, storage, and GPU.
- Demonstrating that coIOMMU not only dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost, but also sustains the desired security as required in specific protection policies.

The rest of the paper is organized as follows. The background and motivation are first provided in section 2. We present the design of coIOMMU in section 3 and its implementation in section 4. Finally, the evaluation results are shown and discussed in section 5, with future work and conclusion drawn in section 6.

2. Motivation

2.1. The Problem

Direct I/O is the best performant I/O virtualization method by enabling direct communication between the guest and the I/O devices. Removal of the software intermediary not only provides much better performance than other I/O virtualization approaches, but also allows faster time-to-market for virtualizing new I/O acceleration capabilities. Direct I/O proliferates via device-side virtualization. Single-Root I/O Virtualization (SR-IOV) [1, 13] allows the device to multiplex its resource into virtual functions, each independently assignable to a guest. Cloud service providers even offload para-virtualized backend drivers into directly assigned devices [11, 12]. With these hardware trends, direct I/O has gained mainstream support in commodity hypervisors and is becoming a cornerstone capability in data centers and clouds.

IOMMUs [3, 14, 16] are introduced by hardware vendors to prevent assigned devices from touching arbitrary memory locations. Use of the IOMMU leads to the static pinning problem due to two factors: (1) most I/O devices do not tolerate DMA faults, and (2) the hypervisor does not know how guest memory is used for DMA. The hypervisor has to pin the entire guest memory upfront, assuming that every guest page might be a DMA page. This heavily hinders the efficiency of memory management and worsens memory utilization, as pinned pages cannot be reclaimed for other purposes.

2.2. Existing Solutions

Previous studies generally tackle this problem in two directions: making the device support DMA page faults or exposing the DMA buffer information to the hypervisor through software approaches.

DMA page faults allow all kinds of memory optimizations that CPU page faults provide. The PCI-SIG standardizes the support of DMA page faults with Address Translation Service (ATS) and Page Request Service (PRS) [2]. It was originally introduced to simplify the programming model on GPUs [27, 41, 42] and now also starts to find its way into NICs [6] and FPGA [9]. However, the latency of handling DMA page faults is 3x-80x higher than that of handling CPU faults [6, 40]. Such long latency, up to hundreds of microseconds, demands a larger on-device buffer to hold in-flight requests and incurs higher device cost. Handling such long latency in all critical paths further complicates the device. Therefore, most commodity devices do not support DMA page faults, or partially support it only for selective workloads. With time, it may become a preferable way for fine-grained pinning, but not anytime soon.

Alternatively, researchers also look at software approaches to expose enlightened guest DMA information to the hypervisor. Knowing when a guest page is mapped or unmapped allows the hypervisor to pin or unpin it dynamically. Willmann et al. [44] evaluates several mapping strategies, revealing that a big performance penalty is incurred when blindly doing hypercalls to notify the hypervisor of every guest mapping/unmapping operation. Yassour et al. [7] dramatically reduces such notifications with a guest-side pin-down cache. However, it puts a complex eviction policy in the guest and provides no intra-guest protection.

Presenting a vIOMMU [23, 29, 60] also provides sufficient information for fine-grained pinning, as a result of emulating its DMA remapping capability for intra-guest protection. However, such emulation may incur significant cost, especially when frequent mapping operations are requested by the guest. To trade off performance and protection, modern OSes typically implement different policies about DMA remapping. For example, Linux [8, 24, 30, 38, 51] implements *strict*, *lazy* and *passthrough* policies. Although DMA remapping is used in *strict* and *lazy* policies, the *passthrough* policy simply disables it to gain best performance. Obviously, the guest cannot provide any DMA buffer information to the hypervisor when the *passthrough* policy is selected. Unfortunately, major guest Linux distributions choose *passthrough* as default and even allow different policies across devices.

Recent studies focus on reducing the cost of emulating DMA remapping in vIOMMU. Tang et al. [52] reduces the remapping overhead by reusing old mappings and delaying their removal, however, at the cost of compromised security. Side-core emulation [8] achieves 100% of 10Gbps line rate with a fully emulated vIOMMU, but with relaxed protection and increased total cost. The overhead of DMA remapping is also tackled on bare metal [24, 30, 38]. While these works generally apply to the guest OS as well, most of them have not been adopted by commodity OSes due to its intrusiveness. In a nutshell, the cost of DMA remapping is still notable in the guest today, leaving the capability disabled or even not exposed in most cloud and data center usages.

2.3. DMA Tracking vs. DMA Remapping

We prefer the vIOMMU approach for two reasons: 1) it supports both intra-guest protection and fine-grained pinning; and 2) DMA page faults are not widely supported by commodity devices. However, we want to go a different direction from previous studies – to enable fine-grained pinning without being encumbered by the intrinsic cost of DMA remapping.

We argue that mixing the requirements of protection and pinning, through the same costly DMA remapping interface, is needlessly constraining. Protection is a guest requirement and relies on the DMA remapping capability, while pinning is for host memory management and needs the capability of tracking guest DMA buffers. The two do not always match, thus favoring one may just break the other, if both are enabled through the same interface. For example, the hypervisor either must fall back to static pinning by assuming that most guests disable protection, or, adopt fine-grained pinning by forcing all guests to enable protection and bear with added cost.

What about inventing a separate DMA buffer tracking mechanism to the vIOMMU, without relying on any semantics of DMA remapping? Separating DMA tracking from DMA remapping allows us to tackle the pinning and protection problems in parallel. If the new tracking mechanism incurs negligible cost, we can expect most guests to always enable it and reliably provide necessary information for fine-grained pinning. If feasible, such an approach would make the vIOMMU as the portal of efficient memory management in future data centers and clouds.

3. Design

We propose coIOMMU, a new vIOMMU architecture that helps the hypervisor achieve efficient memory management in direct I/O. coIOMMU provides a dedicated DMA buffer tracking mechanism that adopts a shared memory interface for efficient communication between host and guest. The guest records the mapping status of its DMA buffers through a shared DMA tracking table (DTT), for the hypervisor to decide its pinning strategy. coIOMMU also introduces two optimizations: smart pinning and lazy unpinning, to dramatically reduce the performance impact when achieving fine-grained pinning.

3.1. Goals

We want the new DMA buffer tracking mechanism to meet these goals:

Orthogonal to DMA Remapping - Our solution should allow DMA buffer tracking and DMA remapping independently configured by the guest. The new tracking mechanism, once enabled, should consistently supply sufficient information for fine-grained pinning, regardless of how DMA remapping is configured to protect guest. Enabling of DMA buffer tracking should not affect the desired protection semantics of DMA remapping.

Low Cost - DMA buffer tracking should incur negligible cost. Otherwise, it faces the same challenge as in DMA remapping: if significant cost is observed, why would one enable it by default? We focus on the efficiency of DMA buffer tracking itself and have no intention to further optimize DMA remapping in this work. The original performance expectation under each guest protection policy is set as the baseline for comparing the cost of DMA buffer tracking in our evaluations.

Non-intrusiveness - We want our solution to minimize the changes in the guest software stack, as a primary factor to gain mainstream support in commodity OSes. Commodity OSes provide a generic DMA API layer [25, 43] to route

DMA mapping requests from device drivers to underlying DMA driver. DMA buffers can be tracked either in the DMA API layer or specific DMA driver. We did not choose DMA API because any change in such common framework usually takes a long time to be adopted by commodity OSes.

Wide Applicability - We prefer a solution that works with all kinds of I/O devices rather than requiring additional changes in hardware or device drivers. We also expect such a solution to make no assumption on any vendor specific characteristics, so it can be easily ported to different vIOMMUs, either emulated or para-virtualized.

Extensibility - The solution should be extensible to help address other limitations in memory management. For example, another challenge in direct I/O is about lively migrating the guest with assigned devices, which requires the ability of tracking the pages that are dirtied by DMAs [20, 26, 28, 35]. We expect our solution can play as a portal of tracking all kinds of DMA buffer status for efficient memory management.

3.2. Architecture

The coIOMMU architecture is illustrated in Figure 1, composed of coIOMMU backend in hypervisor and coIOMMU driver inside the guest. The coIOMMU backend includes three main components: (1) DMA remapping engine (*remapEngine*), the same functionality for intra-guest protection as in established vIOMMUs, over a set of per-device vIOMMU page tables (vIOPTs); (2) DMA tracking engine (*trackEngine*), a new function dedicated for tracking guest DMA buffers over a global DMA tracking table (DTT); and (3) Page-pinning manager (*pManager*), which uses the information gathered by trackEngine to intelligently manage the pinning requirements of guest memory. The remapEngine and trackEngine are independently enumerated and managed by the coIOMMU driver, while pManager is hidden and activated automatically when trackEngine is enabled.

In our prototype, we build coIOMMU by extending an existing vIOMMU, which emulates the Intel VT-d hardware [3]. This allows us to focus on the new trackEngine and pManager, while inheriting the established DMA remapping logic as remapEngine. However, we make no assumption on the specific hardware or vIOMMU type. The design of trackEngine and pManager can be easily ported to any emulated or para-virtualized vIOMMU.

The trackEngine holds the base address of the DTT, which is allocated and registered by the coIOMMU driver. The format of the DTT is a hierarchical page table, containing the mapping information required by fine-grained pinning. trackEngine also includes a doorbell register to notify the hypervisor if necessary. Within the coIOMMU backend, trackEngine provides interfaces for pManager to access the DTT and also notifies pManager when the doorbell is rung. With this design, trackEngine acts as a standard interface solely for conveying the DMA information, while pManager actually uses the information to achieve fine-grained pinning. The separation between these two components allows coIOMMU to be easily extended for other purposes, e.g. by introducing another agent to track dirty pages, alongside pManager, while reusing the same trackEngine interface.

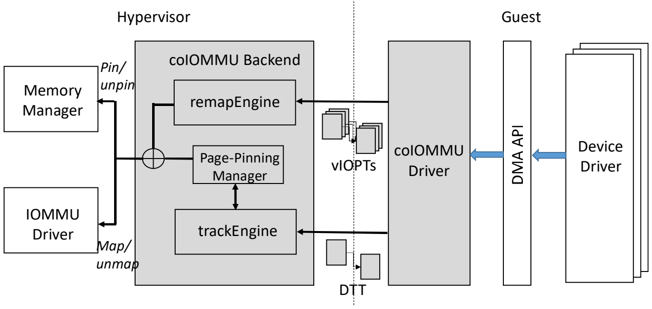


Figure 1: The architecture of coIOMMU

The coIOMMU driver intercepts the DMA API operations in the guest and updates the DTT accordingly. Modern OSes all implement a generic DMA API layer [25, 43], connecting device drivers to the underlying DMA driver to prepare their DMA buffers. The coIOMMU driver registers itself as a DMA driver to capture the latest mapping status of guest DMA buffers. This driver also enforces the desired protection semantics, as other vIOMMU drivers normally do today. In this way, DMA tracking is enabled without any change to the DMA API layer or specific device drivers of the guest.

The pManager contains hypervisor-specific policies for fine-grained pinning. A specific implementation may even include multiple policies and let the hypervisor dynamically choose a policy at runtime. We demonstrate two optimizations in §3.4: smart pinning and lazy unpinning, to minimize the notification overhead. When required, pManager talks to the memory manager for pinning or unpinning a set of guest pages and request the IOMMU driver for mapping or unmapping them in the physical IOPT. When both remapEngine and pManager are enabled, their pinning decisions are ORed together to favor the stricter requirement. Once a guest page is unpinned and unmapped, it can be reclaimed under whatever policy applied by the memory manager.

3.3. DMA Tracking Table (DTT)

The DTT records the mapping status of guest DMA buffers. It is shared by all assigned devices because the hypervisor only wants to know the DMA buffers of the entire guest. It is not necessary to track DMA buffers for virtual devices, assuming their DMAs are emulated by and already known to the hypervisor. The DTT is allocated by the guest, starting as empty and then filled dynamically according to intercepted DMA operations. We choose to track two categories of guest pages in the DTT: 1) the pages that are currently mapped by the guest and 2) the pages that have been unmapped but still pinned by the hypervisor. The latter category is necessary for lazy unpinning introduced in the next section.

One may argue why inventing a new table instead of reusing the vIOPTs, when the latter also carry the information of guest DMA buffers. We considered this approach but gave up for several reasons. First, the vIOPT is designed for intra-guest protection which disallows pinning a page after it is unmapped thus also negates lazy unpinning. Second, the table is indexed by guest I/O Virtual Address (IOVA) for the remapping purpose. The hypervisor has to walk every vIOPT to find out whether a guest page is mapped, which is too costly. Last but not the least, the format of vIOPT is typically vendor-specific, so extending it may not lead to good portability.

The DTT is a 4-level page table in 4KB pages, as shown in Figure 2. The 4KB leaf page consists of 512 DTT PTEs (DTEs) and each 8-bytes DTE is further split into 8 tracking units (TU). Each TU corresponds to one 4KB guest page. In total, the DTT can support up to 51-bits (9+9+9+9+3+12=51) guest physical address width, big enough for prevalent virtualization usages. Such design leaves 8-bits available in each TU. coIOMMU currently uses 3 bits for fine-grained pinning, with the other 5-bits reserved for future extension:

- 'M (mapped)', indicating a page currently mapped by guest for DMA. It is set and cleared by the guest before and after the corresponding DMA and is read-only to the hypervisor. This bit conveys the primary information used by fine-grained pinning.
- 'P (pinned)', marking a page currently pinned by the hypervisor. It is updated by the hypervisor to reflect the pinning status and is read-only to the guest, necessary for smart pinning.
- 'A (accessed)', telling whether a page has ever been used for DMA. The guest sets this bit alongside the setting of M-bit ('mapped' bit). Then it stays sticky until the hypervisor clears it in lazy unpinning.

An entry with both M and P bits cleared marks the page as invalid. If every entry of a DTT page is invalid, the guest may choose to free this page to save space.

3.4. Fine-grained Pinning

Two techniques are introduced in coIOMMU: *smart pinning* and *lazy unpinning*, to minimize the notification overhead of fine-grained pinning. We focus on the scenario where the DMA remapping capability of coIOMMU is disabled by the guest. In this case, there is no intra-guest protection requirement thus the hypervisor can pin more pages than what guest actually maps.

3.4.1. Smart Pinning

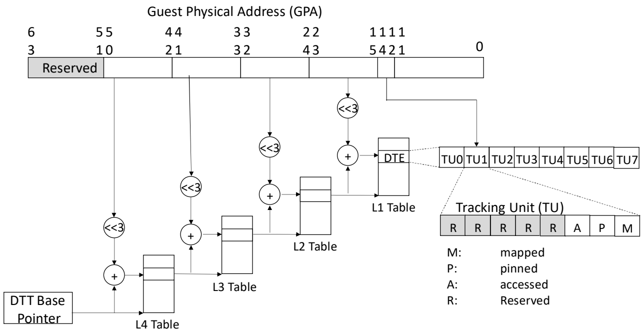


Figure 2: the format of the DTT

coIOMMU manages the pinning of guest pages in three ways: (1) *instantly pinning*: the guest instantly notifies the hypervisor to pin pages when they are being mapped, for correctness; (2) *precise notification*: the guest notifies the hypervisor if and only if the to-be-mapped pages are not pinned, to minimize the notification overhead; and (3) *speculatively pinning*: pManager heuristically pins the frequently used pages for performance.

First, pinning must be *instantly* done before any mapped page is used for DMA, because most devices do not tolerate DMA faults, as aforementioned. In such circumstance, the hypervisor must be notified by the guest to complete the pinning action in a timely manner, if the page has not yet been pinned.

Second, coIOMMU exposes the pinning status to the guest through the P-bit ('pinned' bit) in the DTT, for *precise notification*. If the P-bit is cleared by the hypervisor, the guest must notify the hypervisor instantly when mapping a page. Otherwise, no notification is needed at all. This optimization allows the guest to skip most notifications in its mapping operations.

Last, pManager *speculatively* selects and pins frequently used pages by leveraging the guest DMA locality, which has been identified in both previous studies [7, 44, 51] and our evaluation. The DTT includes an A-bit ('accessed' bit) to mark a page ever used for DMA. The guest sets the A-bit when mapping a page and leaves it set until the hypervisor clears it. pManager determines the ages of unmapped pages by periodically scanning the A-bits (and clears it after a scan). Young pages (with A-bit set) are candidates of frequently used pages and might be accessed soon again. So pManager heuristically pins them to avoid the overhead of another pinning notification in the near future.

Our evaluation shows that precise notification and speculative pinning can dramatically reduce the notification overhead in instant pinning by up to 99.9992% (from 1.5M to 11 notifications, per second), when running memcached with a 40Gbps NIC connection. One notification takes ~2000-4000 cycles in our evaluation, so 1.5M notifications per second may eat up 1-2 CPU cores without such optimization.

3.4.2. Lazy Unpinning

The pManager *lazily* unpins guest pages to completely eliminate the notification overhead in guest unmapping operations. It asynchronously scans the DTT to find out the pages that are unmapped but still pinned, and then unpins them in a batch. In our prototype, we process *lazy unpinning* and *speculative pinning* together in the same thread. Unpinned pages are reclaimable by the memory manager to increase overall memory utilization. In the same example of memcached, lazy unpinning eliminates another 1.5M notifications per second for guest unmapping operations, which means saving another 1-2 CPU cores, with the cost of pinning additional ~1% memory (0.32MB) than the total size of mapped pages (34.68MB), in average.

3.5. Intra-Guest Protection

The DMA remapping engine (remapEngine) can achieve fine-grained pinning as well, as it is required to precisely map and pin DMA buffers per guest protection requirements. However, one cannot solely rely on DMA remapping because the guest may selectively turn it off for certain devices according to its protection strategy. We describe two examples as below.

First, the guest may dynamically enable/disable DMA remapping for an assigned device, leaving the hypervisor to switch back and forth between static pinning and fine-grained pinning. For example, guest Linux typically enables DMA remapping when assigning a device to its user space and then disables remapping when returning the device back to its kernel space [45]. The switch between static and fine-grained pinning may lead to intermittent out-of-memory errors in a budget system. Moreover, the hypervisor needs to unpin all the guest pages when switching away from and then re-pin them when switching back to static pinning, leading to increased overhead.

Second, if the guest enables DMA remapping only for selected devices, DMA remapping cannot provide full DMA buffer information for fine-grained pinning. For example, most Linux distributions enable DMA remapping only for untrusted devices, based on physical characteristics of the device [61, 62]. Such flexible configuration is possible because DMA remapping is typically enabled per device. However, fine-grained pinning needs to know DMA buffers used by all assigned devices in the guest, even for the ones that are not protected with DMA remapping. In such case, the hypervisor must fall back to static pinning with reduced memory utilization.

In both of these examples, DMA buffer tracking of coIOMMU allows reliably providing full DMA buffer information to enable fine-grained pinning. When tracking and remapping are both enabled, it is possible for the two to make different pinning decision for the same page. In such case, the decision from the DMA remapping interface takes precedence, because we must not break any protection semantics desired by the guest.

4. Implementation

We implement coIOMMU by extending the virtual Intel VT-d, which is an emulated vIOMMU in QEMU [58] (the device model of KVM hypervisor [10]), and the intel-iommu driver in the guest Linux. In QEMU, the original DMA remapping logic of the virtual VT-d is reused as remapEngine, while trackEngine and pManager are developed from scratch. Guest-side changes are all contained in the intel-iommu driver and hidden behind the Linux DMA API layer. There is no change required in guest device drivers. Currently, coIOMMU adds ~700 LOC in QEMU and ~1000 LOC in guest.

coIOMMU driver - coIOMMU driver extends guest intel-iommu driver to manage the trackEngine when the capability is detected. The intel-iommu driver registers callbacks to the Linux DMA API layer for mapping and unmapping DMA pages in different forms, e.g. for single page or scatter-gathered page list, for pre-allocated pages or newly allocated pages, etc. We extend the driver by extracting the DMA buffer information from those callbacks and updating the corresponding tracking units (TUs) in DTT. The DTT is allocated in the guest memory, which is always accessible by the commodity KVM hypervisor. If such direct access is prohibited in some specific security related usage cases [55, 56], the DTT should be allocated in a shared memory region. Last, the coIOMMU driver conditionally notifies the hypervisor based on the DTT status.

trackEngine - We extend the virtual VT-d with several changes: (1) a capability bit for enumerating the presence of trackEngine, (2) an enabling bit for activating trackEngine, (3) a register holding the base address of the DTT, (4) a register as the doorbell interface for triggering notification to pManager, and (5) a register pointing to the base address of the notification structure. The notification structure is designed to allow batching requests of multiple pages into one notification, in case of those pages are mapped together. trackEngine also provides function calls for pManager to scan and update the DTT.

pManager – The implementation of pManager can be split into two parts. First, it provides direct function calls for trackEngine to complete instant pinning. The functions are invoked synchronously in the vCPU threads when QEMU emulates the guest write to the doorbell register. Second, pManager also launches a thread for lazy unpinning and speculative pinning, woken up every one second. This thread scans the DTT to find out all the pages that are unmapped but still pinned and speculatively unpin them based on their A-bits. When a pinning decision is made, pManager invokes the VFIO API

[45] to pin/unpin selected pages and map/unmap them in the IOMMU.

Sub-Page Mappings - Multiple DMA buffers may co-locate in the same 4KB guest page, e.g. as widely observed when handling network packets. Sub-page mappings imply that one page might be mapped and unmapped multiple times. In such case, coIOMMU driver tracks the mapping count of each mapped page and clears the "M-bit" of the corresponding entry only when its count reaches zero. We choose to leverage the 5 reserved bits in each TU as the mapping count, holding up to 31 sub-page mappings. Doing so simplifies the implementation and works well in our evaluations. Other implementations may choose different structures for such tracking purpose.

Concurrency - coIOMMU must properly handle concurrent pinning/unpinning requests between multiple vCPU threads and the unpinning thread, as shown in Figure 3.

First, multiple vCPUs may try to map and pin the same DMA page simultaneously, e.g. in sub-page mapping scenario. We employ different locking mechanisms in guest and host for race avoidance. Within the guest kernel, spinlock is required for atomically setting the 'mapped' flag and checking the 'pinned' status of a target page. It is necessary as DMA mappings may happen in the guest interrupt context. On the other hand, a mutex is introduced in QEMU for atomically completing the actual pinning actions: 1) rechecking the 'pinned' status; 2) pinning the page; and 3) updating the 'pinned' flag.

Second, race condition may happen between concurrent pinning requests (from the vCPU threads) and unpinning requests (from the unpinning thread). For example, it is possible seeing an unpinning operation starts before, yet completes after, an in-flight pinning request. Such race may lead to the pinning request completing successfully but with the target page actually unpinned. We introduce two mechanisms to solve this problem. For one, the unpinning thread needs to check the 'mapped' flag before and after clearing the 'pinned' status. We call this special sequence as double-detection, necessary to catch in-flight change of the mapping status in the guest side. For two, the unpinning thread also needs to acquire the aforementioned QEMU mutex for completing its unpinning actions. In particular, the second check of the 'mapped' flag must be done with the mutex acquired and before conducting the unpinning action. If the 'mapped' status becomes true, indicating that a pinning action is in progress for the target page, the unpinning thread should cancel the unpinning operation immediately.

4.1. Discussion

Applicability - coIOMMU applies to all kinds of directly assigned devices, without the need of ad-hoc changes in hardware or software. Porting our Linux implementation to a new guest OS is straightforward, as long as the OS implements a generic DMA API layer which, obviously, is already a common feature in commodity OSes today. On the other hand, the implementation of trackEngine and pManager is vendor-neutral and self-contained. The separation between DMA tracking and DMA remapping allow coIOMMU implementation to be easily portable to other vIOMMUs, regardless of whether remapEngine is emulated or para-virtualized.

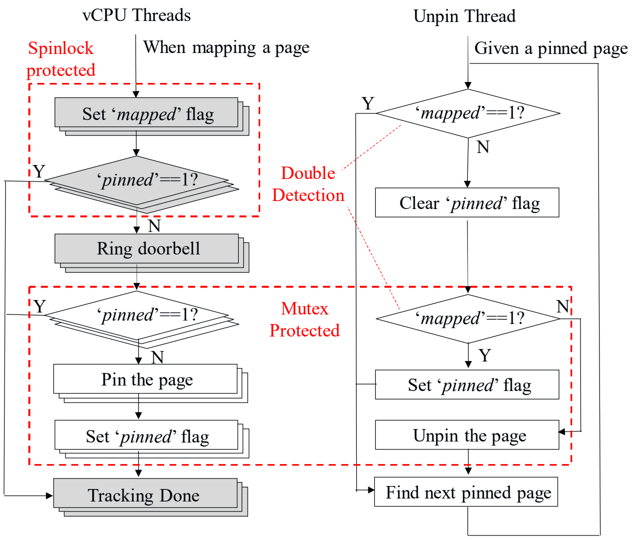


Figure 3: Race avoidance between concurrent pinning and unpinning operations. Gray boxes are guest actions, and white are host.

Extensibility - The page table format of the DTT can be extended to address other limitations in memory management. For example, introducing a "D (dirty)" bit in the TU provides a generic solution for tracking dirty pages when lively migrating VMs in direct I/O. Similarly, using a "W (writable)" bit to indicate read-only page enables the hypervisor to implement copy-on-write features. Ideally, a specific implementation may extend the DTT to include the same set of permission or status bits as available in a CPU page table.

Currently the DTT tracks DMA buffers in 4KB granularity. It is sufficient for most direct I/O usages, as DMA buffers are typically allocated in scattered 4KB pages. When large DMA buffer is used, we rely on pManager to merge batched pinning requests on continuous DMA pages into 2MB-based requests. We observed such optimization leads to ~4.5% FPS improvement in direct GPU benchmark, as illustrated in 5.1. Alternatively, one may also directly extend the DTT format to support 2MB-granular tracking entries.

Kernel Bypassing - coIOMMU also applies to various kernel bypassing techniques [32, 33, 45], which allow applications to directly manage DMA buffers in user space. Applications are untrusted, so they must first register a trunk of memory to the kernel and then manage within that trunk. The registration goes through proper kernel interfaces, e.g. AF_XDP [33] or VFIO [45] in Linux, which finally call into the coIOMMU driver for actual mappings and unmappings thereby are still tracked in the DTT. Kernel bypassing may increase the memory footprint because applications usually register a one-off big buffer pool to avoid calling into the kernel frequently. We leave optimizing such workloads as future work.

DMA Page Faults – For devices which do support DMA page faults, on-demand memory allocation/reclaim can happen at any time thus one could implement fine-grained pinning without using coIOMMU. However, coIOMMU may still provide two benefits in such circumstance. First, the overhead of handling DMA page faults might be non-negligible in hot data paths. coIOMMU allows the guest to reduce the number of faults by proactively requesting pre-pinning of hot pages, based on the knowledge that is easily extracted from DTT, yet invisible or difficult to acquire in legacy host. Second, some devices may allow DMA page faults only in selective data paths. Hypervisor could enable coIOMMU alongside the fault-based pinning scheme, to track DMA pages which are touched in non-faultable data paths in such devices.

Guest Cooperation - coIOMMU is a para-virtualized approach thus requires guest cooperation. We plan to submit our work to Linux and QEMU community, so coIOMMU could be enabled by default in most Linux distributions in the future. However, it is possible that a selfish guest may deliberately report fake DMA pages or simply disable coIOMMU driver to get more pages pinned than a cooperative guest. When required, one may choose to build a quota mechanism alongside the new tracking interface of coIOMMU. For example, the memory ballooning mechanism [57] can be extended to convey the quota information of both total memory and DMA memory, based on the service level agreement of the guest. Afterward, pManager could reject new pinning requests from any guest after its quota is exceeded.

5. Evaluation

Our evaluation aims to answer several questions. How does the overhead imposed by coIOMMU compare to that of established vIOMMUs? How many pages are pinned in various direct I/O usages when using coIOMMU to enable fine-grained pinning? Does coIOMMU sustain the desired performance and security under different intra-guest protection policies? We answer these questions by planning our evaluation to focus on four aspects: footprint, overhead, security and applicability.

Evaluated Modes - We evaluate six modes as shown in Table 1. The guest intel-iommu driver supports three protection policies: 1) *passthrough*, the default policy that disables DMA remapping for performance; 2) *strict*, using DMA remapping to gain full protection; and 3) *lazy*, trading off some security for performance when using DMA remapping (e.g. by deferring and batching IOTLB invalidations). We study the three policies for coIOMMU and a state-of-the-art vIOMMU, respectively, thus leading to six modes in total. In our prototype, coIOMMU inherits the DMA remapping logic of the virtual VT-d, so we choose this emulated vIOMMU solution to represent state-of-the-art vIOMMUs for fair comparison. We use {*PT-O*, *ST-O, LA-O*} to indicate the three protection policies with virtual VT-d and {*PT-N*, *ST-N*, *LA-N*} for the policies with coIOMMU. 'O' stands for the 'old' emulated VT-d while 'N' represents the 'new' coIOMMU.

mode	abbr.	DMA remapping	DMA buffer tracking	pinning model	protection
passthrough (virtual VT-d)	*PT-O*	unused	n/a	static	no
passthrough (coIOMMU)	*PT-N*	unused	used	fine-grained	no
strict (virtual VT-d)	*ST-O*	used	n/a	fine-grained	full
strict (coIOMMU)	*ST-N*	used	used	fine-grained	full
lazy (virtual VT-d)	*LA-O*	used	n/a	fine-grained	relaxed
lazy (coIOMMU)	*LA-N*	used	used	fine-grained	relaxed

Table 1: Evaluated modes in coIOMMU and virtual VT-d

Experimental Setup - Our setup consists of three machines, all running Ubuntu 16.04 with kernel 5.0.0. The primary machine, used for networking and storage tests, is equipped with a 16-core Intel Xeon Cascade Lake CPU at 2.7GHz, one 64GB DDR4 DIMM, an Intel XL710 40Gbps NIC, and two Intel 760P series 1TB NVMe SSDs. The 2nd machine acts as the network traffic generator, with another XL710 NIC connected to the primary machine back-to-back. It includes dual Intel Xeon Gold 6140 CPUs, each with 18 cores at 2.30GHz and 64GB DDR4 memory. The last machine is used for GPU evaluation, equipped with Intel Core i7-7567U CPU with four cores at 3.50GHz, 32GB DDR4 memory, a 256GB Intel 520 series SSD, and an Intel® Iris® Plus graphics 650 GPU.

The VM of the first machine is based on RHEL7.2 with kernel 5.1.0-rc3+, configured with 16 vCPUs, 32GB memory, and a directly assigned device – either a XL710 NIC or a 760P SSD, according to whether direct-networking or direct-storage is under evaluation. The two assigned devices are enabled independently, to avoid mutual interference from section 5.1 to section 5.5. In section 5.6, we evaluated their performance running combined workloads with both devices assigned. The VM for direct GPU includes Ubuntu 18.04 with kernel 5.1.0-rc3+, 4 vCPUs, 4GB memory, and a directly assigned Intel® Iris® Plus graphics 650 GPU. The vCPUs of both VMs are 1:1 pinned to the physical cores for stable results.

Benchmarks - We choose both micro-benchmarks and macro-benchmarks for evaluating the six modes in direct networking, direct storage and direct GPU:

- *Netperf* [63] is a standard micro-benchmark to measure networking throughput. We perform Netperf stream receive (RX) and transmit (TX) tests, using 64KB message size with 16 Netperf client/server instances (one per core) in the guest. Aggregated throughput is reported.

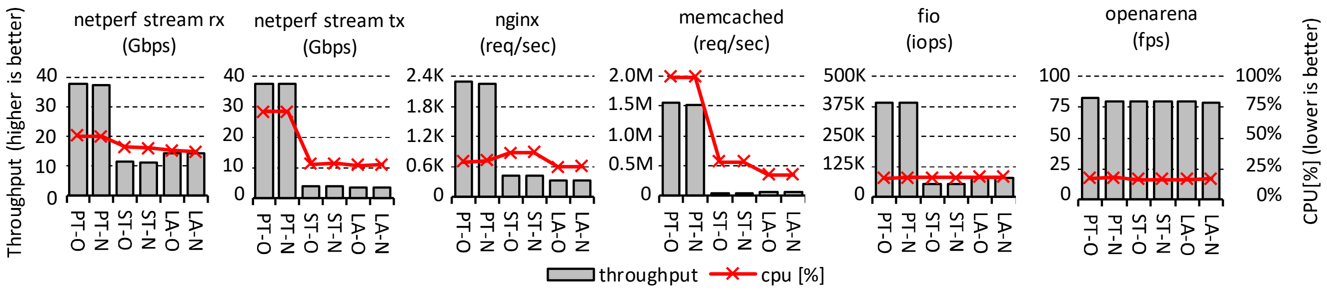


Figure 4: Performance of the six modes (100% CPU is 4 cores in openarena, and 16 cores in all other benchmarks)

mode	netperf stream rx		netperf stream tx		nginx		memcached		fio		openarena	
	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits	dma_ops	VM-exits
PT-O	352,224	0	577,037	0	525,974	0	3,110,716	0	781,055	0	44	0
PT-N	348,335	2,379	572,136	415	525,849	115	3,039,414	11	780,186	9	44	22
ST-O	109,403	109,403	64,448	64,448	72,239	72,239	104,354	104,354	109,864	109,198	44	44
ST-N	108,607	108,607	64,352	64,352	71,682	71,682	103,984	103,984	107,948	107,948	44	44
LA-O	141,844	71,013	59,645	29,896	63,230	31,702	145,309	72,744	163,085	81,655	44	23
LA-N	141,572	70,883	58,398	29,273	62,569	31,370	144,690	72,434	162,417	81,322	44	23

Table 2: The average number of completed DMA operations vs. incurred VM exits, per second.

- *Nginx* [64] is a high-performance HTTP web server. We use ApacheBench [69] to measure the number of concurrent requests that Nginx server can serve. We run ApacheBench to issue 16 concurrent requests of a static 1MB file, through the Nginx server installed in the guest.
- *Memcached* [65] is a popular in-memory key-value store, usually benchmarked using memaslap [70]. We use the default memaslap configuration with 64-byte keys, 1KB values, and 90%/10% GET/SET operations. In the VM, we launch 16 memcached instances driven by 16 memaslap threads each issuing 8 concurrent requests.
- *fio* [66] is a standard micro-benchmark to measure disk performance for wide range of storage types. We configure 16 fio threads, each performing asynchronous direct random reads from the assigned SSD, in 512-byte blocks and 128 in-flight requests.
- *OpenArena* [67] is a 3D first-person shooter game, used to benchmark direct GPU. The throughput is reported in frame-per-second (fps).

In addition, we also selectively run sysbench [68] as a memory benchmark and DPDK [32] for user-space networking stack, for specific evaluation purposes.

5.1. Overhead

We record the performance of aforementioned benchmarks in each evaluation mode, as shown in Figure 4. CPU utilization is aggregated over all cores, i.e. one core at 100% CPU would be reported as 100%/4=25% CPU utilization with 4 cores (for OpenArena) or 100%/16=6.25% CPU utilization with 16 cores (for all other benchmarks). In addition, we also capture the per-second number of completed DMA operations and associated VM-exits when running those benchmarks, in Table 2. All benchmarks run 30 seconds, except OpenArena, which must run to end in around 42 seconds. Next, we compare coIOMMU to virtual VT-d under the three Linux protection policies, respectively.

Passthrough - All networking benchmarks (left four in Figure 4) exhibit consistent results under the passthrough policy: coIOMMU (*PT-N*) retains the performance comparable to that of the virtual VT-d (*PT-O*), with less than 3% throughput degradation and negligible variation in CPU utilization. Such low cost is further explained in Table 2 – although hundreds of thousands of DMA operations are tracked per second, the majority of them do not trigger any VM-exit to notify the hypervisor, due to the optimization of *smart pinning* and *lazy unpinning*. For example, the lowest VM-exit number is observed in memcached, with only 11 VM-exits incurred by ~3M DMA operations.

The overhead of coIOMMU is unrecognizable in FIO but incurs 4.5% FPS drop in OpenArena. We found that OpenArena maps a big buffer (~240MB) in a batch at its launch time, with many pages adjacent to each other. In such case, pinning the buffer in 2MB size is more efficient than pinning in 4KB size, due to increased IOTLB efficiency. Unfortunately, 2MB pinning is not supported in our initial coIOMMU implementation, while it is the preferred option when KVM statically pins the entire guest memory in *PT-O*. After coIOMMU was extended to also conduct 2MB pinning for OpenArena, it then reaches the same performance as the virtual VT-d (not shown in the figure). We do not enable huge page pinning in other benchmarks, because they are observed with frequent mapping operations on many scattered 4KB pages. Blindly doing

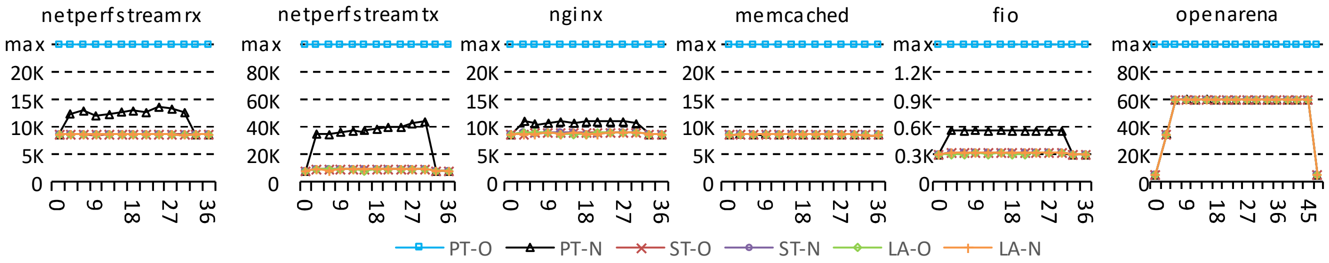


Figure 5: The number of pinned pages sampled in 3 second interval, taken from the beginning of the benchmarks to 6 seconds after their completion. 'max' indicates the total pages of guest memory.

huge page pinning simply adds more cost and footprint in those circumstances.

Strict and Lazy - We did not observe recognizable difference between coIOMMU (*ST-N* and *LA-N*) and virtual VT-d (*ST-O* and *LA-O*) in all benchmarks, regarding to both throughput and CPU utilization. There are much fewer DMA operations completed in the *strict* and *lazy* policy than that in the *passthrough* policy, due to the emulation cost of DMA remapping. As shown in Table 2, the reduction is between 2.46x (in Netperf RX) to 29.8x (in memcached) in all evaluated benchmarks. The tracking overhead in coIOMMU is negligible when comparing to the overhead of DMA remapping.

We also explore an interesting finding between *lazy* and *strict* in Figure 4, although not directly related to coIOMMU. It is a common learning that batching IOTLB invalidations generally brings better performance than strictly invalidating the IOTLB one-by-one. However, it is not always the case in virtualization – we observed 11% and 23% lower throughput when comparing *lazy* to *strict* in Netperf TX and Nginx. We find the batching interface of the virtual VT-d is the root cause. Its emulation requires walking the entire vIOPT to identify every valid mapping. If the walking cost exceeds the cycles of saved invalidations, the performance of *lazy* is instead worse than that of *strict*. We leave studying more efficient batching interface and policy for another research.

5.2. Memory Footprint

We sample the number of pinned pages every 3 seconds, from the beginning of the benchmarks to 6 seconds after its completion, in Figure 5. The extra 6 seconds are used to evaluate the elasticity of the six modes, against transitional system business. One note – the '*max*' mark in the Y-axis indicates the total number of guest pages, representing the case of static pinning. It is 8M (for 32GB memory) in most benchmarks and 1M (for 4GB memory) in OpenArena.

All six modes exhibit the same pattern in all benchmarks, except *PT-N*. First, *PT-O* is tied to static pinning, thereby always sitting in the top 'max' location. Second, all four modes with DMA remapping enabled (*ST-O*, *ST-N*, *LA-O*, and *LA-N*) pin the least number of pages, because they need strictly follow the desired protection semantics. As such, their lines completely overlap in each diagram in Figure 5. The line of *PT-N* (coIOMMU in the passthrough policy) fluctuates in the middle due to smart pinning, which heuristically pins guest pages for balancing performance and footprint. So, it is the focus of our following analysis.

Networking - All four networking benchmarks (left four in Figure 5) start and end with the same number of pinned pages (~8800 pages) in *PT-N*. Those always-pinned pages come from Intel i40e NIC driver, which pre-maps 512 pages per vCPU as the receive buffer pool when the NIC is enabled. The number sums up to 8192 pages with 16 vCPUs in our configuration.

The largest footprint is observed in Netperf stream TX, with up to 44530 pinned pages (174MB). It is ~4.4x of the pages that are actually mapped for DMA at that time. The additionally pinned 34158 pages reflect the DMA temporal locality, occupying only 0.4% of the total 32GB guest memory. coIOMMU recognizes such locality thus sustains the performance of static pinning when keeping a small memory footprint. Netperf stream RX pins fewer pages (up to ~18000) than TX, due to better DMA temporal locality – Intel i40e NIC driver prefers to use the pre-mapped 8192 pages for incoming packets. On the other hand, Nginx and Memcached are less throughput sensitive than Netperf TX/RX, yielding a transfer rate of 2.3Gbps and 1.34 Gbps respectively. Accordingly, there are fewer pages used for DMA in the two benchmarks, leading to smaller footprint in coIOMMU.

Storage - We configure fio to perform asynchronous direct random reads from the assigned SSD, to avoid page cache and readahead optimization in guest Linux. 16 fio threads are launched to read the disk with a 512-byte block size and 128 in-flight requests per I/O queue, summing up to 256 pages for potential DMAs. The guest storage driver pre-maps 302 pages at boot time. Therefore, up to 558 pages may be mapped for DMA simultaneously, at any time. Obviously, coIOMMU precisely captures such temporal locality and constantly pins 558 pages in our test.

GPU - There is no recognizable difference between the line of *PT-N* and the bottom four lines, in OpenArena. The reason is

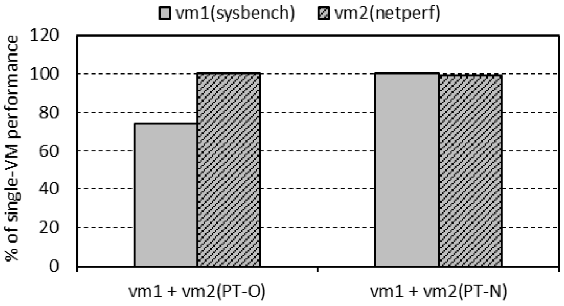


Figure 6: The impact of memory overcommitment: static pinning (*PT-O*) vs. fine-grained pinning (*PT-N*)

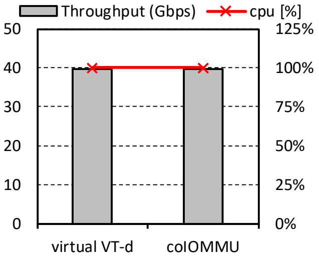
simple, as explained in §5.1, that OpenArena maps most of its DMA pages (~240MB) one-off at launch time and then unmaps them only at exit. In such circumstance, smart pinning and lazy unpinning have no effect at all. Therefore, all five fine-grained pinning modes pin the similar number of guest pages, with only static pinning staying in the top.

5.3. Memory Overcommitment

Overcommitment allows the aggregated size of all VMs to exceed the physical memory, thus improving memory utilization. We explore this configuration in both coIOMMU (*PT-N*) and the virtual VT-d (*PT-O*), to demonstrate the value of fine-grained pinning.

We create two VMs in the test machine with 64GB physical memory. VM1 has no assigned device and is configured with 32GB memory. It runs sysbench to randomly access a 16GB memory region. On the other hand, VM2 is assigned with an Intel i40e NIC and is configured with 48GB memory. It runs Netperf to send packets through the assigned NIC. The total memory size of the two VMs (80GB) exceeds the physical memory limitation.

We compare the performance of running them together to that of running each alone, in Figure 6. With the virtual VT-d, Netperf sustains the single-VM performance while sysbench suffers 25% performance drop. The drop is caused by frequent page swaps due to insufficient host memory. There is only 8.8GB left after statically pinning 48GB memory for VM2. The situation gets worse with random errors reported in VM1, when increasing the memory intensity of sysbench. Conversely, both VMs achieve their desired performance with coIOMMU, with 49GB free memory available even when two benchmarks are both running.



Spent Cycles	virtual VT-d	coIOMMU	ratio
Create VM	7554ms	407ms	18x
Assign NIC	183ms	2ms	91x
Deassign NIC	815ms	2ms	407x

Pinned Pages	virtual VT-d	coIOMMU	ratio
Before DPDK	8388608	548	15307x
In DPDK	186368	186368	1x
After DPDK	8838608	548	15307x

Figure 7: Running DPDK with virtual VT-d and coIOMMU

5.4. Guest User Space Driver

The guest kernel may directly assign a device to its user space for improved performance. However, kernel bypassing imposes the risk of DMA attacks from the user space. In such case, the guest kernel typically turns on DMA remapping of vIOMMU when the device is being assigned to the user space and then turns off remapping after the device is assigned backed to the kernel. In such circumstance, coIOMMU can help the hypervisor maintain fine-grained pinning reliably, while state-of-the-art vIOMMUs suffer from increased overhead by switching back and forth between static pinning and fine-grained pinning. We demonstrate such an example using DPDK pktgen, which offloads TCP packet processing from the guest user space to the assigned NIC. We run DPDK with coIOMMU and with the virtual VT-d respectively and show the comparison in Figure 7.

coIOMMU dramatically reduces the latency in several stages, compared with the virtual VT-d: (1) 18x reduction when the VM is created (407ms vs. 7554ms); (2) 91x reduction when the guest kernel assigns the NIC to user space DPDK (2ms vs. 183ms); and (3) 407x reduction when the NIC is assigned back to the guest kernel (2ms vs. 815ms). The cost of the emulated VT-d is mostly caused by pinning or unpinning the entire guest memory when switching to or away from static pinning. The VM creation phase suffers most because every guest page needs to be allocated and cleared in static pinning. In the meantime, coIOMMU pins no more than 186K pages, while the virtual VT-d pins many more pages varying between 186K and 8M.

5.5. DMA Temporal Locality

Good temporal locality on DMA buffers is crucial for high performance I/O processing, both in virtualization and on bare metal. Commercial OSes are optimized toward this goal, as observed in our evaluation and also reported by previous studies [7, 44, 51]. On the other hand, Markuze et al. [30] observes that many pages may be used to hold DMA buffers, over time, in stock Linux. Hence, we studied the DMA temporal locality of the networking stack in a similar configuration, by running 16 Netperf TX instances for 15 minutes, shown in Figure 8. We also run a Linux 'dd' command alongside Netperf, reading the raw virtual disk into /dev/zero. The 'dd' command constantly causes ~20K page cache misses per second, leading to ~20K new page allocated and heavily contending with the networking stack. The experiment is conducted in *PT-N* mode, i.e. under the passthrough policy.

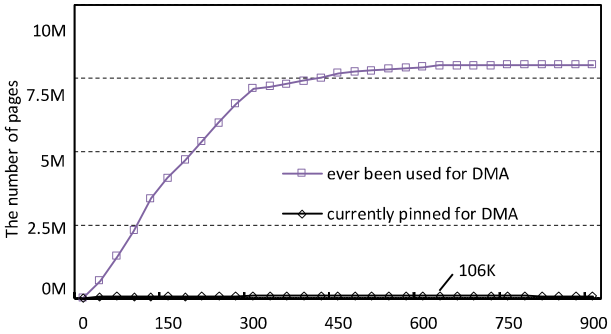


Figure 8: DMA temporal locality when running Netperf with 'dd'

Our data echoes the previous finding [30] – almost the entire guest memory (~7.9M pages, 98.7% of total memory) has ever been used for sending packets, over time. However, the number of pinned pages almost stays flat when coIOMMU is enabled. The peak number is ~106K (424MB), 2.4x of that when running Netperf TX alone and just 1.3% of the total guest memory. The result implies that the DMA locality in a short period is still good in such stress case, allowing the hypervisor to intelligently pin the guest pages with coIOMMU.

5.6. Mixed Workloads

We run Netperf TX and fio together to check how coIOMMU performs in mixed I/O workloads. The tested VM is configured with 16 vCPUs and 32GB memory as previous tests. It is directly assigned two devices: a XL710 NIC and a 760P SSD. We launch 16 netperf instances and 16 fio threads simultaneously in the VM, with each vCPU holding one netperf instance and one fio thread. Here we just compare *PT-O* vs. *PT-N* under the passthrough policy, as the two modes can best demonstrate the coIOMMU benefits according to the baseline data.

The result is promising. First, there is no observable performance difference when comparing Netperf and fio to their baseline performance of running alone. The deviations are less than 1% and within the error bar. Second, the peak number of pinned pages in mixed workloads is 45200 (176.5MB), close to the sum of pinned pages of running Netperf (174MB) and fio (2.2MB) alone. This result proves that coIOMMU can effectively reduce the memory footprint with negligible overhead, even when running mixed direct I/O usages together.

6. Conclusions and Future Work

Established vIOMMUs cannot reliably eliminate static pinning in direct I/O, due to the emulation cost of their DMA remapping interfaces. We instead propose coIOMMU, a new vIOMMU architecture for efficient memory management. coIOMMU introduces a cooperative DMA buffer tracking mechanism for fine-grained pinning, orthogonal to the costly DMA remapping interface. The new mechanism uses a shared DMA tracking table (DTT) for hypervisor and guest to exchange the DMA buffer information, without incurring excessive notifications from the guest, due to smart pinning and lazy unpinning. We demonstrate that coIOMMU not only dramatically improves the efficiency of memory management in wide direct I/O usages with negligible cost, but also sustains the desired security as required in specific protection policies. Last but not the least, although we implement coIOMMU by extending an emulated vIOMMU - the virtual Intel VT-d, this design can be easily ported to other vIOMMUs.

As for future work, we will focus on several areas. First, new IOMMU trends [53, 54] begin to support two-level address translations, allowing the guest to skip certain virtual IOTLB invalidations for improved performance. coIOMMU should provide efficient DMA buffer tracking in two-level translation and maintain its performance benefit. Second, some devices (e.g. GPUs) partially support DMA page faults, e.g. only for selective pages such as those used by applications. We want to study a hybrid approach for fine-grained pinning, by leveraging DMA page faults for faultable pages and using coIOMMU for other non-faultable pages. Last, kernel bypassing usually needs to pre-map a big trunk of memory for the application to manage. We want to extend the coIOMMU concept from the boundary between hypervisor and guest to the boundary between kernel space and user space, to enable finer-grained memory management in such usage.

References

[1] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian and Haibing Guan. High Performance Network Virtualization with SR-IOV. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2010. https://doi.org/10.1109/HPCA.2010.5416637.

[2] PCI-SIG. Address Translation Services Revision 1.1. http://www.pcisig.com/specifications/iov/ats/, 2009.

[3] Intel Corporation. Intel® Virtualization Technology for Directed I/O. Architecture specification, rev. 3.1. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Jun 2019.

[4] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, 2015. https://doi.org/10.1109/SP.2015.43.

[5] Khronos. The OpenCL Specification, rev 2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf, July 2015.

[6] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page Fault Support for Network Controllers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 449–466, 2016. https://doi.org/10.1145/3093337.3037710.

[7] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)*, pages 18:1–18:12, 2010. https://doi.org/10.1145/1815695.1815718.

[8] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011. https://www.usenix.org/legacy/events/atc11/tech/final_files/Amit.pdf.

[9] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. In *IEEE Transactions on Computers*, volume 68, issue 4, 2019. https://doi.org/10.1109/TC.2018.2879080.

[10] Avi Kivity, Yaniv Kamay, and Dor Laor. kvm: the Linux Virtual

Machine Monitor. In *Ottawa Linux Symposium (OLS)*, pages 225-230, 2007. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[11] Chris Schlaeger. AWS EC2 Virtualization: Introducing Nitro. In *AWS Summit*, 2018. http://aws-de-media.s3.amazonaws.com/images/AWS_Summit_2018/June7/Alexandria/Introducing-Nitro.pdf.

[12] Alibaba Corporation. Ali cloud elastic bare metal server – Shenlong architecture (X-Dragon) secret. http://www.programmersought.com/article/7752222651/.

[13] PCI-SIG. Single root I/O virtualization and sharing 1.0 specification. https://pcisig.com/specifications/iov/single_root/, Sep 2007.

[14] AMD Corporation. AMD IOMMU architecture specification, rev 3.00. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf, Dec 2016

[15] Christopher Clark, Keir Fraser, Seven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, volume 2, pages 273-286, 2005. https://www.usenix.org/legacy/event/nsdi05/tech/full_papers/clark/clark.pdf.

[16] ARM Corporation. ARM System Memory Management Unit Architecture Specification, rev 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062d.c/IHI0062D_c_system_mmu_architecture_specification.pdf, 2016

[17] Intel Corporation. Intel Scalable I/O Virtualization Technical Specification, rev 1.0. https://software.intel.com/en-us/download/intel-scalable-io-virtualization-technical-specification, Jun 2018

[18] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-9-20.pdf.

[19] Amazon Corporation. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2019

[20] Xin Xu, Bhavesh Davda. SRVM: Hypervisor Support for Live Migration with Passthrough SR-IOV Network Devices. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 65-77, 2016. https://dl.acm.org/citation.cfm?doid=2892242.2892256.

[21] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-Metal Performance for Virtual Machines with Exitless Interrupts. In *Communications of ACM*, volume 59, issue 1, pages 108-116, 2016. https://doi.org/10.1145/2845648.

[22] Alibaba Corporation. Elastic Compute Service Instance Type Families. https://www.alibabacloud.com/help/doc-detail/25378.htm, Jul 2019.

[23] Eric Auger. vIOMMU/ARM: full emulation and virtio-iommu approaches. In *KVM Forum*, 2017. http://events17.linuxfoundation.org/sites/events/files/slides/viommu_arm.pdf.

[24] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-Free IOMMU Protection for Networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 301–315, 2018. https://doi.org/10.1145/3173162.3173175.

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

[26] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live Migration with Pass-through Device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261-268, 2008. https://landley.net/kdocs/ols/2008/ols2008v2-pages-261-267.pdf.

[27] The HSA Foundation. http://www.hsafoundation.com/.

[28] Asim Kadav and Michael M. Swift. Live Migration of Direct-Access Devices. In *ACM SIGOPS Operating System Review (OSR),* volume 43, issue 3, pages 95-104, 2009. http://pages.cs.wisc.edu/~swift/papers/shadow-migrate-osr.pdf.

[29] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 423-436, 2010. https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Ben-Yehuda.pdf.

[30] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–262, 2016. https://doi.org/10.1145/2872362.2872379.

[31] Ardalan Amri Sani, Kevin Boos, Shaoqu Qin, and Lin Zhong. I/O Paravirtualizatoin at the Device File Boundary. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. http://doi.org/10.1145/2541940.2541943.

[32] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org.

[33] AF_XDP. https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html. Linux networking documentation.

[34] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2003. https://suif.stanford.edu/papers/vmi-ndss03.pdf.

[35] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, Zhijiao Zhang. CompSC: Live Migration with Pass-through Devices. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 109-120, 2012. https://doi.org/10.1145/3139645.3139649.

[36] Yuval Yarom and Katrina Falkner. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX conference on Security Symposium (SEC)*, pages 719–732, 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf.

[37] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, pages 1-15, 2015. https://doi.org/10.1145/2731186.2731189.

[38] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU Scalably. In *USENIX Annual Technical Conference (ATC)*, pages 549–562, 2011. https://www.usenix.org/system/files/conference/atc15/atc15-paper-peleg.pdf.

[39] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 6, pages 2-2, 2004. https://www.usenix.org/legacy/events/osdi04/tech/full_papers/levasseur/levasseur.pdf.

[40] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016. https://doi.org/10.1109/ISPASS.2016.7482091.

[41] Nikolay Sakharnykh. Everything You Need to Know About Unified Memory. In *NVIDIA's GPU Technology Conference (GTC)*, 2018. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

[42] Intel Corporation. Intel Open Source HD Graphics and Intel® Iris® Plus graphics Programmer's Reference Manual, page 139, 2017. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-kbl-vol05-memory_views.pdf.

[43] Vinod Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007.

[44] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *USENIX Annual Technical Conference (ATC)*, 2008. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann_html/.

[45] Alex Williamson. VFIO: A user's perspective. In *KVM Forum*, 2012. http://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf.

[46] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security Implications of Memory Deduplication in a Virtualized Environment. In

Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2013. https://www.eecis.udel.edu/~hnw/paper/memdedup.pdf.

[47] Moshe Malka, Nadav Amit, and Dan Tsafrir. Efficient Intra-Operating System Protection Against Harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 29-44, 2015. https://www.usenix.org/system/files/conference/fast15/fast15-paper-malka.pdf.

[48] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High Performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)*, Pages 3-3, 2006. https://www.usenix.org/legacy/event/usenix06/tech/full_papers/liu/liu_html/usenix06.html.

[49] Himanshu Raj and Karsten Schwan. High Performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC)*, pages 189-188, 2007. https://doi.org/10.1145/1272366.1272390.

[50] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10Gbps Using Safe and Transparent Network Interface Virtualization. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2009. https://www.cs.rice.edu/~rixner/publication/ram-09/.

[51] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. IOMMU: strategies for mitigating the IOTLB bottleneck. In *Proceedings of International Conference on Computer Architecture (ISCA)*, pages 256-274, 2010. https://doi.org/10.1007/978-3-642-24322-6_22.

[52] Hongwei Tang, Qiang Li, Shengzhong Feng, Xiaofang Zhao, and Yan Jin. IOMMU Para-Virtualization for Efficient and Secure DMA in Virtual Machines. *In KSII Transactions on Internet and Information Systems*, vol. 10, no. 12, pp. 5938-5963, 2016. DOI: 10.3837/tiis.2016.12.014.

[53] Eric Auger. SMMUv3 Nested Stage Setup. https://lkml.org/lkml/2019/3/17/124.

[54] Baolu Lu. Use 1st-level for IOVA translation. https://lwn.net/Articles/807079/

[55] Jun Nakajima. Enhancing KVM for Guest Protection and Security. In *KVM Forum*, 2019. https://static.sched.com/hosted_files/kvmforum2019/23/nakajima-enhancing-kvm-for-guest-protection.pdf/.

[56] AMD. Secure Encrypted Virtualization. https://developer.amd.com/sev/.

[57] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2002. https://doi.org/10.1145/844128.844146

[58] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, 2005. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf

[59] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. https://dl.acm.org/doi/pdf/10.1145/2775054.2694355/.

[60] Peter Xu. Device Assignment with Nested Guest and DPDK. In *KVM Forum*, 2017. https://www.linux-kvm.org/images/a/a6/KVM_Forum_2018_viommu_vfio.pdf.

[61] Baolu Lu. IOMMU: Bounce Page for Untrusted Devices. https://lwn.net/Articles/794595/.

[62] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Network and Distributed System Security (NDSS) Symposium*, 2019. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_05A-1_Markettos_paper.pdf.

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

[64] Nginx. https://www.nginx.com/.

[65] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004. https://memcached.org/.

[66] Fio. https://fio.readthedocs.io/en/latest/fio_doc.html.

[67] OpenArena. https://en.wikipedia.org/wiki/OpenArena.

[68] Sysbench. https://wiki.gentoo.org/wiki/Sysbench.

[69] Apachebench. http://en.wikipedia.org/wiki/ApacheBench.

[70] Brian Aker. Memslap – load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html.