The Price of Safety: Evaluating IOMMU Performance

Muli Ben-Yehuda
IBM Haifa Research Lab
muli@il.ibm.com

Jimi Xenidis
IBM Research
jimix@watson.ibm.com

Michal Ostrowski
IBM Research
mostrows@watson.ibm.com

Karl Rister
IBM LTC
krister@us.ibm.com

Alexis Bruemmer
IBM LTC
alexisb@us.ibm.com

Leendert Van Doorn
AMD
Leendert.vanDoorn@amd.com

Abstract

IOMMUs, IO Memory Management Units, are hardware devices that translate device DMA addresses to machine addresses. An isolation capable IOMMU restricts a device so that it can only access parts of memory it has been explicitly granted access to. Isolation capable IOMMUs perform a valuable system service by preventing rogue devices from performing errant or malicious DMAs, thereby substantially increasing the system's reliability and availability. Without an IOMMU a peripheral device could be programmed to overwrite any part of the system's memory. Operating systems utilize IOMMUs to isolate device drivers; hypervisors utilize IOMMUs to grant secure direct hardware access to virtual machines. With the imminent publication of the PCI-SIG's IO Virtualization standard, as well as Intel and AMD's introduction of isolation capable IOMMUs in all new servers, IOMMUs will become ubiquitous.

Although they provide valuable services, IOMMUs can impose a performance penalty due to the extra memory accesses required to perform DMA operations. The exact performance degradation depends on the IOMMU design, its caching architecture, the way it is programmed and the workload. This paper presents the performance characteristics of the Calgary and DART IOMMUs in Linux, both on bare metal and in a hypervisor environment. The throughput and CPU utilization of several IO workloads, with and without an IOMMU, are measured and the results are analyzed. The potential strategies for mitigating the IOMMU's costs are then discussed. In conclusion a set of optimizations and resulting performance improvements are presented.

1 Introduction

An I/O Memory Management Unit (IOMMU) creates one or more unique address spaces which can be used to control how a DMA operation, initiated by a device, accesses host memory. This functionality was originally introduced to increase the addressability of a device or bus, particularly when 64-bit host CPUs were being introduced while most devices were designed to operate in a 32-bit world. The uses of IOMMUs were later extended to restrict the host memory pages that a device can actually access, thus providing an increased level of isolation, protecting the system from user-level device drivers and eventually virtual machines. Unfortunately, this additional logic does impose a performance penalty.

The wide spread introduction of IOMMUs by Intel [1] and AMD [2] and the proliferation of virtual machines will make IOMMUs a part of nearly every computer system. There is no doubt with regards to the benefits IOMMUs bring... but how much do they cost? We seek to quantify, analyze, and eventually overcome the performance penalties inherent in the introduction of this new technology.

1.1 IOMMU design

A broad description of current and future IOMMU hardware and software designs from various companies can be found in the OLS '06 paper entitled *Utilizing IOMMUs for Virtualization in Linux and Xen* [3]. The design of a system with an IOMMU can be broadly broken down into the following areas:

- IOMMU hardware architecture and design.
- Hardware ↔ software interfaces.

- Pure software interfaces (e.g., between userspace and kernelspace or between kernelspace and hypervisor).

It should be noted that these areas can and do affect each other: the hardware/software interface can dictate some aspects of the pure software interfaces, and the hardware design dictates certain aspects of the hardware/software interfaces.

This paper focuses on two different implementations of the same IOMMU architecture that revolves around the basic concept of a Translation Control Entry (TCE). TCEs are described in detail in Section 1.1.2.

1.1.1 IOMMU hardware architecture and design

Just as a CPU-MMU requires a TLB with a very high hit-rate in order to not impose an undue burden on the system, so does an IOMMU require a translation cache to avoid excessive memory lookups. These translation caches are commonly referred to as IOTLBs.

The performance of the system is affected by several cache-related factors:

- The cache size and associativity [13].
- The cache replacement policy.
- The cache invalidation mechanism and the frequency and cost of invalidations.

The optimal cache replacement policy for an IOTLB is probably significantly different than for an MMU-TLB. MMU-TLBs rely on spatial and temporal locality to achieve a very high hit-rate. DMA addresses from devices, however, do not necessarily have temporal or spatial locality. Consider for example a NIC which DMAs received packets directly into application buffers: packets for many applications could arrive in any order and at any time, leading to DMAs to wildly disparate buffers. This is in sharp contrast with the way applications access their memory, where both spatial and temporal locality can be observed: memory accesses to nearby areas tend to occur closely together.

Cache invalidation can have an adverse effect on the performance of the system. For example, the *Calgary* IOMMU (which will be discussed later in detail) does not provide a software mechanism for invalidating a single cache entry—one must flush the entire cache to invalidate an entry. We present a related optimization in Section 4.

It should be mentioned that the PCI-SIG IOV (IO Virtualization) working group is working on an Address Translation Services (ATS) standard. ATS brings in another level of caching, by defining how I/O endpoints (i.e., adapters) inter-operate with the IOMMU to cache translations on the adapter and communicate invalidation requests from the IOMMU to the adapter. This adds another level of complexity to the system, which needs to be overcome in order to find the optimal caching strategy.

1.1.2 Hardware ↔ Software Interface

The main hardware/software interface in the TCE family of IOMMUs is the Translation Control Entry (TCE). TCEs are organized in TCE tables. TCE tables are analogous to page tables in an MMU, and TCEs are similar to page table entries (PTEs). Each TCE identifies a 4KB page of host memory and the access rights that the bus (or device) has to that page. The TCEs are arranged in a contiguous series of host memory pages that comprise the TCE table. The TCE table creates a single unique IO address space (DMA address space) for all the devices that share it.

The translation from a DMA address to a host memory address occurs by computing an index into the TCE table by simply extracting the page number from the DMA address. The index is used to compute a direct offset into the TCE table that results in a TCE that translates that IO page. The access control bits are then used to validate both the translation and the access rights to the host memory page. Finally, the translation is used by the bus to direct a DMA transaction to a specific location in host memory. This process is illustrated in Figure 1.

The TCE architecture can be customized in several ways, resulting in different implementations that are optimized for a specific machine. This paper examines the performance of two TCE implementations. The first one is the *Calgary* family of IOMMUs, which can be found in IBM's high-end *System x* (x86-64 based) servers, and the second one is the *DMA Address Relocation Table (DART)* IOMMU, which is often paired with PowerPC

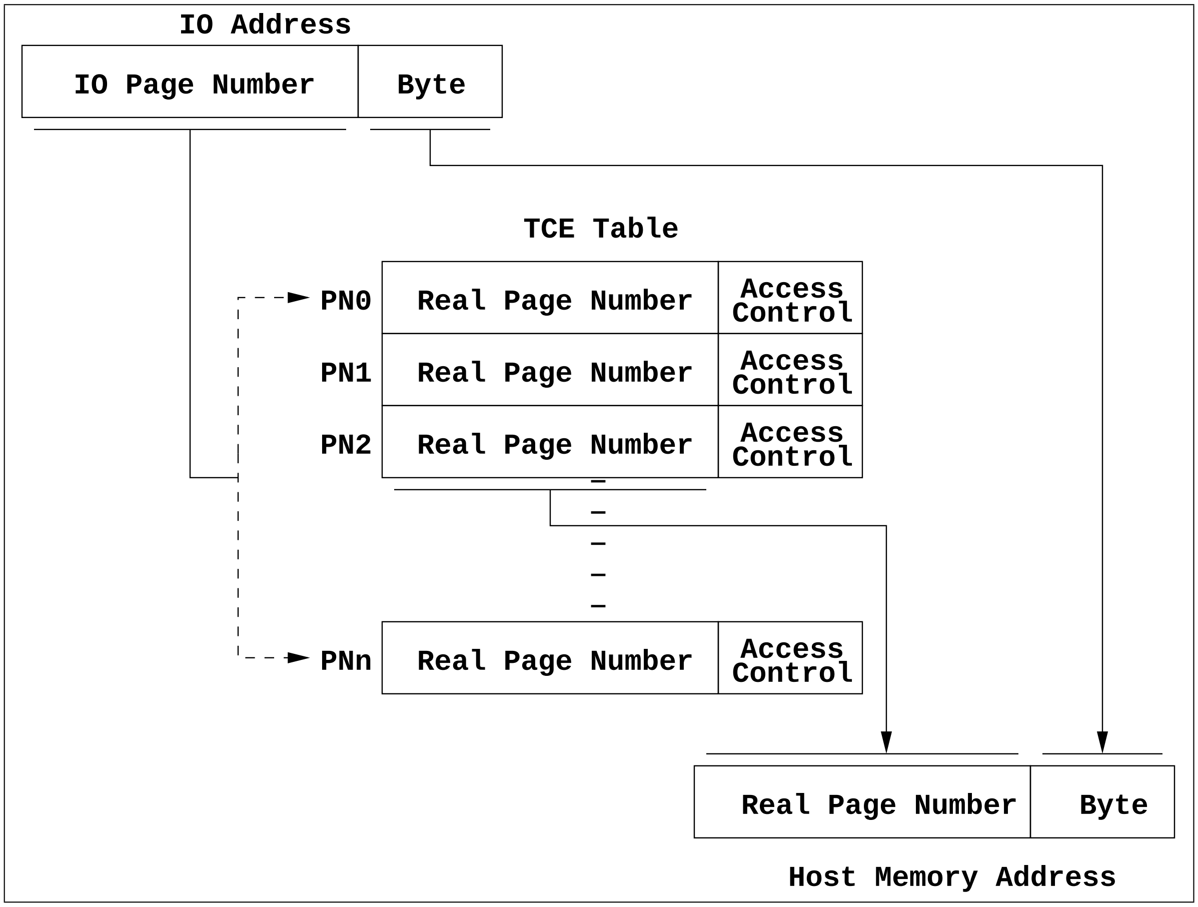

Figure 1: TCE table

970 processors that can be found in Apple G5 and IBM JS2x blades, as implemented by the CPC945 Bridge and Memory Controller.

The format of the TCEs are the first level of customization. Calgary is designed to be integrated with a Host Bridge Adapter or South Bridge that can be paired with several architectures—in particular ones with a huge addressable range. The Calgary TCE has the following format:

Bits	Field	Description
0:15	Unused	
16:51	RPN	Real Page number
52:55	Reserved	
56:61	Hub ID	Used when a single TCE table isolates several busses
62	W*	W=1 ⇒ Write allowed
63	R*	R=1 ⇒ Read allowed
*R=0 and W=0 represent an invalid translation		

Table 1: Calgary TCE format

The 36 bits of RPN represent a generous 48 bits (256 TB) of addressability in host memory. On the other hand, the DART, which is integrated with the North Bridge of the Power970 system, can take advantage of the systems maximum 24-bit RPN for 36-bits (64 GB) of addressability and reduce the TCE size to 4 bytes, as shown in Table 2.

This allows DART to reduce the size of the table by half for the same size of IO address space, leading to better (smaller) host memory consumption and better host cache utilization.

1.1.3 Pure Software Interfaces

The IOMMU is a shared hardware resource, which is used by drivers, which could be implemented in user-space, kernel-space, or hypervisor-mode. Hence the IOMMU needs to be owned, multiplexed and protected

Bits	Field	Description
0	Valid	1 - valid
1	R	R=0 ⇒ Read allowed
2	W	W=0 ⇒ Write allowed
3:7	Reserved	
8:31	RPN	Real Page Number

Table 2: DART TCE format

by system software—typically, an operating system or hypervisor.

In the bare-metal (no hypervisor) case, without any userspace driver, with Linux as the operating system, the relevant interface is Linux's DMA-API [4][5]. In-kernel drivers call into the DMA-API to establish and teardown IOMMU mappings, and the IOMMU's DMA-API implementation maps and unmaps pages in the IOMMU's tables. Further details on this API and the Calgary implementation thereof are provided in the OLS '06 paper entitled *Utilizing IOMMUs for Virtualization in Linux and Xen* [3].

The hypervisor case is implemented similarly, with a hypervisor-aware IOMMU layer which makes hypercalls to establish and tear down IOMMU mappings. As will be discussed in Section 4, these basic schemes can be optimized in several ways.

It should be noted that for the hypervisor case there is also a common alternative implementation tailored for guest operating systems which are not aware of the IOMMU's existence, where the IOMMU's mappings are managed solely by the hypervisor without any involvement of the guest operating system. This mode of operation and its disadvantages are discussed in Section 4.3.1.

2 Performance Results and Analysis

This section presents the performance of IOMMUs, with and without a hypervisor. The benchmarks were run primarily using the *Calgary* IOMMU, although some benchmarks were also run with the *DART* IOMMU. The benchmarks used were FFSB [6] for disk IO and netperf [7] for network IO. Each benchmark was run in two sets of runs, first with the IOMMU disabled and then with the IOMMU enabled. The benchmarks were run on bare-metal Linux (Calgary and DART) and Xen dom0 and domU (Calgary).

For network tests the netperf [7] benchmark was used, using the TCP_STREAM unidirectional bulk data transfer option. The tests were run on an IBM x460 system (with the Hurricane 2.1 chipset), using 4 x dual-core Paxville Processors (with hyperthreading disabled). The system had 16GB RAM, but was limited to 4GB using mem=4G for IO testing. The system was booted and the tests were run from a QLogic 2300 Fiber Card (PCI-X, volumes from a DS3400 hooked to a SAN). The on-board Broadcom Gigabit Ethernet adapter was used. The system ran SLES10 x86_64 Base, with modified kernels and Xen.

The netperf client system was an IBM e326 system, with 2 x 1.8 GHz Opteron CPUs and 6GB RAM. The NIC used was the on-board Broadcom Gigabit Ethernet adapter, and the system ran an unmodified RHEL4 U4 distribution. The two systems were connected through a Cisco 3750 Gigabit Switch stack.

A 2.6.21-rc6 based tree with additional Calgary patches (which are expected to be merged for 2.6.23) was used for bare-metal testing. For Xen testing, the xen-iommu and linux-iommu trees [8] were used. These are IOMMU development trees which track xen-unstable closely. xen-iommu contains the hypervisor bits and linux-iommu contains the xenolinux (both dom0 and domU) bits.

2.1 Results

For the sake of brevity, we present only the network results. The FFSB (disk IO) results were comparable. For Calgary, the system was tested in the following modes:

- netperf server running on a bare-metal kernel.
- netperf server running in Xen dom0, with dom0 driving the IOMMU. This setup measures the performance of the IOMMU for a "direct hardware access" domain—a domain which controls a device for its own use.
- netperf server running in Xen domU, with dom0 driving the IOMMU and domU using virtual-IO (netfront or blkfront). This setup measures the performance of the IOMMU for a "driver domain" scenario, where a "driver domain" (dom0) controls a device on behalf of another domain (domU).

The first test (netperf server running on a bare-metal kernel) was run for DART as well.

Each set of tests was run twice, once with the IOMMU enabled and once with the IOMMU disabled. For each test, the following parameters were measured or calculated: throughput with the IOMMU disabled and enabled (off and on, respectively), CPU utilization with the IOMMU disabled and enabled, and the relative difference in throughput and CPU utilization. Note that due to different setups the CPU utilization numbers are different between bare-metal and Xen. Each CPU utilization number is accompanied by the potential maximum.

For the bare-metal network tests, summarized in Figures 2 and 3, there is practically no difference between the CPU throughput with and without an IOMMU. With an IOMMU, however, the CPU utilization can be as much as 60% more (!), albeit it is usually closer to 30%. These results are for Calgary—for DART, the results are largely the same.

For Xen, tests were run with the netperf server in dom0 as well as in domU. In both cases, dom0 was driving the IOMMU (in the tests where the IOMMU was enabled). In the domU tests domU was using the virtual-IO drivers. The dom0 tests measure the performance of the IOMMU for a "direct hardware access" scenario and the domU tests measure the performance of the IOMMU for a "driver domain" scenario.

Network results for netperf server running in dom0 are summarized in Figures 4 and 5. For messages of sizes 1024 and up, the results strongly resemble the bare-metal case: no noticeable throughput difference except for very small packets and 40–60% more CPU utilization when IOMMU is enabled. For messages with sizes of less than 1024, the throughput is significantly less with the IOMMU enabled than it is with the IOMMU disabled.

For Xen domU, the tests show up to 15% difference in throughput for message sizes smaller than 512 and up to 40% more CPU utilization for larger messages. These results are summarized in Figures 6 and 7.

3 Analysis

The results presented above tell mostly the same story: throughput is the same, but CPU utilization rises when the IOMMU is enabled, leading to up to 60% more CPU utilization. The throughput difference with small network message sizes in the Xen network tests probably stems from the fact that the CPU isn't able to keep up with the network load when the IOMMU is enabled. In other words, dom0's CPU is close to the maximum even with the IOMMU disabled, and enabling the IOMMU pushes it over the edge.

On one hand, these results are discouraging: enabling the IOMMU to get safety and paying up to 60% more in CPU utilization isn't an encouraging prospect. On the other hand, the fact that the throughput is roughly the same when the IOMMU code doesn't overload the system strongly suggests that software is the culprit, rather than hardware. This is good, because software is easy to fix!

Profile results from these tests strongly suggest that mapping and unmapping an entry in the TCE table is the biggest performance hog, possibly due to lock contention on the IOMMU data structures lock. For the bare-metal case this operation does not cross address spaces, but it does require taking a spinlock, searching a bitmap, modifying it, performing several arithmetic operations, and returning to the user. For the hypervisor case, these operations require all of the above, *as well as switching to hypervisor mode*.

As we will see in the next section, most of the optimizations discussed are aimed at reducing both the number and costs of TCE map and unmap requests.

4 Optimizations

This section discusses a set of optimizations that have either already been implemented or are in the process of being implemented. "Deferred Cache Flush" and "Xen multicalls" were implemented during the IOMMU's bring-up phase and are included in the results presented above. The rest of the optimizations are being implemented and were not included in the benchmarks presented above.

4.1 Deferred Cache Flush

The Calgary IOMMU, as it is used in Intel-based servers, does not include software facilities to invalidate selected entries in the TCE cache (IOTLB). The only

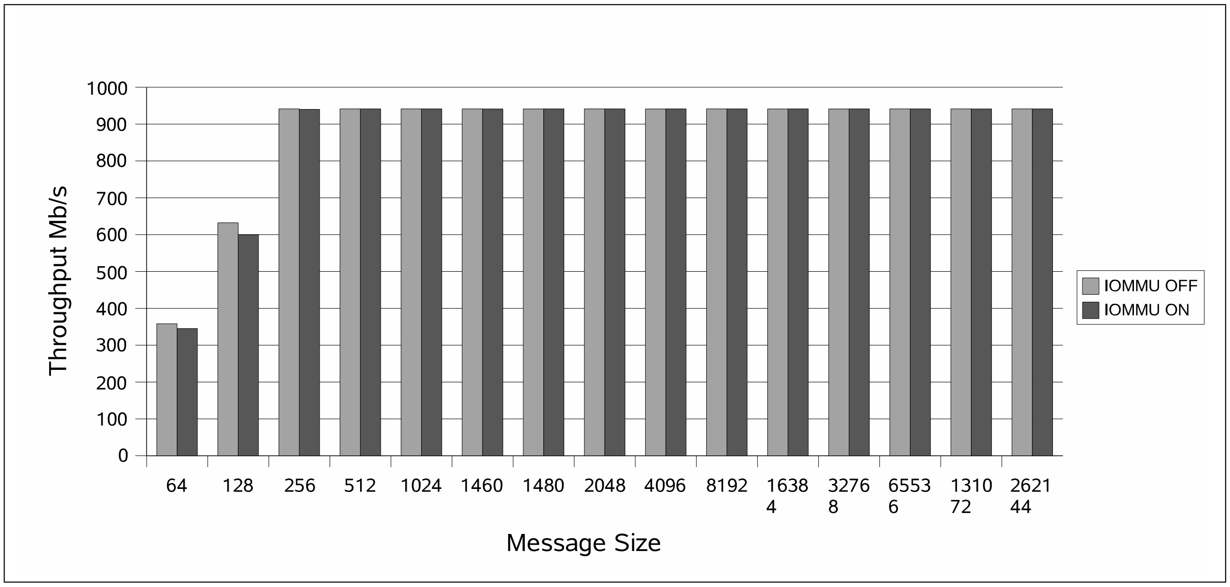

Figure 2: Bare-metal Network Throughput

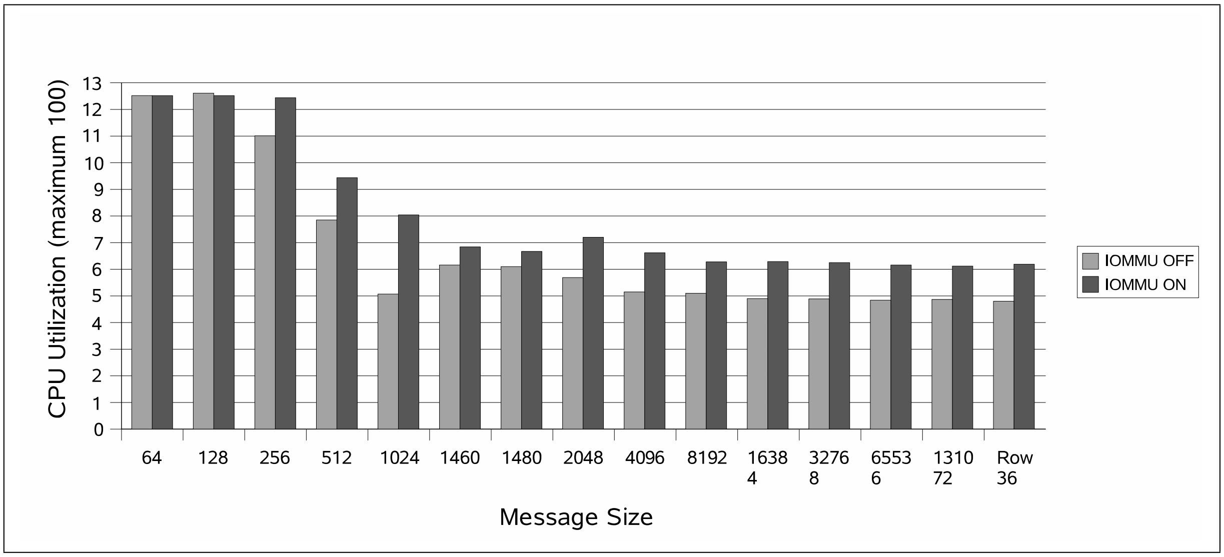

Figure 3: Bare-metal Network CPU Utilization

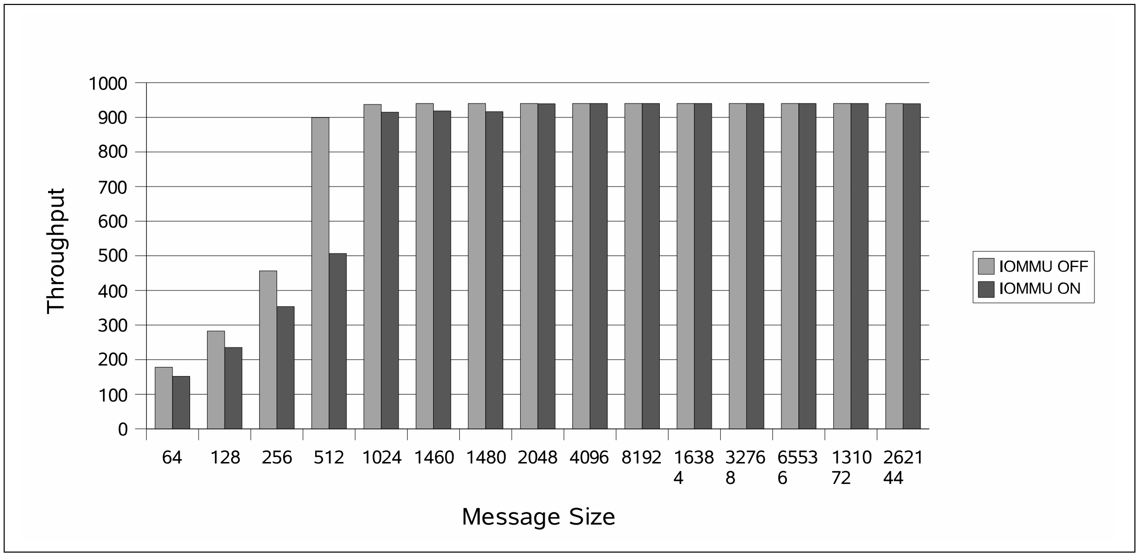

Figure 4: Xen dom0 Network Throughput

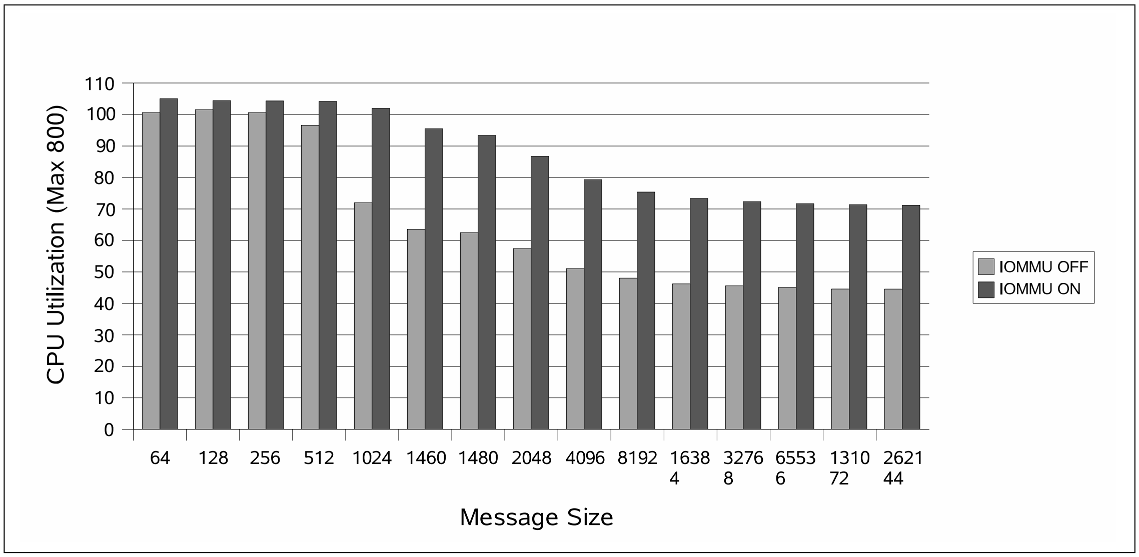

Figure 5: Xen dom0 Network CPU Utilization

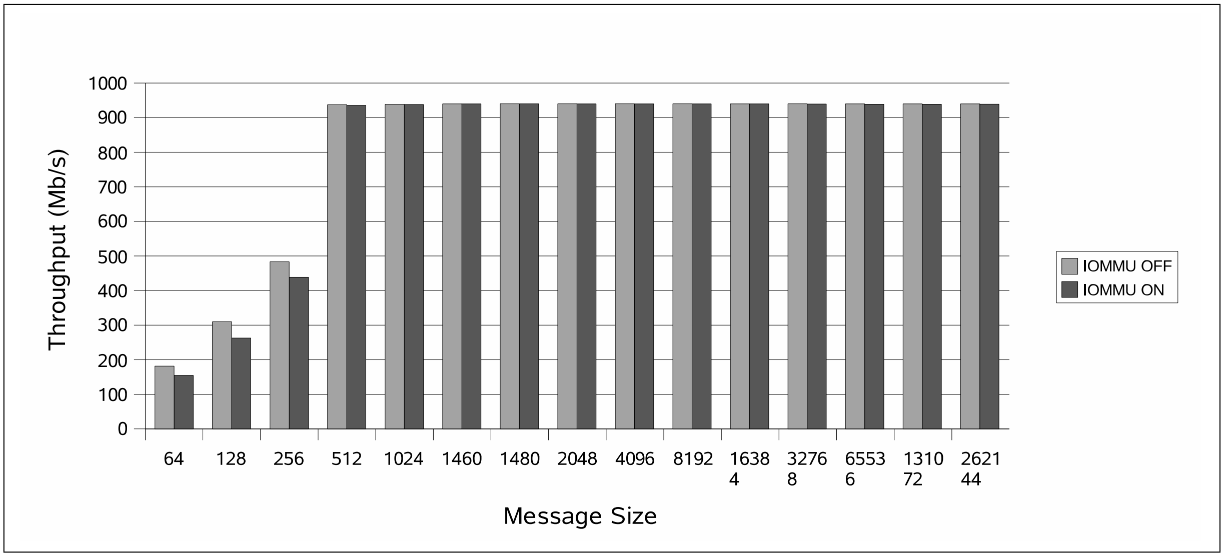

Figure 6: Xen domU Network Throughput

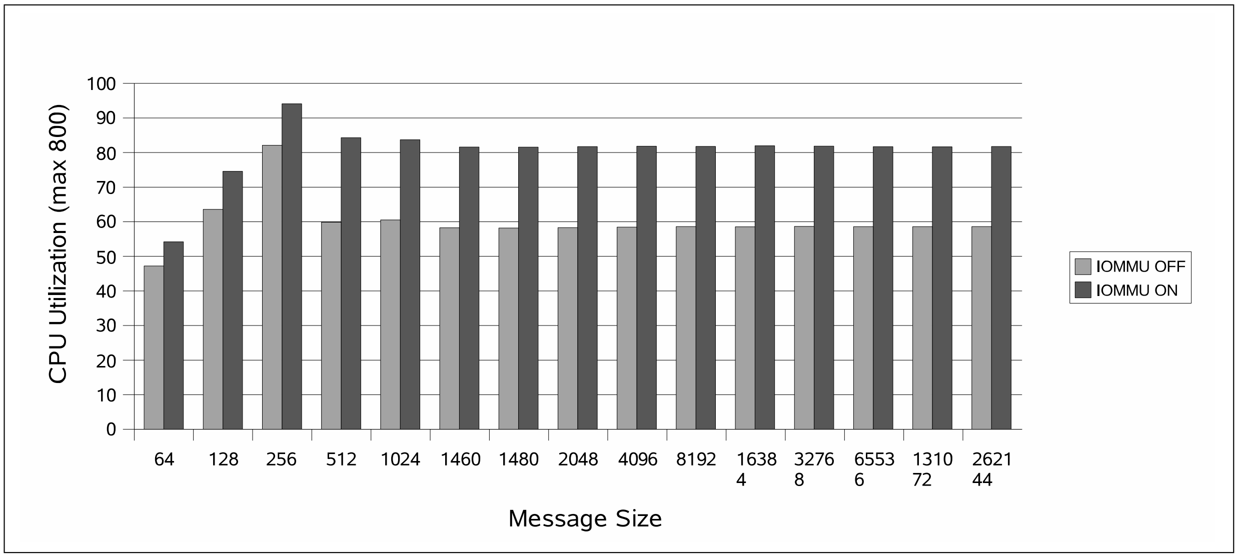

Figure 7: Xen domU Network CPU Utilization

way to invalidate an entry in the TCE cache is to quiesce all DMA activity in the system, wait until all outstanding DMAs are done, and then flush the entire TCE cache. This is a cumbersome and lengthy procedure.

In theory, for maximal safety, one would want to invalidate an entry as soon as that entry is unmapped by the driver. This will allow the system to catch any "use after free" errors. However, flushing the entire cache after every unmap operation proved prohibitive—it brought the system to its knees. Instead, the implementation trades a little bit of safety for a whole lot of usability. Entries in the TCE table are allocated using a next-fit allocator, and the cache is only flushed when the allocator rolls around (starts to allocate from the beginning). This optimization is based on the observation that an entry only needs to be invalidated before it is re-used. Since a given entry will only be reused once the allocator rolls around, roll-around is the point where the cache must be flushed.

The downside to this optimization is that it is possible for a driver to reuse an entry after it has unmapped it, *if* that entry happened to remain in the TCE cache. Unfortunately, closing this hole by invalidating every entry immediately when it is freed, cannot be done with the current generation of the hardware. The hole has never been observed to occur in practice.

This optimization is applicable to both bare-metal and hypervisor scenarios.

4.2 Xen multicalls

The Xen hypervisor supports "multicalls" [12]. A multicall is a single hypercall that includes the parameters of several distinct logical hypercalls. Using multicalls it is possible to reduce the number of hypercalls needed to perform a sequence of operations, thereby reducing the number of address space crossings, which are fairly expensive.

The Calgary Xen implementation uses multicalls to communicate map and unmap requests from a domain to the hypervisor. Unfortunately, profiling has shown that the vast majority of map and unmap requests (over 99%) are for a single entry, making multicalls pointless.

This optimization is only applicable to hypervisor scenarios.

4.3 Overhauling the DMA API

Profiling of the above mentioned benchmarks shows that the number one culprits for CPU utilization are the map and unmap calls. There are several ways to cut down on the overhead of map and unmap calls:

- Get rid of them completely.
- Allocate in advance; free when done.
- Allocate and free in large batches.
- Never free.

4.3.1 Using Pre-allocation to Get Rid of Map and Unmap

Calgary provides somewhat less than a 4GB DMA address space (exactly how much less depends on the system's configuration). If the guest's pseudo-physical address space fits within the DMA address space, one possible optimization is to only allocate TCEs when the guest starts up and free them when the guest shuts down. The TCEs are allocated such that TCE *i* maps the same machine frame as the guest's pseudo-physical address *i*. Then the guest could pretend that it doesn't have an IOMMU and pass the pseudo-physical address directly to the device. No cache flushes are necessary because no entry is ever invalidated.

This optimization, while appealing, has several downsides: first and foremost, it is only applicable to a hypervisor scenario. In a bare-metal scenario, getting rid of map and unmap isn't practical because it renders the IOMMU useless—if one maps all of physical memory, why use an IOMMU at all? Second, even in a hypervisor scenario, pre-allocation is only viable if the set of machine frames owned by the guest is "mostly constant" through the guest's lifetime. If the guest wishes to use page flipping or ballooning, or any other operation which modifies the guest's pseudo-physical to machine mapping, the IOMMU mapping needs to be updated as well so that the IO to machine mapping will again correspond exactly to the pseudo-physical to machine mapping. Another downside of this optimization is that it protects other guests and the hypervisor from the guest, but provides no protection inside the guest itself.

4.3.2 Allocate In Advance And Free When Done

This optimization is fairly simple: rather than using the "streaming" DMA API operations, use the alloc and free operations to allocate and free DMA buffers and then use them for as long as possible. Unfortunately this requires a massive change to the Linux kernel since driver writers have been taught since the days of yore that DMA mappings are a sparse resource and should only be allocated when absolutely needed. A better way to do this might be to add a caching layer inside the DMA API for platforms with many DMA mappings so that driver writers could still use the map and unmap API, but the actual mapping and unmapping will only take place the first time a frame is mapped. This optimization is applicable to both bare-metal and hypervisors.

4.3.3 Allocate And Free In Large Batches

This optimization is a twist on the previous one: rather than modifying drivers to use alloc and free rather than map and unmap, use map_multi and unmap_multi wherever possible to batch the map and unmap operations. Again, this optimization requires fairly large changes to the drivers and subsystems and is applicable to both bare-metal and hypervisor scenarios.

4.3.4 Never Free

One could sacrifice some of the protection afforded by the IOMMU for the sake of performance by simply never unmapping entries from the TCE table. This will reduce the cost of unmap operations (but not eliminate it completely—one would still need to know which entries are mapped and which have been theoretically "unmapped" and could be reused) and will have a particularly large effect on the performance of hypervisor scenarios. However, it will sacrifice a large portion of the IOMMU's advantage: any errant DMA to an address that corresponds with a previously mapped and unmapped entry will go through, causing memory corruption.

4.4 Grant Table Integration

This work has mostly been concerned with "direct hardware access" domains which have direct access to hardware devices. A subset of such domains are Xen "driver domains" [11], which use direct hardware access to perform IO on behalf of other domains. For such "driver domains," using Xen's grant table interface to pre-map TCE entries as part of the grant operation will save an address space crossing to map the TCE through the DMA API later. This optimization is only applicable to hypervisor (specifically, Xen) scenarios.

5 Future Work

Avenues for future exploration include support and performance evaluation for more IOMMUs such as Intel's VT-d [1] and AMD's IOMMU [2], completing the implementations of the various optimizations that have been presented in this paper and studying their effects on performance, coming up with other optimizations and ultimately gaining a better understanding of how to build "zero-cost" IOMMUs.

6 Conclusions

The performance of two IOMMUs, DART on PowerPC and Calgary on x86-64, was presented, through running IO-intensive benchmarks with and without an IOMMU on the IO path. In the common case throughput remained the same whether the IOMMU was enabled or disabled. CPU utilization, however, could be as much as 60% more in a hypervisor environment and 30% more in a bare-metal environment, when the IOMMU was enabled.

The main CPU utilization cost came from too-frequent map and unmap calls (used to create translation entries in the DMA address space). Several optimizations were presented to mitigate that cost, mostly by batching map and unmap calls in different levels or getting rid of them entirely where possible. Analyzing the feasibility of each optimization and the savings it produces is a work in progress.

Acknowledgments

The authors would like to thank Jose Renato Santos and Yoshio Turner for their illuminating comments and questions on an earlier draft of this manuscript.

References

[1] *Intel Virtualization Technology for Directed I/O Architecture Specification*, 2006, `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`.

[2] *AMD I/O Virtualization Technology (IOMMU)* Specification, 2006, `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`.

[3] *Utilizing IOMMUs for Virtualization in Linux and Xen*, by M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig, in Proceedings of the 2006 Ottawa Linux Symposium (OLS), 2006.

[4] *Documentation/DMA-API.txt*.

[5] *Documentation/DMA-mapping.txt*.

[6] *Flexible Filesystem Benchmark (FFSB)* `http://sourceforge.net/projects/ffsb/`

[7] *Netperf Benchmark* `http://www.netperf.org`

[8] *Xen IOMMU trees*, 2007, `http://xenbits.xensource.com/ext/xen-iommu.hg`, `http://xenbits.xensource.com/ext/linux-iommu.hg`

[9] *Xen and the Art of Virtualization*, by B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, in Proceedings of the 19th ASM Symposium on Operating Systems Principles (SOSP), 2003.

[10] *Xen 3.0 and the Art of Virtualization*, by I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, in Proceedings of the 2005 Ottawa Linux Symposium (OLS), 2005.

[11] *Safe Hardware Access with the Xen Virtual Machine Monitor*, by K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, in Proceedings of the OASIS ASPLOS 2004 workshop, 2004.

[12] *Virtualization in Xen 3.0*, by R. Rosen, `http://www.linuxjournal.com/article/8909`

[13] *Computer Architecture, Fourth Edition: A Quantitative Approach*, by J. Hennessy and D. Patterson, Morgan Kaufmann publishers, September, 2006.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*