Page Fault Support for Network Controllers

Ilya Lesokhin[†,◇] Haggai Eran[†,◇] Shachar Raindel[◇] Guy Shapiro[◇] Sagi Grimberg[◇]
Liran Liss[◇] Muli Ben-Yehuda[†] Nadav Amit[†,‡] Dan Tsafrir[†]

[†]Technion – Israel Institute of Technology [◇]Mellanox Technologies [‡]VMware Research

Abstract

Direct network I/O allows network controllers (NICs) to expose multiple instances of themselves, to be used by untrusted software without a trusted intermediary. Direct I/O thus frees researchers from legacy software, fueling studies that innovate in multitenant setups. Such studies, however, overwhelmingly ignore one serious problem: direct memory accesses (DMAs) of NICs disallow page faults, forcing systems to either pin entire address spaces to physical memory and thereby hinder memory utilization, or resort to APIs that pin/unpin memory buffers before/after they are DMAed, which complicates the programming model and hampers performance.

We solve this problem by designing and implementing page fault support for InfiniBand and Ethernet NICs. A main challenge we tackle—unique to NICs—is handling receive DMAs that trigger page faults, leaving the NIC without memory to store the incoming data. We demonstrate that our solution provides all the benefits associated with "regular" virtual memory, notably (1) a simpler programming model that rids users from the need to pin, and (2) the ability to employ all the canonical memory optimizations, such as memory overcommitment and demand-paging based on actual use. We show that, as a result, benchmark performance improves by up to 1.9x.

1. Introduction

Virtual memory provides three key benefits (Table 1). First, it protects applications and virtual machines (VMs) from one another by isolating their address spaces. Second, it simplifies the programming model by providing the illusion that address spaces are contiguous, big as needed, and always available, relieving programmers from having to explicitly decide which portions of their address spaces to place in primary or secondary storage at any given time. The third benefit of virtual memory is allowing the canonical memory optimizations (demand paging etc.), which improve the utilization of the memory and the performance of the system.

The benefits of virtual memory pertain to software that runs on the CPU. But CPUs are not the only processing elements that access the memory. I/O devices do it too via direct memory accesses (DMAs). Thus, programmers who write software that initiates DMAs do not enjoy all the benefits listed in Table 1, because DMAs are typically unable to tolerate page faults. This inability is mostly problematic in the context of network controllers (NICs), due to the proliferation of *direct network I/O*, whose semantics/terminology are defined in Table 2.

Direct network I/O multiplexes the NIC at the hardware level, allowing it to expose multiple instances of itself (denoted IOchannels). The trusted OS or hypervisor (denoted

virtual memory benefit		*page-fault support required*
address space isolation		no
simplified programming model		yes
canonical memory optimizations		yes
- overcommitment	*- swapping*	*- copy on write*
- demand paging	*- mmap-ed files*	*- page migration*
- delayed allocation	*- deduplication*	*- transparent hugepages*

Table 1. *Out of the benefits of virtual memory, only isolation can be provided without page fault support. Gray items list some canonical memory optimizations: overcommitment allows the aggregated size of all address spaces to exceed the physical memory; demand paging and delayed allocation lazily fault-in/allocate pages when they are actually used; swapping dynamically evicts unused pages; memory-mapped files allow applications to access files via load/store operations; deduplication unifies identical pages into one; copy-on-write breaks such unifications when pages cease to be identical; page-migration permits memory hot-(un)plugging and minimizes NUMA traffic; and transparent hugepages promote multiple virtually-adjacent pages into a single physical superpage.*

direct network I/O terminology	
IOchannel	hardware-provided virtual NIC instance
IOuser	untrusted process or VM assigned with IOchannel
IOprovider	trusted operating system (OS) or hypervisor

Table 2. *The IOprovider allocates IOchannels to IOusers but otherwise stays off their I/O paths, notably in multitenant setups.*

ASPLOS '17 April 8–12, 2017, Xi'an, China.

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037710

IOprovider) allocates IOchannels to the untrusted applications or VMs (denoted IOusers), allowing them to bypass the IOprovider and directly interact with the NIC. Direct network I/O is now widely available in commodity NICs via Remote DMA (RDMA), Single-Root I/O Virtualization (SRIOV), and the Data Plane Development Kit (DPDK) [56,87,104]. It frees developers from legacy one-size-fits-all network stacks and the overheads they cause due to their general-purpose nature and their rigid interfaces, which are defined and mediated by the IOprovider. The ability to bypass the IOprovider through direct IOchannels allows the untrusted IOusers to specialize/customize their system software for their specific needs and workloads, even in multitenant setups, like clouds, where IOusers share the physical hardware.

Specialized IOuser software can attain 4–20x the throughput of a well-tuned Linux process using the standard socket POSIX API [13, 90, 93]. Such improvements, along with the commoditization of direct network I/O, has triggered a surge of research that exploits direct IOchannels [6, 13, 16, 29, 40, 41, 43, 64, 82, 89, 90, 105, 106] (§2.1). Understandably, such research focuses on the improvements it delivers. But overwhelmingly, it disregards the cost: losing virtual memory benefits due to lack of DMA page fault support (Table 1). NICs cannot cope with their DMAs experiencing page faults. But an IOuser can initiate DMAs targeting *any* of its virtual addresses, which might not be associated with accessible physical memory, as virtual-to-physical mappings are set by the IOprovider.

Two ways are employed to avoid DMA page faults (§2.2). The first is statically pinning the entire IOuser address space to physical memory [30, 32, 62, 119]. This approach is typical for SRIOV IOchannels, handed to production VMs [52, 113, 114] or to research OSes like Arrakis and IX [13, 90]. It retains the simple programming model of virtual memory but loses its canonical optimizations. Notably, it foregoes memory overcommitment, which is problematic, as memory is often the bottleneck resource in enterprise settings [7, 19, 39, 44, 108, 116]. The second way to avoid DMA page faults is via dynamic pinning. IOusers dynamically ask their IOprovider to pin/unpin DMA target buffers through a special interface [25, 66, 74, 95, 115]. This approach is typical for RDMA setups [14, 37, 81, 83, 103]. It facilitates the canonical optimizations but complicates the programming model. It ruins the illusion of an always-available address space and forces *all* IOusers to continuously engage in cumbersome, explicit memory management activity, to ensure that DMA target buffers never fault. Moreover, frequent dynamic pinning hampers performance substantially [5, 73].

Our goal is to eliminate the unfortunate tradeoff associated with direct network I/O: poor memory utilization and no canonical memory optimizations vs. complicated IOuser code and costly pinning overheads. With commodity NICs bringing IOchannels to the mainstream, we contend that it makes sense for systems to learn how to support DMA page faults of NICs, denoted as *network page faults* (NPFs).

The PCI-SIG recently standardized basic support for DMA page faults [86] (§2.3). We find that this standard may suffice for I/O devices that process *local* data [65, 99] (like GPUs or hard drives) but not for NICs, which process *external* data. Namely, the standard disregards the most challenging aspect of NPFs: *receive NPFs* (rNPFs). When CPU page faults occur, the corresponding thread waits until they are resolved. In principle, devices that process local data can likewise wait. But NICs cannot: they have no available memory to store incoming data upon rNPFs, and more data may arrive subsequently at full line rate, and it too might fault. We focus on resolving this problem for InfiniBand and Ethernet NICs. We specify our solution requirements and explain why naive approaches like simply adding memory buffers to NICs do not constitute viable, satisfactory solutions (§3).

We find that reliable InfiniBand connections (RC) allow for a relatively straightforward solution. Using standard RC commands, we modify the NIC's firmware to instruct the sender to temporarily suspend transmission upon rNPFs, or, when this is impossible, to quickly retransmit the data lost due to the rNPFs (§4). In contrast to InfiniBand, Ethernet is lossy, requiring a more sophisticated solution. We design an interface that, upon rNPFs, allows the NIC to fall back on a small pinned "backup" ring buffer of the IOprovider. The NIC alerts the IOprovider about the rNPFs by raising an interrupt. In response, the IOprovider carefully merges this incoming data into the associated IOchannels, keeping the IOusers unaware. We implement a prototype approximation of such a NIC as well as the matching IOprovider software (§5).

We experimentally evaluate our NIC prototypes (§6). We find that NPFs rarely occur in steady states, and we demonstrate that our system achieves its goals: increasing memory utilization, reducing code complexity of IOusers, and improving their performance.

2. Motivation

2.1 The Rise of Direct Network I/O

A key role of the OS is to abstract and isolate resources. Abstraction delivers a more convenient environment for applications, hiding hardware details and offering higher level semantics. Isolation multiplexes the hardware and allows applications to coexist. The downside of OS-provided abstraction and isolation might be degraded performance [13, 15, 60, 80, 90, 93, 98]. Commodity 10–100 Gbps NICs [28] strikingly exemplify this downside: as noted in §1, customized IOchannels software can be 4–20x more performant than a well-tuned POSIX-compliant process [13, 90, 93]. This improvement is largely due to avoiding the cost of OS-provided abstraction and isolation, as the high network volume leaves the processor with only tens to hundreds of cycles to handle each packet.

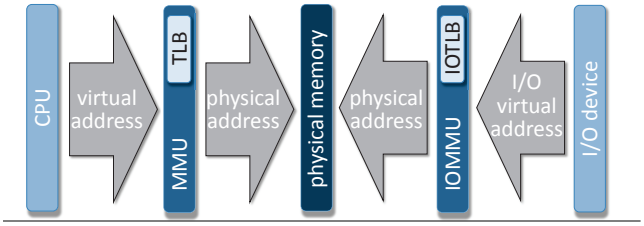

Figure 1. *The IOMMU is for I/O devices what the MMU is for CPUs. There is one important difference, however: page faults.*

Overheads can be avoided if hardware, not software, supports abstraction and isolation directly, as is the case with RDMA [104], which is available over InfiniBand, Ethernet, and TCP fabrics [53, 54, 91, 111]. RDMA provides network isolation and abstraction with reliable message passing and memory transfers semantics. It allows IOusers to access remote virtual memory regions while bypassing the IOproviders in both sides. A newer direct network I/O technology supported by most NIC vendors is SRIOV [55, 87]. An IOprovider can instruct an SRIOV-capable device to create multiple instances of itself, to be assigned to IOusers as PCIe devices for their exclusive use. SRIOV thus provides hardware isolation and imposes no OS abstractions. Another new/popular technology is DPDK [56], consisting of a set of efficient libraries for user-level packet processing on top of an IOchannel.

Dropping RDMA prices [61] and the wide availability of newer direct network I/O flavors that are applicable to Ethernet (SRIOV, DPDK) have brought direct network I/O to the mainstream, sparking a surge of OS research that utilizes IOchannels to improve application performance [6, 13, 16, 29, 40, 41, 43, 64, 67, 68, 82, 84, 89, 90, 96, 105, 106, 110].

2.2 The Problem

In the past, DMAs used physical addresses, which is incompatible with direct I/O, allowing IOusers to indirectly accesses any memory location via their IOchannels [12, 24, 42, 66, 101, 112]. Chip vendors thus introduced I/O memory management units (IOMMUs) [4, 10, 49, 57], which support I/O virtual addresses (IOVAs) for DMA. The role the IOMMU plays for I/O devices is similar to the role the regular MMU plays for processes, as illustrated in Figure 1. Both translate virtual to physical addresses and allow the OS to enforce isolation. Yet, there is one crucial difference: most devices cannot tolerate DMA page faults, which negates virtual memory features (Table 1). In *practice*, there are three zero-copy ways that systems employ to avoid network DMA page faults:[1] (1) static pinning, (2) fine-grained pinning, and (3) coarse-grained pinning, as explained next.

Static Pinning With static pinning, the IOprovider pins the entire address space of IOusers [30, 32, 62, 113, 119]. This approach is standard in production setups when handing SRIOV instances to virtual machines (e.g., under the Linux/KVM [113, 114] and VMware ESX [52] hypervisors) or when giving DPDK channels to applications [92]; it is also used by research OSes like Arrakis [90] and IX [13]. Static pinning is easier and simpler as compared to the alternatives. Notably, it is preferable from the perspective of IOusers, as they remain blissfully unaware of the problem, enjoying a setup where their address spaces are always present in memory. The downside, of course, is that the IOprovider loses its ability to apply the canonical memory optimizations to the pinned memory regions, which hampers memory utilization: pages must be present even if unused, for example. The inability to employ the canonical memory optimizations (Table 1) is highly problematic, e.g., because memory capacity is oftentimes *the* bottleneck resource in today's datacenters [7, 19, 39, 44, 116], and because memory overcommitment is vital in such computational environments [19, 108].

Fine-Grained Pinning With fine-grained pinning, IOusers dynamically pin and unpin each DMA target buffer, and correspondingly map and unmap it in the IOMMU, immediately before and right after the DMA operation. Perceived as the safest operation mode (in the face of potentially malicious or errant I/O devices), this approach is typically the default scheme used by general-purpose kernels as part of their internal DMA API [5, 8, 20, 25, 50, 75, 115]. It is also used to export IOMMU functionality to virtual machines [5, 115]. The advantage of fine-grained pinning is that only a small fraction of the IOuser address space is pinned [74], so the IOprovider can safely apply the canonical memory optimizations to the remaining, bigger, unpinned part. There are two disadvantages, however. First, fine-grained pinning complicates the IOuser programming model: IOusers can no longer assume an address space that is always physically available, which effectively forces them to actively partake in memory management activity with explicit (un)pinning operations. The second disadvantage is that fine-grained pinning might degrade performance significantly, notably due to IOMMU map/unmap overheads [17, 73, 74, 77, 88, 115].

Coarse-Grained Pinning With coarse-grained pinning, systems use a "pin-down cache" capable of enforcing an upper bound on the size of the pinned memory. When this bound is reached, existing pinned buffers are dynamically unpinned—evicted from the cache—to make room for newly pinned buffers. Given a big-enough size, pin-down caches are able to significantly reduce the overheads of dynamic pinning, but they complicate the code considerably. The approach is commonly used in high-performance RDMA setups, and the HPC community has developed a wealth of workload-based pinning and cache-eviction strategies that optimize performance [14, 23, 32, 37, 79, 81, 83, 103, 117]. Coarse-grained pinning can be perceived as a floating point between fine-grained and static pinning: as the upper bound for the pin-down cache gets smaller or bigger, coarse-grained pinning becomes more similar to fine-grained or static pinning, respectively.

[1] We discuss other possibilities in §3.

pinning strategy	*performant*	*memory utilization & canonical optimizations*	*programming simplicity*	*multitenant friendliness*
static	✓	✗	✓	✗
fine-grained	✗	✓	✓̸	✓
coarse-grained	✓̸	✓̸	✗	✓̸
none (NPFs)	✓	✓	✓	✓

Table 3. *Pros and cons of pinning strategies. NPFs provide the only scheme that involves no tradeoffs. The ✓̸ symbol associated with coarse-grained pinning indicates that bigger pin-down caches conflict with multitenant friendliness and the canonical memory optimizations, whereas smaller pin-down caches conflict with performance. The ✓̸ symbol associated with fine-grained pinning indicates that, while this scheme is more complex than NPFs and static pinning, it is still simpler than implementing a pin-down cache.*

Bottom Line Even though (1) the SRIOV standard is ten years old [85] and (2) most NIC vendors support it, we are not aware of any production hypervisor/OS that allows an SRIOV IOchannel to be assigned without statically pinning its *entire* IOusers's address space [52, 113, 114]. A research exploration that allowed SRIOV IOchannels without static pinning reported extremely poor performance due to fine-grained pinning; to improve performance, it dedicated a full extra core for pinning activity (per one 10Gbps port), it employed nontrivial optimizations, and it resorted to various undesirable security compromises [5]. With time, it is possible that SRIOV environments will mature, overcome some of the challenges, and develop pinning strategies that are more performant than fine-grained pinning and less restrictive than static pinning, similarly to pin-down caches in RDMA setups. But then the cost would be likewise similar—programming complexity and more and more memory that is pinned, even if unused.

Our goal is to eliminate all undesirable tradeoffs by providing NPF support, as summarized in Table 3. Our reasoning is simple: page faults make sense.

2.3 Existing DMA Page Faults Support

The PCI-SIG supports our reasoning. It recently acknowledged the significance of DMA page faults by supplementing the ATS (address translation services) PCIe standard with PRI (page request interface) [86]. These standardize device/IOMMU/OS cooperation that allows for basic DMA page fault support suitable for devices that process *local* data (elaborated further in §4). The problem of rNPFs caused by incoming *external* data is outside the scope of ATS/PRI.

The effort to utilize ATS/PRI is spearheaded by AMD's HSA—heterogeneous system architecture [46]. HSA is aimed at unifying the address spaces of CPUs and on-die GPUs, enabling seamless page fault resolution and thereby making pinning/copying between them unnecessary [47, 65]. The proclaimed HSA goals are aligned with ours: making SOCs (that combine CPU and GPU cores) "easier to program; easier to optimize; [and provide] higher performance" [94]. GPUs process local data only and are thus adequately served by ATS/PRI. We aspire to achieve HSA's goals for network programming, which involves external data and hence rNPFs.

2.4 Connection Between NPFs & IOMMU Protection

IOMMUs provides "strict" protection against malicious/errant devices if each DMA is preceded and followed by mapping and unmapping of its target buffer [5, 73, 74, 77, 88]. Such protection will soon be orthogonal and complementary to NPFs. Recent hardware supports 2D IOMMU translations, where host and guest have different I/O page tables [4, 58]. (Guest tables translate guest virtual to guest physical addresses, and host tables translate guest physical to host physical addresses [18].) The hardware concatenates the two for a full IOVA resolution. Thus, IOusers can utilize their tables for strict protection if they wish. Independently, the IOprovider needs NPFs for its own tables in order to apply the canonical memory optimizations. The same reasoning applies to emulated IOMMUs [5].

3. Requirements

Focusing on rNPFs (the more challenging aspect of NPFs), we would like our solution to have the properties outlined next; these properties help justify our design in §4 and §5.

No Additional Hardware Resources We prefer a solution that works with existing resources rather than requires additional hardware on NICs or switches. With rNPFs, it is tempting to consider adding memory to the NIC, for example, to buffer the incoming faulting data until the rNPF is resolved. But such an approach raises the question of how much added memory is enough. Assume that the NIC receives data in line-rate R, that rNPFs occur, and that the time it takes to resolve each rNPF is T. The added memory size should therefore be no less than $S = R \times T$, for each NIC port. Thus, if R is 100 Gbps and T is 10 milliseconds (major page fault), then S should be 125MB. Having to add a buffer of this size for each port would needlessly increase the NIC price. The extra cost makes no economical sense considering (1) rNPFs are infrequent, so this memory would rarely be used, and (2) the host memory can be used instead when necessary.

Stream Isolation The solution should not affect unrelated traffic. Optimally, network channels that do not encounter NPFs should not slow down. This requirement excludes using link-level flow control [2] to block all incoming traffic until the rNPF is resolved. That is, in principle, flow control allows NICs to ask their switch to temporarily buffer incoming traffic

(until the rNPF is resolved), thus preventing data loss. But this approach will suspend all other channels, and it might also lead to congestion spreading—switch buffers might run out, forcing it to ask its neighbors to also stop sending, and then the neighbors of the neighbors, and so on [97, 100].

No IOuser Pinning When implementing or running applications that use direct network I/O, it could be the case that the buffers to pin are easily identifiable and that their combined size in memory is small. For example, a virtual machine that employs a POSIX network stack typically copies packets to user-space from a relatively small set of kernel DMA target buffers. In such cases, it could be claimed that pinning this set should not be an issue, so NPF support is unneeded. This claim is *wrong* for several reasons, as specified next.

The above implied assumption—that the set of memory regions that ever served as DMA target buffers is small and static in POSIX setups—is simply incorrect. Markuze et al. show that the accumulated size of all such buffers quickly becomes gigabytes after only a few minutes of running multiple applications that stream data through TCP connections [76]. One contributing factor to this result is the fact that the OS relies on application contexts to copy incoming data off TCP buffers, so if applications are delayed due to scheduling considerations, the OS keeps the buffers for them, pinning other buffers for DMA as a result.

Even seemingly simple pinning scenarios might become challenging in multitenant setups, where tenants are not allowed to pin as they please. For example, in POSIX, applications pin memory using the mlock system call [70], which is limited to pinning not more than RLIMIT_MEMLOCK bytes [69], which in Linux is only 64KB by default [71].

Regardless, we would like our NPF solution to apply not just to the POSIX network stack but also to other types of workloads, notably to IOusers that utilize RDMA or zero-copy network stacks, which may provide substantial performance improvements [13, 90] while simultaneously mapping many more buffers as DMA targets.

Lastly, we would like to provide IOuser programmers with a simple, easy to program environment. Instead of burdening each and every one of the them with some pinning activity, we would like such programmers to be completely unaware of virtual memory management issues, as is the case with regular applications.

No IOusers TCP Changes Ideally, rNPF recovery time should be close to the time it takes the IOprovider to map the faulting memory: network disruption should be minimized. Under this constraint, upon experiencing an rNPF, we may suspend transmission or even employ retransmission to recover lost data, but only if the protocol allows the entity that experiences the NPF to initiate fast-enough suspension and retransmission. For example, when developing rNPF support for InfiniBand (§4), the NIC itself uses an RC protocol message to stop the sender.

When implementing rNPF support for Ethernet (§5), it could be (wrongfully) claimed that analogous solutions are possible in the TCP stack of IOusers. Fast retransmit [3], for example, allows TCP to quickly recover from loss of a single packet by sending a small number of duplicate acks. Fast retransmit could in principle be used to signal the sender that an NPF has been handled, so it can resume transmission now rather than wait for the long timeout to expire, thereby improving performance. Another idea would be to utilize explicit congestion notification (ECN) [34], which provides means for the receiver to notify the sender to slow down in order to reduce congestion. ECN could be used upon an rNPF to limit the send rate until the rNPF is resolved.

This approach does not work. The IOuser TCP/IP stack consists of software that runs on the CPU. However, it is the NIC (the I/O device hardware) that experiences the rNPF and that must decide what to do with the faulting packet P. Importantly, there is no generally applicable way for the NIC to forward P (or its header) to the appropriate IOuser so as to ask it to make the decision instead. All that current NICs can do is raise an interrupt to notify the IOuser that *some* packet was lost due to an NPF. But this information is not enough. The IOuser is unable to associate the NPF with a specific TCP stream, making the ideas mentioned above irrelevant.

Conceivably, one could propose that the NIC will initiate actions like sending fast retransmission acks or ECN-marked packets. But such NIC involvement would require the hardware to be aware of and to interfere with the TCP state of all IOusers, which is of course unacceptable as a general solution. Moreover, such an approach would only be able to work after the TCP connection is established; it will fail before. Lastly, as noted earlier, we would like our solution to be general and applicable to protocols different than TCP.

No IOusers NPF Handling As noted in §1, our Ethernet solution relies on NIC-IOprovider cooperation, which revolves around a small, pinned "backup" ring buffer that the IOprovider maintains and the NIC uses in order to store faulting packets. A question that may follow is why not use multiple per-IOuser backup rings, such that the NIC will cooperate directly with each IOuser. The are several reasons. Such an approach would needlessly expose each IOuser to the complexities involved in handling NPFs, which goes against our motivation as stated above. The approach would require a pinned ring for each IOuser, which is wasteful. It would additionally require some predetermined IOprovider-dependent interface and policy, which hinders portability. The approach would also be riskier, as interrupt handlers of IOusers might not get scheduled immediately when NPFs fire, so their backup rings might overflow (unlike the IOprovider, IOusers do not controls the physical CPUs). Lastly, the approach would require IOusers to be able to somehow resolve their own physical memory page faults—to our knowledge, no hypervisor of OS provides such a service to their virtual machines or processes.

Completeness There are optimizations that can potentially help to alleviate the rNPFs problem. A notable example is pre-faulting, which, upon encountering an rNPF, pre-faults all subsequent receive buffers that will likely be referenced soon but are currently not preset in memory. Pre-faulting does not provide a complete solution, because it is not applicable to, e.g., RDMA programs that randomly access remote memory, nor to applications sensitive to packet-loss, such as those that use UDP and rely on link-layer flow control. TCP applications can likewise experience NPFs and suffer from slow-start reduced performance due to the "cold ring" problem that is caused by dropped packets (§5), regardless of pre-faulting. Thus, while we make use of such optimizations, we require that our solution will be complete in the sense that it will entirely eliminate packet loss (Ethernet) or alternatively allow for quick recovery of the data (InfiniBand).

4. InfiniBand Page Fault Support

I/O devices trigger interrupts, which device drivers handle. NPF is just another type of of such an interrupt. That is, NPF handling requires understanding the semantics of the NIC. Next, we describe our NPF implementation for a Mellanox Connect-IB 56 Gb/s InfiniBand NIC and Linux. The principles we outline, however, are general.

Basic NPF Support The minimum needed for DMA page faults to work—no rNPFs just yet—is the following. The I/O device (or IOMMU) should be able to tell the IOprovider it encountered a page fault, and the IOprovider should be able respond that the fault is resolved. As the I/O device (or IOMMU) may cache translations, the IOprovider must invalidate these translations when the corresponding mappings change. This is the protocol standardized by ATS/PRI [86].

We modify the driver and firmware of our Connect-IB to provide this functionality. We do not implement ATS/PRI per se, because we do not have a CPU with IOMMU that supports it (ATS/PRI is relatively new), and because we can do better (see below). Having to do without a CPU whose IOMMU supports DMA page faults, we utilize the functionally-equivalent IOMMU of the Connect-IB. In its baseline implementation, all PTEs of the IOMMU page tables must be valid. We allow them to be invalid when supporting NPFs.

Flows Figure 2 (left) illustrates the NPF flow. (1) Processing a new request, the NIC consults the IOMMU page tables and finds and marks that one of the pages involved is not present. (2) Our modified firmware detects this fault and raises an NPF interrupt. (3) The driver's NPF interrupt handler queries the OS regarding the physical address of the faulting IOVAs; if necessary, the OS allocates the pages, possibly retrieving their content from secondary storage. (4) The driver updates the IOMMU page table with these physical addresses and informs the firmware that the NPF has been resolved.

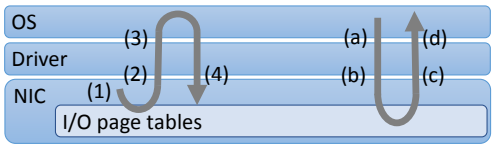

Figure 2. *NPF (1–4) and invalidation (a–d) flows.*

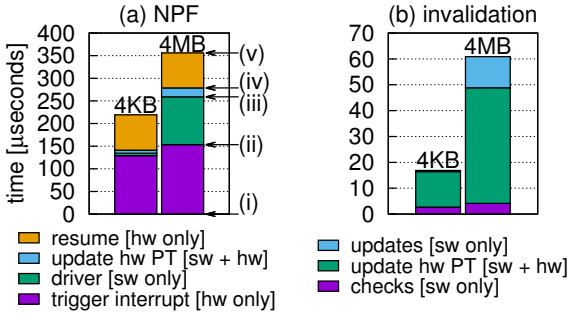

Figure 3. *Execution breakdown of NPF and invalidation.*

message size	*50%*	*95%*	*99%*	*max*
4KB	215μsec	250μsec	261μsec	464μsec
4MB	352μsec	431μsec	440μsec	687μsec

Table 4. *Tail latency of NPFs.*

As DMA buffers are no longer pinned, the OS may unmap and reuse them at will, necessitating an invalidation flow where it notifies the device that their IOVAs are no longer valid. This flow is depicted in Figure 2 (right). (a) The OS asks the driver (via a Linux MMU notifier [9]) to remove the old IOVA and stop the device from using it. (b) The driver updates the IOMMU page tables accordingly and issues the invalidation. (c) The NIC acknowledges, and then (d) the driver notifies the OS that the relevant pages can safely be reused.

Overhead Figure 3(a) shows the average overhead breakdown of minor NPFs (no disk access) when sending 4KB and 4MB messages, consisting of 1 and 1024 pages. Table 4 shows the corresponding tail latencies. The breakdown components depict the time between the following events: (i) the IOMMU observes that an NPF occurred and triggers an interrupt; (ii) the driver's NPF handler is invoked; (iii) the driver receives from the OS the physical addresses of to the IOVAs that should be mapped; (iv) the driver finishes updating the IOMMU page table accordingly (as the IOMMU is on the NIC, the driver must communicate with it when updating its DRAM-residing page tables due to coherency issues); and (v) the NIC identifies the update and resumes transmission.

A minor NPF takes 220μsec for a 4KB message, 90% of which is due to hardware (firmware).[2] The duration increases

[2] This duration is typical for Mellanox NIC firmware activity, not just NPFs, as the goal of the NIC circuitry that runs the firmware is usually to handle error paths, which is why it is allowed to operate relatively slowly.

to 350μsec for 4MB message due to software, as the OS must translate and possibly allocate many more pages. Still, the overhead is dominated by our hardware (firmware); pushing this functionality to silicon would thus make it much faster. We nevertheless find that the overheads are small enough, and that NPFs are rare enough, to allow us to enjoy good performance (§6).

Figure 3(b) shows the invalidation flow breakdown. The driver identifies the (InfiniBand) memory region associated with the invalidation and checks if the page was mapped in the IOMMU (mapping is done lazily via NPFs so the page might not be mapped). If not, no additional overhead is incurred. Otherwise, the driver updates the IOMMU page tables and its own internal state. Invalidations are cheaper than NPFs and are dominated by hardware/software interaction.

rNPFs InfiniBand supports the reliable connection (RC) protocol, which provides reliable multi-packet message delivery through fast acknowledgments and a packet sequencing. We utilize RC to cope with rNPFs. When a sender encounters an NPF, it can simply stop sending and wait until the NPF is resolved, as the faulting data is local. Receiving is trickier, because incoming data is external. With RC, we can nonetheless approximate the local data approach in most cases. RC has an end-to-end mechanism for receivers to quickly stop senders: the receiver sends a receiver-not-ready (RNR) negative acknowledgment packet (NACK), informing the sender to pause transmission for a specified time t. We modify the firmware to leverage RNR NACK for suspending senders upon rNPFs. In data center and HPC setups where InfiniBand is common, RNR NACKs are faster than the basic NPF overhead (Figure 3), so they do not affect performance much.

Some data is still dropped—until the RNR NACK arrives. But retransmission is possible as RC is reliable, so the sender must keep the data until it is acknowledged. We thus do not require more buffers at the receiver or in switches. Importantly, packet loss is decoupled from congestion control in InfiniBand, so retransmission does not reduce the IOchannel's speed.

In addition to send/receive, RC supports RDMA operations; most are handled identically when experiencing NPFs. But in some cases RC does not permit RNR NACKs for RDMA. When an initiator of a remote read request encounters a page fault, RC provides no way for it to ask the responder to stop. The only way to get the sender to retransmit is by asking it to rewind, after the rNPF is resolved. Until then, we must drop all incoming packets. There is no inherent reason for this limitation. We thus recommend to extend the end-to-end flow control RC standard to support remote read operations too.

Note that we are able to immediately send protocol-level control and retransmission messages in response to page faults because NPFs and the transport protocol are both implemented by the same hardware unit. Thus, interfacing them is easy. (As opposed to, say, TCP over Ethernet.)

Optimizations We explored several NPF optimizations, but due to space we describe the most notable three. The first optimization relates to concurrency, which RC supports. For example, a NIC can simultaneously be an RDMA initiator and responder on the same connection. We can thus choose to service multiple NPFs concurrently. It is natural to permit concurrent NPFs in the NIC's initiator and responder paths. But how many per path? (Each may be associated with multiple pending requests.) As each page fault requires more resources from the IOprovider, and for simplicity, our prototype limits the outstanding page faults per IOchannel to four: read and write, for both initiator and responder. (Send corresponds to initiator write, and receive corresponds to responder write.)

We additionally optimize the NPF critical path by temporarily bypassing the firmware when possible. For example, our modified firmware keeps a bitmap of reported in-flight page faults for each connection; if it encounters a new NPF but the bit is set, the firmware handles the NPF (retransmission, RNR NACK, etc.) but does not report it. After resolving the NPF, the driver first informs the hardware it can resume transmission (faster); the firmware still needs to be notified too (slower) to clear the relevant bit and re-enable NPFs of that type.

A third effective optimization was avoiding ATS/PRI restrictions, which dictate one page per PRI request (page fault event). By Figure 3, if we constrained ourselves to doing that, and avoided batching, minor page fault overhead induced by sending a cold 4MB message would have been prohibitive (more than 220 milliseconds). Our IOprovider driver therefore exploits its understanding of the NIC. Upon firing an interrupt, the NIC hands to the driver as much information as possible about the page fault, including the identity of the corresponding work queue (descriptor ring) and the position in that queue. The driver can then parse the relevant work request, which may include multiple scatter-gather memory ranges that are pre-faulted. It then batches the IOMMU page table updates.

Applicability While we focus on InfiniBand, the same RC protocol is also implemented on top of Ethernet as part of the RDMA over Converged Ethernet (RoCE) standard [54]. Our solution applies to RoCE as well. In addition to RC, InfiniBand also supports the unreliable datagram (UD) protocol, which does not guarantee delivery or ordering of the datagrams. The NPF solution described next applies also to UD.

5. Ethernet Page Fault Support

Let us focus on the more mainstream scenario whereby the IOuser utilizes a direct network channel through a regular Ethernet NIC—in our case, a Mellanox ConnectX-3 40 Gbps. The IOuser probably (not necessarily) uses the TCP/IP protocol to drive its direct channel. As in §4, we modify our NIC's

firmware to provide basic NPF support (Figure 2), leaving us with one missing component: how to handle *receive* NPFs.

In the previous section, we coped with rNPFs by relying on the fact that InfiniBand RC handles the transport protocol and the NPF at the same hardware unit, and that it provides a reliable, lossless communication channel C between two endpoints that allows for fast suspension and retransmission of the traffic the flows through C. None of these benefits are available for regular Ethernet NICs. Still, initially, we hoped that—at least with TCP—simply dropping incoming packets that experience rNPFs would constitute a reasonable solution. The reason: like the RC protocol, TCP provides reliable communication between two software entities, so when a packet is lost (due to an rNPF), the other side is guaranteed to retransmit. As it turns out, we were mistaken. Dropping is not a viable solution for direct Ethernet I/O. A better solution is required.

Running Example To demonstrate the problem (and highlight the benefits of NPFs in §6), throughout the remainder of the paper, we use a running example of an IOuser that is coupled with a direct Ethernet channel. Our IOuser is the memcached server [33], which is distributed memory key-value caching system, commonly used in websites to cache database queries and computation results in order to improve response time. Our memcached runs inside a lightweight VM (cgroups Linux container [78]), and it is driven by the memaslap benchmark [1] (90% get, 10% set, 1KB values by default). It utilizes a modified version of lwIP (user-level TCP stack [31]) and kernel bypass direct network technology based on the Linux verbs API to expose Ethernet hardware rings to user-space.

Cold Ring Problem A striking example that demonstrates why dropping packets upon rNPFs is problematic occurs when IOusers start. Recall that no buffers are pinned, and so, on startup, the IOuser's receive ring is "cold"—its buffers are unmapped and therefore rNPFs are triggered one after the other as buffers are demand-paged and mapped for the first time. Meanwhile, packets get dropped. We find that, consequently, TCP retransmission and congestion avoidance nearly deadlock the communication or, worse, completely halt it. The cold ring problem is not limited to startup situations. It can also happen, for example, when the VM is resumed from suspension or brought back from swap, or due to NUMA migration, forking with COW semantics, and so on. Moreover, as the use of direct network I/O gets more widespread, programmers may utilize their direct channels to implement zero-copy stacks [13, 90], in which case occasional cold sequences are likely.

Figure 4(a) demonstrates the problem. It compares the throughput of the baseline memcached whose I/O buffers are pre-pinned, to the "drop" configuration that discards incoming packets that trigger rNPFs. Throughput is presented as a function of time. We can see that pinning reaches the steady state nearly immediately, whereas dropping suffers

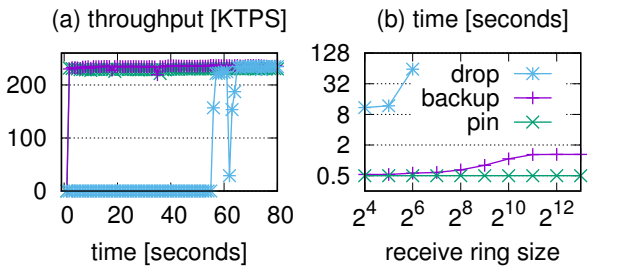

Figure 4. *(a) Startup with 64 entries in receive ring. (b) Time it takes to perform 10,000 operations vs. receive ring size.*

from the cold ring problem, penalizing the IOuser for nearly 60 seconds during which its throughput is effectively zero. In this experiment, the IOuser receive ring consists of 64 entries. Figure 4(b) demonstrates what happens if we vary the ring's size. We configure memaslap to perform 10K operations and display how long they takes along the y-axis. Even with an unreasonably small ring size of 16 entries, the drop configuration takes over 10 seconds. Shortening this duration by reducing the TCP retransmission timeout is problematic, as the timeout is standardized. But even if we did reduce it, the TCP/IP stack still counts retransmissions as failures and gives up due to too many of them with a ring size of 128 or above.

The throughput of dropping is poor due to the way TCP reacts to drops. New TCP connections begin in a slow start mode, sending at a low rate to avoid exceeding the network capacity. Drops are considered a sign of congestion and cause TCP to reduce the transmission rate further. Likewise, during connection establishment stage, TCP utilizes exponential backoff to avoid overloading the network/receiver. If the cause of the packet loss is rNPFs, communication all but stops. The transmitter waits for receiver acks before increasing the transmission speed; instead, due to timeouts, it reduces the transmission rate. The receiver, on the other hand, depends on more packets to arrive to page-in the receive ring. The effective behavior resembles a deadlock. The issue may become so severe that the TCP maximal retry number is exceeded and the stack announces a failure to the application layer.

Backup Ring Our solution for Ethernet NPFs draws inspiration from paravirtual (software-only) NICs, which handle (CPU) page faults on guest receive rings. Incoming packets are stored in pinned memory of the physical NIC. They are then copied by the hypervisor to swappable buffers posted in the paravirtual guest receive rings. The copy is done by the CPU, so page faults are handled transparently.

Figure 5 depicts our proposed solution, which we denote as *backup ring*. Traffic is received from the network (1). For each incoming packet, the NIC inspects the target receive buffer of the IOuser. If this buffer is available, the data is written directly to it (2). Alternatively, if a page fault is encountered,

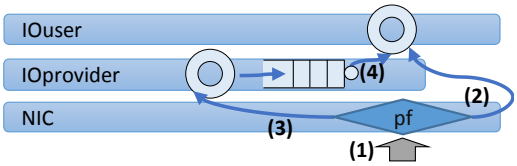

Figure 5. *High level design of the backup ring.*

the packet is written to a small, pinned backup ring owned by the IOprovider (3). After the IOprovider resolves the rNPF, it copies/merges the packet into the original receive buffer (4). The NIC skips IOuser receive descriptors that encounter rNPFs to maintain ordering. For the same reason, the NIC does not report the reception of new packets to the IOuser until all previous page faults have been handled.

Our backup ring is conceptually similar to existing rings and thus enjoys standard optimizations such interrupt coalescing and NAPI [26] that make it fast enough not to run out of space. A key difference is that, with regular rings, packets are steered according to their content, whereas in the backup case, they are steered according to meta data that the NIC adds to allow the IOprovider to find the appropriate IOuser's buffer.

Hardware Figure 6 lists the pseudo-code of a NIC that implements a backup ring. The `head` points to the first descriptor that was not consumed as far as the IOuser is concerned. The NIC tries to store new incoming packets in `head` plus `head_offset`. When there are no pending rNPFs, `head_offset` is zero. With pending rNPFs, `head` keeps pointing to the descriptor associated with the oldest unresolved rNPF: we cannot inform the IOuser that new packets arrived before this rNPF is resolved. The `bitmap` is used to track which descriptors currently experience rNPFs, allowing the NIC to continue storing incoming packets in the IOuser ring regardless of rNPFs. The `bm_size` is the size of the bitmap. It places an upper bound on the number of packets that the IOprovider is willing to store for a specific IOuser ring.

The `recv()` function outlines how the NIC handles a packet `pkt` that is designated for ring *r*. The NIC checks if the packet can be stored in *r*, that is, if the target index does not exceed the tail and if the relevant descriptor and buffers are present. (For brevity we omit such checks as whether the packet fits in the buffer.) Assuming the conditions hold, we store the packet in *r*. If there are pending rNPFs, we only advance `head_offset`. Otherwise, we advance `head` and raise an interrupt to signal reception of new packets.

If the packet cannot be stored in *r*, the NIC attempts to use the backup ring, which is possible if the distance from the first unresolved packet does not exceed the IOprovider's limit in `bm_size`, and if there is room (the packet is dropped otherwise). With the packet, the NIC also stores metadata that the IOprovider needs to resolve the rNPF. It then marks

```
struct ring {
  const int size, bm_size; descriptor_t *descriptor;
  int tail, head, head_offset, bm_index; bit *bitmap;
};
void recv(ring r, Packet pkt) {
  head = r.head + r.head_offset;
  if( r.tail - r.head - r.head_offset < r.size &&
      r.is_descriptor_present(head) ) {
    // store in IOuser ring
    r.descriptor[head % r.size].store(pkt);
    if( r.head_offset ) { r.head_offset++; }
    else                { r.head++; r.raise_isr(); }
  }
  else if( r.head_offset < r.bm_size &&
           backup.tail - backup.head < backup.size) {
    // store in backup ring
    i = backup.head % backup.size;
    bit_index =  bm_index + r.head_offset;
    backup.descriptor[i].store(
      { r.id, head, bit_index, pkt } );
    backup.head++;
    r.bitmap[bit_index % r.bm_size] = 1; r.head_offset++;
    backup.raise_isr();
  } // otherwise drop packet
}
void resolve_rNPFs(ring r, int bm_index) {
   r.bitmap[bm_index % r.bm_size] = 0;
   while (r.head_offset > 0 &&
          r.bitmap[r.bm_index % r.bm_size] == 0) {
     atomic { r.head_offset--; r.head++; r.bm_index++ }
   }
   r.raise_isr();
}
```

Figure 6. *Hardware pseudo-code for the backup ring.*

the bitmap, raises an rNPF, and advances `head_offset` to skip an entry in *r*. We use the field `bm_index` to calculate the offset in the bitmap. This field has a similar role to the `head` field, holding the index of the bit corresponding to the ring entry at `head`. It is required since we support bitmap sizes (`bm_size`) that are different from the associated IOuser ring sizes, thereby making the number of packets the IOprovider agrees to hold for an IOuser independent of the IOuser's ring size.

The IOprovider informs the NIC when finishing to resolve an rNPF of *r*. The NIC then executes `resolve_rNPFs()`, which updates the bitmap, marking that the given index is resolved. The NIC uses the bitmap to update `head` to point to the next unresolved rNPF (or to the top of the ring, if no rNPFs remain). The corresponding iteration might take some time. But it does not exclude packet reception. Only `head` and `head_offset` must be updated together because the destination of new packets is determined by their sum.

Driver We next describe how the IOprovider manages the backup ring. Recall that IOusers are unaware of this ring and can therefore enjoy seamlessly NPF support. As noted, the backup ring is similar to ordinary receive rings and is thus similarly maintained. Its interrupt handler is invoked upon packet arrival. Using the NIC-provided metadata, the handler identifies the associated IOuser for each faulting packet, placing the packet in a software queue *q* of that IOuser. It promptly replenishes the backup ring so as not to run out

of buffers. It then wakes a thread T whose job is to resolve the IOuser's rNPFs. T is required, as rNPF resolution might require sleeping, forbidden in interrupt context.

When resolving an rNPF, T first blocks until there is room in the target IOuser ring r. It then ensures that the corresponding descriptor and buffer(s) are present, and that the IOMMU page tables reflect that. Finally, it copies the packet into the buffer(s) and notifies the NIC that the rNPF has been resolved.

In most cases, we expect r to have room for resolved rNPF packets. But our backup ring mechanism allows the IOprovider to buffer more packets then r's size. The reason underlying this design decision is the following. From the time an rNPF fires until it is resolved, the NIC does not notify the IOuser about newly arriving packets. Hence, the IOuser does not post new buffers in r, risking overflow. As the number of pending packets in q might exceed r's size, T might not be able to process q it in one go. Instead, it might have to process only a few packets and then wait for the IOuser to consume them and post additional buffers on r. Therefore, during this period, T asks the NIC to raise an interrupt whenever the IOuser changes the tail of r.

Prototype As noted, we utilize the Mellanox ConnectX-3 Ethernet NIC as the baseline of our prototype, with modified firmware that provides basic NPF support (Figure 2). Implementing the backup ring flows in firmware, however, is too difficult a task. Unrealistic for us in the scope of a research project. We therefore compromise. We prototype the backup ring flows in the driver of the IOprovider. Doing so is made possible by leveraging the ability of modern NICs to duplicate all incoming packets into *two* receive rings. We build on this feature to accurately emulate hardware-based backup ring functionality for Ethernet rNPFs.

Instead of exclusively redirecting packets that trigger rNPFs to the backup ring (we cannot), our prototype directs *all* packets to two rings: a primary ring p with page faults enabled, and a secondary ring s populated with pinned buffers. In the absence of rNPFs on p, the duplicated packets in s are discarded. But when p gets hit by an rNPF, the driver utilizes s as the backup ring, copying faulting packets from s to an intermediate queue q. Later, after the rNPF is resolved, the driver copies from q to p per the true backup ring design.

There exists a subtle synchronization issue. While the rNPF is resolved, new packets can arrive to p before we complete the processing of s. These packets are also duplicated on both rings. Lacking true rNPFs hardware support, the NIC neither skips faulting receive ring entries, nor reports how many packets are dropped during the rNPF, so there is a question of when to switch back from s to p. We resolve this question by comparing the contents of the packets in s to the packet at the head of p and resume normal operation once the two match.

Our prototype entails a cost. The maximal throughput attainable by the testbed system that houses our ConnectX-3 NIC (see §6) is 24 Gbps, due to PCIe bus limitations. Duplicating every packet halves this throughput, allowing a maximum of only 12 Gbps. Thus, our prototype models a weaker, 12 Gbps NIC. We stress that halving the throughput is *not* a limitation of our proposed rNPF support for Ethernet NICs. Rather, it is a consequence of lacking true hardware support for rNPFs and having to emulate such support somehow. A NIC supporting the backup ring flows in hardware will *not* duplicate the PCIe traffic; it will simply steer incoming faulting packets to a different memory location. Duplicating is turned on in all the evaluated Ethernet configurations—backup, dropping, and pinning—to allow for an apples-to-apples comparison, using the same simulated NIC.

Going back to Figure 4(a), we see that the backup ring solution performs as well as pinning. Figure 4(b) shows that, for much larger ring sizes, while the ring is cold, the overhead of our solution is nonnegligible but still allows the workload to recover after a tolerable delay. When the ring warms up, the performance of our solution becomes identical to that of pinning—a behavior one would expect from demand-paging.

6. Evaluation

In our evaluation, we wish to answer the following questions. Is our mechanism sufficient to sustain good performance? Can we demand-page and utilize the memory effectively? For example, can memory dynamically move to the IOuser that needs it the most? And finally, does NPF support simplifies code as argued? We demonstrate that the answer to all of these questions is yes.

Using several real-world applications, we evaluate the impact of network page faults with respect to memory utilization (§6.1), performance overheads (§6.2), and code complexity (§6.3). We examine use cases of high performance computing (HPC) interconnect fabric, a key-value store workload, and a storage systems workload. We then conduct a what-if analysis that measures the impact of NPFs on the system's network behavior under synthetic load (§6.4).

Our Ethernet experimental setup consists of two Dell PowerEdge R210 II Rack Servers with 8GB 1333MHz memory and a 4-core Intel Xeon E3-1220 CPU at 3.10GHz running Ubuntu 13.10 with a Linux 3.11.4. The server machine communicates through the 12 Gbps NPF-supporting prototype NIC described in §5. The client has the same underlying ConnectX-3 40Gbps without the added NPF support. The NICs are connected back-to-back; the asymmetry between their throughputs might cause packet loss disturbing our measurements. We avoid this problem by enabling flow control [51].

Our InfiniBand setup consists of eight HP ProLiant DL380p Gen8 servers with 128GB memory and a 12-core dual socket Xeon E5-2697 v2 CPU at 2.7GHz running RedHat 7.0 with Linux 3.10. Nodes communicate via NPF-supporting 56 Gbps Connect-IB NICs described in §4. The cluster is connected through a SwitchX-2 SX6036 switch.

memcached instances	*1*	*2*	*3*	*4*
NPF	186	311	407	484
pinning	185	310	N/A	N/A

Table 5. *The aggregated throughput of all the running memcached instances in kilo transactions per second.*

6.1 Memory Utilization

The ability to overcommit the memory—allowing the aggregated size of all virtual address spaces to be bigger than the physical memory such that physical pages are allocated based on actual demand—is crucial for performance [7, 38, 45, 48, 102, 107, 109, 118]. We demonstrate how NPF support improves the IOprovider's ability to overcommitment the memory by using our running example described in §5 of a memcached utilizing direct network I/O (Ethernet). While memcached is not an optimal workload under memory pressure, it serves to demonstrate how cloud operators cannot always tell in advance the memory access pattern and semantics of their applications, when deciding how to overcommit memory. Memory overcommitment also supports easier deployment. The operator can configure the system minimal thresholds for the worst case scenario, while still benefiting from more memory when it is available. We demonstrate this benefit using a storage server (InfiniBand).

Key-value Store As noted in §2.2, all production IOproviders pin the entire address space of IOusers when assigning to them SRIOV or DPDK IOchannels [14, 30, 32, 52, 62, 113, 114, 119]. (Research by Amit et al. overcome this difficulty by exposing a nested IOMMU to guest VMs, finding a massive degradation in performance, which motivated the use of extra "sidecores" for pinning activity as well as security compromises [5].) To demonstrate the problematic nature of this limitation, we conduct an experiment whereby a VM is allocated (thinks it has) 3GB of (guest physical) memory, but its working set is smaller than 2GB. Recall that our testbed host machine is equipped with 8GB of memory. It can therefore adequately support four VMs in this case, in principle. As can be seen in Table 5, with NPF support, it is indeed possible to run such four VMs together in a productive manner. Conversely, without NPF support, the IOprovider is unable to run more than two VMs, because their aggregated virtual memory size is 9GB—bigger than the physical memory.

In the previous experiment, the working sets of the VMs were static. They did not change over time. NPFs, however, additionally support dynamic reallocation of the physical memory as the IOuser working set temporally evolves. To demonstrate this benefit, our current experiment includes two memcached IOusers instances whose working set changes; that is, the aggregated size of the items that memaslap accesses changes (we configure memaslap to use 20KB per item). For the first memcached instance, the set increases by a factor of nine (100MB to 900MB) after 50 seconds of execution. For the other instance, it shrinks by a factor of nine at the same time (900MB to 100MB). Using cgroups, we constrain the aggregated size of both instances to 1GB, simulating a host with that much physical memory.

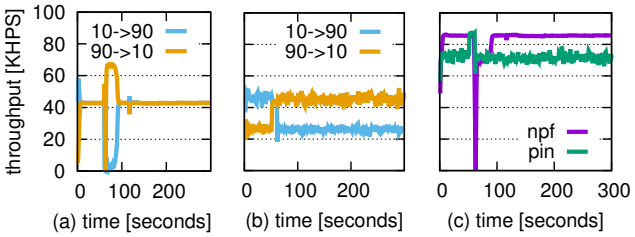

Figure 7. *Pinning vs. NPF with dynamic working set: with NPFs (a), with pinning (b), and combined throughput (c).*

When NPFs are unsupported, pinning is required, so we have no choice but to statically divide the physical memory between the two, 500MB per instance. With NPF support, no such division is necessary, as pages are mapped in the IOMMU on demand based on actual usage. The metric we use to report the results of this experiment is *hits* per second, rather than the default transactions per second, because the latter also include misses. (Recall that memcached is an LRU *cache*, and so its hit rate is affected by its size.)

Figure 7 shows the throughput of the individual memcached instances with NPF support (a) and without (b). The change in working sets is evident in both cases at 50 seconds, resulting in a short transition period after which the system stabilizes. With NPF support, both instances enjoy the same performance: the physical memory is big enough to hold their working sets and DMA page faults of the direct channels are adequately serviced. In contrast, when NPFs are unsupported, one of the two instances always suffer: its statically allocated memory (500MB) is too small to hold its working set (900MB). Figure 7(c) shows the aggregated throughput of the instances, demonstrating why NPF support is advantageous.

Storage Modern data centers often separate compute and storage resources, relying on remote storage using protocols such as iSCSI. RDMA technology has been shown to improve performance in such cases [21, 72], as it reduces CPU utilization. Until now, user-space based arrays that use kernel bypass and RDMA required pinned memory. One had to statically configure the size of the buffers pinned for network communication. These come at the expense of page cache buffers that cache disk content. Our paging-capable solution alleviates this problem by allowing the virtual memory subsystem to manage the communication buffers on demand.

We evaluate the effect of NPFs on storage systems by using tgt [35], an iSCSI target implementation that supports iSCSI extensions for RDMA (iSER) [22]. We use fio [11] to benchmark a tgt storage target. The fio tester is using an unmodified Linux kernel iSER initiator. We evaluate its performance by measuring the random 512KB read bandwidth,

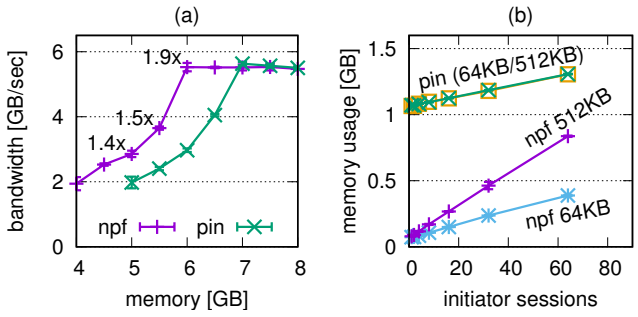

Figure 8. *Storage bandwidth with single initiator and varying memory limit (a). Memory usage with multiple initiators and a fixed memory limit (b).*

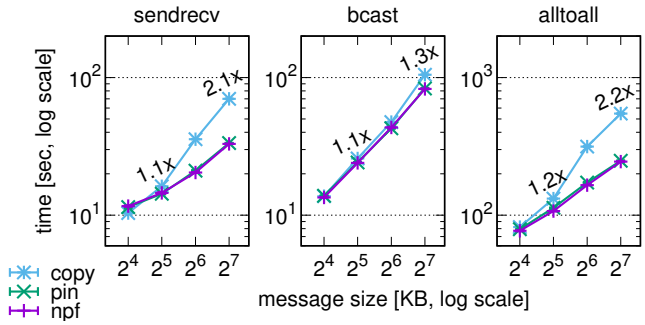

Figure 9. *IMB runtime for IMB benchmarks as a function of message size. The labels show the ratio between the runtimes associated with copying and pinning.*

varying the amount of memory available at the machine. The tgt daemon exposes a single LUN of 4GB, stored on a single high-performance hard drive. A remote machine mounts the iSCSI LUN and performs random reads. We run the baseline tgt with pinned network buffers and compare it against a modified tgt that relies on NPFs for correct DMAing.

Figure 8(a) shows the results. With less than 5GB of available memory, the pinned configuration fails to load the tgt service, whereas the NPF configuration successfully runs with as little as 4GB of memory available. As we increase the amount of memory available to the target machine, it is able to cache a larger number of the blocks from the underlying storage, accelerating the read operations. In the pinned configuration, the static memory allocation leaves a smaller amount of memory to the page cache, inducing more cache misses. In this experiment, NPFs improve performance by up to 1.9x. Only with 7GB or more, the pinned configuration finally has enough memory to cache the entire disk.

The baseline tgt server statically allocates a 1GB buffer for communication. One might wonder if such a large buffer is needed. To answer this question, we run the same experiment with a varying number of iSCSI initiators and a fixed memory limitation of 6 GB. Figure 8(b) shows the amount of resident memory used by the tgt daemon process. Increasing the number of initiators results in more communication buffers used by the target, increasing the total memory usage. Additionally, tgt allocates a fixed size chunk (512KB) for each transaction, regardless of its actual size. Thus, with 64KB blocks, there are communication buffers in the process address space that are never used. With NPFs, these virtual addresses never get backed by physical frames. With 512KB blocks, the memory usage nears the amount used by the pinned configuration, although because the machine is limited to use only 6 GB, there is still enough memory pressure to cause the OS to evacuate pages of the communication buffers.

6.2 Overhead of Pinning and Copying

Next, we compare the performance of our NPF solution to the alternatives of pinning and unpinning dynamically, or copying data in and out of a statically pinned buffer (InfiniBand setup). Direct I/O and RDMA are commonly used in high-performance computing environments, as the gains in communication latency and lower CPU utilization are advantageous for HPC applications. We thus use HPC benchmarks to measure the effect of NPF, focusing on the message passing interface (MPI) standard for communication. We use OpenMPI with a Mellanox communication backend. In order to focus on the inter-machine communication, we run only a single process on each node.

To drive MPI, we use the Intel MPI benchmarks suite (IMB) [27], which contains a collection of benchmarks that measure performance of a specific MPI operation. Figure 9 shows the results of three of these benchmarks. We compare between three configurations: NPF, copying, and pinning—a state-of-the-art heuristic pin-down cache that is part of our MPI communication backend. (Pin-down caches were discussed in §2.2.) We use the IMB "off_cache" mode to increase the working set of the benchmark. This mode prompts the pin-down cache logic to register multiple memory buffers instead of just one. Figure 9, displays the throughput of the three benchmarks as a function of the message size, highlighting the advantage of RDMA (zero copy) over copying, especially for larger messages. The NPF configuration achieves similar performance as the state-of-the-art pin-down cache.

A few of the IMB benchmarks, such as allreduce (not shown), exhibit little difference between copying and pinning. In the case of allreduce, the reason is that this benchmark performs a CPU calculation as part of its reduction operation, forcing it to copy the data into and out of the CPU cache in any case, and thus hampering the performance gain of RDMA. For such primitives, there is likewise no observable difference between the pinning and the NPF configurations.

Another HPC benchmark we use is the effective communication bandwidth benchmark ("beff") [63], designed to model real HPC workloads. This benchmark evaluates the accumulated bandwidth of the communication network by measuring several message sizes, communication patterns, and MPI operations. The results are shown in Table 6, highlighting again

app	*pinning*	*NPF*	*copying*
beff	$16,410 \pm 45$	$16,440 \pm 10$	$8,020 \pm 20$

Table 6. *Performance of beff in MB/sec.*

that there is benefit in using RDMA over copying, and that NPF provides this benefit without the need to pin.

HPC workloads typically do not swap to secondary storage and run with enough permissions to lock the entire physical memory if needed. Because of that, the pin-down cache rarely needs to unpin memory and therefore oftentimes acts more like a solution that pins everything after some warm-up. It seems reasonable to expect, however, that with HPC-like cloud workloads becoming more commonplace, administrators will tighten the limitations on memory use, as well as charge IOusers in a manner that is proportional to the amount of resources they consume. Such setups will drive pin-down caches to exercises their eviction policies and dynamically pin/unpin at a finer granularity, thereby inducing significant overheads [5, 14, 103, 117], which can be avoided on systems that provide NPF support.

6.3 Programming Complexity

Given similar resources, a well-tuned, well-designed pin-down cache may perform similarly to a solution based on NPFs. But why should developers individually invest their efforts in designing pin-down strategies when the virtual memory subsystem can do it collectively for them based on actual use? It is difficult to quantitatively measure the benefit of NPFs in reducing code complexity. One possible way to try is to consider the number of lines of code (LOC) involved. For example, to port the tgt storage daemon to use NPFs, we modified about 40 LOC. The daemon was able to perform significantly better as a result.

How many LOC would we have devoted to resolve the problem without NPFs? In the MPI middleware library that we used, we estimate that thousands of LOC can be disabled when using NPFs. Managing the pin-down cache, the data structures tracking what has been pinned, and the eviction and pinning policies can all be eliminated. Another pin-down cache example is the Firehose algorithm [14], whose implementation took nearly 8.5K LOC [36]. (Our benchmarks are based on MPI, so we cannot compare them directly with the GASNet-based Firehose algorithm, but we expect that NPFs would provide similar performance to Firehose when using the same resources.)

6.4 What-If Analysis

With the exception of cold rings, all of our experiments (those we described above, and those we did not) indicated that spontaneous rNPFs are rare. Still, our experience is limited to the workloads we evaluated. It is possible that other workloads will exhibit different characteristics. Here, we evaluate the consequences of more frequent rNPFs. Our baseline application measures maximal bandwidth, similarly to Netperf TCP stream [59]. The sender iteratively transmits

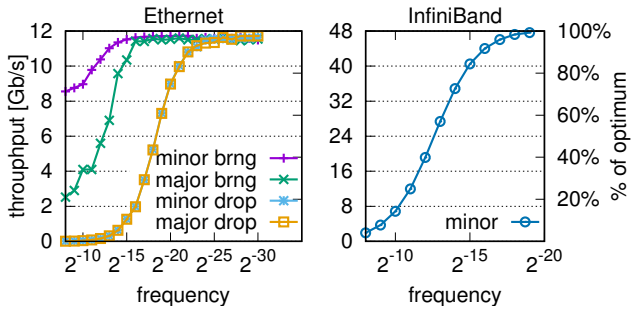

Figure 10. *Throughput of the stream benchmark with rNPFs; "brng" denotes backup ring.*

64KB messages using a standard Linux TCP stack. The receiver runs our lwIP stack and counts how many packets have been received to report throughput. To allow for a comparison between Ethernet and InfiniBand rNPFs, we additionally use the equivalent ib_send_bw InfiniBand stream benchmark from the perftest package. Both benchmarks are modified to synthetically generate rNPFs at a variable specified frequency. They pre-fault the receive ring at startup to eliminate the cold ring problem.

Figure 10 shows the results. The left y-axis denotes raw throughput (benchmarks are differently scaled along this axis). The right y-axis denotes throughput relative to the optimum. On the left, we see that the backup ring significantly improves performance for both major and minor rNPFs (with and without disk access). The page fault type does not matter when dropping, as the TCP retransmission timer is much longer than the time it takes to resolve a major page fault. The hardware implementation, shown on the right, notifies the remote sender immediately upon a page fault. The notification allows the sender to use a relatively short NPF-specific timeout resulting in significant performance improvement relative to dropping. Nonetheless, network utilization-wise, this solution is less efficient then the backup ring solution.

7. Conclusions

As direct network I/O becomes more popular and widely used, we contend that it makes sense for systems to add support for network page faults. We design and implement such support for InfiniBand and Ethernet NICs and show that it improves the utilization of the memory, increases overall performance, and simplifies the program model. Mellanox InfiniBand NICs already provide NPF support as described in this paper (§4); this feature is called "on-demand paging" (ODP).

Acknowledgments

We thank the anonymous reviewers for their helpful feedback and Yuval Dagan, who assisted with the HPC benchmarking. This research is partially funded by the Israeli Ministry of Economics via the HIPER consortium.

References

[1] Brian Aker and Mingqiang Zhuang. Memaslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memaslap.html. libmemcached 1.1.0 documentation. Accessed: May 2016.

[2] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, Rong Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pages 1270–1277, Sept 2008. http://dx.doi.org/10.1109/ALLERTON.2008.4797706.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, April 1999.

[4] AMD Inc. AMD IOMMU architectural specification, rev 2.00. http://developer.amd.com/wordpress/media/2012/10/488821.pdf, Mar 2011. Accessed: May 2016.

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

[6] 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. *Communications of the ACM (CACM)*, 59(1):108–116, Jan 2016. http://dx.doi.org/10.1145/2845648.

[7] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–366, 2014. http://dx.doi.org/10.1145/2541940.2541969.

[8] Apple Inc. Thunderbolt device driver programming guide: Debugging VT-d I/O MMU virtualization. https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/DebuggingThunderboltDrivers/DebuggingThunderboltDrivers.html, 2013. Accessed: May 2014.

[9] Andrea Arcangeli. Integrating KVM with the linux memory management. In *KVM Forum*, 2008.

[10] ARM Holdings. ARM system memory management unit architecture specification — SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062c/IHI0062C_system_mmu_architecture_specification.pdf, 2013. Accessed: Jan 2015.

[11] Jens Axboe. Fio – flexible IO tester. http://git.kernel.dk/?p=fio.git.

[12] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM Eurosys*, pages 73–85, 2006.

[13] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

[14] Christian Bell and Dan Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2003. http://dx.doi.org/10.1109/IPDPS.2003.1213363.

[15] Muli Ben-Yehuda, Orna Agmon Ben-Yehuda, and Dan Tsafrir. The nom profit-maximizing operating system. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 145–160, 2016. http://dx.doi.org/10.1145/2892242.2892250.

[16] 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 System Design and Implementation (OSDI)*, pages 423–436, 2010. http://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf.

[17] 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/mirror/ols2007v1.pdf#page=9.

[18] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–35, 2008. http://dx.doi.org/10.1145/1346281.1346286.

[19] Robert Birke, Lydia Y Chen, and Evgenia Smirni. Data centers in the wild: A large performance study. Technical Report RZ3820, IBM Research, 2012. http://domino.research.ibm.com/library/cyberdig.nsf/papers/0C306B31CF0D3861852579E40045F17F.

[20] James E.J. Bottomley. Dynamic DMA mapping using the generic device. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/DMA-API.txt?id=refs/tags/v3.18.3. Linux kernel documentation. Accessed: Jan 2015.

[21] Ethan Burns. *Implementation and comparison of iSCSI over RDMA*. PhD thesis, University of New Hampshire, 2008.

[22] Mallikarjun Chadalapaka, Uri Elzur, Michael Ku, Hemal Shah, and Patricia Thaler. A Study of iSCSI Extensions for RDMA. In *Computer-Communication Networks*, August 2003.

[23] Yuqun Chen, Angelos Bilas, Stefanos N. Damianakis, Cezary Dubnicki, and Kai Li. UTLB: A mechanism for address translation on network interfaces. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 193–204,

1998. http://dx.doi.org/10.1145/291069.291046.

[24] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001.

[25] Jonathan Corbet. *Linux Device Drivers*, chapter 15: Memory Mapping and DMA. O'Reilly, 3rd edition, 2005.

[26] Jonathan Corbet. Newer, newer NAPI. LWN https://lwn.net/Articles/244640/, Aug 2007. (Accessed: Aug 2016).

[27] Intel Corporation. Intel MPI benchmarks. https://software.intel.com/en-us/articles/intel-mpi-benchmarks, 2013.

[28] Crehan Research. Another year of robust growth and record shipments for branded data center switches. http://www.crehanresearch.com/wp-content/uploads/2015/03/CREHAN-2014-Data-Center-Switching-CR.pdf, Mar 2015. (Accessed: Aug 2015).

[29] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. ReNIC: Architectural extension to SR-IOV I/O virtualization for efficient replication. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):40:1–40:22, Jan 2012. http://dx.doi.org/10.1145/2086696.2086719.

[30] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevic.

[31] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[32] Montse Farreras, George Almasi, Calin Cascaval, and Toni Cortes. Scalable RDMA performance in PGAS languages. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009. http://dx.doi.org/10.1109/IPDPS.2009.5161025.

[33] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, Aug 2004. http://dl.acm.org/citation.cfm?id=1012889.1012894.

[34] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, March 2013.

[35] Tomonori Fujita and Mike Christie. tgt: Framework for Storage Target Drivers. In *Proceedings of the Linux Symposium*, July 2006.

[36] GASNet 1.26.0. https://gasnet.lbl.gov/GASNet-1.26.0.tar.gz, October 2015. (Accessed: May 2016).

[37] Dror Goldenberg, Michael Kagan, Ran Ravid, and Michael S. Tsirkin. Zero copy sockets direct protocol over InfiniBand – preliminary implementation and performance analysis. In *IEEE Symposium on High Performance Interconnects (HOTI)*, pages 128–137, 2005. http://dx.doi.org/10.1109/CONECT.2005.35.

[38] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 154–169, 1999. http://dx.doi.org/10.1145/319344.319162.

[39] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM (CACM)*, pages 85–93, 2010. http://dx.doi.org/10.1145/1831407.1831429.

[40] James Hamilton. AWS innovation at scale. https://www.youtube.com/watch?t=113&v=JIQETrFC_SQ, Nov 2014. (Accessed: Aug 2015).

[41] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. In *USENIX Annual Technical Conference (ATC)*, pages 231–242, 2013. https://www.usenix.org/system/files/conference/atc13/atc13-harel.pdf.

[42] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)*, pages 41–50, 2007.

[43] Gregory D. Hill and Albert H. Chen. High performance network multiplexing with IX++. Research report, Stanford University, 2015. http://hselin.com/resources/CS344g_ixplusplus_final%20paper.pdf.

[44] Michael R. Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 130–137, 2011. http://dx.doi.org/10.1109/CloudCom.2011.27.

[45] Eric Horschman. Hypervisor memory management done right. http://blogs.vmware.com/virtualreality/2011/02/hypervisor-memory-management-done-right.html, 2011. (Accessed: May 2016).

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

[47] HSA Foundation. HSA-Drivers-Linux-AMD. https://github.com/HSAFoundation/HSA-Drivers-Linux-AMD. (Accessed: May 2016).

[48] Woomin Hwang, Yangwoo Roh, Youngwoo Park, Ki-Woong Park, and Kyu Ho Park. HyperDealer: Reference pattern aware instant memory balancing for consolidated virtual machines. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 426–434, 2014. http://dx.doi.org/10.1109/CLOUD.2010.70.

[49] IBM Corporation. PowerLinux servers — 64-bit DMA concepts. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm. Accessed:

May 2014.

[50] IBM Corporation. AIX kernel extensions and device support programming concepts. https://publib.boulder.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.kernelext/doc/kernextc/kernextc_pdf.pdf, 2013. Accssed: May 2014.

[51] IEEE. Specification for 802.3 full duplex operation. IEEE Standard 802.3x http://dx.doi.org, 1997.

[52] VMware Inc. Configuring VMDirectPath I/O pass-through devices on a VMware ESX or VMware ESXi host. https://kb.vmware.com/kb/1010789. VMware Knowledge Base. Accessed: Aug 2016.

[53] InfiniBand Trade Association (IBTA). About InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband. (Accessed: May 2016).

[54] InfiniBand Trade Association (IBTA). About RoCE. http://www.infinibandta.org/content/pages.php?pg=about_us_RoCE. (Accessed: May 2016).

[55] Intel. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html, Jan 2011.

[56] Intel Corporation. DPDK: Data plane development kit. http://dpdk.org. (Accessed: May 2016).

[57] Intel Corporation. Intel virtualization technology for directed I/O - architecture specification - specification - Rev. 2.2. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Sep 2013. Accessed: Jan 2015.

[58] Intel Corporation. Intel virtualization technology for directed I/O - architecture specification - Rev. 2.3. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Oct 2014.

[59] Rick A. Jones. Netperf: A network performance benchmark (Revision 2.0). http://www.netperf.org/netperf/training/Netperf.html, 1995. Accessed: August, 2016.

[60] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1997.

[61] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[62] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 67–81, 2016. http://dx.doi.org/10.1145/2872362.2872367.

[63] Alice E. Koniges, Rolf Rabenseifner, and Karl Solchenbach. Benchmark design for characterization of balanced high-performance architectures. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*, IPDPS '01, pages 196–, Washington, DC, USA, 2001. IEEE Computer Society.

[64] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote I/O. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–65, 2016. http://dx.doi.org/10.1145/2872362.2872378.

[65] George Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD Inc., Aug 2012. Rev. 1.0 http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf (Accessed: May 2016).

[66] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2004. https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/levasseur/levasseur.pdf.

[67] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 476–488, 2015. https://doi.org/10.1145/2749469.2750416.

[68] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[69] getrlimit(2) – Linux man page. http://linux.die.net/man/2/getrlimit. (Accessed: May 2016).

[70] mlock(2) – Linux man page. http://linux.die.net/man/2/mlock. (Accessed: May 2016).

[71] The include/uapi/linux/resource.h header file of Linux 4.5. http://lxr.free-electrons.com/source/include/uapi/linux/resource.h?v=4.5#L71. (Accessed: May 2016).

[72] Jiuxing Liu, Dhabaleswar K. Panda, Jiuxing Liu Dhabaleswar K. P, and Mohammad Banikazemi. Evaluating the impact of RDMA on storage I/O over InfiniBand. In *SAN-03 Workshop (in conjunction with HPCA), 2004*, 2004.

[73] 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)*, pages 355–368, 2015.

[74] 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.

[75] Vinod Mamtani. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007. Accessed: May 2014.

[76] Alex Markuze, Adam Morrison, and Dan Tsafrir. It's DAMN time for overhead-free IOMMU protection. Submitted.

[77] 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. http://dx.doi.org/10.1145/2872362.2872379.

[78] Paul Menage. Cgroups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[79] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 124–133, 2006. http://dx.doi.org/10.1007/11823285_13.

[80] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 89–104, 2002.

[81] Jarek Nieplocha, Vinod Tipparaju, Amina Saify, and Dhabaleswar K. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2002. http://dx.doi.org/10.1109/IPDPS.2002.1016563.

[82] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. FasTrak: Enabling express lanes in multi-tenant data centers. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 139–150, 2013. http://dx.doi.org/10.1145/2535372.2535386.

[83] Li Ou, Xubin He, and Jizhong Han. An efficient design for fast memory registration in RDMA. *Journal of Network and Computer Applications*, 2009.

[84] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2015. https://doi.org/10.1145/2815400.2815423.

[85] PCI-SIG. Single root I/O virtualization and sharing 1.0 specification. http://www.pcisig.com/specifications/iov/single_root/, Sep 2007. (Accessed: Aug 2016).

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

[87] PCI-SIG. Single root I/O virtualization and sharing 1.1 specification. http://www.pcisig.com/specifications/iov/single_root/, Jan 2010. (Accessed: Aug 2016).

[88] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU Scalably. In *USENIX Annual Technical Conference (ATC)*, 2015.

[89] Simon Peter, Jialin Li, Doug Woos, Irene Zhang, Dan R. K. Ports, Thomas Anderson, Arvind Krishnamurthy, and Mark Zbikowski. Towards high-performance application-level storage management. In *USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2014. https://www.usenix.org/system/files/conference/hotstorage14/hotstorage14-paper-peter.pdf.

[90] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[91] Renato J. Recio, Bernard Metzler, Paul R. Culley, Jeff Hilland, and Dave Garcia. A remote direct memory access protocol specification. RFC 5040, The Internet Engineering Task Force (IETF) Network Working Group, 2007. https://tools.ietf.org/html/rfc5040 (Accessed: May 2016).

[92] Bruce Richardson. [dpdk-dev] memory pinning. http://dpdk.org/ml/archives/dev/2014-June/003937.html, 2014. (Accessed: Aug 2016).

[93] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference (ATC)*, pages 101–112, 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[94] Phil Rogers. Heterogeneous System Architecture (HSA): Overview and implementation. In *Hot Chips*, 2013. HC25. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.0T1-Hetero-epub/HC25.25.100-Intro-Rogers-HSA%20Intro%20HotChips2013_Final.pdf (Accessed: May 2016).

[95] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, 2007. http://dx.doi.org/10.1145/1294261.1294294.

[96] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 317–332, 2016. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi.

[97] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. Securing self-virtualizing Ethernet devices. In *USENIX Security Symposium*, pages 335–350, 2015.

[98] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 33–46, 2010. https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Soares.pdf.

[99] Vaidyanathan Srinivasan, Anand K. Santhanam, and Madhavan Srinivasan. Cell Broadband Engine processor DMA engines, Part 1: The little engines that move data. http://www.ibm.com/developerworks/library/pa-celldmas, 2005. (Accessed: May 2016).

[100] B. Stephens, A.L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *INFOCOM, 2014 Proceedings IEEE*, pages 1824–1832, April 2014.

[101] Michael Swift, Brian Bershad, and Henry Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, Feb 2005.

[102] Taneja Group. Hypervisor shootout: Maximizing workload density in the virtualization platform. http://www.vmware.com/files/pdf/vmware-maximize-workload-density-tg.pdf, 2010. (Accessed: May 2016).

[103] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *IEEE International Parallel Processing Symposium (IPPS)*, pages 308–314, 1998. http://dx.doi.org/10.1109/IPPS.1998.669932.

[104] Animesh Trivedi. Remote direct memory access (RDMA) 101 – quick history lesson and introduction. http://0x8086.blogspot.com/2011/11/remote-direct-memory-access-rdma-101.html, 2011. (Accessed: May 2016).

[105] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzicker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 1–15, 2016. http://dx.doi.org/10.1145/2731186.2731189.

[106] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Marlin: A memory-based rack area network. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 125–136, 2014. http://doi.acm.org/10.1145/2658260.2658262.

[107] Gabrie van Zanten. Memory overcommit in production? YES YES YES. http://www.gabesvirtualworld.com/memory-overcommit-in-production-yes-yes-yes/, 2010. (Accessed: May 2016).

[108] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[109] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002. https://www.usenix.org/legacy/events/osdi02/tech/waldspurger.html.

[110] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015. https://doi.org/10.1145/2815400.2815419.

[111] Wikipedia. iWARP – internet Wide Area RDMA Protocol. https://en.wikipedia.org/wiki/IWARP. (Accessed: Aug 2016).

[112] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 241–254, 2008.

[113] Alex Williamson. VFIO: A user's perspective. In *KVM Forum*, 2012. http://www.linux-kvm.org/images/b/b4/2012-forum-VFIO.pdf. (Accsessed: May 2016).

[114] Alex Williamson. [qemu-devel] Intel IOMMU guest emulation and vfio-pci passthrough. https://lists.gnu.org/archive/html/qemu-devel/2015-11/msg04284.html, Nov 2015. (Accessed: Aug 2016).

[115] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann.pdf.

[116] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 31–40, 2009. http://dx.doi.org/10.1145/1508293.1508299.

[117] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand: Design and performance evaluation. In *International Conference on Parallel Processing (ICPP)*, pages 125–132, 2003. http://dx.doi.org/10.1109/ICPP.2003.1240573.

[118] Xiaowei Yang, Chuan Ye, and Qiangmin Lin. Evaluation and enhancement to memory sharing and swapping in Xen 4.1. In *Xen Summit*, 2011. http://tinyurl.com/xen-mem-share-swap (Accessed: May 2016).

[119] 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. http://dx.doi.org/10.1145/1815695.1815718.